UNIX Shell Programming

Reminder that there are some useful tutorials for UNIX Shell use and programming. See the Software Carpentries tutorial and give it a try.

Basic UNIX programming in the BASH shell can help you do make some simple things more possible. More complicated programming is probably better achieved in a scripting language like Python which will be covered in the rest of the course, but BASH can be very powerful and useful to apply these to improve the tools.

See the Software Carpentry tutorial.

Variables

Variables are used to store information in Variables. To access a value of a variable in UNIX you can prefix it with `$`.

For example to assign a variable a value

```
NAME="GeneA"
NAME2="GeneB"
echo "$NAME $NAME2"

NAME="GeneC"
NAME3=$NAME.$NAME2
echo "$NAME $NAME2 $NAME3"
```

Loops and Logic

`if [ TEST ]; then DOSOMETHING fi` can be used to test for a logical statement. This testing structure also allows for other conditions to be met with `elif` or "else if" and `else`.

For example:

```
if [ $NAME == "GeneC" ]
then
   echo "Name is C"
fi
if [ $NAME != "GeneA" ]
then
   echo "Name is not GeneA"
fi
NAME="GeneA"
if [ $NAME == "GeneA" ]; then
  echo "A"
```

```
elif [ $NAME == "GeneB" ]; then
  echo "B"
else
  echo "had another class for NAME: $NAME"
fi
NAME="genea"
if [ $NAME == "GeneA" ]; then
  echo "A"
elif [ $NAME == "GeneB" ]; then
  echo "B"
else
  echo "had another class for NAME: $NAME"
fi
```

The structure requires the `[ ]` and there is expected to be a space between the `[` or `]` and the options; The `then` is also require but if you want to compact this slightly differently.

```
if [ $NAME == "GeneC" ]; then echo "Name is C"; fi
```

Multiple tests can be applied in same if statement but require double brackets.

```
NAME=GeneC
if [[ $NAME == "GeneC" || $NAME == "GeneB" ]]; then
  echo "Name is $NAME"
fi
```

More logical operator such as testing if a number if smaller or greater with `-gt` and `-lt`.

```
NUM=10
if [ $NUM -gt "0" ]; then
  echo "NUM is greater than 0"
fi
```

Can test if one file is newer than another with `-nt`. Also showing how to use `else`.

```
touch fileA.fasta
echo "second file" > fileB.fasta
if [ fileA.fasta -nt fileB.fasta ]; then
  echo "File A is newer"
else
  echo "File B is newer"
fi
```

Really useful testing options

- `-f` - if the variable is a file and exists

- `-s` - if file exists and is not zero
- `-d` - if the variable is a directory
- `-z` - if variable is empty

```
if [ ! -f $file1 ]; then
 echo "file $file1 does NOT exist"
fi

if [ -z $var1 ]; then
 echo "variable $var1 is empty"
fi

if [ -s $file2 ]; then
 echo "file $file2 exists and is not empty"
fi
```

Loops

Loops are important components for iterating through data. For loops we can specify a list to go through explicitly. for loops are structured with `for VARIABLE in LIST; do DOSOMETHING done`

```
for n in A B C D
do
 echo "$n"
done
```

Can also use the results of a function to loop through a dataset, folder of files, etc. For loops are used when the specific list is available at the start of the loop.

```
for file in $(ls *.fa)
do
  echo "file $file is found"
done
```

Can use the `seq` function to make a list of numbers. Arguments are either the ending number, or start and end, or start, end, and offset.

```
seq 3 # start at 1 and count to 3
1
2
3
```

```
seq 5 7 # start at 5 end at 7
5
6
7
```

```
seq 5 2 10  # start at 5, end at 10, offset by 2
5
7
9
```

So if you want to iterate through a bunch of numbers.

```
for m in $(seq 3 15)
do
  echo "m is $m"
done
```

Using UNIX tools with Variables

Capturing output from a program is also a useful. For example if you want to do simple mathematical arithmetic with the UNIX tool `expr` (or "evaluate expression"). It takes arguments for simple math.

To save the result from a command you can use the `$( )` structure and also can use the "‘" backquote, they both will work for taking the output from an application and saving it in a variable.

```
n=$(echo "ABCDEFG" | wc -c) # this prints out the number of characters
echo "$n characters"
n=`echo "ABCD" | wc -c`
echo "$n characters"
```

```
a=1
echo "a is $a"
expr $a + 1
a=$(expr $a + 1)
echo "a is now $a"
```

Loops again

While loops can be used which can run

```
N=1
while [ $N -lt 10 ]
do
  echo "N is $N"
  N=$(expr $N + 1)
done
```

Can also use while to read data from a file using the `read` directive.

```
echo "wolf tooth animal" > data.txt
echo "snake fang animal" >> data.txt
echo "mantis mandible insect" >> data.txt
while read COL1 COL2 COL3
do
 echo "COL1 is $COL1; COL3 is $COL3"
done < data.txt
```

How these columns are delimited are dependent on an environment variable defined `$IFS`. For example to separate columns based on comma:

```
echo "wolf,tooth,animal" > data.csv
echo "snake,fang,animal" >> data.csv
echo "mantis,mandible,insect" >> data.csv
IFS=,
while read COL1 COL2 COL3
do
 echo "COL1 is $COL1; COL3 is $COL3"
done < data.csv
```

Can also pass data INTO the while loop with pipes. This is a really useful way to parse out columns of data.

```
IFS=,
echo "Hop,Skip,Jump" | while read COL1 COL2 COL3;
do
   echo "COL1=$COL1 ... COL2 is $COL2"
done
```

Data Processing

https://www.safaribooksonline.com/library/view/bioinformatics-data-skills/9781449367480/ch07.html#chapter-07

https://github.com/biodataprog/GEN220_data/tree/main/data

sort Sort data and files.

```
sort file.txt > file.sorted.txt
```

Type of sorting: * -d/–dictionary_order : consider only blanks & alphanumeric characters * -n/–numeric-sort : compare according to string numerical value * -f/–ignore-case : upper/lower doesn't matter * -r/–reverse : reverse the order * -k : specify the key positions to sort by

```
#generate some random numbers between 0 and 100
for n in $(seq 100); do echo $(($RANDOM%100)); done > numbers.txt

sort numbers.txt | head -n 10
10
```

```
10
12
25
30
34
39
42
49
49
```

But if sort by numeric - you see there are some numbers < 10 which weren't shown.

```
sort -n numbers.txt | head -n 10
0
1
2
3
4
6
6
8
8
13
```

uniq - Collapse runs of words/numbers into unique list. This only works if the data are sorted.

```
sort -n numbers.txt | uniq | head -n 10
0
1
2
3
4
6
8
13
15
16
```

To see the numbers (or words) uniquely with counts of the occurrences use '-c'.

```
sort -n numbers.txt | uniq -c | head -n 10
   1 0
   1 1
   1 2
   1 3
   1 4
   2 6
   2 8
   3 13
   2 15
   2 16
```

Hey let's sort this list so we know the numbers that show up most frequently

```
$ sort -n numbers.txt | uniq -c | sort -r -n | head -n 8
   4 91
   4 54
   4 32
   3 57
   3 22
   3 17
   3 13
   2 95
```

Sort Multicolumn data - you can sort by the 2nd or 3rd column.

```
head -n 10 data/rice_random_exons.bed
Chr7    21408673    21408826
Chr9    16031526    16031938
Chr11    4762531    4762595
Chr8    54040    54193
Chr10    19815475    19815747
Chr3    16171331    16172869
Chr10    2077882    2077938
Chr3    20517604    20517936
Chr10    9777446    9777527
Chr2    4967096    4967246
$ sort -k1,1 -k2,2n data/rice_random_exons.bed | head -n 5
Chr1    12152    12435
Chr1    98088    98558
Chr1    216884    217664
Chr1    291398    291534
Chr1    338180    338310
$ sort -k1,1 -k2,2n data/rice_random_exons.bed | tail -n 5
Chr9    22369724    22369776
Chr9    22508926    22509014
Chr9    22753347    22753458
Chr9    22924316    22924424
ChrSy    136034    136323
```

cut Cut - subselect and print certain columns from a file

Here we will process output from NCBI blast - this is default output for the tab delimited format. It has 12 columns.

```
YAR060C Chr_I 100.00 336 0   0    336    1   217148    217483    8.6e-83    298.8
```

```
YAR060C Chr_I 64.00  325 95 22    330   14   198385    198695    4.1e-18    84.0
YAR060C Chr_I 74.07  108 25  3    110    6   211012    211119    2.1e-10    58.4
YAR060C Chr_I 97.02  336 8   2    1    336    14799    15132    1.3e-77    281.6
YAR060C Chr_I 72.48  109 25  5    6    110    20974    21081    2.3e-10    58.2
YAR061W Chr_I 100.00 204 0   0    1    204   218131    218334    3.4e-54    203.1
YAR061W Chr_I 70.62  194 57  0    1    194   203400    203593    6.5e-23    99.2
YAR061W Chr_I 94.61  204 7   4    204    1    13951    14150    5e-48    182.6
YAR061W Chr_I 67.88  193 62  0    194    2    27770    27962    3.9e-20    90.0
YAL030W Chr_I 100.00 252 0   0    103  354    87502    87753    2.5e-55    207.7
```

Just print out the first column of sequence names. The `-f1` option specifies only to print Column 1

```
cut -f1 data/yeast_orfs-to-chr1.FASTA.tab | head -n 7
YAR060C
YAR060C
YAR060C
YAR060C
YAR060C
YAR061W
```

Print out Column 2

```
cut -f2 data/yeast_orfs-to-chr1.FASTA.tab | head -n 5
Chr_I
Chr_I
Chr_I
Chr_I
Chr_I
```

Get the Query name and Percent Identity which are contained in Column 3

```
cut -f1,3 data/yeast_orfs-to-chr1.FASTA.tab | head -n 5
YAR060C    100.00
YAR060C    64.00
YAR060C    74.07
YAR060C    97.02
YAR060C    72.48
YAR061W    100.00
YAR061W    70.62
YAR061W    94.61
YAR061W    67.88
YAL030W    100.00
YAL030W    98.15
```

Sort data on the percent identity column (number 3 -specify this is a numeric sort). The `-k3,3` means sort the first (and only in this example) sort key starts at column 3 and ends at column 3. If you wanted the key to span multiple fields you can specify it with '-kSTART,END'. For more complicated sorting scheme see some answers 1 2.

After sorting, cut two columns out (columns 1 and 3), and only print out top 5 for our example with `head -n 5`.

```
sort -k3,3nr data/yeast_orfs-to-chr1.FASTA.tab | cut -f1,3 |  head -n 5
HRA1	100.00
YAL001C	100.00
YAL002W	100.00
YAL003W	100.00
YAL003W	100.00

# 9th column is Bitscore (a measure of similarity in the alignment)
$ sort -k9,9n data/yeast_orfs-to-chr1.FASTA.tab  | cut -f1-3,9 | head -n 5
YAL069W Chr_I   100.00  335
YAL068W-A   Chr_I   100.00  538
YAL068C Chr_I   100.00  1807
YAR020C Chr_I   79.76   2008
YAL067W-A   Chr_I   100.00  2480
```

Made up example, but you can cut two columns out. And also use Paste to combine things back together.

```
cut -f1,3,4 data/yeast_orfs-to-chr1.FASTA.tab > first_cols.tab
cut -f1,7 data/yeast_orfs-to-chr1.FASTA.tab > second_cols.tab

paste first_cols.tab second_cols.tab | head -n 5
YAL027W 100.00  786     YAL027W 1
tL(CAA)A        100.00  44      tL(CAA)A        39
tL(CAA)A        100.00  38      tL(CAA)A        1
YAL028W 100.00  1587    YAL028W 1
YAL029C 100.00  4416    YAL029C 4416
```

AWK

Can also use `awk` to process column data.

```
awk '{print $1}' yeast_orfs-to-chr1.FASTA.tab   # print out the first column of a file
# specificy a different delimiter (,)
head -n 3 data/random_exons.csv
Chr5,27781790,27781800
Chr11,14656670,14656870

$ awk -F, '{print $1,$2}' data/random_exons.csv | head -n 3
Chr5 27781790
Chr11 14656670
Chr3 14560358
```

Here get the length of an aligmment (column 6 is the START and column 7 is the end) using awk.

```
awk '{print $7-$6}' data/yeast_orfs-to-chr1.FASTA.tab | sort -n | head -n 4
-42
-42
-26
-26
awk '{print $7-$6}' data/yeast_orfs-to-chr1.FASTA.tab | sort -nr | head -n 5
4614
4607
4416
4308
4286
```

Advanced Variable usage

BASH also supports the concepts of Arrays. This tutorial provides useful summary of how to use arrays.

A simple example is like this

```
animals=(dog cat mouse)
for name in ${animals[@]};
do
 echo "name is $name"
done
# add to the array
animals+=(snake)
for name in ${animals[@]};
do
 echo "name is $name"
done
```