Introductory Perl

Boston University
Information Services & Technology

Course Coordinator: Timothy Kohl

Last Modified: September 2021

1

What is Perl?

- General purpose scripting language developed by Larry Wall in 1987.
- Has many of the characteristics of C, the various Unix shells, as well as text processing utilities like sed and awk

2

A very basic Perl script

Start up your favorite text editor and call this 'hello' and enter in the following two lines.

```
#!/usr/bin/perl
print "Hello world!\n";
```

After saving this file, exit the editor and do the following:

```
>chmod u+x hello
```

• Perl programs or 'scripts' are not compiled, but interpreted.

• In Unix, the u+x permission must be set to run the script.

• In Windows, perl scripts have a **.pl** as the file extension so you would call this script **hello.pl** and the chmod command would not be needed.

3

We run this script simply by typing:

```
>hello
```

If '.' (current directory) is not in your path, then you must invoke the program as follows:

```
>./hello
```

Assuming no mistakes you should get:

```
Hello world!
```

In Windows, one could also just double click on **hello.pl** (which won't work as expected) or issue the command

```
>hello.pl
```

from within a command shell.

So what's going on?

4

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

tells Unix that the script which follows is to be processed with the program /usr/bin/perl

- Common mechanism used by Unix scripting languages, utilities and shells
- It may be **/usr/bin/perl** or **/usr/local/bin/perl** depending on your system
- script is run after its syntax is checked first
- In Windows, the **#** isn't needed, but the script is still checked for correctness first.

```
print "Hello world!\n";   # produces output on screen
```

- **\n** is the newline character which puts the cursor at the start of next line
- A semi-colon is needed at the end of (almost) every line in a Perl script.
- Comments can be put on any line, and must start with a **#** character.

5

Let's modify our hello script top make it interactive.

```
#!/usr/bin/perl
print "What is your name? ";
$name=<STDIN>;
chomp($name);
print "Hello there $name.\n";
```

If we run this, we get

```
>hello  ( or ./hello if your shell is misconfigured)

What is your name?  Tim

Hello there Tim.
>
```

6

So what's happening here?

First we prompt the user for their name,

```
print "What is your name? ";
```

and then take input from the keyboard:

```
$name=<STDIN>;
```

This takes a line of **standard input** and assign it to the variable **`$name`**

(We'll discuss variable nomenclature in the next section.)

7

Since the line of standard input includes a **`\n`** at the end (when we hit **ENTER**) this gets removed or '**chomped**' by the command

```
chomp($name);
```

(This 'chomping' is something you should get used to seeing and using in any perl script which takes input.)

Finally, we say hello

```
print "Hello there $name.\n";
```

8

Perl Variables and Operators

In Perl, there are three basic data types:

- Scalars
- Arrays
- Associative arrays (also called hashes)

Unlike C or C++, for example, there is no need to specify names or types of variables at the beginning of a program.

9

Scalars

Scalars consist of integer or floating point numbers or text strings.

Scalar variables begin with a **$** followed by their name which can consist of either letters (upper or lower case) or **_** or numbers, with some exceptions which we'll discuss.

Ex:

```
$x = 3.5;
$name = "Tim";
$A_very_long_and_silly_looking_variable_name = 2;
```

10

All numbers in Perl are double precision floating point numbers (integers too!)

Ex:

```
$x=3;
$y=-5.5;
$z=6.0E23;    # exponential notation for 6 x 10^23
```

One can also work in Octal (base 8) or Hexadecimal (base 16) as well.

11

As for strings, the only two types are single and double quoted.

Ex:

```
$x = "Hello\n";     # Hello followed by newline
$y = 'Hello\n';     # literally Hello\n
```

Within double quotes, special characters like **\n**, are interpreted properly.

Ex:

\n	newline
\t	tab
\"	literally "
****	literally \

12

So if we have

```
print "Left\tMiddle\tRight\n";
```

we get

```
Left  Middle      Right
```

For single quoted strings, however, what's in quotes gets printed as is.

```
print
'Left\tMiddle\tRight\n';
```

yields

```
Left\tMiddle\tRight\n
```

13

Also, if you wish to embed variables inside strings and have the value substituted in properly, you must use double quotes.

Ex:

```
$name="Tim";
print "Hello $name\n";
```

will produce

```
Hello Tim
```

14

The typical operators for numerical values are present:

```
+,-,*,/
```

There is also an exponentiation operator,

```
2**3;      # 8 since 2^3 = 8
```

as well as a 'modulus' operator for taking remainders

```
5 % 2;     # 1, since 5 divided by 2 leaves remainder 1
```

15

Additionally, there are the autoincrement **++** and autodecrement **--** operators as in C.

```
$a=2;
++$a; # $a now equals 3
--$a; # $a now equals 2 again
```

Note, these also can be applied to character values as well.

Ex:

```
$x="A";
++$x;   # $x now equals B
```

16

For strings, there is a concatenation operator for combining two (or more) strings

It is given by **.** (a period)

Ex:

```
$x="Hello";

$y="There";

$z=$x.$y;   # $z is now "HelloThere"
```

Note, if you want a space in between, you can do this

```
$z=$x." ".$y;     # $z is now "Hello There"
```

17

We saw earlier the **chomp()** function removes a trailing newline character if one is present.

Ex:

```
$a="Hello There\n";
chomp($a);  # $a now equals "Hello There"


$b="Hi There";
chomp($b);  # $b still equals "Hi There"
```

There is also the function, **chop()**,which removes the last character in a string, whether it is a newline or not, but this is deprecated.

18

Making Comparisons

If we wish to compare two scalars then we **must** choose the appropriate comparison operator.

Comparison	Number	String
equal	`==`	`eq`
not equal	`!=`	`neq`
less than	`<`	`lt`
greater than	`>`	`gt`
less than or equal	`<=`	`le`
greater than or equal	`>=`	`ge`

Ex: `"023" < "23"` is false, but
`"023" lt "23"` is true

so be aware of the data you are working with when making comparisons.

We'll use these later, in the section on control structures.

19

Arrays

In Perl, arrays are lists of scalar values, either strings, or numbers.

Array variables, as a whole, are prefixed with the `@` sign followed by the array name which can consist of either letters, numbers, or `_` characters.

They can be created and modified in a variety of ways, the simplest is to just list the elements in the array.

Ex:

```
@X=(5,11,-6,12);

@People=("Tom","Dick","Harry");

@DaysOfWeek=("Mon","Tue","Wed","Thu","Fri","Sat","Sun");

@stuff=("Hi",3.1415,6,"Bye\n"); # mix and match!
```

20

Array elements are indexed starting from **0** and are accessed as follows:

Ex:

```
@X=(5,11,-6,12);
print "$X[2]\n";
```

yields

```
-6
```

That is, if the array is named **@X** then the i^{th} element is **$X[i]**

21

Adding elements to an array can be done in several ways.

Ex:

```
@People=("Tom","Dick");
@People=(@People,"Harry")
```

So now,

```
@People=("Tom","Dick","Harry");
```

Note, if one instead did

```
@People=("Harry", @People);
```

then

```
@People=("Harry","Tom","Dick");
```

22

One can also add an element by means of the array index.

Ex:

```
@X=(3,8,-2);
$X[3]=5;
```

So now

```
@X=(3,8,-2,5);
```

That is, we have added a **fourth** element to the array. (at array index **3**)

23

One can also copy arrays in a very simple manner.

```
@Names=("Tom","Dick","Harry");
@CopyOfNames=@Names;
```

So now,

```
@CopyOfNames=("Tom","Dick","Harry");
```

24

One can also take a 'slice' of an array.

Ex:

```
@Planets=("Mercury","Venus","Earth","Mars",
          "Jupiter","Saturn","Uranus",
          "Neptune","Pluto");

@InnerPlanets=@Planets[0..3];
```

So now, `@InnerPlanets=("Mercury","Venus","Earth","Mars");`

Also, one may include other ranges, e.g.

`@SomePlanets=@Planets[0..1,7..8];`

thus `@SomePlanets=("Mercury","Venus","Neptune","Pluto");`

(Keep in mind, element **0** is the first element in the array.)

25

Combining two arrays is also very easy:

Ex:

```
@People=("Tom","Dick","Harry");

@MorePeople=("John","Jim");

@Combined=(@People,@MorePeople);
```

So now,

```
@Combined=("Tom","Dick","Harry","John","Jim");
```

26

There is a built-in **`sort()`** function for sorting the elements of an array.

Ex:

```
@People=("Tom","Dick","Harry");
@People=sort(@People);
```

`@People` now equals **`("Dick","Harry","Tom");`**

• By default, the sorting is based on the ASCII (i.e. dictionary) value of the strings.

• There is also a way to sort arrays in numerical order.

27

Associative Arrays

An associative array is a structure consisting of pairs of scalars, a key and a value, such that each value is associated to a key.

Associative array variables, as a whole, are prefixed with `%` followed by the name which can consist of either letters or numbers or **_** characters.

28

As with regular arrays, individual elements are accessed with a **$**.

Typically, associative arrays are created and augmented on the fly, just by giving key and value pairs.

Ex:

```
$Grade{"Tom"}="A";

$Grade{"Dick"}="B";
```

note {} instead of **[]** for associative arrays

29

That is, **%Grade** is an associative array with (right now) two key and value pairs, which were given by the two assignment statements.

We could have also done this with the following statement:

```
%Grade = ("Tom" => "A","Dick" => "B");
```

30

A very useful function to apply to an associative array is **`keys()`**

As the name suggests, this returns all the keys in a given associative array, in ordinary array form.

Ex:

```
%Grade=("Tom"=>"A","Dick"=>"B","Harry"=>"C");

@Students=keys(%Grade);
```

`@Students` now equals **`("Tom","Dick","Harry")`**

31

undefined values

If a scalar value is referred to, but has not been assigned a value, Perl gives it the default value of **undef** which literally means undefined.

So, for example, if **`$a`** has not been defined, then

```
print "$a";
```

will produce no output, but will not generate an error either.

32

Likewise

```
@X=(3,7,9,2);
print "$X[10]";
```

will produce no output.

The point being that any array element not yet defined has the value **undef**.

And if

```
%Grade=("Tom" => "A","Dick"=>"B");
```

then **`$Grade{"Harry"}`** is **undef** since we have not given it a value.

33

Perl Control Structures

In Perl, there are a variety of familiar loop structures and conditionals. Some of the syntax is similar to C.

All of these are built around what's known as a statement block which is simply a sequence of statements, surrounded by **`{`** and **`}`**

Conditionals

Ex:

```
$entry=<STDIN>;
chomp($entry);
if($entry eq "Thank You"){
    print "You are Welcome!\n";
}
```

34

The conditional itself

```
$entry eq "Thank You"
```

is within parentheses and the value returned is either true or false.

If true, then the block within `{` and `}` is executed.

35

Before going further, here is a basic guide as to what is true or false in Perl:

- `"0"` and `""` (the empty string) and **undef** are false.
- all else is true*

What Perl does, is to first convert any scalar to string, then apply the above rules.

* Note, "0.0" evaluates to true since, as a string, "0.0" is not "0"

36

Why should we care that **"0"** and **undef** are false?

Ex:

```
if($go){
      print "Time to go!\n";
}
```

This print statement won't be invoked if the variable **$go** has not been set. e.g the value of this variable is based upon some input from the user.

This can be useful as we will see in subsequent tutorials.

37

In addition to **if,** one also has an **else** construction.

```
print "What\'s the password? ";
$entry=<STDIN>;
chomp($entry);
if($entry eq "FOOBAR"){
      print "Access Granted\n";
}else{
      print "Incorrect Password!\n";
}
```

If the conditional is true, **($entry eq "FOOBAR")** then the print statement inside the first set of **{** and **}** is executed,

otherwise the **"Incorrect Password!"** message gets printed.

38

Also, one can combine conditionals using

`&&`	logical and
`||`	logical or

```
if(($day eq "Monday") && ($time eq "7AM")){
      print "Time to get up!\n";
}
```

Logical **not** is given via `!`

```
if(!($password eq ”FOOBAR")){
      print "Access Denied\n";
}
```

39

loops

One has many of the familiar loop constructions.

Consider the following examples.

Ex:

```
$n=1;
$sum=0;
while($n<=10){
      $sum = $sum + $n;
      $n++;
}
print "The sum of the numbers from 1 to 10 is $sum\n";
```

40

A useful example of a while loop is one which takes multiple lines of standard input and process each line in some fashion. For example:

```
#!/usr/bin/perl
while($line=<STDIN>){
       chomp($line);
       print "[$line]\n";
}
```

This keeps repeating as long as there is input to be read in.

If we call this script 'bracket' then we can take input from a Unix pipe and surround each line with `[]` for example

```
> ls –al | bracket
```

41

There is also a **`for`** statement.

Ex:

```
$sum=0;
for($n=1;$n<=10;$n++){
       $sum = $sum + $n;
}
print "The sum is $sum\n";
```

The general syntax is:

```
for(initial_expression;test_expression;increment_expression){
       statement block
}
```

42

There is a nice generalization of **`for()`** used to loop over the elements of an array.

Ex:

```
@People=("Tom","Dick”,"Harry");
foreach $person (@People){
      print "$person\n";
}
```

yields (as you might expect)

```
Tom
Dick
Harry
```

Note, this works regardless of the size of the array.

Also, one does not need to keep track of the array index.

43

One can use the **`foreach()`** function together with the **`keys()`** function to examine the contents of an associative array.

Ex:

```
%Grade=("Tom"=>"A",
        "Dick"=>"B",
        "Harry"=>"C"
        );
@People=keys(%Grade);
foreach $person (@People){
      print "$person received a $Grade{$person} \n";
}
```

i.e.

`keys(%Grade)` is the array **`("Tom","Dick","Harry")`**
extracted from the associative array **`%Grade`**;

44

Regular Expressions (a.k.a. 'regexps')

- one of the most powerful features of Perl
- process text using what are known as regular expressions
- regular expressions are a means of doing pattern matching on strings.

The general syntax for a pattern is

```
/pattern/
```

where **pattern** is the text pattern we are trying to describe.

45

The general syntax to see if a string matches a certain pattern is:

```
$x =~ /pattern/
```

pattern matching operator

For example, to see if **$x** contains the word **hello** we might write:

```
if($x =~ /hello/){
      #do something
}
```

i.e. If the pattern matches, then the conditional has value true.

46

By default, pattern matching is case sensitive, so the following strings would match:

```
$x="hello there"
$x="I just called to say hello"
$x="Othello by William Shakespeare"
```

yes! this is a match

but something like

```
$x="Hello to you!"
```

would not (the capital H makes a difference)

47

Note, to ignore the distinction between upper and lower case one can do the following:

```
if($x =~ /hello/i){
      #do something
}
```

The **i** after the **/** means *ignore* case.

48

One way to make the pattern more flexible is to use alternation.

Ex:

```
$x =~ /th(is|at)/
```

is true if **$x** matches either

this or that

The **(|)** allows us to choose one or more possibilities.
For example, we could do:

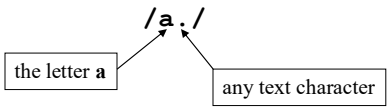
```
$x =~ /th(is|at|en)/
```

to look for '**this**' '**that**' or '**then**'

49

Regular expressions allow us to be quite general about the patterns we look for.

Ex: Match all strings which have the letter **a** followed by *at least one* text character. (i.e. something other than \n)



So these would match

```
"apple"
"this and that"
```

but not

```
"a"
"a\n"
```

} no text characters after the **a**

50

For more variability, we can also match on multiples of characters.

multipliers.

`*`	**zero or more** occurrences of the *previous* entity
`+`	**at least one** of the *previous* entity
`?`	**0** or **1** instances of the *previous* entity
`{n}`	**n** instances of the *previous* entity
`{m,n}`	between **m** and **n** instances of the *previous* entity

Ex:

```
/be*t/
```

would match **"bet"** and **"beet"** or even **"bt"**

51

If we change ***** to **+** then

```
/be+t/
```

matches **"bet"** and **"beet"** but not **"bt"**
since the **e+** means at least one instance of the letter **e**

If we change this to say

```
/b.+t/
```

then this would match **"boot", "belt", "bet", "bat", "b t"** etc.
since **.+** means match one or more of any character

Again, the pattern just has to exist somewhere in the string in order to match.

52

classes

Say we wish to see if there is a vowel somewhere in a given string.

We could do this as follows.

```
if($x=~/[aeiou]/){
      print "Found a vowel!\n";
}
```

The `[ ]` indicates a specific **class** of characters which we want to match.

In this case, one of the five vowels.

53

If we wish to match any lower case letter, then we can use

```
/[a-z]/     # i.e. all the letter from a to z
```

to include upper case letters we use

```
/[a-zA-Z]/ # all letter from a to z and A to Z
```

Likewise, we can also match digits.

```
/[0-9]/
```

54

There are also a number of pre-defined classes one can use which have abbreviations.

description	class	abbreviation
digits	**[0-9]**	**\d**
words	**[a-zA-Z_]**	**\w**
space	**[\n\r\t\f]**	**\s**

literally a space

These classes are useful particularly when constructing complicated patterns.

55

To negate a class, use **[^]**

ex. **[^x]** everything <u>but</u> the letter x

\W	non-word characters	**[^a-zA-Z_]**
\S	non-space characters	**[^ \r\n\t\f]**
\D	non-digit characters	**[^0-9]**

One can combine pre-defined classes to make larger classes.

Ex:

```
$x=~/[\w\d]/
```

matches words **and** digits

56

anchoring patterns

Suppose we wish to specify where in a string a given pattern is matched.

For example, say we wish to see if a given string starts with a capital letter.

```
$sentence =~ /^[A-Z]/
```

The **^** is to test if the pattern is matched at the beginning of the string.

Note, due to an unfortunate reuse of symbols, this is *not* the same as class negation seen earlier.

i.e. **`/[^A-Z]/`** means match everything *but* A-Z !!

57

Likewise, we could test if a certain pattern is matched at the end of a string.

i.e. Say we wish to check if a certain string ends with the letter **e**

We could use the following:

```
$x =~ /e$/;
```

So this would match if

```
$x = "the"
```

but not if

```
$x = "the rest"
```

58

One can also anchor a pattern at a **word boundary** using the directive **\b**

Such a boundary occurs at the beginning of a string (or end) or at a transition from a **\w** to a **\W** or vice versa.

Ex:

```
$x =~ /the\b/;
```

matches if

$x="the"** or **$x="the end"

but not

$x="then"

Matching somewhere that is **not** a word boundary can be done with **\B**

59

References for further information on Perl

Books

- Learning Perl by Randal L. Schwartz & Tom Christiansen (O'Reilly)
- Programming Perl by Larry Wall, Tom Christiansen and Jon Orwant (O' Reilly)
- Perl in a Nutshell by Ellen Siever, Stephen Spainhour, and Nathan Patwardhan (O' Reilly)

Web

http://www.perl.com

http://math.bu.edu/people/tkohl/perl ← My Perl Page!

60

Introductory Perl

Boston University
Information Services & Technology

Course Coordinator: Timothy Kohl

61