Developers

Developed by: Yuan Zhen

Supervisors: Pro. Fangjie Zhu

Horticultural Plant Biology and Metabolomics Center,Haixia Institute of Science and Technology,Fujian Agriculture and Forestry University

########################

This pipeline is to make a software from shell and R scripts by yourself,you can do it easily!

########################

Basic knowledge

First of all,a good beginning is half done

Linux Bash Scripting :Parameters and Options

Read parameters

you can use variable to process input parameters:

- $0 is the script's name
- $1 is the 1st parameter
- $2 is the 2nd parameter

The shell can read the 9th parameter,which is $9.

```
##touch a file name yuanzhen
#!/bin/bash
echo $0 ##script name
echo $1 ## 1st parameter
echo $2 ## 2nd parameter
echo $3 ## 3rd parameter

##Check the results of the following command:
./myscript 5 10 15

##The parameters are not restricted to numbers; they could be strings like this:
#!/bin/bash
echo Hello $1, how do you do

##Check the results of the following command:
./myscript yuanzhen
```

What if our parameter contains a space, and we want to pass it as one value? I guess you know the answer from the previous posts. The answer is to use quotations.

If your script requires over nine parameters, you should use braces like this ${10}.

Check parameters

If you don’t pass parameters and your code is expecting it, your script will exit with an error.

That’s why we should use a Linux if statement to make sure that they exist.

```
## you can have a try by youself
#!/bin/bash

if [ -n "$1" ]; then # If first parameter passed then print Hi

    echo Hi $1.

else

    echo "No parameters found. "

fi
```

Counting parameters

To get how many parameters passed, you can use this variable ($#).

```
#!/bin/bash

echo There were $# parameters passed.
```

Process options with values

When you dig deep into Linux bash scripting, sometimes you need options with additional parameter values like this:

```
./myscript -a value1 -b -c value2
```

There should be a way to identify any additional parameter for the options and be able to process it.

```
#!/bin/bash

while [ -n "$1" ]; do # while loop starts

    case "$1" in

    -a) echo "-a option passed" ;;

    -b)
        param="$2"

        echo "-b option passed, with value $param"
        shift
        ;;

    -c) echo "-c option passed" ;;

    --)
        shift # The double dash makes them parameters

        break
        ;;

    *) echo "Option $1 not recognized" ;;

    esac

    shift

done

total=1

for param in "$@"; do

    echo "#$total: $param"

    total=$(($total + 1))

done
```

From the results, you can see that we get the parameter for the -b option using the $2 variable.

Getting user input using the read command

Sometimes you need data from the user while the bash scripting is running.

The bash shell uses the read command for this purpose.

The read command reads input from standard input (the keyboard) or a file descriptor and stores it in a variable:

```
#!/bin/bash

echo -n "What's your name: "

read name

echo "Hi $name,"
```

You can specify multiple inputs like this:

```
#!/bin/bash

read -p "What's your name: " first last

echo "Your data for $last, $first…"
```

If you don’t specify variables for the read command, it will save all incoming inputs in the REPLY variable.

You can use the -t option to specify a timeout for input in seconds

```
#!/bin/bash

if read -t 5 -p "What's your name: " name; then

    echo "Hi $name, how are you?"

else

    echo "You took much time!"

fi
```

Read files

The read command can read files one line on each call.

Now, if you want to get all file data, you can get the content using the cat command, then send it to the read command using while loop like this:

```
#!/bin/bash

count=1

# Get file content then pass to read command by iterating over lines using while
command

cat myfile | while read line; do

    echo "#$count: $line"

    count=$(($count + 1))

done

echo "Finished"
```

Or we can make it simpler by redirecting the file content to the while loop like this:

```
#!/bin/bash

while read line; do

    echo $line

done <myfile
```

We just pass the file content to the while loop and iterate over every line and print the line number and the content, and each time you increase the count by one.

Linux shell command

set

set: set [-abefhkmnptuvxBCHP] [-o option-name] [--] [arg ...]
  Set or unset values of shell options and positional parameters.

```
Change the value of shell attributes and positional parameters, or
display the names and values of shell variables.

Options:
  -a  Mark variables which are modified or created for export.
  -b  Notify of job termination immediately.
  -e  Exit immediately if a command exits with a non-zero status.
  -f  Disable file name generation (globbing).
  -h  Remember the location of commands as they are looked up.
  -k  All assignment arguments are placed in the environment for a
      command, not just those that precede the command name.
  -m  Job control is enabled.
  -n  Read commands but do not execute them.
  -o option-name
      Set the variable corresponding to option-name:
          allexport    same as -a
          braceexpand  same as -B
          emacs        use an emacs-style line editing interface
          errexit      same as -e
          errtrace     same as -E
          functrace    same as -T
          hashall      same as -h
          histexpand   same as -H
          history      enable command history
          ignoreeof    the shell will not exit upon reading EOF
          interactive-comments
                       allow comments to appear in interactive commands
          keyword      same as -k
          monitor      same as -m
          noclobber    same as -C
          noexec       same as -n
          noglob       same as -f
          nolog        currently accepted but ignored
          notify       same as -b
          nounset      same as -u
          onecmd       same as -t
          physical     same as -P
          pipefail     the return value of a pipeline is the status of
                       the last command to exit with a non-zero status,
                       or zero if no command exited with a non-zero status
          posix        change the behavior of bash where the default
                       operation differs from the Posix standard to
                       match the standard
          privileged   same as -p
          verbose      same as -v
          vi           use a vi-style line editing interface
          xtrace       same as -x
```

```
    -p  Turned on whenever the real and effective user ids do not match.
        Disables processing of the $ENV file and importing of shell
        functions.  Turning this option off causes the effective uid and
        gid to be set to the real uid and gid.
    -t  Exit after reading and executing one command.
    -u  Treat unset variables as an error when substituting.
    -v  Print shell input lines as they are read.
    -x  Print commands and their arguments as they are executed.
    -B  the shell will perform brace expansion
    -C  If set, disallow existing regular files to be overwritten
        by redirection of output.
    -E  If set, the ERR trap is inherited by shell functions.
    -H  Enable ! style history substitution.  This flag is on
        by default when the shell is interactive.
    -P  If set, do not resolve symbolic links when executing commands
        such as cd which change the current directory.
    -T  If set, the DEBUG and RETURN traps are inherited by shell functions.
    --  Assign any remaining arguments to the positional parameters.
        If there are no remaining arguments, the positional parameters
        are unset.
    -   Assign any remaining arguments to the positional parameters.
        The -x and -v options are turned off.

Using + rather than - causes these flags to be turned off.  The
flags can also be used upon invocation of the shell.  The current
set of flags may be found in $-.  The remaining n ARGs are positional
parameters and are assigned, in order, to $1, $2, .. $n.  If no
ARGs are given, all shell variables are printed.

Exit Status:
Returns success unless an invalid option is given.
```

you can read more about this part

On Unix-like operating systems, the **set** command is a built-in function of the Bourne shell (sh), C shell (csh), and Korn shell (ksh), which is used to define and determine the values of the system environment.

In **sh**, the **set** built-in command has the following options:

--	**An option of a double-dash ("--") signifies the end of an option list. This option is primarily useful when values listed after the options start with a dash themselves.**
-a	Mark variables that are modified or created for "export"; environment variables set in this way will be passed on to the environments of any subsequent commands.
-e	Exit immediately if a command exits with a non-zero exit status.
-f	Disable file name generation (globbing).
-h	Locate and remember function commands as functions are defined (function commands are normally located when the function is executed).
-k	All keyword arguments are placed in the environment for a command, not only those that precede the command name.
-n	Read commands but do not execute them.
-t	Exit after reading and executing one command.
-u	Treat unset variables as an error when substituting.
-v	Print shell input lines as they are read.
-x	Print commands and their arguments as they are executed.

Using **+** rather than **-** causes these flags to be turned off. These flags can also be used upon invocation of the shell itself. The current set of flags are found in the variable **$-**. The remaining arguments are positional parameters and are assigned, in order, to **$1**, **$2**, etc. If no arguments are given the values of all names are printed.

For each name, the **unset** command removes the corresponding variable or function value. The special variables **PATH**, **PS1**, **PS2**, **MAILCHECK**, and **IF** cannot be unset.

With the **export** built-in command, the given names are marked for automatic export to the environment of subsequently executed commands. If no arguments are given, variable names that are marked for export during the current shell's execution are listed. Function names are not exporte

GUN parallel

you can learn more about this part or a Chinese intrudction .

There are ways to run commands in parallel that are not built into Bash. GNU Parallel is a tool to do just that.

GUN Parallel ,as its name suggests ,can be used to build and run commads in parallel.You may run the same command with different arguments,whether they are filenames,username, hostname,or line read from files.GUN Parallel provides shorthand reference to many of the most common operations(input lines, various portions of the input line, different ways to specify the input source , and so on). Parallel can replace xargs or feed commands from its input sources to several different instance of bash.

For example, it is easy to replace `xargs` to gzip all html files in the current directory and its subdirectories:

```
find . -type f -name '*.html' -print | parallel gzip
```

If you need to protect special characters such as newlines in file names, use find's `-print0` option and `parallel's -0` option.

You can use Parallel to move files from the current directory when the number of files is too large to process with one `mv` invocation:

```
printf '%s\n' * | parallel mv {} destdir
```

As you can see, the {} is replaced with each line read from standard input. While using `ls` will work in most instances, it is not sufficient to deal with all filenames. `printf` is a shell builtin, and therefore is not subject to the kernel's limit on the number of arguments to a program, so you can use '*' (but see below about the `dotglob` shell option). If you need to accommodate special characters in filenames, you can use

```
printf '%s\0' * | parallel -0 mv {} destdir
```

This will run as many `mv` commands as there are files in the current directory. You can emulate a parallel `xargs` by adding the -X option:

```
printf '%s\0' * | parallel -0 -X mv {} destdir
```

(You may have to modify the pattern if you have the `dotglob` option enabled.)

GNU Parallel can replace certain common idioms that operate on lines read from a file (in this case, filenames listed one per line):

```
while IFS= read -r x; do
        do-something1 "$x" "config-$x"
        do-something2 < "$x"
    done < file | process-output
```

with a more compact syntax reminiscent of lambdas:

```
cat list | parallel "do-something1 {} config-{} ; do-something2 < {}" |
            process-output
```

Parallel provides a built-in mechanism to remove filename extensions, which lends itself to batch file transformations or renaming:

```
ls *.gz | parallel -j+0 "zcat {} | bzip2 >{.}.bz2 && rm {}"
```

This will recompress all files in the current directory with names ending in .gz using bzip2, running one job per CPU (-j+0) in parallel. (We use `ls` for brevity here; using `find` as above is more robust in the face of filenames containing unexpected characters.) Parallel can take arguments from the command line; the above can also be written as

```
parallel "zcat {} | bzip2 >{.}.bz2 && rm {}" ::: *.gz
```

If a command generates output, you may want to preserve the input order in the output. For instance, the following command

```
{
    echo foss.org.my ;
    echo debian.org ;
    echo freenetproject.org ;
} | parallel traceroute
```

will display as output the traceroute invocation that finishes first. Adding the `-k` option

```
{
    echo foss.org.my ;
    echo debian.org ;
    echo freenetproject.org ;
} | parallel -k traceroute
```

will ensure that the output of `traceroute foss.org.my` is displayed first.

Finally, Parallel can be used to run a sequence of shell commands in parallel, similar to `cat file | bash`. It is not uncommon to take a list of filenames, create a series of shell commands to operate on them, and feed that list of commands to a shell. Parallel can speed this up. Assuming that file contains a list of shell commands, one per line,

```
parallel -j 10 < file
```

will evaluate the commands using the shell (since no explicit command is supplied as an argument), in blocks of ten shell jobs at a time.

nice

you can read more about this part

nice runs command *COMMAND* with an adjusted "niceness", which affects process scheduling. A process with a lower niceness value is given higher priority and more CPU time. A process with a higher niceness value (a "nicer" process) is given a lower priority and less CPU time, freeing up resources for processes that are more demanding.

```
nice [-n adjustment] [-adjustment] [--adjustment=adjustment] [--help] [--
version] [command [arg...]]
```

-n*N, --*adjustment*=N*	**Add integer *N* to the niceness (default is 10).**
--help	Display a help message and exit.
--version	Output version information and exit.

```
nice -n13 pico myfile.txt
###Runs the pico command on myfile.txt with a niceness increment of 13. Since we
already saw the default niceness level was zero, this runs pico with a niceness
level of zero plus 13, which is 13. As a result, pico can use CPU resources with
a higher priority than any process running with a niceness level greater than 14,
but has a lower priority than processes with a value less than 14.Effectively,
this tells the system to treat pico as a low-priority process, but not the
lowest.
```

basename

learn more about this part

The basename command in Linux prints the final component in a file path. This is particularly helpful in bash scripts where you want to extract the file name from the long file path.

The basename command has two kinds of syntax. First one involves a suffix:

```
basename PATH [suffix]
```

Second one allows you to add options:

```
basename OPTION PATH
```

You cannot combine the options with suffix. Don't be confused just yet. Follow the examples and then you'll understand what I want to say.

Using basename command with a file path will give the file name:

```
basename /home/user/data/filename.txt
filename.txt
```

The basename command is quite stupid actually. It doesn't really recognize file path. It just looks for the slashes (/) and prints the whatever is after the last slash.

For example, if I run the above example by removing the file name, here's what it will yield.

```
basename /home/user/data
data
```

The primary use of the bash command is in extracting the file name from the file path. You can also remove the file extension while extracting the file name.

Just mention what you want to remove from the end of the output. So let's say, you want to remove the .txt from filename.txt. Just add it at the end of the basename command:

```
basename /home/user/data/filename.txt .txt
filename
```

You can also use the -s option for suffix:

```
basename -s .txt /home/user/data/filename.txt
filename
```

The suffix is removed from the end of the final component of the input. It doesn't really figure out the extension of the file. If you provide txt (without the dot) instead of .txt, you'll get 'filename.' (with the dot at the end).

Also, if you provide a suffix that is not at the end of the component, the output remains as if there was no suffix.

```
basename /home/user/data/filename.txt name
filename.txt
```

With the option `-a`, you can use multiple paths simultaneously.

```
basename -a /home/user/data/filename1.txt /home/user/data/filename2.txt
filename1.txt
filename2.txt
```

You can use suffix option `-s` with `-a` but with some limitations. You can only provide one suffix to all the file paths.

```
basename -as .txt /home/user/data/filename1.txt /home/user/data/filename2.txt
filename1
filename2
```

You cannot assign individual suffices. It won't work.

You can also separate output with NULL instead of the newline with `-z` option.

I showed some examples of the basename command. Let's see a couple of examples of basename in bash scripts.

Suppose you have a file path variable and you want to store the file name from the path in a variable. This could be a simple script:

```
pathname="/home/dir/data/filename"

result=$(basename "$pathname")

echo $result
```

Another example is where you want to rename the file extensions. Of course, you can use the rename command to batch rename files but this is just an example.

So, I wrote this sample script with the purpose of replacing the file extensions:

```
for file in *$1; do
if [ -f $file ]; then
 mv $file `basename $file .$1`.$2
fi
done
```

Did you notice that I put a check if it is file or not in bash script so that it doesn't change a matching directory?

You can use the above script like this:

```
./myscript.sh html htm
```

And it will rename all the files in current directory with html at the end to htm.

These were just a few examples. You can use it as per your requirements.

The basename command is complemented with dirname command. Unlike basename, the dirname command prints all the path except the last component.

I hope you liked this tutorial. As always, feel free to ask questions or provide suggestions in the comment section.

read

you can learn this part

What is the read command in Linux?

The ***read command in Linux*** is a way for the users to interact with input taken from the keyboard, which you might see referred to as stdin (standard input) or other similar descriptions.

In other words, if you want that your bash script takes input from the user, you'll have to use the read command.

I am going to write some simple bash scripts to show you the practical usage of the read command.

Read command examples

The read command can be confusing to get started with, especially for those who are new to shell scripting. The scripts I am going to use here are very simple to understand and should be easy to follow, especially if you are practicing along with the tutorial.

Basic Programming Concepts

With almost every program or script, you want to take information from a user (input) and tell the computer what to do with that information (output).

When you use read, you are communicating to the bash terminal that you want to capture the input from the user. By default, the command will create a variable to save that input to.

```
read [options] variable_name
```

Now let's see some examples of read command to understand how you can use it in different situations.

1.Read command without options

When you type read without any additional options, you will need to hit enter to start the capture. The system will capture input until you hit enter again.

By default this information will be stored in a variable named `$REPLY`.

To make things easier to follow for the first example, I will use the `↲` symbol will show when the enter key is pressed.

```
read
hello world
echo $REPLY
hello world
```

More About Variables

As I mentioned earlier, the `$REPLY` variable is built into `read`, so you don't have to declare it.

That might be fine if you have only one application in mind, but more than likely you'll want to use your own variables. When you declare the variable with read, you don't need to do anything other than type the name of the variable.

When you want to call the variable, you will use a `$` in front of the name. Here's an example where I create the variable `Linux_Handbook` and assign it the value of the input.

You can use echo command to verify that the read command did its magic:

```
read Linux_Handbook
for easy to follow Linux tutorials.
echo $Linux_Handbook
for easy to follow Linux tutorials.
```

Reminder: Variable names are case-senstive.

2. Prompt option `-p`

If you're writing a script and you want to capture user input, there is a read option to create a prompt that can simplify your code. Coding is all about efficiency, right?

Instead of using additional lines and echo commands, you can simply use the `-p` option flag. The text you type in quotes will display as intended and the user will not need to hit enter to begin capturing input.

So instead of writing two lines of code like this:

```
echo "What is your desired username? "
read username
```

You can use the `-p` option with read command like this:

```
read -p "What is your desired username? " username
```

The input will be saved to the variable $username.

3. "Secret"/Silent option `-s`

I wrote a simpe bash script to demonstrate the next flag. First take a look at the output.

```
bash secret.sh
What is your desired username? tuxy_boy
Your username will be tuxy_boy.
Please enter the password you would like to use:

You entered Pass123 for your password.
Masking what's entered does not obscure the data in anyway.
```

Here is the content of `secret.sh` if you'd like to recreate it.

```
#!/bin/bash
read -p "What is your desired username? " username
echo "Your username will be" $username"."
read -s -p "Please enter the password you would like to use: " password
echo
echo "You entered" $password "for your password."
echo "Masking what's entered does not obscure the data in anyway."
```

As you can see, the `-s` option masked the input when the password was entered. However, this is a superficial technique and doesn't offer a real security.

4. Using a character limit with read option `-n`

You can add a constraint to the input and limit it to n number of characters in length.

Let's use the same script from before but modify it so that inputs are limited to 5 characters.

```
read -n 5 -p "What is your desired username? " username
```

Simply add `-n N` where N is the number of your choice.

I've done the same thing for our password.

```
bash secret.sh
What is your desired username? tuxy_Your username will be tuxy_.
Please enter the password you would like to use:
You entered boy for your password.
```

As you can see the program stopped collecting input after 5 characters for the username.

However, I could still write LESS than 5 characters as long as I hit `↵` after the input.

If you want to restrict that, you can use `-N` (instead of -n) This modification makes it so that exactly 5 characters are required, neither less, nor more.

5.Storing information in an array `-a`

You can also use read command in Linux to create your own arrays. This means we can assign chunks of input to elements in an array. By default, the space key will separate elements.

```
christopher@pop-os:~$ read -a array
abc def 123 x y z
christopher@pop-os:~$ echo  ${array[@]}
abc def 123 x y z
christopher@pop-os:~$ echo  ${array[@]:0:3}
abc def 123
christopher@pop-os:~$ echo  ${array[0]}
abc
christopher@pop-os:~$ echo  ${array[5]}
z
```

If you're new to arrays, or seeing how them in bash for the first time, I'll break down what's happening.

- Enter desired elements, separated by spaces.
- If we put only the @ variable, it will iterate and print the entire loop.
- The @ symbol represents the element number and with the colons after, we can tell iterate from index 0 to index 3 (as written here).
- Prints element at index 0.
- Similar to above, but demonstrates that the elements are seperated by the space

Bonus Tip: Adding a timeout function

You can also add a timeout to our read. If no input is captured in the allotted time, the program will move on or end.

```
christopher@pop-os:~$ read -t 3
christopher@pop-os:~$
```

It may not be obvious looking at the output, but the terminal waited three seconds before timing out and ending the read program.

Conclusion

I hope this tutorial was helpful in getting you started with the read command in Linux. As always, we love to hear from our readers about content they're interested in. Leave a comment below and share your thoughts with us!

practice

I use my own pipeline to make scripts with four parameters: **input_dir**,**output_dir**,**genome_dir**,**threads**(all of these path of directory should be *ABSOLUTE PATH*)

input_dir: this directory include your *.fastaq* or *.fastaq.gz* rawdata

output_dir : this directory include the data_output.specifically,when you use my scripts , in this output_dir,I will give you two results,one is the a alignment results with *bowtie*,another is the peak calling results.

genome_dir:you should give me a reference genome

threads:threads per bowtie align

I met some problems in this scripts

```
##the follow is not properly,the file1 should not with $
read -p "Please input your sequence data (you file include .fastq.gz) :"  $file1
### the right one
read -p "Please input your sequence data (you file include .fastq.gz) :"  file1
#### if you want to use this variable,you should ${file1}

##if you have many variable path,you'd better use the absolute path
```

```
Usage () {
  echo "Usage"
  echo "mnase_first.sh [-h] [-v] [-i <directoryname>] [-o <directoryname>] [-g
<reference genome>] [-t threads]"
    echo "-h   you can find help"
    echo "-v   this software version"
    echo "-i   input a directory include your sequence data (you directory
include .fastq.gz)"
    echo "-o   output your results"
    echo "-g   input your reference genome directory and genome"
    echo "-t   input your threads "
  exit 1
}

while getopts hvi:o:g:t: varname ##tell the scripts you have six options and -
i,-o,-g,-t need parameters
do
 case $varname in
   "h")
   echo "$varname"
   Usage
   exit
   ;;
   "v")
   echo"v 1.0"
   exit
   ;;
    "i")
   echo "$varname"
   echo "$OPTARG"
   input=$OPTARG  ### put the parameters to input
   if [ ! -d "$input" ];then ### whether the parameter is a directory or exit
    echo "the source directory $input not exist!"
    exit
   fi
   ;;
    "o")
   echo "$varname"
   echo "$OPTARG"
   output=$OPTARG
   if [ ! -d "$output" ];then
    echo "the output path $output not exist"
    exit
   fi
   ;;
     "g")
      echo "$varname"
      echo "$OPTARG"
      genome=$OPTARG
      if [ ! -f "$genome" ];then
         echo "YOU MUST input a exit genome file(fasta file)"
         exit
      fi
      ;;
     "t")
      echo "$varname"
      echo "$OPTARG"
      threads=$OPTARG
      if [[ $threads == *[!0-9]* ]];then
         echo "the threads should be a number"
```

```
          exit
       fi
       ;;
       :)
    echo "$varname"

   echo "the option -$OPTARG require an arguement"
    exit 1
    ;;
       ?)
    echo "$varname"
    echo "Invaild option: -$OPTARG"
    Usage
    exit 2
    ;;
   esac
 done
```