GSE 760–S01

Advanced Methods in Geospatial Modeling:

Computation for Remote Sensing Analysis and Product Generation

February 19, 2020

Class Schedule

Date	Lecture (Friday)	Date	Exercises (Friday)
Jan 15	**Lecture 1**: Course overview and introduction to remote sensing processing on Linux system	Jan 15	Lab assignment 1- Linux system setup
Jan 22	**Lecture 2**: Getting start with Linux system	Jan 25	Lab assignment 2 - Command line syntax
Jan 29	**Lecture 3:** Linux files and file utilities	Jan 29	Lab assignment 3 – File utilities
Feb 5	**Lecture 4:** File system and processes	Feb 5	Lab assignment 4 – File system and processes
Feb 12	**Lecture 5:** Shell scripting	Feb 12	Lab assignment 5- Shell scripting
Feb 19	**Lecture 6:** Perl scripting (1)	Feb 19	Lab assignment 6- Perl scripting
Feb 26	**Lecture 7:** Perl scripting (2)	Feb 26	Lab assignment 7- Perl scripting
March 5	**Lecture 8:** Python scripting	March 5	Midterm Exam
March 12	**Spring Break**		**Spring Break**
March 19	**Lecture 9:** Satellite data and file format	March 19	Lab assignment 8- Python scripting
March 26	**Lecture 10:** Satellite data processing	March 26	Lab assignment 9- Satellite data processing
April 2	**No class/Easter Recess**	April 2	**No class/Easter Recess**
April 9	**Lecture 11:** Operational product generation	April 9	Lab assignment 10- Programing for product generation
April 16	**Lecture 12:** Software and product documentation	April 16	Project work overview
April 23	Work on projects	April 23	Work on projects
April 30	Work on projects	April 30	Work on projects
May 7	**Lecture 13:** Final presentation		

Final Exam: We will schedule final presentations during the final exam period.

Note: Recommended to readings to accompany each chapter will be assigned on the class D2L site.

Perl Scripting

- Perl is known as the Swiss Army chainsaw of scripting languages: powerful and adaptable.
- Perl was first developed by Larry Wall, a linguist working as a system administrator for NASA in the late 1980s, as a way to make report processing easier.
- Larry Wall combined useful features of several existing languages with a syntax designed to sound as much as possible like English.
- Perl is designed to be flexible, intuitive, easy, and fast; this makes it somewhat "messy".

http://www.tutorialspoint.com/perl/
https://www.cs.tut.fi/~jkorpela/perl/
http://www.tutorialspoint.com/perl/perl_tutorial.pdf

Perl Scripting

- Perl: **P**ractical **E**xtraction and **R**eporting **L**anguage.

- **Perl** is a family of high-level, general-purpose, interpreted, dynamic programming languages.

- Perl is a language that's designed for text processing.
- Perl is an **interpreted** programming language

Writing a script

- Perl scripts/programs are plain-text documents (generally ASCII), usually with the extension **.pl**, though this isn't strictly necessary.

A Perl script is generally started with the "**shebang**" line, which looks something like one of these:

#!/usr/bin/perl
#!/usr/local/bin/perl
#!/usr/local/bin/perl5
#!/usr/bin/perl -w
etc.

Shell Scripting
#!/bin/sh

The purpose of this line is to tell the server which version of Perl to use, by pointing to the location in the server directory of the Perl executable.

Writing a script

The first Perl program. Open your text-editor (VI, gedit, or emacs) with name **perlscript1.pl**:

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

Then save it.

To tell the interpreter (linux Shell) to execute the script at the **command line**:

$ ./perscript1.pl

or

Using absolute directory

or

$ perl perscript1.pl

Basic Perl Syntax

- Perl thinks of "lines" differently from humans. Perl takes semicolon as lines.
- All lines in a code in Perl must end with one of two things: a **semicolon** or **curly braces**:

print "Hello.";
sub {print "Hello."}

- A line in a code, called a **statement**, is basically a single instruction to the computer; it says, "Do **x**." It can extend over more than one line of the page.
- Anything in **{ }** is called a **block**. A block can contain several statements.

Any line that begins with the character **#** is treated by Perl as a **comment** and is ignored. It can be put any comments.

```
# This next line sorts alphabetically
@names = sort (@names);   # () optional
```

Basic Perl syntax

Perl makes use of three basic kinds of quote marks: single (' ') and double (“ ”).

- **Double quotes** allow **variable interpretation**.

```
$name = "Alejna";
```

```
$name = "Alejna";
Print “Hello, $name!\n”;
OUTPUT ?
```

- **Single quotes** will prevent interpretation.

```
$name = "Alejna";
print 'Hello, $name!\n';
```

will print this:

Hello, $name!\n

Basic Perl syntax

Backslash interpretation

- Perl has a number of special characters that have a particular meaning within double quotes. Here are two important ones:

\n means "newline"

a.k.a. "carriage return"

\t means "tab"

- What we can type and what we'll get:

```
print "Name:\tBecky\nEyes:\thazel\n";
```

?

Name: Becky
Eyes: hazel

Variables

Scalars

Names of scalar variables begin with **$**.

Scalars are integers, floating-point numbers and strings.

```
$x=1;
$x=1.11;
$x="Hello";
```

```
($n, $n5, $n6) = (15, 16, 60);
```

Reserved words

- In Python:

and, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while

- In Shell script

System Variable	Meaning
BASH=/bin/bash	Our shell name
BASH_VERSION=1.14.7(1)	Our shell version name
COLUMNS=80	No. of columns for our screen
HOME=/home/vivek	Our home directory
LINES=25	No. of columns for our screen
LOGNAME=students	Our logging name
OSTYPE=Linux	Our o/s type : -)
PATH=/usr/bin:/sbin:/bin:/usr/sbin	Our path settings
PS1=[\u@\h \W]\$	Our prompt settings
PWD=/home/students/Common	Our current working directory
SHELL=/bin/bash	Our shell name
USERNAME=vivek	User name who is currently login to this PC

Reserved words

- In Shell script
  - $0 is the *basename* of the program as it was called.
  - $1 .. $9 are the first 9 additional parameters the script was called with.
  - $@ is all parameters from $1 .. .
  - $*, is similar to $@, but does not preserve any whitespace, and quoting, so "File with spaces" becomes "File" "with" "spaces".
  - $# is the number of parameters the script was called with.

Global Special Scalar Variables

$_ $ARG	The default input and pattern-searching space.
$. $NR	The current input line number of the last filehandle that was read. An explicit close on the filehandle resets the line number.
$/ $RS	The input record separator; newline by default. If set to the null string, it treats blank lines as delimiters.
$, $OFS	The output field separator for the print operator.
$\ $ORS	The output record separator for the print operator.
$" $LIST_SEPARATOR	Like "$," except that it applies to list values interpolated into a double-quoted string (or similar interpreted string). Default is a space.
$; $SUBSCRIPT_SEPARATOR	The subscript separator for multidimensional array emulation. Default is "\034".
$^L $FORMAT_FORMFEED	What a format outputs to perform a formfeed. Default is "\f".
$: $FORMAT_LINE_BREAK_CHARACTERS	The current set of characters after which a string may be broken to fill continuation fields (starting with ^) in a format. Default is "\n"".
$^A $ACCUMULATOR	The current value of the write accumulator for format lines.

Global Special Scalar Variables

$#	Contains the output format for printed numbers (deprecated).
$OFMT	
$?	The status returned by the last pipe close, backtick (``) command, or system operator.
$CHILD_ERROR	
$!	If used in a numeric context, yields the current value of the errno variable, identifying the last system call error. If used in a string context, yields the corresponding system error string.
$OS_ERROR or $ERRNO	
$@	The Perl syntax error message from the last eval command.
$EVAL_ERROR	
$$	The pid of the Perl process running this script.
$PROCESS_ID or $PID	
$<	The real user ID (uid) of this process.
$REAL_USER_ID or $UID	
$>	The effective user ID of this process.
$EFFECTIVE_USER_ID or $EUID	
$(	The real group ID (gid) of this process.
$REAL_GROUP_ID or $GID	
$)	The effective gid of this process.
$EFFECTIVE_GROUP_ID or $EGID	

Global Special Scalar Variables

$0 $PROGRAM_NAME	Contains the name of the file containing the Perl script being executed.
$[	The index of the first element in an array and of the first character in a substring. Default is 0.
$] $PERL_VERSION	Returns the version plus patchlevel divided by 1000.
$^D $DEBUGGING	The current value of the debugging flags.
$^E $EXTENDED_OS_ERROR	Extended error message on some platforms.
$^F $SYSTEM_FD_MAX	The maximum system file descriptor, ordinarily 2.
$^H	Contains internal compiler hints enabled by certain pragmatic modules.
$^I $INPLACE_EDIT	The current value of the inplace-edit extension. Use undef to disable inplace editing.
$^M	The contents of $M can be used as an emergency memory pool in case Perl dies with an out-of-memory error. Use of $M requires a special compilation of Perl. See the INSTALL document for more information.
$^O $OSNAME	Contains the name of the operating system that the current Perl binary was compiled for.
$^P $PERLDB	The internal flag that the debugger clears so that it doesn't debug itself.

Global Special Scalar Variables

$^T	The time at which the script began running, in seconds since the epoch.
$BASETIME	
$^W	The current value of the warning switch, either true or false.
$WARNING	
$^X	The name that the Perl binary itself was executed as.
$EXECUTABLE_NAME	
$ARGV	Contains the name of the current file when reading from .

Scope of Variables

Variables in Perl are global by default.

To make a variable local to subroutine or block, the my construct is used.

For instance,

my $x;

my $y;

```
#!/usr/bin/perl
my $name = "Bar";
print "$name\n"; # Bar
{
print "$name\n"; # Bar
$name = "Other";
print "$name\n"; # Other
}
print "$name\n"; # ???
```

```
#!/usr/bin/perl
$name = "Bar";
print "$name\n"; # Bar
{
print "$name\n"; # Bar
my $name = "Other";
print "$name\n"; # Other
}
print "$name\n"; # ???
```

Variables

Array names begin with **@**. Indices are integers beginning at 0. Array elements can only be scalars, and not for instance other arrays.

@x=(1,2,3,4); -------- @x=((1,2),3,4)

Since array elements are scalars, their names begin with **$**.

@x=(1,2,3,4),

$x[0]=1;

$x[1]=2;

$x[2]=3;

$x[3]=4;

String array:

@str=(“How”, “are”, “you”, “?”);

$str[0]=“How”;

Size of Array

Perl array size: the number of elements of a Perl array.

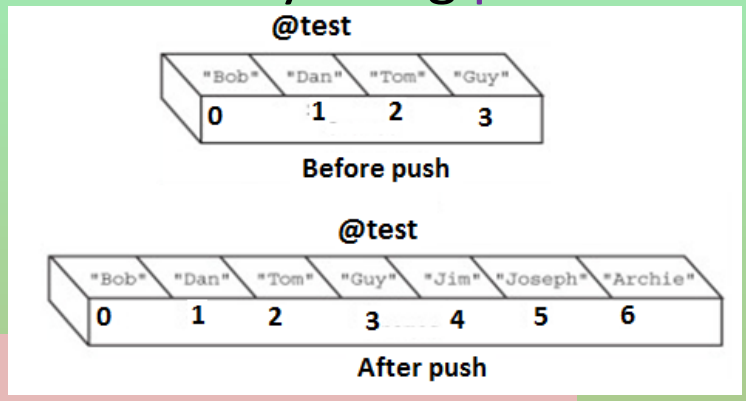
```
$arraySize = @array;
$arraySize = scalar(@array);
$arraySize = $#array + 1;
```

```
@words = qw(Perl array size);
$size = length(@words);
print "$size\n";
```

scalar context in **@words** means 3
length(3) is equal to 1

push, shift, pop, slicing

An array can be treated as a **queue data** structure, using the Perl operations **push** and **shift** (usage of the latter is especially common in the Perl idiom), or treated it as a **stack** by using push and pop.



Syntax:
push ARRAY, LIST

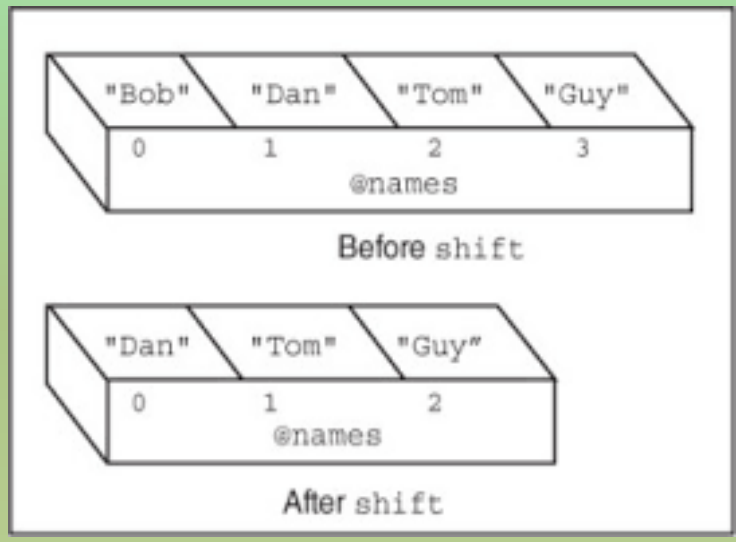
```
#!/usr/bin/perl -w
@array = ( 1, 2 );
print "Before pushing elements @array \n";
push(@array, (3, 4, 5));
 print "After pushing elements @array \n";
```

```
Before pushing elements 1 2
 After pushing elements 1 2 3 4 5
```

push, shift, pop, slicing

"**Shift**" function returns the first value in an array, deleting it and shifting the elements of the array list to the left by one. shift is essentially identical to pop, except values are taken from the start of the array instead of the end.

Syntax:
shift ([ARRAY])
shift



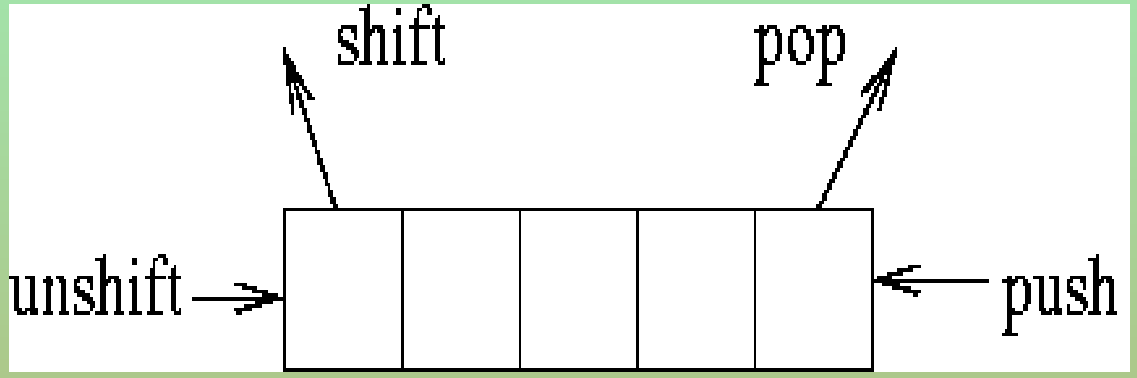

```
#!/usr/bin/perl
@array = (1..5);
while ($element = shift(@array))
{
print("$element - ");
}
print("The End\n");
```

1 – 2 – 3 – 4 – 5 – The End

push, shift, pop, slicing

pop function returns the last element of ARRAY, removing the value from the array. Note that ARRAY must explicitly be an array, not a list.

Syntax:
pop ([ARRAY])
pop




```
#!/usr/bin/perl -w
@a = (1, 2, 3, 4);
print("pop() ", pop(@a), " leaves ",@a, "\n");
```

```
pop()  4  leaves  123
```

push, shift, pop, slicing

Extract a "**slice**" from an array - that is to select more than one item from an array in order to produce another array. Subsets of arrays may be accessed—“sliced”—via commas and a range operator

```
@z = (5,12,13,125);

@w = @z[1..3];          # @w will be (12,13,125)
@q = @z[0..1];          # @q will be (5,12)
@y = @z[0,2];           # @y will be (5,13)
@w[0,2] = @w[2,0];      # swaps elements 0 and 2
@g = @z[0,2..3];        # can mix "," and "..."
print "@g\n"            # prints 0 13 125
```

Sort Array

The **sort()** function sorts each element of an array according to ASCII Numeric standards. This function has following syntax:

```
sort [ SUBROUTINE ] LIST
```

```
#!/usr/bin/perl
# define an array
@foods = qw(pizza steak chicken burgers);
print "Before: @foods\n";
@foods = sort(@foods); # sort this array
print "After: @foods\n";
```

```
Before: pizza steak chicken burgers
After: burgers chicken pizza steak
```

Merging Arrays

```
#!/usr/bin/perl
@numbers = (1,3,(4,5,6));
print "numbers = @numbers\n";
```

```
#!/usr/bin/perl
@odd = (1,3,5);
@even = (2, 4, 6);
@numbers = (@odd, @even);
print "numbers = @numbers\n";
```

numbers = 1 3 5 2 4 6

Selecting Elements from Lists

```
#!/usr/bin/perl
$var = (5,4,3,2,1)[4];
print "value of var = $var\n“;
```

```
#!/usr/bin/perl
@list = (5,4,3,2,1)[1..3];
print "Value of list = @list\n";
```

Split

"**Split**" splits up a string and places it into an array. The function uses a regular expression and as usual works on the **$_** variable unless otherwise specified.

$_	The default input and pattern-searching space.

```
$info = "Caine:Michael:Actor:14, Leafy Drive";
@personal = split(/:/, $info);
```

```
my $line = "file1.gz  file1.gz  file3.gz";
my @abc = split(' ', $line);
print "@abc\n";
```

```
#!/usr/bin/perl
print "content-type: text/html \n\n";
$astring = "Rain-Drops-On-Roses-And-Whiskers-On-Kittens";
$namelist = "Larry,David,Roger,Ken,Michael,Tom";
@array = split('-',$astring);
@names = split(',',$namelist);
print @array."\n";
print "@names";
```

join

```
#!/usr/bin/perl
print "content-type: text/html \n\n";
@array =("David","Larry","Roger","Ken","Michael","Tom");
@array2 = qw(Pizza Steak Chicken Burgers);

$firststring = join(",",@array);
$secondstring = join(" ",@array2);
print "$firststring \n";
print "$secondstring";
```

```
David,Larry,Roger,Ken,Michael,Tom
Pizza Steak Chicken Burgers
```

Global Array Special Variables

@ARGV	The array containing the command line arguments intended for the script
@INC	The Array containing the list of places to look for Perl scripts to be evaluated by the do, require, or use constructs
@F	The array into which the input lines are split when the –a command line switch is given

@ARGV

```
#!/usr/bin/perl
my ($name, $number) = @ARGV;
 if (not defined $name) {
 die "Need name\n";
 }

if (defined  $number) {
print "Save '$name' and '$number'\n";
# save name/number in database
exit;
}
 print "Fetch '$name'\n";
# look up the name in the database and print it out
```

```
$ perl Test1.pl Foo 123
Save 'Foo' and '123'
$ perl Test1.pl Bar 456
Save 'Bar' and '456'
```

Examples Variables and Arrays

```
1 $x[0] = 15;
2 $x[1] = 16;
3 $y = shift @x
4 print $y, "\n";
5 print $x[0], "\n";
6 push(@x,9);
7 print  scalar(@x), "\n";
8 print @x, "\n";
9 $k = @x;
10 print $k, "\n";
11 @x = ();
```

```
 #
###
#  the element shifted out
# prints 15
# prints 16
 # sets $x[1] to 9
# prints 2
 # prints 169 (no space)
 ##
 # prints 2
 # @x will now be empty
```

Examples Variables and Arrays

```
12 print scalar(@x), "\n";
13 @rt = (’abc’,15,20,95);
14 delete $rt[2];
15 print "scalar(@rt) \n";
16 print @rt, "\n";
17 print "@rt\n";
18 print "$rt[-1]\n";
19 $m = @rt;
20 print $m, "\n";
21 ($m) = @rt;
22 print $m, "\n";
```

```
12# prints 0
13 #
14 # $rt[2] now = undef
15 # prints 3
16 #prints abc1595
17 #prints abc 15 95,due to quotes
18 # prints 95
19 #
20  # prints 3
21  #  a 1-element array
22 # prints abc
```

Hashes

A hash is a set of **key/value** pairs. Hash variables are preceded by a percent (%) sign. A single element of a hash use the hash variable name preceded by a "$" sign and followed by the "key" associated with the value in curly brackets.

```
#!/usr/bin/perl
%data = ('John Paul', 45, 'Lisa', 30, 'Kumar', 40);
print " \$data{'John Paul'} = $data{'John Paul'} \ n";
print " \$data{'Lisa'} = $data{'Lisa'} \n";
print " \$data{'Kumar'} = $data{'Kumar'} \n";

$data{'John Paul'} = 45;
$data{'Lisa'} = 30;
$data{'Kumar'} = 40;
```

```
#!/usr/bin/perl          ##Use => as an alias
%data = ('John Paul' => 45, 'Lisa' => 30, 'Kumar' => 40);
print "$data{'John Paul'}\n";
print "$data{'Lisa'}\n";
print "$data{'Kumar'}\n";
```

Extracting Slices

Extracting slices of a hash just as extracting slices from an array.

Use @ prefix for the variable to store returned value because they will be a list of values

```
#!/uer/bin/perl
%data = (-JohnPaul => 45, -Lisa => 30, -Kumar => 40);
@array = @data{-JohnPaul, -Lisa};
print "Array : @array\n";
```

```
Array : 45 30
```

Extracting Keys and Values

Get a list of all of the keys from a hash by using **keys** function which has the following syntax:

keys %HASH

```
#!/usr/bin/perl
%data = ('John Paul' => 45, 'Lisa' => 30, 'Kumar' => 40);
@names = keys %data;
print "$names[0]\n";
print "$names[1]\n";
print "$names[2]\n";
```

```
Lisa
John
Paul
Kumar
```

Extracting Keys and Values

use **values** function to get a list of all the values. This function has following syntax:

values %HASH

```
#!/usr/bin/perl
%data = ('John Paul' => 45, 'Lisa' => 30, 'Kumar' => 40);
@ages = values %data;
print "$ages[0]\n";
print "$ages[1]\n";
print "$ages[2]\n";
```

```
45
30
40
```

Operators

Perl Arithmetic Operators:
$a=10;
$b=20;

Operator	Example
+ addition	$a + $b will give 30
- subtraction	$a - $b will give -10
*	$a * $b will give 200
/	$b / $a will give 2
% Modulus	$b % $a will give 0
**	$a**$b will give 10 to the power 20

++ $a++
-- $b--

Perl Arithmetic Operators:
$a=5;
$b=2;

Combined Assignment Operators

$a+=3; ========= $a=$a+3;

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	$c = $a + $b will assigned value of $a + $b into $c
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	$c += $a is equivalent to $c = $c + $a
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	$c -= $a is equivalent to $c = $c - $a
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	$c *= $a is equivalent to $c = $c * $a
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	$c /= $a is equivalent to $c = $c / $a
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	$c %= $a is equivalent to $c = $c % a
**=	Exponent AND assignment operator, Performs exponential (power) calculation on operators and assign value to the left operand	$c **= $a is equivalent to $c = $c ** $a

Combined Assignment Operators

Perl Bitwise Operators

$a = 60; and $b = 13;

$a = 0011 1100
$b = 0000 1101

$a&$b = 0000 1100

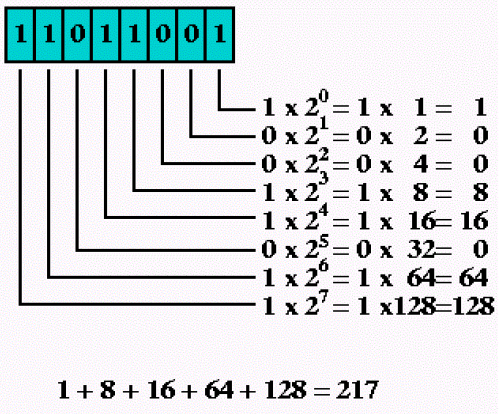
$a|$b = 0011 1101

$a^$b = 0011 0001

~$a = 1100 0011

$a<<2 = 1111 0000

$a>>2= 0000 1111



Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	($a & $b) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	($a | $b) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	($a ^ $b) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the efect of 'flipping' bits.	(~$a) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	$a << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	$a >> 2 will give 15 which is 0000 1111

Operators

Perl Logical Operators

Operator	Example
and	($a and $b) is false.
&&	($a && $b) is false.
or	($a or $b) is true.
||	($a || $b) is true.
not	not($a and $b) is true.

Operators

Perl Equality Operators

Number	example	Strings	example
==	($a==$b)	eq	($a eq $b)
!=	($a!=$b)	ne	($a! ne $b)
<	($a<$b)	lt	($a lt $b)
<=	($a<=$b)	le	($a le $b)
>	($a>$b)	gt	($a gt $b)
>=	($a>=$b)	ge	($a ge $b)
		cmp	($a cmp $b)

Conditions

Statement	Description
if statement	An **if statement** consists of a boolean expression followed by one or more statements.
if...else statement	An **if statement** can be followed by an optional **else statement**.
if...elsif...else statement	An **if statement** can be followed by an optional **elsif statement** and then by an optional **else statement**.
unless statement	An **unless statement** consists of a boolean expression followed by one or more statements.
unless...else statement	An **unless statement** can be followed by an optional **else statement**.
unless...elsif..else statement	An **unless statement** can be followed by an optional **elsif statement** and then by an optional **else statement**.
switch statement	With latest versions of Perl, you can make use of **switch** statment which allows a simple way of comparing a variable value against various conditions.

Conditions

```
#!/usr/bin/perl
$x=7;
$y=10;
if ($x == $y) {$z = 2;}
elsif ($x > 9) {$z = 3;}
else {$z = 4;}

$str1=“num”;
$str2=“nun”;
If($str1 eq $str2){
Print “$str1\n”;
}else{
Print “$str2\n”;
}
```

Conditions

```
unless(condition_1){
    unless code block
}elsif(condition_2){
    elsif code block
}else{
   else code block
}
```

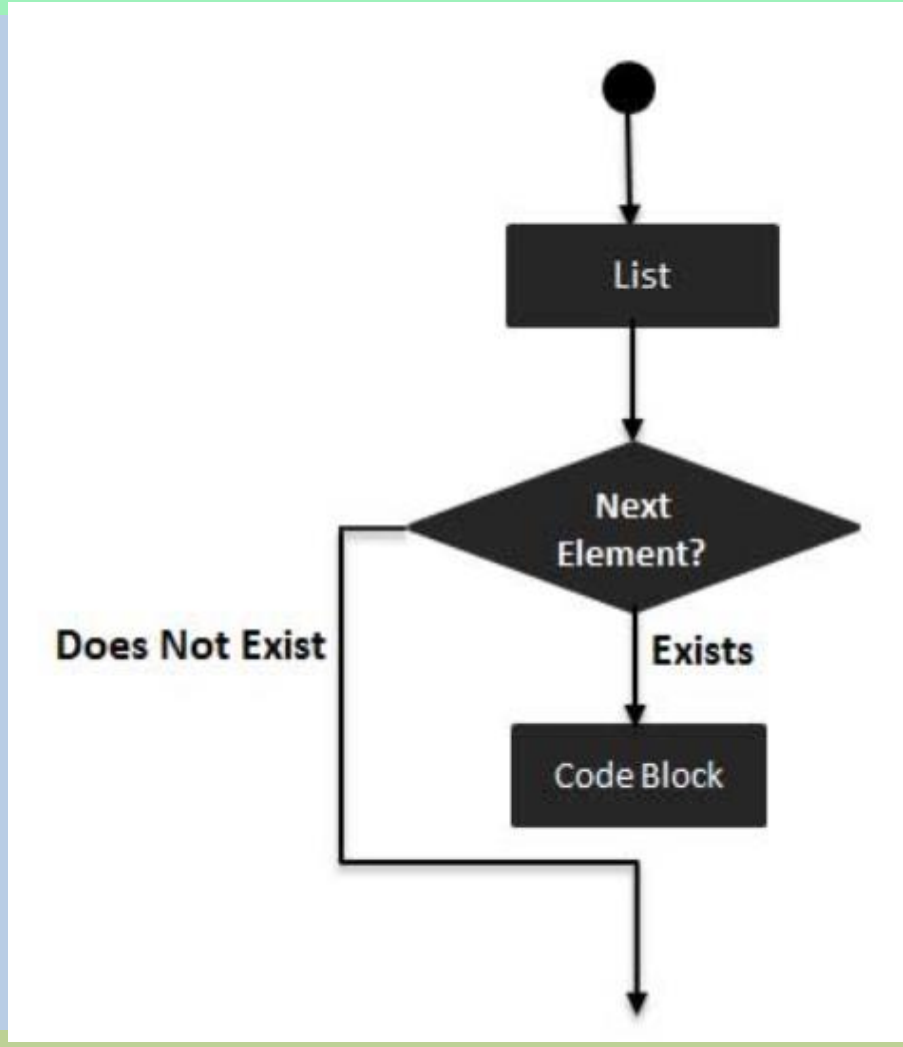
```
#!/usr/bin/perl
my $a = 1;
unless($a > 0){
   print("a is less than 0\n");
}elsif($a == 0){
   print("a is 0\n");
}else{
   print("a is greater
than0\n");
}
```

Perl Loop

Loop Type	Description
while loop	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
until loop	Repeats a statement or group of statements until a given condition becomes true. It tests the condition before executing the loop body.
for loop	Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
foreach loop	The foreach loop iterates over a normal list value and sets the variable VAR to be each element of the list in turn.
do...while loop	Like a while statement, except that it tests the condition at the end of the loop body
nested loops	You can use one or more loop inside any another while, for or do..while loop.

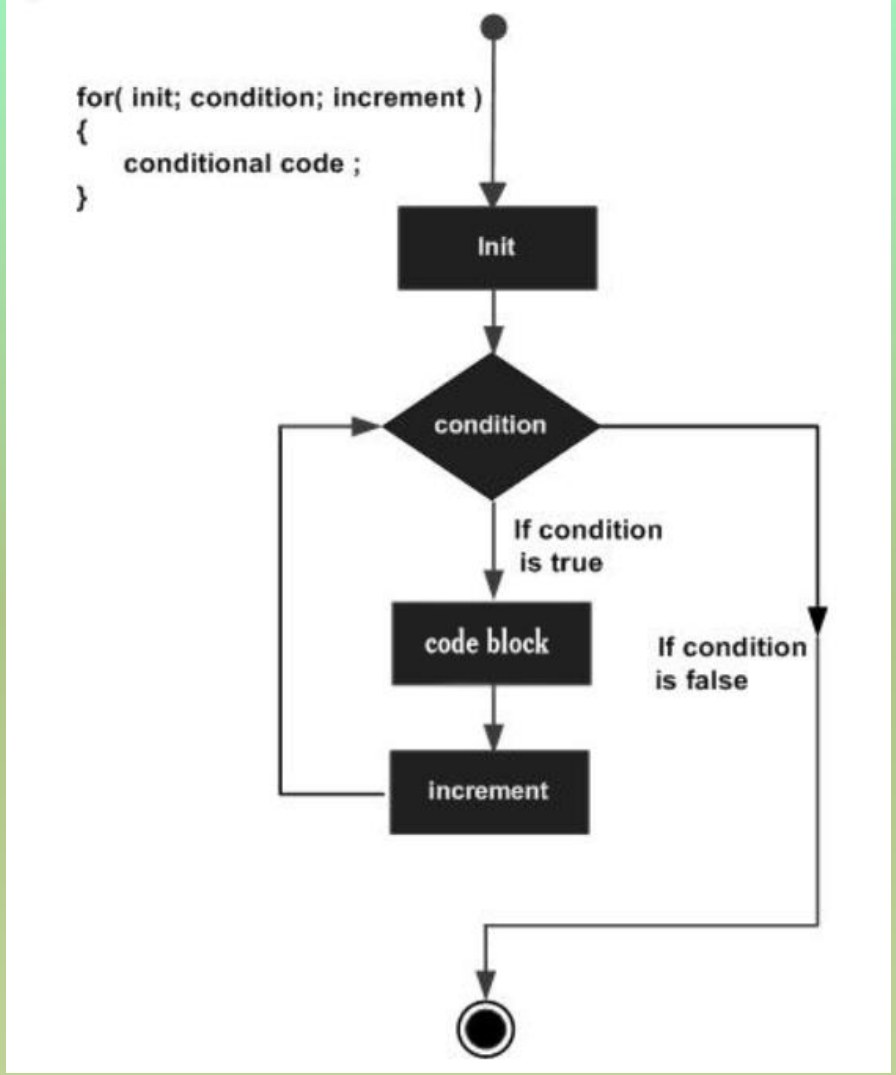
Perl Loop

```
# #!/usr/bin/perl
@recs = (1, 2, 3, 4, 5, 6 , 7);
$sum = 0;
# foreach loop

foreach $rec (@recs) {
chomp($rec)
$sum = $sum + $rec;
}
# print the sum
print "sum = $sum\n";
```



Perl Loop

```
# #!/usr/bin/perl
@recs = (1, 2, 3, 4, 5, 6 , 7);
$sum = 0;
# For  loop,

 for($i=0;$i<7;$i++) {
$sum = $sum + $recs[$i];
}
# print the sum
print "sum = $sum\n";
```



Perl Loop

```
#!/usr/local/bin/perl
$a = 10;
# do...while loop execution
do{
print "Value of a: $a\n";
$a = $a + 1;
}while( $a < 20 );
```

```
Value of a: 10
Value of a: 11
Value of a: 12
Value of a: 13
Value of a: 14
Value of a: 15
Value of a: 16
Value of a: 17
Value of a: 18
Value of a: 19
```

Perl Loop

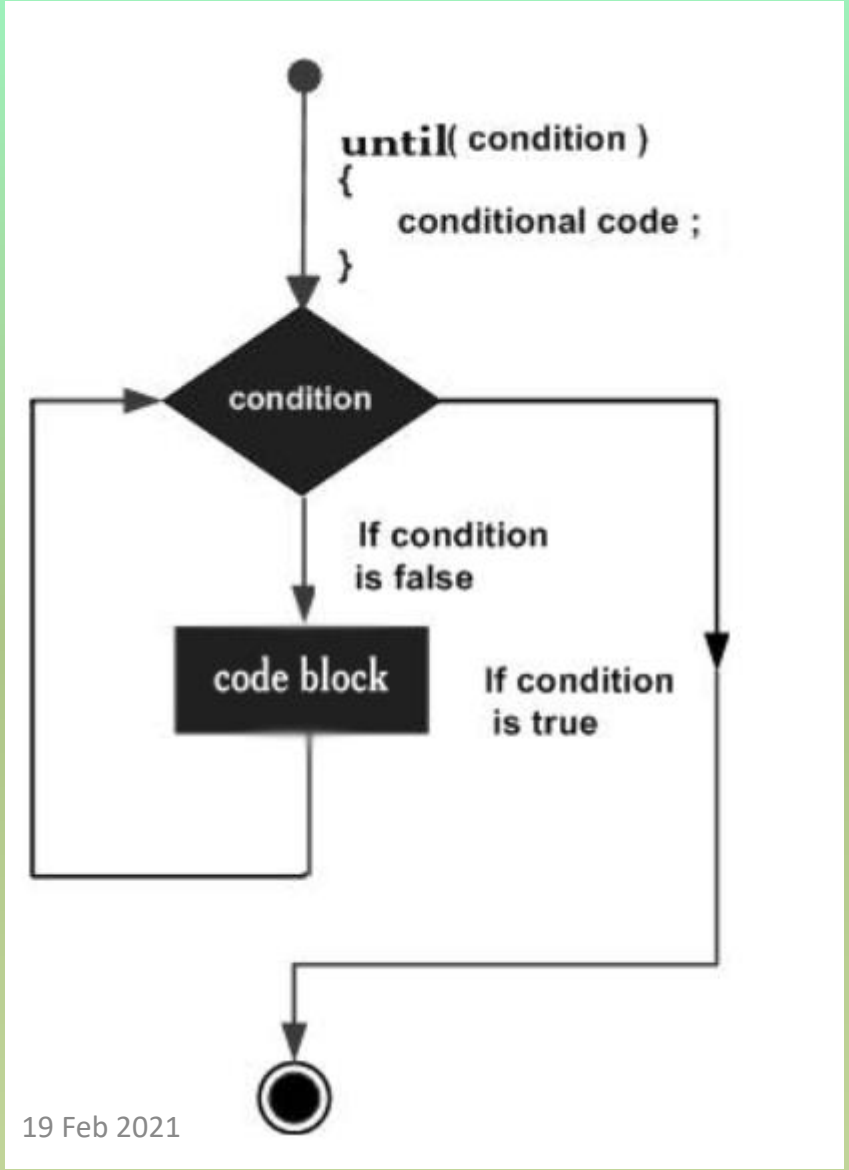
```
#!/usr/bin/perl
print "content-type: text/html \n\n";
$count = 0;
while ($count <= 7) {
print "$count<br />";
$count ++;
}
print "Finished Counting!";
```

```
#!/usr/bin/perl
print "content-type: text/html \n\n";
print "<table border='1'>";
@names = qw(Steve Bill Connor Bradley);
$count = 1;
$n = 0;
while ($names[$n]) {
print"<tr><td>$count</td><td>$names[$n]</td></tr>";
$n++;
$count++;
}
print "</table>“;
```

Perl Loop

```
#!/usr/local/bin/perl
$a = 5; # until loop execution
until( $a > 10 )
{
printf "Value of a: $a\n";
$a = $a + 1;
}
```

```
Value of a: 5
Value of a: 6
Value of a: 7
Value of a: 8
Value of a: 9
Value of a: 10
```



Summary

Scalar Variables
Array
Hash
Operators
Conditional Statements and Looping
Subroutines and Parameters

END