Introduction to Programming in Perl

Diego Diez

June 5, 2008

Variables

Variables are places where we store values. We want to store them to use them later. These values can be numbers or string characters. Numeric variables can be used in equations. String variables can be combined with other strings. Perl does not really distinguish between numbers or strings; it evaluates the type based on what we want to do. This makes thing simpler. There are three type of variables:

Scalar

Scalars are used to store single values, either a number or a word (or a single character word). Scalars are identifies by the **$** symbol. For example:

```
# scalar variable $a storing the value "10"
$a = 10;

# scalar variable $b storing the value "exon"
$b = "exon";
```

Arithmetic operators You can apply to scalar several arithmetic operators. With numbers you have sum, subtraction, module, etc. With strings the basic operator is the concatenation operator . that combines two strings together.

```
# with numbers you can use + - / * %
$a = $a + 3 # now $a is "13"

# with strings you may want to concatenate them.
$b = "false ".$b # now $b is "false exon"
```

Arrays

Arrays are used to store multiple values in the same place. Although in principle the values can be a mix of different types (like numbers and strings), arrays are usually employed with values of the same type. That is, all numbers or all strings or all objects of a given type (e.g. bioperl Seq objects). Arrays are identified by the **@** character:

```
# array variable @a containing 3 values.
@a = (10, 20, 30);

# array variable @a containing 4 values:
@b = ("Peter", "David", "John", "Anne");

# although it could be also this way:
# heterogeneous arrays like this one use to come from file parsing.
@c = ("Gene1", 1000, 1200, "H-Ras pseudogene");
```

Arrays have a dimension, i.e. the number of elements in the array. Individual array values are accessed using the index. The index is the position of the placeholder we want to obtain. To access values be use the square brackets "[" and "]". Because the returned value is a single value, i.e. a scalar it is not longer an array. Therefore when we use the variable that way we put the "$" prefix:

```
# this will print the number of element of @a, in the example
# above, will be "3"
print scalar @a;

OUTPUT
$ 3

# to access the first element of the array (indexes begin from zero):
print $a[0];

OUTPUT
$ 10

# to access the last one:
print $a[2];

OUTPUT
$ 30

# or alternatively, using an alias to the last one (whatever it is):
print $#a;

OUTPUT
$ 30
```

Hashes

A hash is a variable used to store heterogeneous data in a key/value fashion. They are useful to group a set of related but heterogeneous variables. For example, think about the attributes of a person. We have the name, age, hair color, eyes color, etc. We can use a hash to represent all these values related to a person:

```
%Person = (
    name => 'Peter',
```

```
    age  => 25,
    hair => 'dark',
    eyes => 'blue',
    # and so on...
);

# To access one's hash value we use the "{" brackets with the key we want.
# Again, the returned value is an scalar, hence the "$" notation.

print $Person{"name"};

OUTPUT
$ Peter

print $Person{"age"};

OUTPUT
$  25
```

Hashes are the basis of object oriented (OO) classes in Perl. Bioperl uses OO programming, therefore that means that all those object in its essence are hashes.

Built in functions

Perl provide a set of built in functions that can be used for specific tasks. Like the *print* function above to output the value to the standard output (stdout). There are also arithmetic functions to compute square root, sin, logarithm, etc. String functions to split a string into several substrings, based on a character, join independent characters into one single string, etc. A list of built in functions can be consulted with the command;

```
(console prompt)$ man perlfunc

# this example will get the square root of the value in the variable:
$a = 100;
print sqrt $a;

OUTPUT
$ 10
```

Standard streams

Before we referred to STDOUT, this is a default stream. This indicates where the output of a function will go by default. STDOUT use to be associated to the console, and the *print* function in Perl by default prints to STDOUT. This means that *print* by default will output in the console.

Similarly STDERR is an stream used to output program errors. By default outputs to the console, i.e. it appears in the same place than the output of STDOUT.

STDIN indicates the default input stream. It is used to obtain input from the users. For example:

```
# this will prompt the user for a radius and then compute some circle values.
print "What is the radius of the circle? ";
chomp ($r = <STDIN>);
$diameter = (2 * $r);
$area = (3.14 * ($r ** 2));
$cir = ($diameter * 3.14);
print "Radius: $r\n Diameter: $diameter\n Circumference: $cir\n Area: $area\n";
```

Flux control

Flux control refers to the methods we have to alter the linear events in our programs. We want our programs to behave different based on user input, variable values, etc. To accomplish that we need some way to control the flux of the code. These are the most common methods in Perl:

If ... elsif ... else

Sometimes, we need to alternate between different options, depending on the value of one variable. This is the purpose of the "If ... elsif ... else" statement. The following example is self-explanatory.

```
if ($a < 0) {
    print "Variable A must be negative. Do something!";
} elsif ($a > 0) {
    print "Variable A must be positive. Great!";
} else {
    print "Variable A must be zero!. Let do something else...";
}
```

Logical operators It is convenient to talk now about logical operators. Logical operators allow us to evaluate if certain expression are *TRUE* or *FALSE*. This is very useful in combination with the flux control structures above. Logical operators evaluate the expression on their left and right based on some specific rule. Table 1 has a list with the most common used ones.

Examples:

```
if ($a < 0 or $a > 0) {
    print "Variable A is NOT zero. Do something!";
} else {
    print "Variable A must be zero!. Lets do something else...";
}

if ($a != 0) {
    print "Variable A is NOT zero. Do something!";
} else {
    print "Variable A must be zero!. Let do something else...";
}
```

Operator	*Description*	*Returned value*
and	*AND* operator	TRUE if both A and B are TRUE
or	*OR* operator	TRUE if either A or B are TRUE
eq	*EQUAL*	TRUE if string A is the same as B
==	*EQUAL* operator	TRUE if number A is equal to number B
ne	*NOT EQUAL* operator	TRUE if string A is different from B
!=	*NOT EQUAL* operator	TRUE if number A is different from B
$<$	less than	TRUE if A in smaller than B
$>$	greater than	TRUE if A is greater than B
$\leq$	less or equal than	TRUE if A is smaller OR equal to B
$\geq$	greater or equal than	TRUE if A is greater OR equal to B

Table 1: Some logical operators in Perl.

For, While and other loops

Sometimes what we want is to cycle through a loop, until one variable reach some value. There are two ways to do this. In the first case, we want to control the number of times the loop is iterated:

```
# this is a C-style for loop:
for ($a = 0; $a < 10; $a++) {
    print "A now is $a!\n";
}
# the same thing more Perl-like:
for $a (0 .. 9) {
    print "A now is $a!\n";
}
# with arrays we can use foreach (well we can use also just 'for').
@a = ("Peter", "David", "Anne");
foreach $name (@a) {
    print "Name is: $name\n";
}

# sometimes we don't want to loop a fixed number of times but to
# finish whenever a condition is met. For this, 'while' is more convenient.
$a = 10;
while ($a > 0) {
    # do something...
    print "do something...\n";
    $a--;
}
```

Functions

When we write a program it is inevitable to write the same code again and again, with small changes. Functions allow to reuse the code, making it easier to read, to understand and to maintain. A function is defined with the command **sub**

followed by the function name. Functions can receive parameters, i.e. variables that are going to be used by the function internally to produce some output. All the built in functions mentioned above are just simple functions that are available to the user. Functions may not need any parameter at all, and be meant for things that we do several times in exactly the same way (for example, printing a help message).

```
# compute the melting temperature (tm) of the sequence ATCTTTCGGGCGCCAAACT
# this sequence has 4 A, 6 C, 5 T and 4 G:
print tm(4, 6, 5, 4);

# computes the melting temperature of a DNA strand by the Wallance method:
# TM = 2(A+T)+3(C+G)
# needs the number of A, C, T, G bases in the sequence.
sub tm {
    my ($a, $c, $t, $g) = @_; # get the parameters.
    if ($a < 0 or $c < 0 or $t < 0 or $g < 0) {
     help();
     return undef;
    }

    $tm = 2 * ($a + $t) + 3 * ($g + $c);
    return $tm;
}

sub help {
    print "Usage of the program: program.pl <file>"
    # a lot of more print statements...
}
```

Regular expressions

Regular expressions (regexp) is the real reason why Perl was used in the first place as a language for analysis of Genomic data. Regular expression is a way to query about the existence of specific motifs or patterns and to do something with them. The drawback is that they cryptic syntax look difficult to understand to the newcomer. The syntax is the string to query followed by an regexp operator (either '= ' or '! ' to indicate match or not match). After that is an expression with the motif to find, surrounded by the '/' characters. In the left side, the type of matching and in the right side some fine tuning parameters:

```
# define a dna string (usually this will come from a file).
$a = "ATTCGTCACTGCTAGGACTAGCGGCATTAGCTAGGCGCTGAGCGGGCGCT";
# lets see if it contains an stop codon:
# the 'm' means match and is the default regexp type,
# so we can skip using it.
if ($a =~ m/TGA/) {
    print "STOP codon at position @-\n";
} else {
    print "No STOP codon found in DNA string\n";
```

```
}

OUTPUT:
$ DNA has STOP codon at position 38
```

We can combine the knowledge accumulated to do this in a nicer manner. For example, defining a function that receives a character string as parameter and sees if it contains an stop codon.

```
#define function to query for STOP codons:
sub has_stop {
    my $dna = shift;
    if ($dna =~ /TGA/) {
        print "DNA has STOP codon at position @-\n";
    } else {
        print "No STOP codon found in DNA string\n";
    }
}
# then we just do:
$a = "ATTCGTCACTGCTAGGACTAGCGGCATTAGCTAGGCGCTGAGCGGGCGCT";
$b = "ATTCGTCACTGCTAGGACTAGCGGCATTAGCTAGGCGCTAAGCGGGCGCT";
has_stop($a);
has_stop($b);

OUTPUT:
$ DNA has STOP codon at position 38
$ No STOP codon found in DNA string
```

This regular expression will only find *one* stop codon. To get all possible matches we can use the 'g' modifier in the regular expression: /ATG/g

Documentation

There is an extensive documentation about perl, plus many beginners tutorials online. The best documentation is the internal reference:

```
# this will point to the other manual pages with specific information
$ man perl
# for example:
# perl for beginners.
$ man perlintro
# perl built in functions:
$ man perlfunc

# also, the perlfunc command is very useful to know how to use a specific command:
$ perldoc -f pop

OUTPUT:

       pop ARRAY
```

```
        pop     Pops and returns the last value of the array, shortening the
                array by one element.  Has an effect similar to

                    $ARRAY[$#ARRAY--]

                If there are no elements in the array, returns the undefined
                value (although this may happen at other times as well).  If
                ARRAY is omitted, pops the @ARGV array in the main program, and
                the @_ array in subroutines, just like "shift".

# There are also many web sites where you can find information, and help.
#  The most famous is probably:
http://www.perlmonks.org
```