## pwd (Working Directory)

Enter the line: pwd

That is, type the text above and then press the Enter key.

The command should show you *home* directory: /home/jovyan

~ jovyan$ pwd

/home/jovyan

pwd stands for print working directory.

Unlike a file browser, bash does not automatically display the current directory in a textbox – hence the need for this command.

Execute the following commands, which we will understand later:

echo $PATH

export PATH=$HOME/Bash/bin:$PATH

If the commands have worked properly when you type:

echo $PATH

youshouldsee the output:

/home/jovyan/Bash/bin:/opt/conda/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/bin/TDM-GCC-64/bin

## bash vs Operating System and the hostname command

Enter the command: hostname

The response will tell you the name of your computer:

$ hostname

cc5b57da-44e0-4a30-baab-6b7667d8e775-7f9d594bc8-lzjm6

This bash tool is interacting with a virtual personal computer on the CyVerse cloud that is running Linux.

The long length of the host name is an indication of the large number of such computers that can be created in the cloud.

This is a personal computer in that it there is only one home directory in this computer. It is a virtual computer because it has been created on a physical computer on the cloud that pretends to be multiple virtual computers, one of which is our computer.

bash applications also exist to interact with personal desktop computers running a variety of operating systems. They all provide the same user interface. For example, if I type hostname and pwd in my home directory on my Windows computer, I see the name of my Windows 10 computer and my home directory:

$ hostname

DEWANX380YOGA

$ pwd

/c/Users/dewan

bash is, in fact, the native command-line interface to computers running Mac operating systems.

bash user-interfaces are a subset of a set of user-interfaces called *shell* user-interfaces (which also include the *shell*, and *cshell*), which are a subset of user-interfaces called *command-line interfaces* or command interpreters (which also includes the Windows native command-line user-interfaces).

Discovery provides three different GUI-based tools to interact with the Discovery System – Data, Apps and Analysis. bash provides a command-line alternative to all three of these tools.

## Discovery bash Launch vs Atmosphere

As implied above, a bash process is connected to an operating system (OS) instance –a set of predefined data files and applications. This one is connected to one preloaded with those applications and data relevant to this study.

In this one-user OS instance, there is only one home directory, called /home/jovyan, which has a copy of the input directories.

In addition, it has several predefined files such as stdout.txt. We can add data to the home directory, which will be copied back to the process output directory when we terminate the process. If we want a bash process to be connected to all applications and data available from CyVerse, we need to start a separate CyVerse system called Atmosphere, for which we need special access from CyVerse. In Atmosphere, a bash process is connected to home directories of all CyVerse users, and each user’s home directory is different and derived from the user name.

The concepts we will learn here are independent of the operating system. In fact, they are independent even of the bash user-interface, and will transfer easily to shell and other command-line user-interfaces or command interpreters.

## Select and Right Menu

While holding the SHIFT key, right click, that is, click the right mouse button. You will see the interpreter menu, which can be used to copy and paste text.

The standard short cuts of CTRL+ C and CTRL+Vfor copy and paste also work in this bash tool; in others they do not. In this document CTRL+Key means pressing CTRL and Key at the same time, where Key can be any key on your keyboard such as C, V, Z, and D.

## Tracking Progress using SuperShell

Copy the following line from this web page using CTRL + C:

/SuperShell/linux\_install\_supershell\_docker.sh

Use CTRL+V to paste this command in your bash/terminal window and press Enter.

$ /SuperShell/linux\_install\_supershell\_docker.sh

The response should be a welcome message from SuperShell, which is an extension of bash that tracks your progress and provides commands to receive help.

## ls command

A file browser devotes one or more panes to show the children of the current directory. Enter the command: ls /home/jovyan

The response of this command should show the children of the specified directory: the directory whose absolute name is /home/jovyan**.**

:~ jovyan$ ls /home/jovyan

Bash

Discovery

DNASubway

downloads.stderr.log

downloads.stdout.log

RNA\_SEQ\_SAMPLES

stderr.txt

stdout.txt

SuperShell

As the specified directory is the home directory, the children include all the children of the home directory. These include all the directories we input when we launched the Jupyter application to create our personal computer in the cloud. It includes additional predefined files and directories such as stderr.txt**.**

## Many ways to specify directories

Enter the command:

ls .

It is equivalent to the previous command as **.** stands for the current directory, and the current directory is the home directory.

Enter simply:

ls

It too is equivalent to the first ls command – if no file/directory name is provided, the **ls** command assumes the specified directory is the current directory.

Enter:

ls ..

**..** stands for the parent of the current directory. This command shows the only child of this directory – jovyan.

jovyan$ ls ..

jovyan

This command is of course equivalent to:

ls /home

There is only one home directory because this is a personal rather than shared cloud computer.

Enter:

ls /home/jovyan/RNA\_SEQ\_SAMPLES

The response should show the names of the children of the specified directory.

:~jovyan$ ls /home/jovyan/RNA\_SEQ\_SAMPLES

Drug\_A

Drug\_B

Enter the command:

ls RNA\_SEQ\_SAMPLES

It is equivalent to the earlier command. Any bash command that expects a file or directory name can be given the absolute name of the file node (e.g. /home/jovyan/RNA\_SEQ\_SAMPLES) or a local name (e.g. RNA\_SEQ\_SAMPLES**)** in the current directory (e.g. /home/jovyan/RNA)

Enter the command: ls stdout,txt**.**

It should just echo back the specified file name:

:~ jovyan$ ls stdout,txt

stdout.txt

## Linux: Case Matters

Enter the command: ls Stdout.txt

You will get an error response.

:~jovyan$ ls Stdout.txt

ls: cannot access 'Stdout.txt': No such file or directory

It seems you have an error. Enter 'shelp' for any suggestion

As mentioned earlier, the virtual cloud computer created for you runs the Linux operating system. In this operating system case matters in file and folder names. Thus name *stdout.txt* is legal but not *Stdout.txt*.

Ignore messages that suggest the use of shelp**.** This is not yet a developed feature – one we hope to mature based on usage of this system.

## Tab Completion

Enter the command: ls RNA^t

where ^t stands for the invisible tab character. *Press tab, do not enter ^t***.**  That is, click on the tab key after entering RNA**.** bash does tab completion, that is, automatically expands a unique prefix of a local directory or file name to the full local name.

Enter the text

ls D^t

bash does not do tab completion as Dis the prefix of two local names.

Enter another tab:

ls D^t^t

bash shows all local names to which the prefix can be expanded:

:~ jovyan$ ls D

Discovery/ DNASubway/

Now we can enter either i or N after D to choose Discovery or DNASubway, respectively.

Choose Discovery.

## Viewing and Editing Previous Commands

Press the up arrow character on your keyboard. bash should show the last command executed: ls Discovery/

Press it once more. bash should now show the second last command executed.

Press the down arrow characterit should show again the last command executed.

Use the left and right arrow keys to insert –l between ls and Discoveryin the last command and then press Enter to compose the command: ls –l Discovery/

As we see above, the up, down, left and right arrows can be used to edit a previously executed command to issue a new command.

$ ls -l Discovery/

total 16

drwxr-sr-x 2 jovyan 1000 4096 Apr 16 00:34 FilteredSortedOutput

drwxr-sr-x 4 jovyan 1000 4096 Apr 16 00:34 RNASequenceOutput

drwxr-sr-x 2 jovyan 1000 4096 Apr 16 00:34 TSVFiles

drwxr-sr-x 4 jovyan 1000 4096 Apr 16 00:35 UncompressedSamples

Select the text TSVFiles in the output and Use CTRL+C to copy it.

Use the up arrow to view the command: $ ls Discovery/

Use CTRL+V to paste TSVFiles and press Enter.

$ls Discovery/TSVFiles

A\_1\_abundance.tsv

A\_2\_abundance.tsv

B\_1\_abundance.tsv

B\_2\_abundance.tsv

Thus, tab completion, display and editing of previously executed commands, and the familiar copying and pasting can be used to reduce the amount of text we have to enter.

The up arrow key allows you see the history of previously commands, one at a time. What if you want to see the entire command history together?

Enter the command history.

You will see all of the commands you have entered so far, numbered.

Enter the command !!**.**

The last command (history in this case) is re-executed.

Enter !pw**.**

The last command starting with the prefix string pw in re-executed, which should be pwd.

In your history output, let N be the number of the last ls command.

Enter !N**.**

The command numbered N in the history is executed.

The ability to view and edit commands using the up and down arrows is a new feature, Using the **! (**pronounced bang**)** symbol to re-execute commands in the traditional way.They both should be useful in different contexts.

## File Attributes and Long Listing

In the listings seen so far, we see only the names of files and directories, whereas several file browsers such as the Discovery file browser show also their attributes such as size and date of modification.

Execute **ls – l**

It displays the attributes of the children of the current directory**.**

$ ls –l

total 340

drwxr-sr-x 3 jovyan 1000 74 Apr 10 19:16 Bash

drwxr-sr-x 6 jovyan 1000 122 Apr 10 19:15 Discovery

drwxr-sr-x 3 jovyan 1000 72 Apr 10 19:20 DNASubway

-rw-r--r-- 1 jovyan 1000 0 Apr 10 19:14 downloads.stderr.log

-rw-r--r-- 1 jovyan 1000 341095 Apr 10 19:20 downloads.stdout.log

drwxr-sr-x 4 jovyan 1000 46 Apr 10 19:15 RNA\_SEQ\_SAMPLES

-rw-r--r-- 1 jovyan 1000 0 Apr 10 19:43 stderr.txt

-rw-r--r-- 1 jovyan 1000 0 Apr 10 19:43 stdout.txt

drwxr-sr-x 6 jovyan 1000 4096 Apr 10 19:31 SuperShell

ls stands for listing(of child nodes) and –lstands for long**.** Thus, ls –l stands for long listingof children of current directory.

The first line in the listing gives an indication of how much space (in physical “blocks” rather than bytes) is taken by the descendants of the specified directory.

Like a regular directory listing, a long directory listing shows a line for each child of the specified directory. These lines, together, form a table in which each column corresponds to an attribute of a child.

The last column, column 7, of course, is the local name of the child (e.g. DNASubway).

The column before that, column 6, shows the time of last modification (e.g. April 10. 19:20).

Column 5 is the size of the file or directory in bytes (e.g. 72). The size of a directory is the space required to store the names of its children and their attributes, and does not include the size of its children.

Columns 3 and 4 identify the owner (e.g. jovyan) and group (e.g. 1000) of the child.

The second column gives the number of links or parents of the child. In a Linux system, a file can have multiple parents while a directory has only one parent. (This number seems wrong in the implementation we are using)

The first column, containing 10 characters, conveys two kinds of information.

The first character is d or – for a directory or file child, respectively.

The other nine characters describe the access list associated with the child.

We will learn what access lists and ownership means in the Linux world later.

Enter:

ls –l stdout.txt

The command will show the attributes of the file stdout.txtbut not its contents**.**

Thus it makes sense to list not only list a directory but also a file.

If the ls command is given multiple specified file/directory names, the result is equivalent to executing the command on each of the specified nodes.

Enter:

ls –l RNA\_SEQ\_SAMPLES stdout.txt

to try this feature.

Enter:

ls -l –R RNA\_SEQ\_SAMPLES

This command executes ls –l on the RNA\_SEQ\_SAMPLES andeach of its descendant directories. It can be used to explore your directory structure.

## Anatomy of a Command: Natural Language Analogy

We have seen above several examples of simple commands, one starting with the word pwd**,** and others starting with the word ls**.**

The first word in a simple command such as ls –l -R is an operation name.

It corresponds to the verb of an imperative sentence such as Play soccer competitively and energetically.

Some verbs – in particular transitive verbs – operate on zero more objects (e.g. play; playsoccer***,*** play soccer and tennis).

Similarly, some bash operations operate on optional input and output files/directories (e.g. ls; ls RNA\_SEQ\_SAMPLES*;* ls RNA\_SEQ\_SAMPLES 1\_abundance.tsv).

Some verbs can be qualified by zero or more adverbs (e.g. play competitively***;*** play soccer competitively and energetically).

Similarly some operations can be customized by optional *parameters* (e.g. ls –l -a; ls -1 RNA\_SEQ\_SAMPLES).

Thus, commands, operations, (input and output), file nodes, and parameters correspond to imperative sentences, verbs, objects, and adverbs.

## Anatomy of a Command: Implementation-Based

A more implementation-oriented decomposition of a bash command is based on the notion of *tokens*. A sequence of characters without spaces and tabs is a token. Thus, in the command:ls –l -R RNA\_SEQ\_SAMPLES

has four different tokens:

ls

-l

-R

RNA\_SEQ\_SAMPLES

A sequence of characters enclosed in double quotes is also a token. Thus, the command: ls “RNA SEQ SAMPLES”

has two tokens:

ls

RNA SEQ SAMPLES

A bash command is a sequence of tokens.

The first token is an *operation* name.

It can be followed by parameters and names of input and output files. These are all called the operation *arguments*or*operands*. The sequence of arguments of an operation is its *argument*or *operand list*.

In DNA Subway, we clicked on preselected applications to run them. In Discovery, we used a special application manager to search for and launch arbitrary predefined and programmer-defined applications known to Discovery’s operating system.

In bash, we execute an application by entering its name as the first word in a command and specify parameters and input and output files and directories after the application name.

DNASubway, Discovery, and bash are all united in that they have notions of application names, input and output files/directories, and application parameters – they simply provide user-interfaces to specify these components of application-launch data. DNASubway and Discovery can be viewed as *GUI-based*rather than line-based*interpreters* of commands, where commands are instructions that can be used to start and compose applications and view and manipulate the file system.

## Change to Child and Parent Directory

As we saw above, the notion of a current directory is common to both GUI and line-based interpreters.

In a GUI file browser, we can click on a child directory to make it the current directory.

In bash we enter cd followed by the local name of the child:

cd RNA\_SEQ\_SAMPLES

Execute the pwd command to show the new current directory, and ls to show the children of this directory.

$ cd RNA\_SEQ\_SAMPLES/

$ pwd

/home/jovyan/RNA\_SEQ\_SAMPLES

$ ls

Drug\_A

Drug\_B

Execute cd Drug\_A to change the current directory one more time and type ls one more time to see the names of the uncompressed files.

As you may have noticed, the prompt after each command shows the local name of the current directory.

File browsers provide buttons to change the current directory to the parent of the current directory. Recall that in DNA Subway, this button was labelled .. , the name of the parent directory in file systems. Other file browsers label it Up. In bash, we enter cd followed by the name of the parent: cd ..

Execute it repeatedly to return to your home directory.

$ cd ..

$ pwd

/home/jovyan/RNA\_SEQ\_SAMPLES

$ cd ..

$ pwd

/home/jovyan

Commands to change the current directory to a child or parent directory allows us to support arbitrary navigations through the system, though these can be tedious if we are navigating far from the home directory.

## Change to Arbitrary and Home Directory

Many file browsers like the Discovery file browser (but unlike the Subway browser) allow us to edit the displayed absolute name of the current directory to go to an arbitrary directory in the file system. In bash we can do so by entering **cd** followed by the absolute name a directory.

Enter: cd /home/jovyan/RNA\_SEQ\_SAMPLES/Drug\_B

No matter what the current directory was before the command was executed, the current directory after it was executed should now be Drug\_B.

We often or almost always name files that are descendants of our home directory.

bash allows us to use the character **~** as a shorthand for this directory.

Type: cd ~

This should return to the home directory.

Enter: cd ~/ RNA\_SEQ\_SAMPLES/Drug\_B

This should return to Drug\_B.

Enter: cd

You return to the home directory.

Entering cd without an argument is a shorthand for: cd ~ , which in turn is a shorthand for: cd /home/jovyan/

## File Name Completion with cd

While a GUI allows us to simply point and click to specify file names, bash requires us to specify their names. To reduce this overhead, it provides file name completion, as we saw when we used the **ls** command. This facility is provided by bash for file-name operands of a command. Let us get some more practice with by using it in cd commands.

in the home directory, enter: cd R^t where ^t stands for the invisible tab character. Press tab, do not enter^t**.**

As the local name of only one child of the home directory starts withR, bash automatically completes the remaining characters of the name.

Now enter: cd D^t

As there are two children of RNA\_SEQ\_SAMPLES starting with D, bash partially completes the name. Type tab again:cd Drug\_^t .bash does nothing as it does not know whether it should complete with A or B. Type tab once again: cd Drug\_^t^t**.** bash shows the names of two children that match the entered prefix, so that we can expand our prefix to be unique. Complete it to change the current directory to **Drug\_A**.

## which Operation and Built-In vs. External Applications

Enter: which ls

The response will be the absolute name of a file named /bin/ls.

$ which ls

/bin/ls

The ls operation is implemented by an application program, and the response of whichgives the name of the file that holds the code that contains the program.

While we have used bash to list a directory and file, bash actually knows nothing about how to list them! The **ls** application program is the one responsible for this task. The reason it seems bash performed the operation is that **ls** used bash’s window to interact with the user.

Enter: which cd

No file name will be output.

$ which cd

$

The reason for no output is that cd is actually implemented by bash as a built-in bashoperation**.** Thus, a bash command may invoke an internal built-in operation or invoke an external application program. Both kinds of operations have the same syntax, so the user can tell the difference only by understanding how they work and/or executing which. We will refer to the code that implements an operation as the operation implementation or simply operation. This code may be part of bash or an independent program.

## Multi-Link Relative Names

Change your directory to Drug\_A in RNA\_SEQ\_SAMPLES.

Suppose we now need to change the current directory from Drug\_A to Drug\_B.

We can enter: cd ~/RNA\_SEQ\_SAMPLES/Drug\_B

Here, we used the absolute name of the destination directory.

Another alternative is to first enter the command: cd ..and then enter the command: cd Drug\_B

In these two commands, we used local names.

The first one tells bash to go one link up to the parent and the other tells it to go one link down to a child.

A local name is a special case of a relative name**,** that is, a file or folder name is interpreted with respect to the current directory of the program (e.g. bash) with which you are interacting. In general such as name can contain multiple links separate by the **/** character.

Instead of executing the two **cd** commands above, execute a single command:

**cd ../Drug\_B**

Thus, this is yet another alternative to go from Drug\_A to Drug\_B**,** which uses a multi-link name relative to the current directory, with one up link and one down link.

Enter the following command with two up links: cd ../..

We have used a relative name to go to the grand parent of the current directory.

Enter: cd RNA\_SEQ\_SAMPLES/Drug\_B

As we have not preceded this name with **~**, again we have used a multi-link relative name, with two down links.

A relative name, like an absolute name, can contain multiple links separated by the file separator character (**/**) but it cannot start with the file separator. In general, if the absolute name of the current directory is d, the relative name ris a shorthand for the absolute name d/r**.**

Enter: ls –a.

This display shows not only the links to the children of Drug\_B but also the double-period (**..)** link to the parent. In addition, it shows a link displayed as a single period (.). This is a self link that points to the current directory.

Enter: cd ./../Drug\_A

This now takes us to Drug\_B. The self-link is redundant in most cases

## Creating Directories

Change the directory to the Bash subdirectory of the home directory. Execute the command: mkdir misc

List the Bash directory to confirm that a new child of the home directory has been created.

Change your current directory to misc.

Execute: mkdir misc\_child

Confirm child has been created in misc.

Execute: mkdir misc\_child/misc\_child\_child

Confirm that misc\_child\_child has been created in misc\_child.

Create the following directories:

1. In misc\_child\_child, create a new directory named misc\_child\_child\_child.
2. In the Bash directory in your home directory, create sub-directories named UncompressedSamples, FilteredSortedOutput, TSVFiles, PermissionsTest, and bin.
3. In RNASeqOutput in Bash, create a subdirectory called Advanced**.**
4. In PermissionsTest create a subdirectory called PermissionsChild.

When the local names of directories are unique, we will use short names instead of their long absolute names to refer to them. Thus, when we use the name **Bash**, we mean the **Bash** subdirectory in the home directory.

## Creating and Editing a Text File

Go to the misc directory in your Bash directory in the home directory, that is, make misc the current directory.

Enter: pico hello.txt

The entire bash window will now be taken over by the pico editor.

Enter the lines helloandworld**,** using the keyboard arrows to move the cursors.

Enter CTRL+X to exit.

Type y to confirm you want to exit, and then press Enter to return to bash. *Be sure to press Enter at the end!*

Create in misc another file called goobye.txt with the lines goodbyebeautiful andworld.

This is not a lesson on nano**.** If you have not used nano before, we recommend you use it only to delete existing text in a file and paste text you have copied from a text editor on your local computer. For example, the hello.txt file in the cloud could have been edited by using your favorite editor on your local computer to create the text, copying it, and then using CTRL+V to paste it into the cloud file.

This lesson teaches three aspects of nano:

1. How to use nano to edit file f (execute: nano f)
2. How to delete text in nano (use the arrow keys and Delete and Backspace keys)
3. How to insert test in nano (use arrow keys and CTRL+V)
4. How to exit nano: CTRL+X, Y, Enter

## (cat) Displaying Files

Make sure misc is still your current directory.

Enter the command: cat hello.txt

You should see the contents of hello.txt.

$ cat hello.txt

hello

world

Thecatcommand displays the contents of a file. It can also be used to concatenate the contents of files – hence the name.

Enter: cat hello.txt goodbye.txt

You should see a concatenation of contents of both files in the bash window.

$ cat hello.txt goodbye.txt

hello

world

goodbye beautiful

world

## Standard Input/Output (I/O)

Unlike applications executed by Discovery, applications executed by bash can not only store results in one or more files but also show output in the console/terminal window after the command line that launches them. This information is called the **standard output** of the application. This, the concatenated output displayed by our last command forms its standard output.

The standard output of an application is displayed as part of the bash output in the bash window, so it seems as if bash is the one that produced the response.

Enter the cat command without any input file argument: cat

The cursor will start blinking below the command-line to indicate that the launched application is expecting input.

Enter the following input:

abc

def

CTRL+D

The CTL+ D character is used to signal the end of the input.

As we see here, an application can not only process information provided in input files identified as command arguments, but also receive input interactively in the bash console/terminal window. This information is called the standard input of the application.

As the bash window contains both standard input and output, it is also called the console and terminal.

Previous executions of cat displayed the contents of the files whose names were passed to it as arguments. This invocation, without any file argument, instead displays its standard input, which is the sequence of two lines we input in the console. Thus, each input line entered will be reproduced as an output line.

$ cat

abc

abc

def

def

An application can have multiple input and output files passed to it as arguments, but has at most one standard input and one standard output. Whether an application processes’s standard input or output may depend on the arguments passed to the application. For example, **cat** processes standard input only if no file names are passed as arguments to it. An application, of course, may process no file or standard input, and no file or standard output.

## Output Redirection

Make sure misc is your current directory for all of these commands.

Enter: ls > ls.out

No console output follows the command entry.

$ ls >ls.out

$

The> directive tells bash to redirectthe standard output of the preceding command (displayed in the console)to the file, ls\_out.txt**,** that follows the directive.

Confirm that this file exists and display its contents.

Enter: cat hello.txt goodbye.txt > hello\_goodbye.txt

As the output has again been redirected, it is not displayed in the console (bash window), but instead stored in the file, hello\_goodbye.txt**.**

Display the concatenated file, hello.txtgoodbye.txt, and verify it has the four lines of the concatenated contents.

## Input Redirection

Again, make sure misc is your current directory.

Execute: cat < hello.txt

Even though cat is not given a file argument, you will not be prompted for user input. This time the standard input is redirected and is taken from **hello.txt** rather than the console. Thus, the redirected standard input is displayed as standard output.

$ cat < hello.txt

hello

world

Execute**:**

cat > cat.out

This is cat input

More cat input

CTRL+D

As no input file argument has been given, and input has not been redirected, **cat** prompts the user for input. The two input lines are now stored in the file **cat.out** instead of being displayed in the console**.** Verify this is the case.

## I/O Redirection vs. File Arguments

Input and output redirection seems to blur the distinction between file and standard I/O (input/output). There are several differences.

There is one standard input and output, and it may be redirected to a file. This redirection is setup using bash and done by the operating system, and not the code that implements the operation. The arguments of the operation do not include the input and output redirection symbols (<, >) and the tokens that follow them. For instance, **>** and hello\_goodbye.txt is not an argument to the command cat hello.txt goodbye.txt > hello\_goodbye.txt**.** It is processed by bash, which sets up the input/output redirection oblivious to the code implementing cat. When the operation code reads from standard input or writes to standard output, it does not know whether these are the terminal or whether they have been redirected. It can use special mechanisms to determine if such redirection has occurred but typically does not. As we will see later, the standard input and output of an operation can also be redirected to another process through pipes!

An operation can take input from and store output to an arbitrary number of files specified as arguments. The I/O involving file arguments cannot be redirected to the terminal or another process, and the operation implementation must be aware of the names of the input and output files to read and write them respectively. bash does not know how many such arguments the operations the command is expected to take or whether they are input files, output files, or options such as -l. It is the implementation of the operation that is aware of these aspects of the command and decides, based on the arguments supplied, whether it should read standard input and write to standard output.

## echo

Make sure misc is your current directory.

Enter: echo “hello world”

The output will be the single argument passed to echo.

$ echo “hello world”

“hello world”

Enter: echo 1 2

The output will be the two arguments separated by a single space.

$ echo 1 2

1 2

In general, echo displays its argument list in standard output, using a single space to separate each pair of adjacent arguments.

echo, with output redirection, is a useful alternative to an editor to create a text filebecause its standard output can be redirected.

Enter: echo “hello world” > misc\_child/echo.out

Verify that Bash/misc/echo\_output.txt hasbeen created and has the contents, hello world.

## The >> Append directive\*

Execute this time: echo “goodbye world” > misc\_child/echo\_output.txt

Should goodbye world be appended to misc\_child/ echo\_output.txt or should it replace the existing text?

Verify that, in fact, the text is replaced, that is, hello world in the file has been replaced with goodbye world.

When output is redirected with> to a file in a command, the current contents of the file are deleted before feeding the output of the command to the file.

The **>>** directive can be used to append rather than replace redirected standard output to a file.

echo “hello world” >> misc\_child/echo\_output.txt

Verify that the argument to echo is now appended to the file and the file now has hello world appended to goodbye world**.**

## Star Expansion\*

Suppose we wanted to concatenate all files in the misc directory. We can of course list all files in the cat command:

cat hello.txt goodbye.txt hello\_goodbye.txt ls.out cat

It would be nice if we could provide the directory as an argument, but cat ignores a directory argument.

Fortunately, bash allows the name of a directory to be followed by a shorthand, the star symbol (\*), which stands for all children of the named directory.

Enter: cat ~/Bash/misc/\*

The output will show that this command is equivalent to the above cat command.

Enter: cat \*

The output will show that this command is also equivalent to the above two cat commands. If no directory appears before the star symbol, then the current directory is assumed.

Enter: cat \*.txt

Only the files whose names end with the .txt suffix will be displayed. Thus cat.out will not be displayed.

Execute: cat hello\*

Only the files whose names begin with hello will be displayed. Thus, goodbye.txt will not be displayed but hello.echo will be displayed.

Execute: cat hello\*.txt

Only the files whose names begin with hello and end with .txt will be displayed.

In general \* can be preceded or followed with any prefix and/or suffix, and only files in whose names have that prefix and/or suffix will be provided as arguments to the application executed.

Star expansion is done by bash, and thus applies to all operations invoked from bash. The operation is unaware that \* was used. It just sees the expanded names.

Execute:ls –l ~/\*.log

Since \*is preceded with a name of the home directory this time, ls gets the list of all file names in the home directory ending with the .log suffix. This list does not include the names of subdirectories of the home directory, nor the names of other files in the home directories. ls then shows the properties of each of the named files.

echo is particularly useful to understand how \* expansion works.

Enter**:** echo \*

The standard output will show that bash expands the \* argument to an argument list that contains the names of all children in the current directory. echo then simply echoes or displays the expanded argument list, and does not even know it was passed file names.

Enter: echo ~/\*.txt

Since **\*** is preceded with a name of the home directory, the standard output shows that \*.txt is expanded to the names of all children of the home directory whose names end with .txt.

## Copying File to File and Directory

Make sure misc is still the current directory.

Execute: cp goodbye.txt copied\_ goodbye.txt

Confirm that copied\_ goodbye.txt now exists in current directory and has a copy of the contents of goodbye.txt**.**

When cp is called with two file names, the command copies the contents of the file whose name is specified as the first argument to a file whose name is the second argument.

Execute: cp goodbye.txt misc\_child/copied\_goodbye.txt

Confirm that the copy of goodbye.txtis created in the subdirectory, misc\_childand has the name misc\_child/copied\_goodbye.txt

Execute**:** cp goodbye.txt misc\_child

Confirm that a new copy named goodbye.txt exists in the misc\_child subdirectory of the misc directory. When cpis called with a names of an existing fileand a directory, it creates a copy of the file named by the first argument in the directory specified by the second argument whose local name is the same as the local name of the copied file.

## Moving/Renaming Files

Again, change the bash directory to ~/Bash/misc if this is not the current directory.

The mv command is like cp except the file specified by the first argument is moved or renamed rather than copied.

Execute: mv goodbye.txt renamed\_goodbye.txt

Confirm that the file goodbye.txt is now renamed to renamed\_goodbye.txt.

Execute: mv goodbye.txt misc\_child

Confirm that the file goodbye.txt has moved from the misc directory to the misc\_child subdirectory. Thus, as with cp, what mv does depends on its arguments. When mv is called with two file names, it changes the name of file specified in the first argument to the name specified in the second argument. When mv is called with a file name and a directory name argument, it moves the file to the directory, keeping the local name. Each directory keeps a pointer or link to its children. The local name of a file is simply the name of the link.

## Linking Files

Again, change the bash directory to ~/Bash/misc if this is not the current directory.

The ln command can be used to create another link/name to an existing file.

Execute:

ln –s hello.txt linked \_hello.txt

cat linked\_hello.txt

As this interaction shows, we have created another name or link or shortcut to the file linked\_hello.txt**.**

Display a long listing of the properties of this new link and you should see that: the first character of the listing is the character **l**, for link, and (b) the name of this new child of misc directory shows not only its name but also the file to which it points.

$ ls -l linked\_hello.txt

lrwxrwxrwx 1 jovyan 1000 11 Apr 16 22:44 linked\_hello.txt -> hello.txt

The Windows operating systems use the term *shortcut* for a link. In fact, creating a link in the home directory to a file in another directory is like creating a shortcut on the desktop for a file in another directory. It is possible to create a link to a file in a different directory.

Change your directory to misc\_child and create in it a link called hello.txt to the file with the same name in its parent directory:

$ cd misc\_child

$ ln -s ../hello.txt hello.txt

$ cat hello.txt

$ ls -l hello.txt

lrwxrwxrwx 1 jovyan 1000 12 Apr 17 20:29 hello.txt -> ../hello.txt

Now the short cut hello.txt can be used to refer to the file with the same name in its parent directory.

There are two versions of ln, one that takes the parameter –s and one that does not. The former creates a “hard link” while the latter creates a “soft link”. A soft link is more flexible, and works for directories, and thus we have focused on it.

What if you rename hello.txt to renamed\_hello.txt? Will linked\_hello.txt now point to renamed\_hello.txt**?** Because linked\_hello.txt is soft link, the answer is no. If you now create another file called hello.txt**,** linked\_hello.txtwill now point to this new file**.**

## Comparing Files Using diff\*

Make misc the current directory.

Use pico to edit goodbye.txt.Change beautiful to wonderful.

At this point, goodbye.txt and link\_goodbye.txt should both point to the new contents.

On the other hand copied\_goodbye.txt should point to the old text.

Display these files to confirm this is the case.

To do so, you have to manually look for differences in the compared files. We can use the **diff** application to do so automatically.

Execute: diff goodbye.txt copied\_goodbye.txt

This command asks diff to tell us what lines must be inserted into, deleted from, or changed in the contents of the first file to create the contents of the second file.

$ diff goodbye.txt copied\_goodbye.txt

1c1

< goodbye wonderful

---

> goodbye beautiful

The output indicates that line 1 of the first file (goodbye.txt), goodbye wonderful, should be replaced with goodbye beautiful to create the second file.

If we reverse the arguments of diff we get:

$ diff copied\_goodbye.txt goodbye.txt

1c1

< goodbye beautiful

---

> goodbye wonderful

The output now indicates that line 1 of the first file (copied\_goodbye.txt), goodbyebeautiful, should be replaced with goodbye wonderful to create the second file (goodbye.txt).

Execute: diff goodbye.txt linked\_goodbye.txt

There should be no output, telling us that no lines should be inserted or deleted from the contents to which the first name points to create the contents to which the second name points.

$ diff goodbye.txt linked\_goodbye.txt

$

Operations such as **diff** that show the difference between files play a very important role in computing. Such an operation is built-into Word so that we can see the difference between different versions of a document created by the same user or different users. For those familiar with GitHub, it is built into that system. When we commit or check-in a new version of some directory, GitHub does not store the entire new version. Instead, it stores the differences between the new and previous version to save space.

## Copying Moving, Linking Directories\*

Again, all commands illustrated here assume miscis the current directory.

Execute: cp misc\_child copied\_misc\_child

The response will be that you need to use the –r parameter when the first file argument passed to cp is an existing directory.

Execute in misc: cp -r misc\_child copied misc\_child

Verify that copied\_misc\_childhas been created in misc.

Verify also that this new directory has the descendants of misc, that is, has the child and grandchild misc\_child\_child and misc\_child\_child\_child, respectively.

Execute in misc: mv misc\_child renamed\_misc\_child

As the first argument of mv is an existing directory and the second argument is a new name, the command changes the name of the existing directory to the new name. The previous name or link is removed.

Confirm misc\_child has been renamed to renamed\_misc\_child.

Execute in misc: ln –s renamed\_misc\_child linked\_misc\_child

This command creates the name, linked\_misc\_childto whichrenamed\_misc\_childrefers**.**

Create a new child, misc\_child\_another\_child in renamed\_misc\_child.

Do you expect the name to appear in the listing of renamed\_misc\_child, copied\_misc\_child, and shortcut\_misc\_child?

List these directories to confirm your expectation.

## Removing a File/Directory

We are interested here in how files and directories *can* be deleted, not in really deleting any of our work. The –i (interactive) option will make sure you are asked if you want to really remove the file node. Choose n as the answer so the node is not removed. Always use this option even if you really want to delete the node. It is dangerous to remove files/directories without the –i option as the deleted files do not go to a recycling bin when deleted using bash.

Execute in misc: rm –i hello.txt

You will be asked if you want to delete the specified file. The answer of course is no.

Execute in misc: rm –i renamed\_misc\_child

Like cp, rm asks you to use the –r option to delete a directory.

Execute in misc: rm –i –r copied\_misc

This time you will be asked if you want to delete the directory. Again, the answer is of course is no.

If you remove a file or directory, then any link to it created using the –s option becomes invalid unless you create another file or directory with the same name.

## Recursion and –R/-r option\*

Many applications provide the –r and and/or -R parameter, also called the recursive option. We have seen three such applications, ls, cp, and rm. This parameter stands for recursive execution of the operation. What this means can be explained considering how the ls application works.

When invoked without the recursive option on directory d, ls invokes itself only on d. When invoked with this option on d(ls –R d), ls first invokes itself (ls) on d, and then invokes itself with the recursive option (ls –R) on each child directory of ls, which in turn first invokes ls on itself, and then invokes **(**ls – R) on each of its child directories, and so on. Thus, we can summarize the ls –R algorithm as:

ls –R d:

ls d;

for each child c of d do

ls –R c

In general, an operation can execute other operations to perform the task it implements. When the operation calls itself, then we say the operation uses *recursion*. Thus, the recursive option of an operation asks it to use recursion.

Recursion is a concept that goes beyond bash. If any operation called with some arguments called itself with the same arguments, then the operation will never terminate. Such recursion is called ***infinite recursion***. We would have to use CTRL+C to terminate it!

Usually, it will call itself with different arguments that *reduce* the problem. In bash, the recursive operations call themselves on child directories – recursively listing/copying/removing a child directory is a smaller problem than recursively listing/copying/removing the parent directory.

ls provides a single parameter to specify the recursive option, -R. cp and rm provide both the –R and –r parameters to specify this option. It is convention to provide one or both of these parameters in bash operations that can be executed recursively on directories.

It does not, of course, make sense to invoke a bash operation on a file recursively, as it has no children. It does makes sense to execute some directory operations *not recursively*. Consider mv and ln. mv renames the link only to the specified directory, and its descendants “move” with it. Similarly, ln creates an additional link only to the specified directory, which can be used to refer also to its descendants.

## Execution Path

We have seen above bash’s role as a file manager based on the command line. Let us next study its abilities to provide the functions an application manager of the kind we saw in Discovery. In fact, in many ways, it goes far beyond the Discovery application manager.

In any directory, execute: /bin/ls RNA\_SEQ\_SAMPLES

Now execute: /usr/bin/which ls

As we see above, we can enter the full/absolute name of the file containing the application code to execute it.

If bash does not have built-in knowledge of ls, which and other applications, how did it find the files /usr/bin/which and /bin/ls when we typed the short names of these files, which and bin?

Execute: echo $PATH

PATH is the name of an *environment variable***.** The termvariable indicates it that it is associated with a value. The term environment indicates it is a name visible to all applications. Usually the name consists of only uppercase letters. The value of a variable is denoted by prefixing its name with the symbol$**.** Thus $PATH is the value of the variable PATH.

This variable stores a list of directories including /usr/binand/bin**,** called the path**.**

$echo $PATH

opt/conda/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/bin/TDM-GCC-6

When an application file is specified by its short name, the path is looked up for a file with that name. If more than directory has a file with that name, the directory that appears earlier in the path is used.

If you have used java earlier, you might have seen the analogous concept of *classpath*, which is a list of directories that java searches for external classes your program accesses.

Execute: echo PATH

echo simply prints its argument, as it normally does.

$ echo PATH

PATH

When bash encounters a token preceded with the $ symbols, it treats the token as an environment variable, looks up its value, and replaces the token with its value. Thus, when we typed: echo $PATH

bash replaced $PATH with the looked up value and supplied that value to echo**.**

Execute: echo $HOME

$ echo $HOME

/home/jovyan

The HOME environment variable holds our home directory.

## Exporting Application Directories

Execute the following command to indicate that the **bin** directory in our Bash directory should be searched for applications: export PATH="$PATH:$HOME/Bash/bin"

Confirm that the path has been changed.

The bin directory in our Bash directory will store the executable files of the composite applications we will create. It is convention in many systems to store executable files in directories named bin as these files normally contain the binaryrepresentation of code. The composite applications we create will actually contain textual bash commands, but we fill follow this conventions. In the Windows operating systems, bin corresponds to the Program Files directory.

## Linking Discovery TSV files in Bash\*

Change your directory to the TSVFiles subdirectory of the Bash directory in your home directory.

Without changing directory, use ls to list the four tsv file descendants of the TSVFiles subdirectory in the Discovery subdirectory in your home directory by executing lsone or more times**:**

A\_1\_abundance.tsv

A\_2\_abundance.tsv

B\_1\_abundance.tsv

B\_2\_abundance.tsv

In the TSVFiles subdirectory of the Bash directory, create four links to these files with the same local names:

612c4dc4-ce35-4571-952d-cfedc92809d3-5fbf7c6db6-nzw89:TSVFiles jovyan$ ls -l

total 0

lrwxrwxrwx 1 jovyan 1000 49 Apr 17 21:53 A\_1\_abundance.tsv -> /home/jovyan/Discovery/TSVFiles/A\_1\_abundance.tsv

lrwxrwxrwx 1 jovyan 1000 49 Apr 17 21:54 A\_2\_abundance.tsv -> /home/jovyan/Discovery/TSVFiles/A\_2\_abundance.tsv

lrwxrwxrwx 1 jovyan 1000 49 Apr 17 21:55 A\_3\_abundance.tsv -> /home/jovyan/Discovery/TSVFiles/A\_3\_abundance.tsv

lrwxrwxrwx 1 jovyan 1000 49 Apr 17 21:55 A\_4\_abundance.tsv -> /home/jovyan/Discovery/TSVFiles/A\_4\_abundance.tsv

The fourln commands will be very similar to each other, so make use of tab completion, editing of previous commands, and copying and pasting to reduce your effort and mistakes.

## Cancelling/Stopping a Process: CTRL+C

In the TSVFiles subdirectory of **Discovery**, execute: cat A\_1\_abundance.tsv

The entire large file will be displayed by bash. We do not need to actually see the fileso stop this time-consuming process by typingCTRL + C.

This, CTRL + C can not only be used to copy text but also stop a process. When a process is running, it is used to stop the process. Otherwise, it is used to copy text.

## Head and Tail

In Discovery/TSVFiles, enter: head -20 A\_1\_abundance.tsv to see only the first 20 lines of the input file.

Enter: tail -20 A\_1\_abundance.tsvto see the last 20 lines of the input file.

The head and tail commands are particularly relevant in data science, which involves input and output of large data files.

## Discovery Launch vs Bash Command Line

In TSVFiles subdirectory of Discovery execute:

grep --ignore-case enst00000391< A\_1\_abundance.tsv > ~/Bash/FilteredSortedOutput/Greped.tsv

Without changing the directory, confirm that the file Greped.tsv has been created in the directory ~/Bash/FilteredSortedOutput**.**

Display its contents. It should have only those lines that contain letter sequence enst00000391**.**

The following table explains what this command does.

|  |  |  |  |
| --- | --- | --- | --- |
| Application Name | Standard Input | Standard Output | Parameters |
| grep | The tsv file produced for the first sample of Drug\_A | Greped.tsv file in  FilteredSortedOutput subdirectory of the Bashfolder in the home directory | enst00000391  --ignore-case |

It is similar to the analogous table we created for the Discovery grep.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Application Name | Analysis Name | Input file | Output Folder | Output file | Parameters |
| Grep 3.1-2 | Chosen by you | The tsv file produced for the first sample of Drug A | FilteredSortedOutput subdirectory of the Discovery folder in the home directory | Greped.tsv | Search string:  enst00000391  Case insensitive match: **Yes** |

The differences are minor and mostly syntactic. The names of the two applications are different,

There is no notion of user-defined analysis or process name in bash.

The input file is redirected standard input in bash, but an argument input file in Discovery. Similarly, the output file is redirected output file in bash. In Discovery, two different file arguments are given to specify a file, the directory in which the file is located and the local name of the file in the argument.

A bigger difference is in how non-file arguments are specified. Consistent with the terminology we used in the Discovery tutorial, we will use the term *parameters* for them.

Consider first Boolean parameters - those that like the grep case -match take True/False or Yes/No values. Discovery shows the default value of the parameter through a checkbox, which we change to override the value. A Boolean parameter is called a *flag*in bash. The absence of its name in the command line indicates the false value and its presence indicates the true value. Thus, the presence of –ignore-case in the bash command above says that is default value of false should be overridden with true.

Specification of bash non-Boolean parameters require the parameter name followed by its value as we will see when we execute the **sort** command and specify the column value.

grep --ignore-case enst00000391< A\_1\_abundance.tsv > ~/Bash/FilteredSortedOutput/Greped.tsv

Its syntax can be described as follows:

Operation-name optional-file-arguments optional-parameters < input file > output-file

This is a common (but not required) syntax for bash commands, and one we will use in the exercises below. Once we know this syntax, the name of the application we need to execute, and the names of its parameters, we can execute any command that runs the application and follows this syntax.

## Operation Description: -- help and –h

How does one know the parameters a bash operation takes? Discovery prompts us through form labels for the various components of the application launch (input, output directories, parameter names) whereas bash does not, which can specially be an issue when entering parameters names**.** On the other hand, most applications provide the –help option to print the parameter names and other information about an operation.

Enter: grep --help and ls --help

You should see a description of the operations – in particular all arguments of the operation.

Enter: fastx\_trimmer --help

You should see an error message saying use –h instead.

Enter: fastx\_trimmer -h

Again you should see a description of the operation.

Most operations take a --help and/or --h parameter for describing their behavior.

Consider again the grep command:

## Running Sort

Now that we understand in some depth how to run in bash an operation and find and display its output, let us further practice this skill by running the sort application to sort the file produced by the grep program.

Here is the application-specific information we need to execute this second step in the grep-sort workflow.

|  |  |  |  |
| --- | --- | --- | --- |
| Application Name | Standard Input | Standard Output | Parameters |
| sort | The tsv file output by grep | Sorted.tsv file in FilteredSortedOutput in the Bash subdirectory | --key=4  --numeric-sort |

*Table 2: Specification for the Bash sort command*

As we see here, a non-Boolean parameter (key**)** requires a value (4) to be specified after the parameter name. The --numeric-sort Boolean parameter asks for a numeric sort; on the specified column/key.

Compose and execute the bash command line from this table, just as you composed the launch contents from the analogous Discovery table.

You can execute the command from any directory including the home directory, the directory containing the input file, or the directory containing the output file. Remember you can use tab completion and copying and pasting to reduce effort and errors.

Confirm that the output file exists and view its contents.

## Pipes

Recall that in Discovery we created composite applications consisting of a sequence of composed applications, in which each composed applications was fed the output of the preceding application. In bash, such composition is performed using the |symbol, called *pipe*, which “pipes” the output of the application on its left as input to the application on the right.

In the FilteredSortedOutput subdirectory in the Bash folder, enter the following piping command-line:

grep --ignore-case enst00000391< ~/Discovery/TSVFiles/A\_1\_abundance.tsv | head -50 > GrepedFirst50.tsv

Here, the operation on the left of the pipe is grep and the one on the right is head.

We have specified the standard input of the application on the left of the pipe (grep) and the standard output of the application on the right of the pipe (head), (Both of these are files rather than the console)

We have not specified the standard output of the first operation or the standard input of the second application. These are not the console. Because of the pipe symbol between the two operations, the standard output of the first operation is automatically fed or “piped” as input to the second application.

Thus, the grep application sends the filtered output of grep as input to head, which stores the first 50 lines of its input in the output file GrepedFirst50.tsv. After receiving the first 50 lines, the head process terminates and thus breaks the pipe connection to grep**,** which terminates asits remaining output will not be consumed.

This is the reason you may see the message: write error: Broken pipe

Such broken connection and early termination is exactly what we want, notwithstanding this error message, which not all shell implementations give. Confirm that the command worked by looking at the output file GrepedFirst50.tsv.

As in Discovery, it is possible to string together an arbitrary number of applications.

Enter now the following piping command-line:

grep --ignore-case enst00000391< ~/Discovery/TSVFiles/A\_1\_abundance.tsv | head -50 | tail -25 > GrepedSecond25.tsv

This command is like the previous one except for the following. Instead of storing the output of head in a file, it feeds it as input to tail, which stores the last 25 lines of head’s output (which is the second 25 lines of the grep’s output) in its output file (GrepedSecond25.tsv).

Thus, standard output of the operation can go to the console or be redirected to a file or the standard output of a succeeding operation. Similarly, the standard input of an operation can come from the console or be redirected to a file or the standard output of a preceding piped operation.

Exercise: Previously, you first ran one command to run the grep operation, whose standard output was stored in a file. You then ran another command to run the sort operation, redirecting its standard output to create an output file. You then

Pipe the output of the grep application to the sort application that has the parameters and output file specified in the table above.

**Exercise**: Execute a piped command involving ls and grepthat shows all descendants of the home directory that have the suffix .tsvand store the output in a file called TSVFileNames.txt in the TSVFiles subdirectory in Bash**.** Executing the cat command on the output file should create a display of the kind shown below,

$ cat ~/Bash/TSVFiles/TSVFileNames.txt

GrepedFirst50.tsv

GrepedSecond25.tsv

Greped.tsv

A\_1\_abundance.tsv

A\_2\_abundance.tsv

B\_1\_abundance.tsv

…

## Pipes and Unix Single-Mindedness\*

The notion of pipes, along with other concepts we see here such as I/O redirection, was invented in the Unix operating system – bash simply provides a user-interface to use it and other Unix features. It is consistent with the Unix name and philosophy that (1) each application should focus on *one* task – for example ls on listing a directory, grep on searching text, sort on sorting text, history on displaying the command history – and (2) the functions of these single-minded, specializing, text-based applications should be combined through pipes, which allow the text-based composed applications to be unaware of each other.

The notion of independent plugins such as the Chrome plugin required for this lesson is consistent extends this philosophy to non-text data.

## Command Files and Composite Applications

Recall that in Discovery, we not only connected the output and input of applications, but also stored such connections as new composite applications, which could be invoked like other applications. Pipes, on their own, allow us only to perform the first step. For the second step, we need to store commands in **command files**. As we have exported the **bin** subdirectory in the home directory, in this exercise, create all command files in this directory so that it can be automatically found by bash. We will refer to this directory to simply as **your bin** directory.

Change your current directory to your bin subdirectory.

In this directory**,** usepicoto create a file called grep\_head\_tailwith the following contents:

grep --ignore-case enst00000391 | head -50 | tail -25

This is like the piped command we saw earlier:

grep --ignore-case enst00000391< ~/Bash/TSVFiles/A\_1\_abundance.tsv | head -50 | tail -25 > GrepedSecond25.tsv

The difference is that we have omitted the (a) input file that forms the standard output of the first piped application and (b) the file that forms the stanrard output of the last application. By doing so, we are telling bash that these two files will be specified by the user when the composite command is executed.

Like other applications, a bash composite application containing a piped command may be associated with standard input and output. The standard input of the composite application becomes the standard input of the first piped application and the standard output of the composite application becomes the standard output of the last piped application.

## Executable Files and Permission String

In the bin directory, execute this new application without an argument: grep\_head\_tail

The response will tell you that the permission to execute this file is denied.

$ grep\_head\_tail

grep\_head\_tail: command not found

It seems you have an error. Enter 'shelp' for any suggestions

Execute:which grep\_head\_tail

No output will be shown even though our bin directory is exported.

$ which grep\_head\_tail

$

To understand why and fix these two related problems, display a long listing of grep\_head\_tail.

-rw-r--r-- 1 jovyan 1000 55 Apr 18 14:48 grep\_head\_tail

Recall that the first character of a long listing of a file-system node indicates its type (file, directory link).

The other nine characters, forming the *permission string*, are Unix permissions associated with the node, with each permission denoted by a single character. This string, discussed in more depth later, shows the characters, r, w, and **-**. There is no x**,** which stands for the e*X*ecute permission, in this string.

To allow the command file to become an executable application, execute the following command: chmod a+x grep\_head\_tail

The parameter a+xasks chmod to make the argument file (grep\_head\_tail**)** eXecutablefor *a*ll users.

Display again the long listing of the file – it should show three new occurrences of the execute permission.

$ ls -l grep\_head\_tail

-rwxr-xr-x 1 jovyan 1000 55 Apr 18 20:51 grep\_head\_tail

In any directory, execute the which command again on this now executable file. The response this time should be more promising and give the full name of the application.

$which grep\_head\_tail

/home/jovyan/Bash/bin/grep\_head\_tail

Make sure you use the chmod command for all command files created in this lesson. Otherwise, you will get the permission is denied message when you use the file as an application name.

Return to the ~/Discovery/TSVFiles directory, and execute:

$ grep\_head\_tail < A\_1\_abundance.tsv > ~/Bash/FilteredSortedOutput/Second25\_1\_GrepedSecond.tsv

Here we have specified the standard input and output of the new composite application as A\_1\_abundance.tsvand ~/Bash/FilteredSortedOutput/Second25\_1\_GrepedSecond.tsvrespectively.

bash responds to this line by executing the operation implementation stored in Bash/bin/grep\_head\_tail.

It redirects the standard input of the first piped application, grep, to A\_1\_abundance.tsvand redirects the standard output of the last piped application, tail, to ~/Bash/FilteredSortedOutput/Second25\_1\_GrepedSecond.tsv

Exercise:

Execute grep\_head\_tailwith the standard input redirected to ~/Discovery/TSVFiles/A\_2\_abundance.tsvand standard output redirected to Second25\_2\_GrepedSecond.tsv**.**

By storing an application composition in a command file, we do not have to redo the work of connecting the applications for each input file. In the example above, we simply name the composite application and input file, and are in fact, unaware that the composition occurred. This is hidden in the composite application command file.

The above example illustrates the two main steps in defining a new command/executable file named C.

1. In an *exported directory* (such as our bin directory), create a file named C. If you do not perform this step, bash will say that the command is not found, when you type a command line starting with C.
2. Use the chmod file to make it executable. If you do not perform this step, bash will say that permission to execute the command is denied when you type a command line starting with C.

## Looping\*

In the example above, using **grep\_head\_tail**, we executed the command for each file in the TSVFiles directory.

$ grep\_head\_tail < A\_1\_abundance.tsv > ~/Bash/FilteredSortedOutput/Second25\_1\_GrepedSecond.tsv

$ grep\_head\_tail < A\_2\_abundance.tsv > ~/Bash/FilteredSortedOutput /Second25\_2\_GrepedSecond.tsv

If the number of input files in the TSVFiles directory is large, entering a command for each file can be cumbersome.

In ~/Bash/bin create an executable file called directory\_grep\_head\_tailwith the following contents:

cd $1

**for** tsv\_file **in** A\_1\_abundance.tsv A\_2\_abundance.tsv;

**do**

grep\_head\_tail < $tsv\_file > $2/Second25\_$tsv\_file;

cat $2/Second25\_$tsv\_file;

**done**

This command defines a composite application that takes two arguments, an input ($1) and an output directory ($2). It executes grep\_head\_tail on selected files in the input directory A\_1\_abundance.tsv A\_2\_abundance.tsv) and creates an output file in the output directory ($2) whose name is derived from the name of the input file ($tsv\_file)/

Now execute:

directory\_grep\_head\_tail ~/Discovery/TSVFiles ~/Bash/FilteredSortedOutput

This is equivalent to executing:

cd ~/Discovery/TSVFiles

grep\_head\_tail < A\_1\_abundance.tsv > ~/Bash/FilteredSortedOutput /Second25\_ A\_1\_abundance.tsv;

cat ~/Bash/FilteredSortedOutput /Second25\_ A\_1\_abundance.tsv;

grep\_head\_tail < A\_2\_abundance.tsv > ~/Bash/FilteredSortedOutput /Second25\_ A\_2\_abundance.tsv;

cat ~/Bash/FilteredSortedOutput /Second25\_ A\_2\_abundance.tsv;

Confirm that the command runs the command grep\_head\_tail on each tsv file in the input directory, stores the output of the command in the output directory, and displays the output.

$ ls ~/Bash/FilteredSortedOutput/Second25\_1\_GrepedSecond.tsv

Second25\_2\_GrepedSecond.tsv

Second25\_A\_1\_abundance.tsv

Second25\_A\_2\_abundance.tsv

Now that we have seen what the command file does, let us understand the lines in it. These are relatively complicated lines, with many new features, so let us understand them in stages.

## Variables and References\*

Let us focus first on the tokens beginning with the symbol $: $1, $2 and $tsv\_file. Such tokens are *variable-value references*, and the string following the **$** are names of variables to which they refer. We have seen the concept of variable name and associated value earlier, when we studied the environment variables PATH and HOME. Unlike those variables, the ones in our file are not environment variables. Instead they are *local variables* known only to our composite application.

Thus, $1, $2 and $tsv\_file are (variable-value) references to the (local) variables 1, 2and tsv\_file, respectively.

Variables are so called because they are associated with or assigned values that can vary. Before a file is executed, all local variable references in it are replaced (by bash) with the values of the referenced variables. Thus, the references $1, $2and$tsv\_file are not used literally, these are replaced with values of the variables 1, 2 and tsv\_file, respectively.

## Numbered Argument Variables\*

1and2 are *argument variables*, which is always denoted by numbers.

Consider execution of the command directory\_head\_tail:

directory\_grep\_head\_tail ~/Discovery/TSVFiles ~/Bash/FilteredSortedOutput

Argument variable1in the named command fileis assigned the first argument, ~/Discovery/TSVFiles, and variable **2** is assigned the second argument, ~/Bash/FilteredSortedOutput.

When a command file such as directory\_grep\_head\_tail is executed, argument variable N in it is assigned the Nth argument in the command that executes it.

Thus the lines in the command file:

cd **$1**

**for** tsv\_file **in** A\_1\_abundance.tsv A\_2\_abundance.tsv;

**do**

grep\_head\_tail < $tsv\_file > **$2**/Second25\_$tsv\_file;

cat **$2**/Second25\_$tsv\_file;

**done**

are first expanded to:

cd ~/Discovery/TSVFiles

for tsv\_file in A\_1\_abundance.tsv A\_2\_abundance.tsv;

do

grep\_head\_tail < $tsv\_file > ~/Bash/FilteredSortedOutput/Second25\_$tsv\_file;

cat ~/Bash/FilteredSortedOutput/Second25\_$tsv\_file;

done

The first expanded line above is a familiar command to change directory. Like bash, every application has a working directory, which by default is the same as that of the bash that executed it. This application changes this default directory (without changing the directory of the invoking bash). Thus, the first line changes the current directory to the ~/Discovery/TSVFiles directory.

## Loop Header, Body, Variable, Iteration\*

The remaining lines are a *bash loop***,** which consists of a *loop header* and a *loop body*.

The line between the for and ;tokensform *the loop header*.

**for** tsv\_file **in** A\_1\_abundance.tsv A\_2\_abundance.tsv;

The loop header declares a *loop variable*, tsv\_file, whose name appear between the for and in tokens of the loop header. Only one loop variable can occur in a loop.

Unlike an argument variable such as $1 and $2, a loop variable can be assigned multiple values, which are determined by the *loop range*. A loop range is a sequence of tokens that appears between the in and semicolon tokens. Here, the loop range is A\_1\_abundance.tsvA\_2\_abundance.tsv.

The lines between do and done tokens form the loop body. A loop body contains a sequence of semicolon separated bash (basic or complex) commands.

Let us consider the expanded loop body with the argument variable references:

grep\_head\_tail < **$tsv\_file** > ~/Bash/FilteredSortedOutput/Second25\_**$tsv\_file**;

cat ~/Bash/FilteredSortedOutput/Second25**\_$tsv\_file**;

These are basic commands executing applications, grep\_head\_tail and cat, which reference the tsv\_file loop variable thrice.

The loop assigns the loop variable successive values in the loop range. After each assignment, it executes the loop body. The variable assignment and loop body execution together is called a *loop iteration.*

In the first iteration, the loop assigns to tsv\_files the first value in the loop range: A\_1\_abundance.tsv**.**

It then executes the loop body, with all references to this variable to it replaced with this value, executing the first desired step.

grep\_head\_tail < A\_1\_abundance.tsv > ~/Bash/FilteredSortedOutput/Second25\_ A\_1\_abundance.tsv;

cat ~/Bash/FilteredSortedOutput/Second25\_ A\_1\_abundance.tsv;

In the second iteration it replaces both loop variable references with the second and last range value, A\_2\_abundance.tsv, and executes the second and last desired step.

grep\_head\_tail < A\_2\_abundance.tsv > ~/Bash/FilteredSortedOutput/Second25\_ A\_2\_abundance.tsv;

cat ~/Bash/FilteredSortedOutput/Second25\_ A\_2\_abundance.tsv;

.

The loop terminates when all values in the range have been assigned to the variable.

## Star Expansion in Loop Range\*

A problem with the solution above is that it requires us to list all the files in the directory in the loop header:

**for** tsv\_file **in** A\_1\_abundance.tsv A\_2\_abundance.tsv

To overcome this problem, replace this header with:

**for** tsv\_file **in** \*.tsv;

Based on the discussion of \*expansion we saw earlier, these two versions of the header are equivalent. Make this change and confirm it does not change the behavior.

Exercise**:**

Define a new composite application, called cd\_ls in the bin directory that takes an argument specifying a directory. It changes the current directory to the specified directory and then lists the contents of the directory.

## Natural, Command, and Programming Languages\*

We have described several rules for constraining the tokens in command languages. For example, operation names, operation arguments, and redirection symbols should occur in this order; and for, variable name, in**,** variable range, and a semicolon should occur in this order. Rules constraining acceptable token sequences are called syntactic or grammar rules, and the set of all token sequences allowed by these rules is called a *language*.

Languages we speak such as English are called natural languages. Languages defined by computer programs are called computing languages. A language defined by a line-based interpreter for browsing files and launching applications is called a *command language***.** A language forwriting arbitrary atomic and composite applications is called a *programming language.*

A command and programming language can have very similar features, like the for loop above, but they serve different through related purposes. Perhaps less obvious, other features in the bash command language have counterparts in programming languages. For example, any programming language has as an assignment statement. Depending on the language and the type of variable assigned, this statement corresponds either to execution of the cp command or the ln command.

There has been research in trying to create a single language that subsumes both a command and programming language. However, it has not so far gained widespread use. One reason is that bash is today more or less a standard command language, with the DOS command language being the only other alternative in use. There is, today, no programming language that is accepted as widely. So if the bash command language was to be integrated with a programming language and still be considered an as well-accepted standard, there will be little agreement on which that programming language should be.

## Different Ways of Executing Workflows

We have above seen above three ways of executing a workflow such as the grep-head-tail workflow and the grep-sort workflow:

1. *Individual Application Execution:* We run applications in each workflow individually, by entering a command line for each application in the workflow.
2. *Direct Piped Execution*: We directly enter a piped command.
3. *Executable File*: We store the piped command in a file, make it executable, and run this file as a composite application zero or more times.

The following exercises apply these concepts to the RNA-Seq workflow.

## RNA-Seq: Individual Application Execution

All of the RNA\_Seq applications described in the tables below provide the **–h** parameter to describe other parameters. Use this feature to understand the role of the parameters specified in the tables.

Our first four steps are to uncompresses all four sample files. The following four tables describes the processes we need to create.

|  |  |  |  |
| --- | --- | --- | --- |
| Application Name | Standard Input | Standard Ouput | Parameters |
| gunzip | Drug\_A\_1.fastq.gz in Drug\_A in RNASeqSamples in Bash | Drug\_A\_1.fq in UncompressedSamples in Bash | -k |

|  |  |  |  |
| --- | --- | --- | --- |
| Application Name | Standard Input | Standard Ouput | Parameters |
| gunzip | Drug\_A\_2.fastq.gz in Drug\_A in RNASeqSamples in Bash | Drug\_A\_2.fq in UncompressedSamples in Bash | -k |

|  |  |  |  |
| --- | --- | --- | --- |
| Application Name | Standard Input | Standard Ouput | Parameters |
| gunzip | Drug\_B\_1.fastq.gz in Drug\_B in RNASeqSamples in Bash | Drug\_B\_1.fq in UncompressedSamples in Bash | -k |

|  |  |  |  |
| --- | --- | --- | --- |
| Application Name | Standard Input | Standard Ouput | Parameters |
| gunzip | Drug\_B\_2.fastq.gz in Drug\_B in RNASeqSamples in Bash | Drug\_B\_2.fq in UncompressedSamples in Bash | -k |

For each table, compose a bash command with the specified application name, input and output file names, and parameters. After each command, confirm that the output file is produced and has the expected contents.

Recall that you can execute a command from any directory including the home directory, the directory containing the input file, or the directory containing the output file; and that you can use tab completion and copying and pasting to reduce effort and errors.

Our next step is to trim the output of the first gunzip as follows.

|  |  |  |  |
| --- | --- | --- | --- |
| Application Name | Standard Input | Standard Output | Parameters |
| fastx\_trimmer | Drug\_A\_1.fq in UncompressedSamplesin Bash | Drug\_A\_1.fq.trimmed in RNASeqOutputin Bash | -m 20  -f 1  -Q33 |

The next step is to run the quality checker:

|  |  |  |  |
| --- | --- | --- | --- |
| Application Name | Standard Input | Output File/Directory | Parameters |
| fastq\_quality\_filter | Drug\_A\_1.fq.trimmed in RNASeqOutput in Bash | Drug\_A\_1.fq.filtered in RNASeqOutput in Bash | -q 50  -Q33 |

## RNA-Seq: Piping

Execute the above workflow on the second compressed Drug A sample, in a single piped command. This means you must execute the operations gunzip, fastx\_trimmer, and fastq\_quality\_filter, with parameters, -k, -m 20 –f 1, and –q 20, respectively. The input of the piped command should be Drug\_A\_2.fastq.gz in RNASeqSamples in Bash. The output should now be Drug\_A\_2\_filtered.fq in RNASeqOutputin Bash**.**

## RNA-Seq: Composite Command

Create a composite application, called my\_fastx, implementing the pipe above. The application reads from standard input and writes to standard output. It does not take any file or other arguments. Thus, the application will not refer to argument or other variables,

Execute my\_fastx in a command giving it as standard input, Drug\_B\_1.fastq.gz in Drug\_B in RNASeqSamplesin Bash, and as standard output, the file Drug\_B\_1.fq.filteredin RNASeqOutput in Bash**.**

Execute my\_fastx in another command giving it as standard input, Drug\_B\_2.fastq.gz in Drug\_B in RNASeqSamplesin Bash, and as standard output, the file Drug\_B\_2.fq.filteredin RNASeqOutput in Bash**.**

## RNA-Seq: Loop and Diff\*

Create a composite command, called directory\_trimmer\_filter, that takes as the first and second arguments an input and output directory, respectively. It trims and filters each uncompressed fastq file in the input directory, which is known to end with the suffix, .fq, It writes the output of the filter to the output directory. The name of the file is the same as the fastq file except it as the additional suffix .filteredappended to it.

Run the command as follows:

directory\_trimmer\_filter ~/Bash/UncompressedSamples ~/Bash/RNASequOutput/Advanced

Confirm that the output directory has the expected trimmed and filtered files.

At this point, each uncompressed sample has been filtered twice, once using the loop, and once without the loop. The suffixes of the filtered files indicate whether a loop was used or not.

Use the diff application to verify that the two filtering of each file are the same.

Use also diffto also verify that the looped filtering of the two samples of Drug\_A are different.

## Regular User vs. Super User\*

Let us try to better understand the concepts in the permission string and the chmod command in some depth, invented in the context of the Unix operating system.

As we saw in Discovery, permissions are associated with groups of users. A Unix file system has one administrator user, called *root*or*super user*. This personal virtual computer OS instance has only one non-root, or *regular user*, jovyan. In general, a Unix-based OS instance can have multiple regular users.

The super user, being the administrator, has arbitrary permissions to a file. Regular users, as expected may not have the same permission to each file or directory.

## Owner, Group and Other Users of File/Directory\*

Create a directory, PermissionsTest, in your home directory.

Change your directory to this directory.

Use the echo command to create in this directory file, hello.txt, with the text hello.

You will change the permission string and contents of this file, so before you do that, copy its contents to original\_hello.txt**.**

Display a long listing of hello.txt.

-rw-r--r-- 1 jovyan 1000 6 Apr 21 00:35 hello.txt

Let us try and understand its attributes in more depth.

Unix divides the set of all users who can be assigned permissions to a file into three disjoint user-partitions – the file’s *owner*, the file’s *group*, and *other* users. The file’s owner is also called the file’s **user**.

The first token after the permission string (e.g. jovyan) has a name that identifies the file’s owner, and the token after that (e.g.1000) identifies the file’s group, with a number or a name. The listing says that the owner of hello.txt owner is jovyan, the name assigned to you in this OS instance, and its group is denoted by the number 1000. By default, you are the owner of files and directories in your home directory. By default, each file and directory in your home directory is assigned you as the owner, and a default group as the group. A file system node has exactly one owner.

Display a long listing of the executable file /bin/ls.

$ ls -l /bin/ls

-rwxr-xr-x 1 root root 133792 Jan 18 2018 /bin/ls

Its long listing should show that owner of this executable file is root and its group is also named root.

Execute the application addgroupwithout any operand:

The response says only root can create a group.

$ addgroup

addgroup: Only root may add a user or group to the system.

Thus, we see here the difference between the rights assigned to a regular and super user.

It is possible for root to create arbitrary groups on our behalf and assign them to different file. However, a file cannot be assigned multiple groups, a major drawback of the Unix protection Discovery overcomes this problem by using a protection system that implements a different scheme.

## Owner, Group, and Other Permission Substrings\*

Let us study in more depth the permission string of /bin/lsin its long listing**.**

The permission string is divided into three successive sequences of three, each of which represents an access record of the kind we saw in Discovery. While Discovery supports access lists containing an arbitrary number of groups, Unix supports access lists containing exactly three access records, a limitation that allows the long listing to be compact. The three sequences of three-characters represent access records describing permissions given to the file’s owner, group, and other users respectively.

Thus, in the permission string of /bin/ls**,** rwxr-xr-x, is decomposed as follows:

1. rwx: owner permissions
2. r-x: group permissions
3. r-x: other permissions.

The permission characters r, w, x, and – stand for read, write, execute and no permission, respectively. Thus, the first, second and third characters of each access record/permission sub-string of a file-system node indicates if the associated set of users (owner, group and other) has the read, write or execute access to the file-system node.

We have used the chmod operation earlier to give all users a permission. For example, we used the command chmod a+x grep\_head\_tailto give all users the execute permission to grep\_head\_tail**.** In this command, before the permission qualifier, **+** and  **-,** we used the user abbreviation, a, which stands for all users. Other variations are u, g, and o**,** which stand for owner/user, group, and other**,** respectively**.**

Execute**:** chmod a-r hello.txt

Confirm that the read right is taken from all three user partitions.

--w------- 1 jovyan 1000 6 Apr 21 00:35 hello.txt

Execute: chmod g+r hello.txt

Confirm that the right has been given to the file’s group but not the owner or other users.

--w-r----- 1 jovyan 1000 6 Apr 21 00:35 hello.txt

Execute: chmod o+r hello.txt

Confirm that the read right has been given also to the file’s other users:

--w-r--r-- 1 jovyan 1000 6 Apr 21 00:35 hello.txt

Execute: chmod u+r hello.txt

Confirm that the file’s owner/user also has this right:

-rw-r--r-- 1 jovyan 1000 6 Apr 21 00:35 hello.txt

As we see here, it is possible for

(a) owners to deny themselves certain rights

(b) a user or group to have the write right to a file but not the read right.

An owner can always change the rights of any of the three user partitions.

We have seen how to change permissions. Let us try to understand what they mean exactly.

What these permissions mean depends on the type of the file-system node. Let us first consider files.

## Meaning of Unix File Permissions

Remove from hello.text the owner write permission.

Use the echo command to try and append again the text hellotohello.txt.

As you, the owner, no longer have the write right, you will get a permission denied message.

hello.txt: Permission denied

Display the file to ensure you can still read the file.

Deny yourself the read right to this file.

Verify that the file can no longer be displayed by you.

Give the owner write permission to hello.txt**.**

Use the echo command to try and append again the text hellotohello.txt.

Perhaps surprisingly, the command will succeed in that you get no permission denied message.

Verify that the file can no longer be displayed by you. This means you cannot display the file to validate the file was correctly modified!

This might seem unintuitive – write seems like a stronger right than read. For this reason, many systems do not allow the write right to be given without the read right. But the file appending example shows why this might be useful. A professor might want each student to append an answer to a file without seeing others’ answers. Unfortunately, in Unix, it is not possible to prevent the student from (accidentally or maliciously) overwriting the contents of the file by for instance using the **>** directive instead of **>>**. For this reason, some systems differentiate between the right to append and write.

## Directory Write Permission

Remove from PermissionsTest the user (owner) write permission.

Execute an echo command to create a file named goodbye.txt with the contents goodbye.

You should get a permission denied message.

goodbye.txt: Permission denied

Execute the rm command to remove hello.txt from PermissionsTest.

You should not be able to remove the file from the directory.

rm: 'hello.txt': Permission denied

Use echo to replace the text in hello.txt with “”, the empty string within quotes.

Display hello.txt to confirm indeed the file is empty.

Thus, we see here that you cannot remove the file hello.txt even though you have permission to change the file itself, including making it empty!

Execute the echo command to create a file calledremove\_me.txtin PermissionsTest with the contents remove\_me.

The attempt to create a new child in PermissionsTestshould fail.

remove\_me.txt: Permission denied

Restore the owner/user write right to PermissionsTest**.**

Again, execute the echo command to create a file calledremove\_me.txtin PermissionsTest with the contents remove\_me.

This time it should succeed.

Remove the owner (user) write right to remove\_me.txt.

-r--r--r-- 1 jovyan 1000 10 Apr 21 20:22 remove\_me.txt

Execute the echo command to change the contents of remove\_me.txt to You cannot change me.

The attempt to change the file should fail.

remove\_me.txt: Permission denied

Execute the rm command to remove remove\_me.txt.

Bash will check with you if you really want to remove a write protected file.

rm: remove write-protected regular file 'remove\_me.txt'?

If you way yes, you should be able to remove the fileeven though you cannot change it!

We can deduct from these examples that the write permission to a directory determines if the set of children file-nodes in it can be changed, not if the child nodes themselves can change.

## Directory Read Permission

Change the current directory to Bash.

Display a long listing of the directory PermissionTest**.**

drwxr-sr-x 2 jovyan 1000 74 Apr 21 13:02 PermissionsTest

The directory permission string is similar to the file permission string except it has the **s** group permission, which is beyond the scope of this lesson.

Remove the owner read permission from PermissionsTest.

d-wxr-sr-x 2 jovyan 1000 44 Apr 21 20:24 PermissionsTest

Try to list PermissionsTest.

The operation should fail.

ls: cannot open directory 'PermissionsTest/': Permission denied

Change the current directory to PermissionsTest.

Execute a command to view hello.txt in PermissionsTest**.**

Execute a command append hello to hello.txt.

All of these operations should succeed.

Return to the Bash directory.

Restore the owner read permission in PermissionsTest**.**

drwxr-sr-x 2 jovyan 1000 44 Apr 21 20:24 PermissionsTest

## Directory Execute Permission

Now remove the owner (user) execute permission from PermissionsTest.

drw-r-sr-x 2 jovyan 1000 44 Apr 21 20:24 PermissionsTest

What could this mean?

Execute a command to change the current directory to PermissionsTest**.**

This operation should fail.

cd: PermissionsTest/: Permission denied

Execute a command to list the directory PermissionsTest

The operation should succeed.

Execute a command to view hello.txt in PermissionsTest.

The operation should fail.

cat: PermissionsTest/hello.txt: Permission denied

Execute a command append hello to hello.txt.

The operations should fail.

PermissionsTest/hello.txt: Permission denied

Unintuitively, the execute permission to a directory determines if the current directory can be changed to it, and whether any file can be accessed in it (for reading or writing). It does not determine if you can read its contents, which of course, is determined by the read permission. Without execute permissions, you cannot create a file in it or remove a file in it.

Restore the owner execute permission in PermissionsTest**.**

## Directory Permissions Inheritance vs Recursion and Star Expansion?\*

Use echo to create in PermissionsChilda file called permissions\_grand\_child.txtwith the contents permissions\_grand\_child.

Display a long listing of the directory PermissionTest.

Give others the write right to PermissionsTest**.**

drwxr-srwx 3 jovyan 1000 68 Apr 21 22:50 PermissionsTest

Display again the long listing of hello.txt.

-rw-r--r-- 1 jovyan 1000 6 Apr 21 00:35 hello.txt

The directory write permissions for others are not inherited by the file.

Perhaps that is because we used chmod earlier to specify a custom permissions for this file

Look at the permission string of original\_hello.txt**.**

-rw-r--r-- 1 jovyan 1000 6 Apr 21 13:01 original\_hello.txt

It also does not inherit the permissions, even though we did not override the default permissions for it.

Perhaps that is because file and directory permissions are interpreted differently.

Look at the permission string of the directoryPermissionsChild**.**

drwxr-sr-x 2 jovyan 1000 6 Apr 21 22:50 PermissionsChild

It also does not inherit the permissions, even though we did not override the default permissions for it.

Recall that in Discovery, a file-system node inherited its parent directory’s access records if it did not override it with a custom access record. This is not the case in the Unix protection scheme.

One of the reasons for associating permissions with directories is to allow their grant and denial to be inherited by the descendants – we need to make the specification once for the directory than multiple times for each descendant.

Bash’s star expansion provides a way to overcome this apparent limitation of the Unix scheme.

Make sure your current directory is PermissionsTest**.**

Execute: chmod o+w \*

Verify that all of the children of the current directory now have the others write right

-rw-r--rw- 1 jovyan 1000 6 Apr 21 21:47 hello.txt

drwxr-srwx 2 jovyan 1000 6 Apr 21 22:50 PermissionsChild

-r--r--rw- 1 jovyan 1000 10 Apr 21 20:22 remove\_me.txt

Thus, permissions can be taken or given to from children of the current directory with one specification.

So, is star expansion equivalent to inheritance?

List the permission string of permissions\_grand\_childin PermissionsChild**.**

-rw-r--r-- 1 jovyan 1000 24 Apr 21 23:44 permissions\_grand\_child

It does not have the write permission for others.

Use the echo command to create a new child, new\_hello.txt, in PermissionsTest with the contents hello.

Look at its permission string. It does not have the others write right.

-rw-r--r-- 1 jovyan 1000 6 Apr 22 00:00 new\_hello.txt

Thus, star expansion is not equivalent to inheritance. Star expansion generates *current* children of a directory, while inheritance applies to *current* and *future* descendants of a directory. Ideally both should be supported.

Make PermissionsTest your current directory.

List it to verify it has the following child permissions:

-rw-r--r-- 1 jovyan 1000 6 Apr 21 21:47 hello.txt

drwxr-sr-x 2 jovyan 1000 37 Apr 21 23:44 PermissionsChild

-r--r--r-- 1 jovyan 1000 10 Apr 21 20:22 remove\_me.txt

Execute: chmod -R o+w .

Here we are recursively executing the chmod command on the current directory (.) and all of its descendants.

.:

total 12

-rw-r--rw- 1 jovyan 1000 6 Apr 21 21:47 hello.txt

-rw-r--rw- 1 jovyan 1000 6 Apr 22 00:00 new\_hello.txt

drwxr-srwx 2 jovyan 1000 37 Apr 21 23:44 PermissionsChild

-r--r--rw- 1 jovyan 1000 10 Apr 21 20:22 remove\_me.txt

./PermissionsChild:

total 4

-rw-r--rw- 1 jovyan 1000 24 Apr 21 23:44 permissions\_grand\_child

Use echo to create newer\_hello.txt with the contents hello.

As expected, it does not have the others write right, showing inheritance is a much more powerful technique.

-rw-r--r-- 1 jovyan 1000 6 Apr 22 00:07 newer\_hello.txt

## Meeting Access Requirements\*

Exercise 1:

Let us try to solve a variation of the access control problem we tried to address in the Discovery tutorial.

1. The ownershould have the readright on every file descendant of Discovery.
2. otherusersshould be able to read all files related to Drug\_B.

No other access should be allowed to descendants of Discovery. In particular, othersshould not be allowed to read or execute any file related to Drug\_A including the sort\_output.txt file in FilteredSortedOutput.

Exercise 2:

Change your directory to the Discovery subdirectory in the home directory. Use your knowledge of Unix access control to share parts of the CompressedSamplesfolder in it with all users.

No user should have write or execute access to any descendent of this folder.

All users should be able to determine which two drugs are involved in the experiment, that is, read the names of the two children of UncompressedSamples (Drug\_A and Drug\_B). No user should have read access to Drug\_A or any of its descendants such as Drug\_A\_1.fastq and Drug\_A\_2.fastq. All users should have read access to the Drug\_B output folder and to only two descendants of this folder - the uncompressed fastq files, Drug\_B\_1.fastq and Drug\_B\_2.fastq.  Thus, no user should have read access to all other descendants of Drug\_B, such as Drug\_B\_1.fastq and Drug\_B\_2.fastq.

Given star expansion, there are multiple ways in which we can change default access control to meet these requirements. Try to choose one that involves fewest executions of chmod.

Verify that these requirements by trying owner accesses that are legal and illegal under these requirements.

## Exit Bash and Shell

Type the command **exit** to terminate your bash session. This also terminates this long lesson.

Done!