VCU Bioinformatics and Bioengineering Summer Institute

Introduction to Personal Programming

Why personal programming?

Computer programming is the most general and most powerful tool in bioinformatics and bioengineering. It is the tool that makes tools. One may work productively in these fields without knowing how to program a computer, just as one can write great literature with a small, set vocabulary, but so much more is possible with the ability to make new tools in response to new situations.

Fortunately, computer programming is easy. Don't misunderstand – it is extraordinarily difficult to write a program that will be used on a variety of platforms by a variety of people in a way that is easy to use and idiot proof. Programmers get paid a lot of money to do this sort of thing, and they deserve every penny. However, it is quite a different matter to write a program that will be used only by yourself on your own computer, and if it encounters an unforeseen situation – no problem: you're there to deal with it. Anyone who can master a natural human language can accomplish the far easier task of gaining a working knowledge of a computer language. You can too.

Some of you know several computer languages. Others don't know any. None of us are masters of the computer language called Perl (though some have a passing acquaintance). By the end of this summer, all of you should be able to understand a well-written program in Perl and to make modifications in it to suit your needs. It would be nice if you could also write certain kinds of programs from scratch, but that may be beyond what we can accomplish in our limited time. We will use programs written in Perl as teaching aids, windows into the concepts and tools that underlie your research projects.

Computer languages

Knowing nothing of the languages of the Mowri or Basque, you may nonetheless feel certain that they possess words that describe hunger and any other condition at the core of human experience. So it is with computer languages. All must connect with the same machine, the digital computer. For our purposes we may imagine the computer as a string of boxes, each box labeled with a number, the address, by which it is known. You can put things into specific boxes, take things out of specific boxes, and manipulate the contents of boxes in a variety of ways. In the end, the contents of boxes are numbers, but we can tell the computer to interpret the numbers as letters or other symbols. Most powerfully, we can also tell the computer to interpret the numbers inside boxes as instructions how to manipulate the contents of ***other*** boxes.

There's a very limited number of things the computer knows intrinsically about manipulating numbers. The rest it must be taught, as combinations of the things it ***does*** know intrinsically. A low-level computer language, such as Assembler, helps us to combine these intrinsic instructions into useful programs. A high-level computer language, such as Perl, has many combinations of instructions built in and are given names to make them easier to remember. Some of these combinations, e.g. *sort* or *print*, may comprise hundreds of individual machine instructions that you would otherwise have to write yourself. That's a lot of work saved!

All languages have to do certain things, at least:

- Put information into specific locations in memory
- Take information from specific locations in memory
- Manipulate the data in certain ways (e.g. arithmetic)
- Communicate with peripheral devices (e.g. the screen, a disk, or a printer)
- Allow instructions to be executed in a logical fashion
- Allow blocks of code to be reused

Every language has its particular set of peculiarities and strengths. For Perl, one outstanding strength is its ability to manipulate strings (sequences of characters). It's not so good in number crunching.

How to learn Perl

To my mind, the most effective way to learn a language is to use it, exploring its limits as you do. Therefore, your first step is to gain access to Perl. Fortunately, it's free and readily available. To get it, click on the link on today's web page. You'll probably be much happier using an editor to input programs, as it will help you identify inevitable errors. You can get a free editor from a link on the web page, but many other editors are also available. You can also write programs using any text editor (e.g. *Notepad* supplied by Windows).

The rest of these notes presumes that you have already downloaded and installed Perl.

First programs

Let's write and run a few programs to see Perl in action, then we'll step back and note some important generalities.

<u>Program 1: Variations on *print*</u>

Copy and paste the following one-line program into a text editor, save it as print.pl,[1] and run the program.

```
print "I am Sam";
```

If you're running in the editor, it will offer you the first line:

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

Leave it there. It does no harm and is the traditional first line of all Perl programs (it enables programs to be run more easily in a Unix environment).

If you run within DOS (get there by clicking the `Start` button, then `Run`, then `Command`[2]), then the session should look something like this:

<u>Program 1</u>

```
C:\my-programs>perl print.pl
I am Sam
C:\my-programs>
```

[1] I suggest that you create a special directory for saving all your programs.
[2] Windows-XP may have other notions on how to get to a DOS prompt. Never mind Macs.

If you run within an Perl Editor, then the output will appear in the bottom window if you Run with output going to STDOUT (Standerd Output) only. Henceforth I'll give output presuming you're running with the editor.

This program is more complicated than it might seem – computer languages are designed to make complicated things seem easy. What you did was to set aside a number of boxes in memory into which you put "I" " " "a" "m" " " "S" "a" "m", then you told the computer to send the contents of those boxes outside the computer to some device. That device is not specified so presumed to be the screen. Communication with devices is very complex and we (thankfully!) are spared the behind-the-scenes give and take.

Copy, save, and run the following very similar two-line program:

<u>Variation 1</u>

```
my $sentence = "I am Sam";
print $sentence;
```

You should get exactly the same results as before when you run this program, but the program that gave the results is now more involved than the first program. Here is an expansion of what each line says to Perl:

```
my $sentence = "I am Sam";
```

Convert the symbols "I" " " "a" "m" " " "S" "a" "m" into numerical equivalents and store them into consecutive boxes in memory. Remember: (a) the address where you stored the first symbol, (b) how many symbols you stored, and (c) that I want you to interpret the stored numbers as symbols in the future. Keep all this information in a place in memory we'll refer to as the variable $*sentence*. By starting the word with "$" I'm warning you that $*sentence* is a variable that I made up and, by preceding *$sentence* with *my*, I'm warning you that this is the first time you've seen this variable.

```
print $sentence;
```

Look up what you know about $*sentence*, then take the contents of the variable (interpreted in the way you remember is appropriate) and send the contents to the screen.

The ability to store and manipulate variables gives you a lot of power. Let's play, altering the program in a variety of ways and running them:

<u>Variation 2</u>

```
my $sentence = "I am Sam";
print $sentence, $sentence;
```

Translation: Look up what you know about $*sentence*, then take the contents of the variable (interpreted in the way you remember is appropriate) and send the contents to the screen. Then look up the same variable and send its contents (again) to the screen.

If you were Perl, you may well have interpreted the last statement differently and come up with somewhat more intelligible results. You are ***not*** Perl, a fact that you need to keep at the forefront of your mind at all times.

<u>Variation 3</u>

```
my $sentence = "I am Sam";
print reverse (split (" ", $sentence));
```

Translation: Give the contents of the variable $*sentence* to the Perl function called *split*, which splits up a string of characters at the given character (in this case " "); take the result of that function and give it to another Perl function *reverse*. Take the result of ***that*** function and send it to the screen.

A function takes what is given to it (optionally within parentheses) and spits out a result, which can be used by the remainder of the statement.

Variation 4

```
my $sentence = "I am Sam";
if ($sentence eq "I am Sam") {
   print "I'm really not Sam";
} else {
   print $sentence;
}
```

> **Translation**: If the condition within the parentheses is true, then do the statement between the first set of brackets; otherwise do the statement between the set of brackets following *else*.
>
> **Translation (expanded):** If the contents of the variable $*sentence* happens to be "I am Sam" then print the string "I'm really not Sam", but if the contents of $*sentence* is not "I am Sam", then print the contents of that variable.

SQ1. What happens if you change the first statement of Variation 4 to read:

```
my $sentence = "I am Pam"
```

SQ2. What happens if you change the first statement of Variation 3 to read:

```
print reverse (split ("", $sentence)));
```

SQ3. What happens if you change the original program to read more like English sentences by capitalizing the first word:

```
My $sentence = "I am Sam";
Print $sentence;
```

Variation 3 may seem somewhat complicated. If you don't understand what it is trying to do, then unpack it by writing simpler programs from its parts. For example, try:

Variation 5

```
my $sentence = "I am Sam";
print split (" ", $sentence);
```

Play with the program until you form an idea of what *split* does. For example, change " " to "a" or to "".

IMPORTANT PRECEPT #1

There is no better way to understand how a function works than by playing with it (unless it is by playing with it and reading its documentation)

IMPORTANT PRECEPT #2

There is nothing you're likely to do that will cause any lasting harm whatsoever. So play!

Now take the displayed results from Variation 5 and stick it back into Variation 3:

Variation 6

```
my $sentence = "I am Sam";
print reverse (results of Variation 5);
```

The results of this program may surprise you. Perl has an overwhelming desire to figure out what you are trying to say, and sometimes its exuberance leads to unexpected results (and sometimes absolute atrocities). Perl's penchant for magic often irritates those used to cleaner, more predictable languages. We won't figure out right now how this result came about, but store it away in your mind somewhere to be considered later.

Data Aggregates

It is extraordinarily useful to be able to refer to aggregates of data. Here's an example why:

Suppose you wanted to teach Perl how to calculate the molecular weight of a protein. You already have a function, `MW_of`, that takes the name of an amino acid and returns its molecular weight. Your present task is to do this for all the amino acids of a protein and add them up. So let's get started:

Program 2

```
my total_MW = 0;                              # Sum is initialized to 0
total_MW = total_MW + MW_of (amino_acid_1);   # Add in first amino acid
total_MW = total_MW + MW_of (amino_acid_2);   # Add in second
total_MW = total_MW + MW_of (amino_acid_3);   # Add in third
…
```

… for a protein with several hundred amino acids, this can get really boring! Suppose instead you conceived of a ***list*** of amino acids, call them amino_acid$_1$, amino_acid$_2$, etc. Then you can teach Perl everything it needs to know (in principle) as follows:

```
For all n from 1 to the number of amino acids in the protein
   total_MW = total_MW + MW_of (amino_acid_n)
```

We might want to know what all those amino acids are doing something like this (in principle):

Print amino_acids

where this is intended to be short hand for:

```
For all n from 1 to the number of amino acids in the protein
   Print amino_acid_n
```

This list, amino_acids, is an example of a data aggregate. Highly useful! We'll talk about different ways of organizing and using such aggregates.

SQ4: Does Program 2 run properly? [No] Why not?

SQ5: The first line of Program 2 should not raise any errors, though it doesn't do anything interesting by itself. What happens if you remove the "#" from the line?

A few generalities

- Case distinctions: Perl ***cares*** about upper/lower case!
- Statement formatting: Perl statements generally end with ";". Otherwise, you can format them how you like – add spaces, spill over onto multiple lines – Perl doesn't care.
- Variables: Simple variables are recognized by Perl by the tag "$".
- Declaring variables: The first use of a variable should be preceded by "my"
- Functions/arguments: Functions take arguments within parentheses (though they are optional with Perl built-in functions
- Comments: Anything after "#" is ignored by Perl, hence can be used for documentation.