Introduction to Perl Basics I

Georgia Advanced Computing Resources Center (GACRC)
EITS/UGA
Zhuofei Hou
Outline

• GACRC
• Perl Overview
• Run Perl Script
• Perl Data Types
GACRC

- We are a high-performance-computing (HPC) center at UGA
- We provide to the UGA research and education community an advanced computing environment:
  - HPC computing and networking infrastructure located at the Boyd Data Center
  - Comprehensive collection of scientific, engineering and business applications
  - Consulting and training services
- [http://wiki.gacrc.uga.edu](http://wiki.gacrc.uga.edu) (GACRC Wiki)
- [https://wiki.gacrc.uga.edu/wiki/Getting_Help](https://wiki.gacrc.uga.edu/wiki/Getting_Help) (GACRC Support)
- [http://gacrc.uga.edu](http://gacrc.uga.edu) (GACRC Web)
Perl Overview

• What does “Perl” Stand for?

• Perl’s Brief History

• Tasks Perl is Good and Not Good for

• Perl’s “Hello, World!”
What does “Perl” Stand for?

- Option 1: Practical Extraction and Report Language
- Option 2: Pathologically Eclectic Rubbish Lister

Which would you select?
Perl’s Brief History

• In the mid-1980s, *awk* ran out of steam to produce large reports

• Low-level C/C++  \(\leftrightarrow\) High-level shell scripting, *awk*, *grep*, *sed*

Creator: Larry Wall
Tasks Perl is Good and Not Good for

• Good for:
  Optimized for working with text (90%) and everything else (10%)

• Not Good for:
  Binary coding, GUI programing, OOP programming

• Easy to use, but take some time to learn
Perl’s “Hello, World!”

```perl
#!/usr/bin/perl
@lines=`ls -l`; # outputs of command ls -l are read into @lines, @ defines array
foreach(@lines) # for each item in array @lines
{
    chomp; # remove tailing New Line character from each line
    print "$_\n" if /^d/; # print the line to screen if the begins with 'd' (for directory)
}
print “Hello World!\n”; # print a message to screen
```
Run Perl Script

**Option 1:**

- Create a Perl script `my_script.pl`:
  ```perl
  #!/usr/bin/perl
  print "Hello World!\n";
  ```
- Make `my_script.pl` executable:
  ```bash
  chmod a+x myScript.pl
  ```
- Run `my_script.pl`:
  ```bash
  ./my_script.pl
  ```

**Option 2:**

- Create a Perl script `my_script.pl`:
  ```perl
  #!/usr/bin/env perl
  print "Hello World!\n";
  ```
- Make `my_script.pl` executable:
  ```bash
  chmod a+x myScript.pl
  ```
- Run `my_script.pl`:
  ```bash
  ./my_script.pl
  ```
Run Perl Script

Option 3:

• Create a Perl script `my_script.pl`:

```
print "Hello World!\n";
```

• Run `my_script.pl` by calling `perl`:

```
perl ./my_script.pl
```

Option 4:

• For a small script with several lines, you can run it directly on the command line:

```
perl -e 'print "Hello World!\n";
'
Perl Data Types

• Scalar Data

• List Data and Array

• Scalar and List Context
Scalar Data

• Integer Number: 0, 2017, 255, -42, 786251429018 (you can have long integer!)

• Floating-Point Number: 1.25, 255.0, 7.31e2, -12e-10, 3.2E5 (default double-precision!)

• Single-Quoted String: ‘’, ‘fred’, ‘hello, world!\n’, ‘I didn’t see.’

• Double-Quoted String: “”, “barney”, “hello world!\n”, “coke\tsprite”

• Undefined Scalar Data: undef
Scalar Data

• Scalar Variable: `$variableName`, a scalar variable holds a scalar.

• Perl identifier: Made up of letters, underscores, and digits.
  
  Starts with a letter (a-z, A-Z) or underscore (_ ) only!

• Scalar Assignment = with +, -, *, / (for numbers) and ., x (for strings):

```
$i=10;
$x= 10.11;
$y =$x + $i;  # $y is 20.11
$y = $y * 2;  # $y is 40.22
```

```
$s1 = 'Hello';
$s2 = "World!" ;
$s3 = $s1 . "", " . $s2;  # $s3 is "Hello, World!"
$s3 = $s3 x 2;  # $s3 is "Hello, World!Hello, World!"
```

Note: When you use . and x, surrounding spaces are suggested!
Scalar Data

• Scalar Binary Assignment with +=, -=, *=, /=, **=(for numbers) and .=, x=(for strings):

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i=10;</td>
<td></td>
<td>$i=10</td>
</tr>
<tr>
<td>$i+=5;</td>
<td># $i is 15</td>
<td>$i=15</td>
</tr>
<tr>
<td>$i*=2;</td>
<td># $i is 30</td>
<td>$i=30</td>
</tr>
<tr>
<td>$i-=20;</td>
<td># $i is 10</td>
<td>$i=10</td>
</tr>
<tr>
<td>$i**=3;</td>
<td># $i is 1000</td>
<td>$i=1000</td>
</tr>
</tbody>
</table>

$s = 'Hello';
$s .= "World!";     # $s is “Hello, World!”
$s x= 2;           # $s is “Hello, World!Hello, World!”

Note: When you use .= and x=, *surrounding spaces are suggested!*
Scalar Data

- Auto Conversion between Numbers and Strings:

Perl uses them nearly **interchangeably**! Numbers or Strings?

It is depend on the **operator** that you are using! ➔ *Operator Overloading*

```
$i=10;          # $i is an integer scalar
$s=“17”;       # $s is a numeric string scalar
$i + $s;       # + is a numeric operator ➔ result is 27
$i . $s;       # . is a string operator ➔ result is “1017”
$s * 2;         # * is a numeric operator ➔ results is 34
$s x 2;         # x is a string operator ➔ result is “1717”
```
List Data

• List: A collection of scalars. List is a data type!

(1, 2, 3, 4, 5); # a list of 5 integer scalars
(“fred”, 3.14, 1); # a list of 1 string and 2 number scalars
(); # an empty list
(1..100); # a list of 100 integer scalars 1, 2, 3, ….., 100
$m=1; $n=10; ($m..$n); # a list of 10 integer scalars 1, 2, 3, ….., 10
qw / fred barney rock / # qw: quoted words, equivalent to (“fred”, “barney”, “rock”)
qw / 1 2 3 4 5 / # equivalent to (“1”, “2”, “3”, “4”, “5”)

Note: You can choose any punctuation character as the delimiter, e.g., qw ( ... ) or qw ! ... ! or qw # ... #
Array

• Array Variable: @arrayName

• An array variable holds a list, just like a scalar variable holds a scalar.

@arr1 = ("fred", 3.14, 1);  # @arr1 holds a list of 1 string and 2 number scalars
@arr2 = ();                 # @arr2 holds an empty list
@arr3 = (1..100);          # @arr3 holds (1, 2, 3, ……, 100)
@arr5 = qw / fred barney rock /;  # @arr5 holds ("fred", "barney", "rock")
**Array**

- **Array Indexing Range:**

  \[
  @arr = (1, 2, 3, 4, 5);
  \]

  \[
  \begin{array}{c|c|c|c|c|c}
  \text{Index} & 0 & 1 & 2 & 3 & 4 \\
  \end{array}
  \]

  1\text{st} scalar : $\text{arr}[0] \rightarrow 1$
  2\text{nd} scalar : $\text{arr}[1] \rightarrow 2$
  3\text{rd} scalar : $\text{arr}[2] \rightarrow 3$
  4\text{th} scalar : $\text{arr}[3] \rightarrow 4$
  5\text{th} scalar : $\text{arr}[4] \rightarrow 5$

  $\text{arr}[1] + \text{arr}[4] = ?$

- **Special Array Index:**

  $\#\text{arr} \rightarrow 4$

  5\text{th} scalar : $\text{arr}[\#\text{arr}]$

- **Quiz 1:** How many items in arr? $\#\text{arr}$ or $\#\text{arr} + 1$?

- **Quiz 2:** What’s the difference?

  \[
  (\text{arr}[0], \text{arr}[1], \text{arr}[2]) = ("fred", 3.14, 1) \\
  (\text{arr}[0], \text{arr}[1], \text{arr}[2]) = ("fred", 3.14, 1)
  \]

  list assignment
Array

• Adding and Deleting Elements:
  - pop and push
  - shift and unshift
  - splice
Array

• pop and push on @a = (1, 2, 3, 4, 5):

  pop @a;       # @a is (1, 2, 3, 4); 5 is discarded
  $v = pop @a;  # @a is (1, 2, 3); $v is 4
  push @a, “fred”; # @a is (1, 2, 3, “fred”)  
  push @a, qw ( a b c ); # @a is (1, 2, 3, “fred”, “a”, “b”, “c”)

• shift and unshift on @a = (1, 2, 3, 4, 5):

  shift @a;      # @a is (2, 3, 4, 5); 1 is discarded
  $v = shift @a; # @a is (3, 4, 5); $v is 2
  unshift @a, “barney”; # @a is (“barney”, 3, 4, 5)
  unshift @a, qw ( a b c ); # @a is (“a”, “b”, “c”, “barney”, 3, 4, 5)
Array

• splice on array \@a \Rightarrow \text{splice } \@a \text{ START, LEN, REPLIST} \quad (\text{LEN and REPLIST are optional!})
Array

- splice on @a = ("fred", "barney", 3, 4, "tom", 7):

  ```perl
  @b = splice @a, 2, 3, qw( a b );  # @a is ("fred", "barney", "a", "b", 7)
  # @b is (3, 4, "tom")
  @b = splice @a, 2;                # @a is ("fred", "barney", 7)
  # @b is (3, 4, "tom")
  @b = splice @a, 2;                # @a is ("fred", "barney")
  # @b is (3, 4, "tom", 7)
  ```

*Note: Each command line on this slide is operating on the initial array @a*
Array

• Common Operations:
  ➢ foreach
  ➢ sort and reverse
  ➢ join and split
  ➢ map
Array

• foreach:

  @fruits = qw /apple pear orange banana/;
  foreach $fruit (@fruits)
  {
    print "fruit is $fruit\n";
  }

• sort and reverse:

  @sorted = sort @fruits
  @reversed= reverse @sorted

# @fruits is ("apple", "pear", "orange", "banana")
# for each element, print messages to screen:
# "fruit is apple"
# "fruit is pear"
# ……

# @sorted is ("apple", "banana", "orange", "pear")
# @reversed is ("pear", "orange", "banana", "apple")
Array

• join (array ➔ string) and split (string ➔ array):

```perl
@fruits = qw /apple pear orange banana/;  # @fruits is ("apple", "pear", "orange", "banana")
$fruitString = join ",", @fruits;        # $fruitString is "apple-pear-orange-banana"

$line = "John Smith:jsmith:Physics:grad";
@info = split ":", $line;                # @info is ("John Smith", "jsmith", "Physics", "grad")
```

• map: evaluates an EXPR for each list element and returns the list with evaluated values

```perl
@negatives = (-10..0);                 # @negatives is (-10, -9, -8, ..., 0)
@positives = map abs, @negatives;      # @positives is (10, 9, 8, ..., 0)
# abs is one of built-in numeric functions of sqrt, log, sin, cos, tan, etc.
```
Scalar and List Context

• When Perl is parsing an expression, it’s always expecting either a scalar or a list.

• What Perl expects is called the context of the expression.

• How to decide? It is the operation that decides the context, for example:

Scalar Context:  
- $a = …
- $arr[3] = …
- 123 + …
- 5 += 10
- ... * 2.7
- “Hello”. …
- “Hello” x …

List Context:  
- @fred = …
- ($a, $b) = …
- push @arr, …
- foreach $i (...) 
- sort …
- reverse …
- print …
Scalar and List Context

• List in Scalar Context:

```perl
@a = qw/1 2 fred barney/;  # @a is ("1", "2", "fred", "barney")
$n = @a;                    # $n is 4 ➞ length of @arr = $#arr + 1
$s = "Hello! " x @a;       # $s is "Hello! Hello! Hello! Hello! Hello!"

@b = reverse @a;            # @b is ("barney", "fred", "2", "1")
$s = reverse @a;            # $s is "yenrabderf21"
$s = sort @a;               # $s is undef

$n = scalar @a;            # $n is 4; built-in function scalar to force scalar context
```

8/31/2017
Scalar and List Context

• Scalar in List Context:

Scalar value will be automatically used to make a *one-element list*:

```perl
@a = 1;
# @a is (1)
@a = 5 * 7;
# @a is (35)
@a = undef;
# @a is (undef)
@a = ();
# @a is an empty array

foreach $i (3)
# for each element of (3), print messages to screen:
{
    print "now it is $i\n";
}
```

Thank You!

Basics I: Perl overview, fundamental data types

Basics II: programming structures (if, unless, while, until, foreach, for), Input and Output

Basics III: Perl Regular Expression, text processing, file test and directory operation