



Basic Shell Scripting

Jason Li

HPC User Services

LSU HPC / LONI

sys-help@loni.org

Louisiana State University, Baton Rouge

Jul 23, 2025

▪ HPC User Environment 1

1. Intro to HPC
2. Getting started
3. Into the cluster
4. Software environment (modules)

▪ HPC User Environment 2

1. Basic concepts
2. Preparing my job
3. Submitting my job
4. Managing my jobs

1. Introduction

- 1) What's Shell?
- 2) What can Shell do?

2. Basic Knowledge

- 1) Interactive vs Non-interactive (Shell Script)
- 2) Basic Commands & Syntax
- 3) Variables
- 4) Arrays
- 5) Arithmetic Operations

3. Beyond Basics

- 1) Subshells
- 2) Flow Control
- 3) Advanced Text Processing Commands

4. BONUS: Where to Get Help

- **Example and exercises:**

- <http://www.hpc.lsu.edu/training/weekly-materials/Downloads/ShellScripting.zip>

1. Introduction

- 1) What's Shell?
- 2) What can Shell do?

2. Basic Knowledge

- 1) Interactive vs Non-interactive (Shell Script)
- 2) Basic Commands & Syntax
- 3) Variables
- 4) Arrays
- 5) Arithmetic Operations

3. Beyond Basics

- 1) Subshells
- 2) Flow Control
- 3) Advanced Text Processing Commands

4. BONUS: Where to Get Help

1) What's Shell?

- Previously in HPC User Environment 2...
 - Two types of jobs

1) Interactive job

```
(base) [jasonli3@qbd454 pi]$  
salloc: Granted job allocation 23480  
salloc: Waiting for resource configuration  
salloc: Nodes qbd454 are ready for job  
salloc: lua: Submitted job 23480  
(base) [jasonli3@qbd454 pi]$
```

2) Batch job

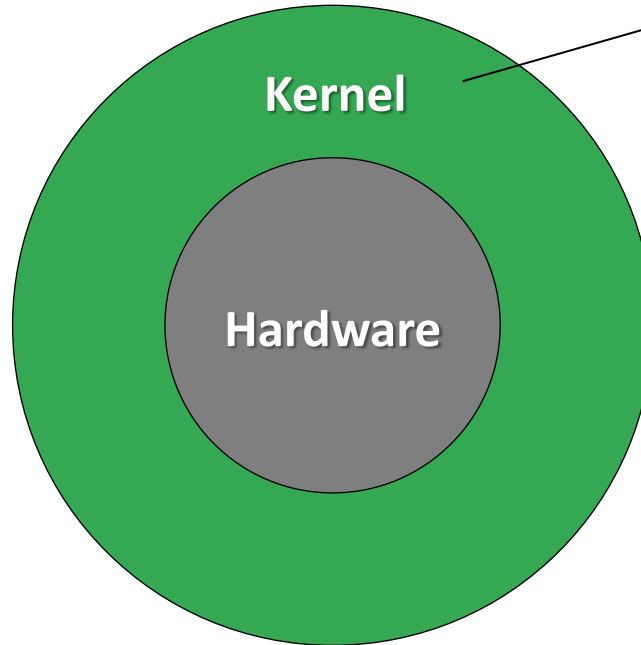
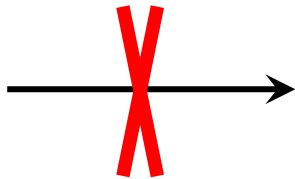
```
#SBATCH -n 64  
  
module load python  
  
cd $SLURM_SUBMIT_DIR  
./pi_serial.out 100000000
```

In both cases, you are accessing a Linux system **through Shell**

1) What's Shell?



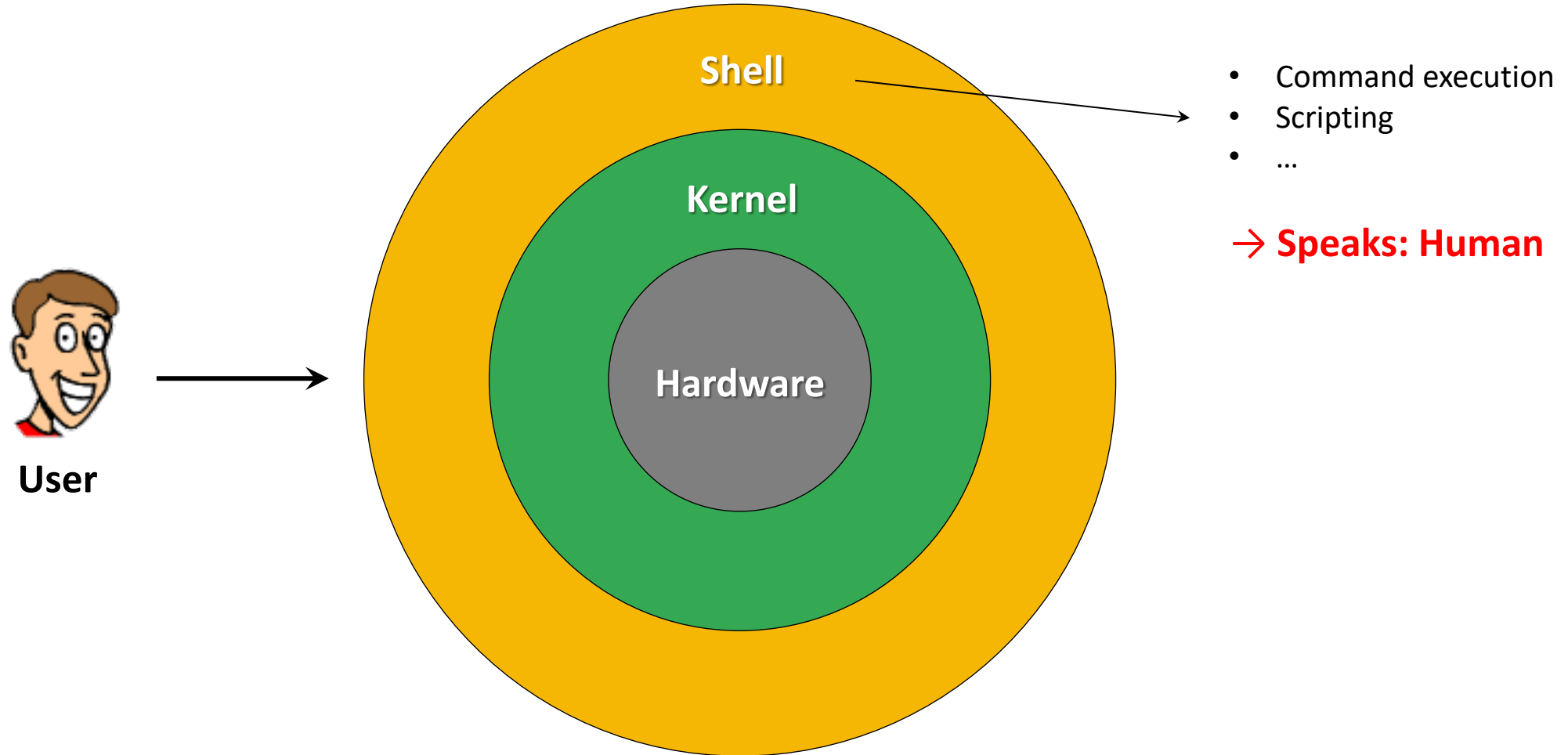
User



- Resource management
- Process management
- Device Drivers
- System Calls
- ...

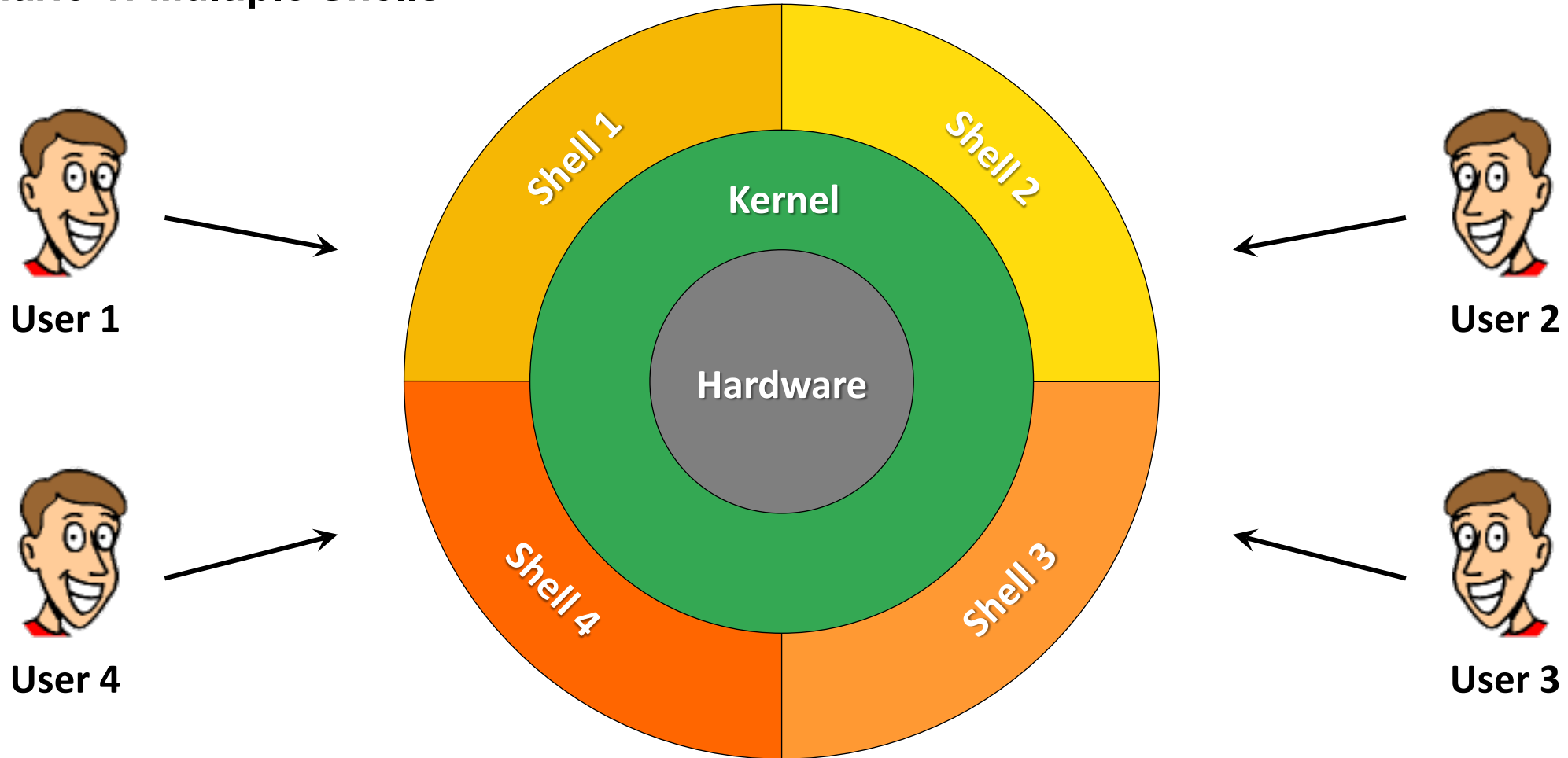
→ **Speaks: Machine**

1) What's Shell?



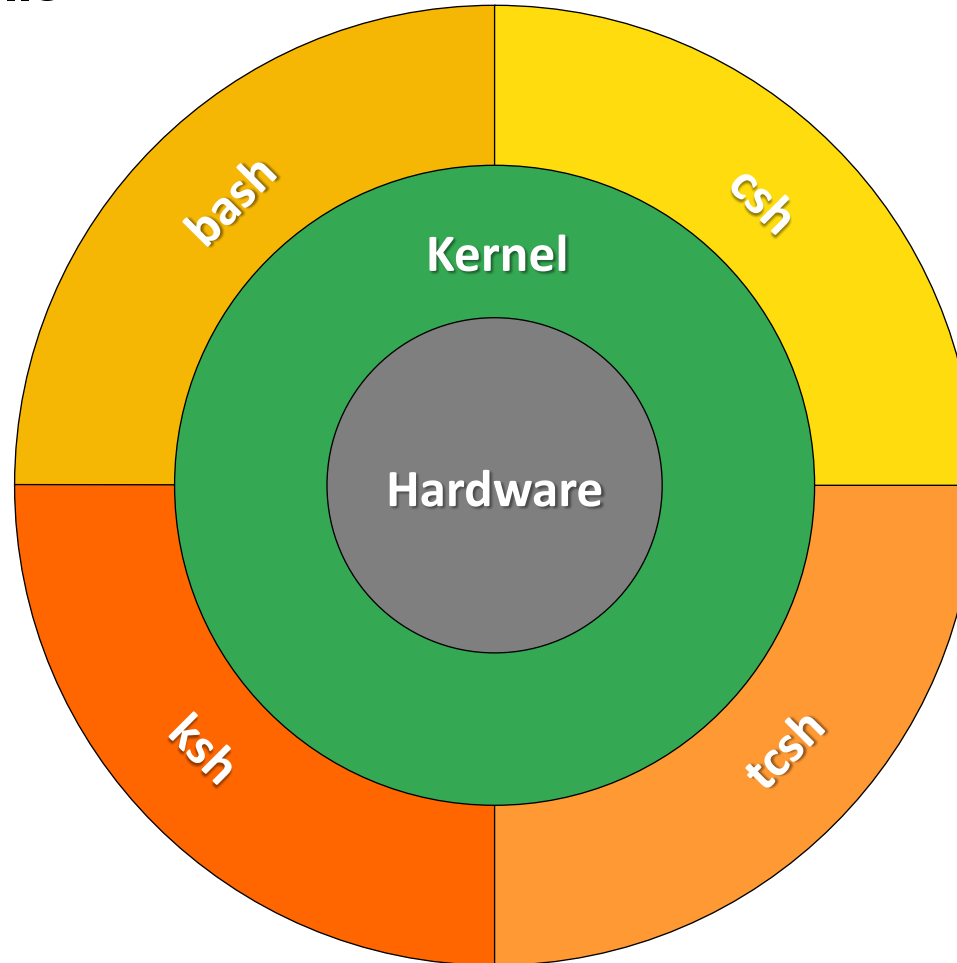
1) What's Shell?

- Scenario 1: Multiple Shells



1) What's Shell?

- Scenario 1: Multiple Shells

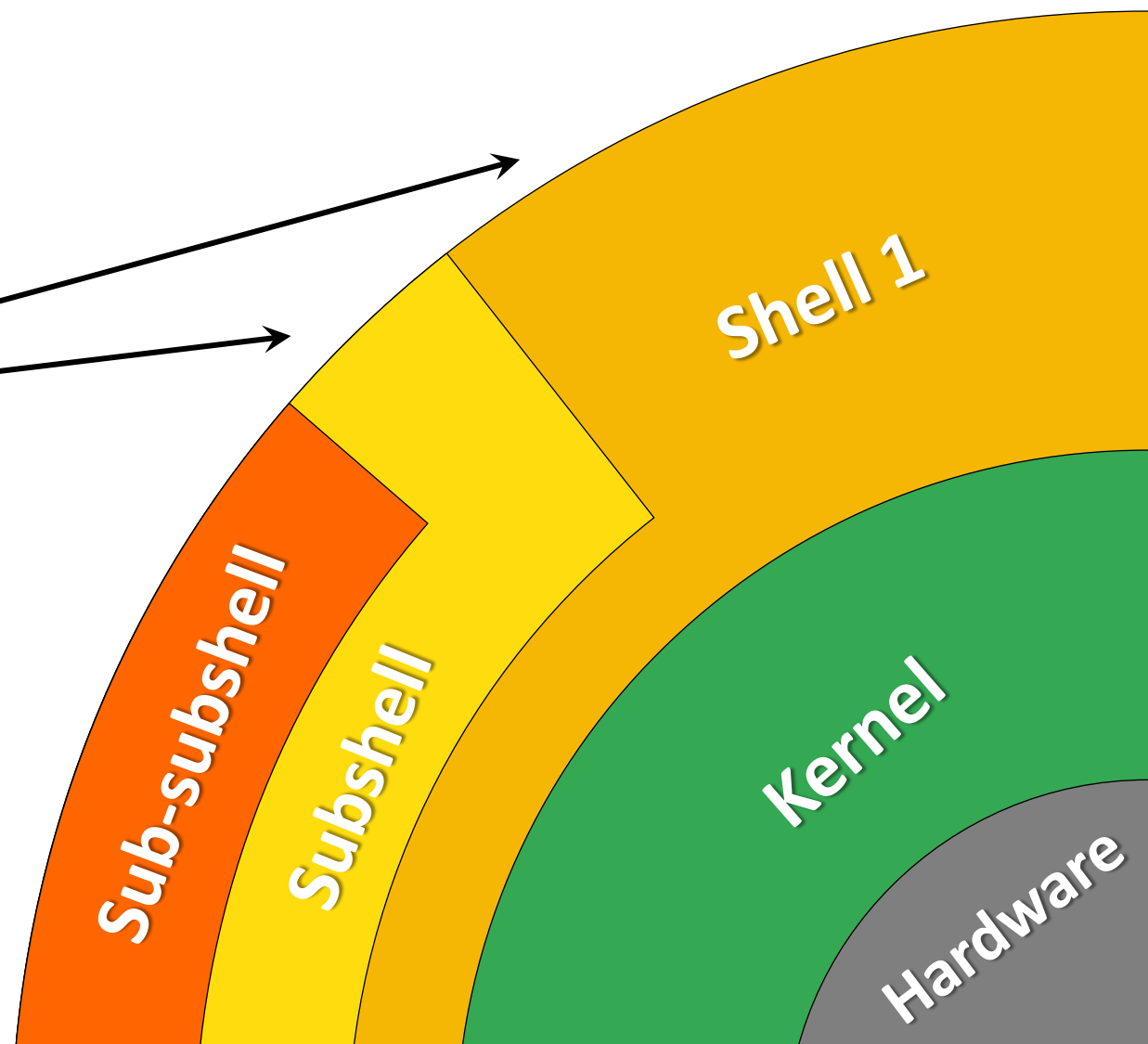


1) What's Shell?

- Scenario 2: Shells within Shells (Subshells)



User 1



1) What's Shell?

- **Shell:**
 - A **user interface** to access UNIX-like systems (e.g., Linux) by executing commands.

1. Introduction

- 1) What's Shell?
- 2) What can Shell do?

2. Basic Knowledge

- 1) Interactive vs Non-interactive (Shell Script)
- 2) Basic Commands & Syntax
- 3) Variables
- 4) Arrays
- 5) Arithmetic Operations

3. Beyond Basics

- 1) Subshells
- 2) Flow Control
- 3) Advanced Text Processing Commands

4. BONUS: Where to Get Help

2) What can Shell do?

- Shell can do this ...

- Typing commands one by one

```
(base) [jasonli3@qbd1 ShellScripting]$ ls
1.1-ShellExamples 1.2-Goal exec README.txt
(base) [jasonli3@qbd1 ShellScripting]$ date
Tue Sep 17 15:17:47 CDT 2024
(base) [jasonli3@qbd1 ShellScripting]$ echo $SHELL
/bin/bash
(base) [jasonli3@qbd1 ShellScripting]$ cd exec
(base) [jasonli3@qbd1 exec]$ ls
pi_bash.sh pi_c pi_c.sbatch
(base) [jasonli3@qbd1 exec]$ ./pi_c 100000000
niter=100000000
count in circle:78547994
Pi: 3.141920
(base) [jasonli3@qbd1 exec]$
```

2) What can Shell do?

- Shell can also do this ...
 - A much more complicated program / script

```
#!/bin/bash

# URL and file name
URL=$1
FILENAME=`basename $URL`

# Number of parts
PARTS=$SLURM_NTASKS
echo "Downloading in $PARTS parts..."

# Create temporary directory for parts
TEMP_DIR=$(mktemp -d)

# Calculate the range for each part
FILE_SIZE=$(curl -sIk $URL | awk '/Content-Length/ {print $2}' | tr -d '\r') # Get file size
PART_SIZE=$((FILE_SIZE / PARTS)) # Calculate part size
LAST_PART_SIZE=$((FILE_SIZE - PART_SIZE * (PARTS - 1))) # Calculate size of the last part

# Download each part concurrently
for ((i = 0; i < PARTS; i++)); do
    start=$((i * PART_SIZE))
    end=$((start + PART_SIZE - 1))

    if [[ $i -eq $((PARTS - 1)) ]]; then
        end=$((start + LAST_PART_SIZE - 1))
    fi

    curl -ks -o "$TEMP_DIR/part$i" --range "$start-$end" "$URL" &
done

# Wait for all downloads to finish
wait

# Merge the parts into a single file
for ((i = 0; i < PARTS; i++)); do
    cat "$TEMP_DIR/part$i" >> "data/$FILENAME"
done

# Clean up temporary directory
rm -rf "$TEMP_DIR"

echo "Download completed!"
```

2) What can Shell do?

- **Shell Scripting:**
 - A practice to **automate tasks** with Shell commands.

2) What can Shell do?

- Take a closer look at this:

```
#!/bin/bash

# URL and file name
URL=$1
FILENAME=`basename $URL`

# Number of parts
PARTS=$SLURM_NTASKS
echo "Downloading in $PARTS parts..."

# Create temporary directory for parts
TEMP_DIR=$(mktemp -d)

# Calculate the range for each part
FILE_SIZE=$(curl -sIk $URL | awk '/Content-Length/ {print $2}' | tr -d '\r') # Get file size
PART_SIZE=$((FILE_SIZE / PARTS)) # Calculate part size
LAST_PART_SIZE=$((FILE_SIZE - PART_SIZE * (PARTS - 1))) # Calculate size of the last part

# Download each part concurrently
for ((i = 0; i < PARTS; i++)); do
    start=$((i * PART_SIZE))
    end=$((start + PART_SIZE - 1))

    if [[ $i -eq $((PARTS - 1)) ]]; then
        end=$((start + LAST_PART_SIZE - 1))
    fi

    curl -ks -o "$TEMP_DIR/part$i" --range "$start-$end" "$URL" &
done

# Wait for all downloads to finish
wait

# Merge the parts into a single file
for ((i = 0; i < PARTS; i++)); do
    cat "$TEMP_DIR/part$i" >> "data/$FILENAME"
done

# Clean up temporary directory
rm -rf "$TEMP_DIR"

echo "Download completed!"
```

2) What can Shell do?

- Take a closer look at this:

```
#!/bin/bash

# URL and file name
URL=$1
FILENAME=`basename $URL`

# Number of parts
PARTS=$SLURM_NTASKS
echo "Downloading in $PARTS parts..."

# Create temporary directory for parts
TEMP_DIR=$(mktemp -d)

# Calculate the range for each part
FILE_SIZE=$(curl -sIk $URL | awk '/Content-Length/ {print $2}' | tr -d '\r') # Get file size
PART_SIZE=$((FILE_SIZE / PARTS)) # Calculate part size
LAST_PART_SIZE=$((FILE_SIZE - PART_SIZE * (PARTS - 1))) # Calculate size of the last part

# Download each part concurrently
for ((i = 0; i < PARTS; i++)); do
    start=$((i * PART_SIZE))
    end=$((start + PART_SIZE - 1))

    if [[ $i -eq $(PARTS - 1) ]]; then
        end=$((start + LAST_PART_SIZE - 1))
    fi

    curl -ks -o "$TEMP_DIR/part$i" --range "$start-$end" "$URL" &
done

# Wait for all downloads to finish
wait

# Merge the parts into a single file
for ((i = 0; i < PARTS; i++)); do
    cat "$TEMP_DIR/part$i" >> "data/$FILENAME"
done

# Clean up temporary directory
rm -rf "$TEMP_DIR"

echo "Download completed!"
```

2) What can Shell do?

- Take a closer look at this:

```
#!/bin/bash

# URL and file name
URL=$1
FILENAME=`basename $URL`

# Number of parts
PARTS=$SLURM_NTASKS
echo "Downloading in $PARTS parts..."

# Create temporary directory for parts
TEMP_DIR=$(mktemp -d)

# Calculate the range for each part
FILE_SIZE=$(curl -sIk $URL | awk '/Content-Length/ {print $2}' | tr -d '\r') # Get file size
PART_SIZE=$((FILE_SIZE / PARTS)) # Calculate part size
LAST_PART_SIZE=$((FILE_SIZE - PART_SIZE * (PARTS - 1))) # Calculate size of the last part

# Download each part concurrently
for ((i = 0; i < PARTS; i++)); do
    start=$((i * PART_SIZE))
    end=$((start + PART_SIZE - 1))

    if [[ $i -eq $((PARTS - 1)) ]]; then
        end=$((start + LAST_PART_SIZE - 1))
    fi

    curl -ks -o "$TEMP_DIR/part$i" --range "$start-$end" "$URL" &
done

# Wait for all downloads to finish
wait

# Merge the parts into a single file
for ((i = 0; i < PARTS; i++)); do
    cat "$TEMP_DIR/part$i" >> "data/$FILENAME"
done

# Clean up temporary directory
rm -rf "$TEMP_DIR"

echo "Download completed!"
```

2) What can Shell do?

- Take a closer look at this:

Isn't it basically a programming language?

```
#!/bin/bash

# URL and file name
URL=$1
FILENAME=`basename $URL`

# Number of parts
PARTS=$SLURM_NTASKS
echo "Downloading in $PARTS parts..."

# Create temporary directory for parts
TEMP_DIR=$(mktemp -d)

# Calculate the range for each part
FILE_SIZE=$(curl -sIk $URL | awk '/Content-Length/ {print $2}' | tr -d '\r') # Get file size
PART_SIZE=$((FILE_SIZE / PARTS)) # Calculate part size
LAST_PART_SIZE=$((FILE_SIZE - PART_SIZE * (PARTS - 1))) # Calculate size of the last part

# Download each part concurrently
for ((i = 0; i < PARTS; i++)); do
    start=$((i * PART_SIZE))
    end=$((start + PART_SIZE - 1))

    if [[ $i -eq $((PARTS - 1)) ]]; then
        end=$((start + LAST_PART_SIZE - 1))
    fi

    curl -ks -o "$TEMP_DIR/part$i" --range "$start-$end" "$URL" &
done

# Wait for all downloads to finish
wait

# Merge the parts into a single file
for ((i = 0; i < PARTS; i++)); do
    cat "$TEMP_DIR/part$i" >> "data/$FILENAME"
done

# Clean up temporary directory
rm -rf "$TEMP_DIR"

echo "Download completed!"
```

2) What can Shell do?

- Questions:

Why would I need ...	If I can just use ...
Shell	Another language (Python / C++ / Fortran)
Another language (Python / C++ / Fortran)	Shell

2) What can Shell do?

a) Why would I need **Shell** if I can just use **another language**?

- Shell is a “**quick and dirty**” way to get things done!
 - *Example:* Change all text “/ddnB/work” to “/work” in all files in folder “~/mycode/” and subfolders.

Python	Shell
<pre>import os folder = os.path.expanduser('~mycode') for dirpath, _, filenames in os.walk(folder): for filename in filenames: filepath = os.path.join(dirpath, filename) with open(filepath, 'r') as file: content = file.read() new_content = content.replace('/ddnB/work', '/work') with open(filepath, 'w') as file: file.write(new_content)</pre>	<pre>find ~/mycode/ -type f -exec sed -i 's /ddnB/work /work g' {} +</pre>

2) What can Shell do?

- Rule of thumb:

Anything you wish to run faster, you should NOT use shell!

2) What can Shell do?

- **Goal of Shell scripting:**

Shell scripting is NOT for...	Shell scripting IS for...
• Heavy calculation (basically, anything you wish to run faster!) • Replacing your known language / software	• Automating job workflow with minimum scripting (e.g., set up environment, call proper executables, etc.) • Pre-processing / Post-processing (e.g., trim data, edit config files in batch, etc.)

- **Goal of this training:**

We do NOT expect you to be...	We DO expect you to be...
• An expert in Linux or Shell language.	• Familiar with Shell's basic usage. • Able to use Shell scripting to optimize job workflow.

1. Introduction

- 1) What's Shell?
- 2) What can Shell do?

2. Basic Knowledge

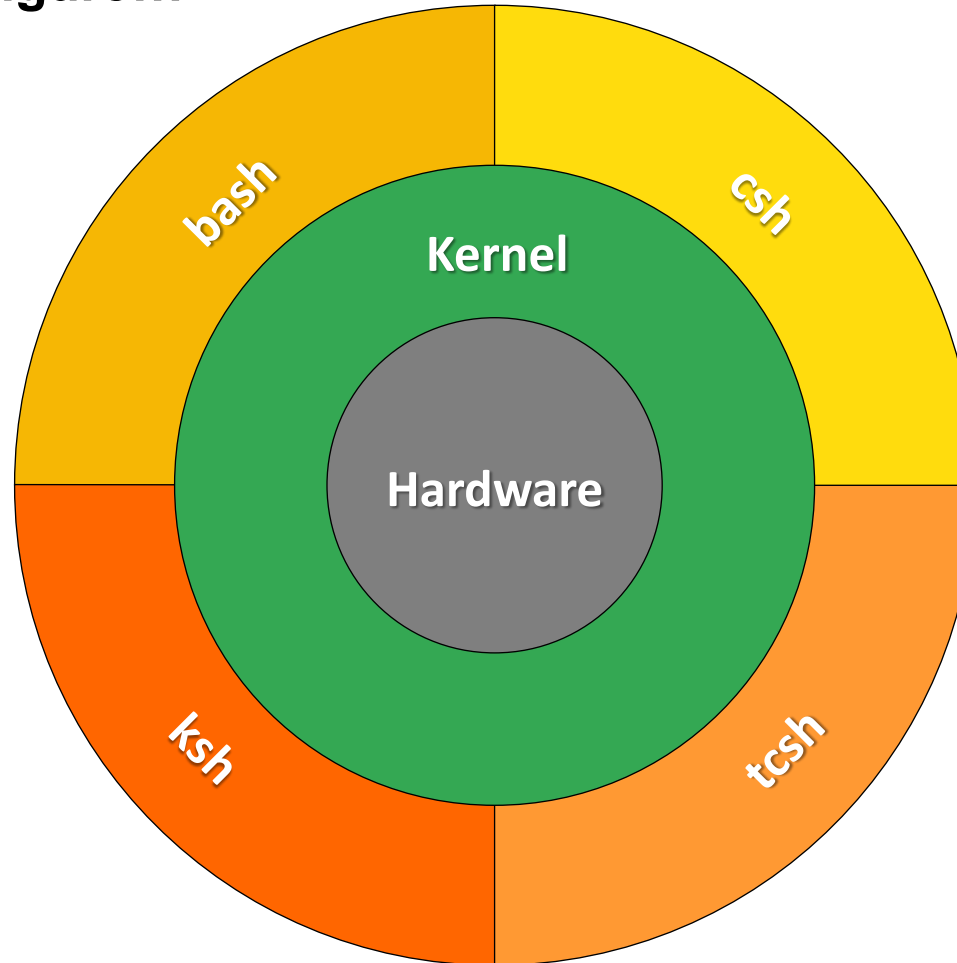
- 1) Interactive vs Non-interactive (Shell Script)
- 2) Basic Commands & Syntax
- 3) Variables
- 4) Arrays
- 5) Arithmetic Operations

3. Beyond Basics

- 1) Subshells
- 2) Flow Control
- 3) Advanced Text Processing Commands

4. BONUS: Where to Get Help

- Remember we had this figure...



- There are many Shell implementations

- **sh** (Original Bourne Shell)
- **bash** (Bourne Again Shell)
- **csh** (C Shell)
- **tcsh** (TENEX C Shell, more features)
- **ksh** (KornShell)
- **zsh** (Z Shell)
- **dash** (Debian Almquist Shell)
- **fish** (Friendly Interactive Shell)
- ...

- Supported by our clusters
- Feel free to use whichever you like!
- Can set your own default Shell

- There are many Shell implementations

- **sh** (Original Bourne Shell)
- **bash** (Bourne Again Shell)
- **csh** (C Shell)
- **tcsh** (TENEX C Shell, more features)
- **ksh** (KornShell)
- **zsh** (Z Shell)
- **dash** (Debian Almquist Shell)
- **fish** (Friendly Interactive Shell)
- ...

- Default Shell on all clusters
- Will only talk about it today

1. Introduction

- 1) What's Shell?
- 2) What can Shell do?

2. Basic Knowledge

- 1) Interactive vs Non-interactive (Shell Script)
- 2) Basic Commands & Syntax
- 3) Variables
- 4) Arrays
- 5) Arithmetic Operations

3. Beyond Basics

- 1) Subshells
- 2) Flow Control
- 3) Advanced Text Processing Commands

4. BONUS: Where to Get Help

1) Interactive vs Non-interactive Shell

a) Two ways to access Shell

Interactive

```
(base) [jasonli3@qbd1 ShellScripting]$ ls
1.1-ShellExamples 1.2-Goal exec README.txt
(base) [jasonli3@qbd1 ShellScripting]$ date
Tue Sep 17 15:17:47 CDT 2024
(base) [jasonli3@qbd1 ShellScripting]$ echo $SHELL
/bin/bash
(base) [jasonli3@qbd1 ShellScripting]$ cd exec
(base) [jasonli3@qbd1 exec]$ ls
pi_bash.sh pi_c pi_c.sbatch
(base) [jasonli3@qbd1 exec]$ ./pi_c 100000000
niter=100000000
count in circle:78547994
Pi: 3.141920
(base) [jasonli3@qbd1 exec]$
```

Non-interactive

```
#!/bin/bash
# URL and file name
URL=$1
FILENAME=`basename $URL`

# Number of parts
PARTS=$SLURM_NTASKS
echo "Downloading in $PARTS parts..."

# Create temporary directory for parts
TEMP_DIR=$(mktemp -d)

# Calculate the range for each part
FILE_SIZE=$(curl -sIk $URL | awk '/Content-Length/ {print $2}' | tr -d '\r') #
PART_SIZE=$((FILE_SIZE / PARTS)) # Calculate part size
LAST_PART_SIZE=$((FILE_SIZE - PART_SIZE * (PARTS - 1))) # Calculate size of th

# Download each part concurrently
for ((i = 0; i < PARTS; i++)); do
    start=$((i * PART_SIZE))
    end=$((start + PART_SIZE - 1))
```

1) Interactive vs Non-interactive Shell

a) Two ways to access Shell

Interactive

- Runs in terminal
- Can interact in real time
- Type commands one-by-one
- E.g., every time you log in in terminal

Non-interactive

```
#!/bin/bash
# URL and file name
URL=$1
FILENAME=`basename $URL`

# Number of parts
PARTS=$SLURM_NTASKS
echo "Downloading in $PARTS parts..."

# Create temporary directory for parts
TEMP_DIR=$(mktemp -d)

# Calculate the range for each part
FILE_SIZE=$(curl -sIk $URL | awk '/Content-Length/ {print $2}' | tr -d '\r') #
PART_SIZE=$((FILE_SIZE / PARTS)) # Calculate part size
LAST_PART_SIZE=$((FILE_SIZE - PART_SIZE * (PARTS - 1))) # Calculate size of th

# Download each part concurrently
for ((i = 0; i < PARTS; i++)); do
    start=$((i * PART_SIZE))
    end=$((start + PART_SIZE - 1))
```

1) Interactive vs Non-interactive Shell

a) Two ways to access Shell

Interactive

- Runs in terminal
- Can interact in real time
- Type commands one-by-one
- E.g., every time you log in in terminal

Non-interactive

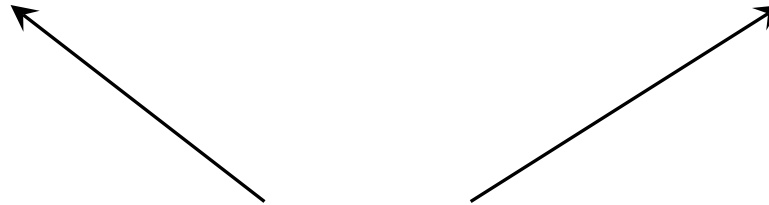
- Prewritten script (Shell script)
- Cannot interact while it is running
- Runs by itself (line-by-line)

1) Interactive vs Non-interactive Shell

a) Two ways to access Shell

Interactive

Non-interactive



Shell scripting works the same way in both! *

* A few features may be slightly different. But for now, don't worry about that.



1) Interactive vs Non-interactive Shell

b) How to write a Shell script

```
#!/bin/bash
```

```
date  
echo "Hello World!"  
█
```

" **Shebang** " -

- Shell to run this script with

1) Interactive vs Non-interactive Shell

b) How to write a Shell script

```
#!/bin/bash
```

```
date  
echo "Hello World!"  
█
```

Commands to run

c) How to run a Shell script (four methods)

Method		Example	Remarks			
			Must be executable?	Which Shell?	Start subshell?	Others
1	Use full path (Most common)	\$./helloworld.sh \$ /path/to/helloworld.sh	✓	Shebang (if exist) or default Shell	✓	-
2	Use specific Shell	\$ bash helloworld.sh \$ cs h helloworld.sh	✗	Specified Shell	✓	-
3	Use "source" or "."	\$ source helloworld.sh \$. helloworld.sh	✗	Current Shell	✗	-
4	Run as Shell command	\$ helloworld.sh	✓	Shebang (if exist) or default Shell	✓	Parent directory must be included in \$PATH environment variable

Run "**chmod u+x [filename]**" if file is not executable

1) Interactive vs Non-interactive Shell

- Pop quiz: What is this?

→ Anything you learned about Shell today, applies to your batch job files!

```
#!/bin/bash
#SBATCH -A loni_loniadmin1
#SBATCH -p single
#SBATCH -t 1:00:00
#SBATCH -N 1
#SBATCH -n 12

# Time stamp at the beginning
date

# Run the code
./pi_c 1000000

# Time stamp at the end
date
```

1. Introduction

- 1) What's Shell?
- 2) What can Shell do?

2. Basic Knowledge

- 1) Interactive vs Non-interactive (Shell Script)
- 2) Basic Commands & Syntax
- 3) Variables
- 4) Arrays
- 5) Arithmetic Operations

3. Beyond Basics

- 1) Subshells
- 2) Flow Control
- 3) Advanced Text Processing Commands

4. BONUS: Where to Get Help

2) Basic Commands & Syntax

a) Basic commands

Command		Description
File	<code>ls</code>	List files at a given location .
	<code>cp / mv</code>	Copy / Move files.
	<code>rm</code>	Remove files.
	<code>find</code>	Search for files.
Directory	<code>cd</code>	Change directory.
	<code>mkdir</code>	Create a directory.
	<code>pwd</code>	Print current directory in standard output.
Display	<code>cat</code>	Print out an entire file in standard output.
	<code>head / tail</code>	Show first / last several lines of a file.
	<code>more / less</code>	Display file one page at a time.
System	<code>echo</code>	Print out strings in standard output.
	<code>date</code>	Print out current date & time in standard output.
...		

2) Basic Commands & Syntax

b) Commonly used **special characters** that works with commands

Character	Description	Example
#	Comment: Anything follows in the same line will not be executed.	\$ date # Print time stamp
;	Command separator: Allows multiple commands in one line.	\$ module purge; module load python
	Pipeline: Use output of first command as input of the second.	\$ queue -u \$USER wc -l
>	Redirect (Output): Redirect standard output / error to file. This method overwrites the file.	\$./testoutput > out.txt \$./testoutput 1> out.txt 2> err.txt
>>	Redirect (Output): Redirect standard output / error to file. This method appends to the file.	\$./testoutput >> out.txt \$./testoutput 1>> out.txt 2>> err.txt
<	Redirect (Input): Read input from a file instead of standard input.	\$./testinput < input.txt
&	Send to background: Send a command to background, and do not wait for it to finish.	\$ sleep 10 &

1. Introduction

- 1) What's Shell?
- 2) What can Shell do?

2. Basic Knowledge

- 1) Interactive vs Non-interactive (Shell Script)
- 2) Basic Commands & Syntax
- 3) Variables
- 4) Arrays
- 5) Arithmetic Operations

3. Beyond Basics

- 1) Subshells
- 2) Flow Control
- 3) Advanced Text Processing Commands

4. BONUS: Where to Get Help

a) Variable basics

	To assign	To access	To delete
Syntax	var=value	\$ var	unset var
Examples	\$ str="Hello World!"	\$ echo \$str	
	\$ workdir="/work/jasonli3/test/"	\$ cd \$workdir	
	\$ mycmd="/home/jasonli3/myexec"	\$ \$mycmd > \$myout	
	\$ myout="/work/jasonli3/out.txt"		

– ATTENTION!

- All Shell variables are treated as **strings**! (No integer, float, Boolean...)
- **No space** allowed in assignment!
- Use **{ }** to explicitly mark variable name. (e.g., **\${var}** instead of **\$var**)
 - Think about it. When can this be useful?

b) Naming rules

- Allowed characters: **letters** (a-z, A-Z), **numbers** (0-9), **underscore** (_)
- Must begin with a **letter** or an **underscore**.
- No other special characters (e.g., #, @, %, \$, ...)
 - **Allowed:** `varname`, `var_name`, `_varName`, `var123`
 - **Not allowed:** `123var`, `#var`, `var@name`, `var-123`
- Case sensitive
 - `VAR` and `var` are different variables!

c) Global & local variables

	Local	Global
Syntax	\$ var=value	\$ export VAR=value
Differences	Exist only in current shell	Copied to all subshells
	Lowercase*	Uppercase*

* **Convention**, to avoid conflict

d) Environment variables

- **Definition:**

- Specific variables used by Shell or other programs to regulate certain functionalities.

- **Remarks:**

- Usually **global** (Convention)
- **Customizable**, will change Shell or program behavior (Caution!)
- Programs may have their own environment variables (e.g., Conda / Python / R / MPI ...)

d) Environment variables

Variable		Functionality
Shell	USER	Username.
	PWD	Full path to current directory.
	HOME	Full path to user's home directory.
	SHELL	Default Shell
	PATH	A list of paths to look for executables as Shell commands (separated by ":").
	LD_LIBRARY_PATH	A list of paths to look for shared libraries (separated by ":").
Slurm	SLURM_JOB_ID	Slurm job ID.
	SLURM_JOB_NODELIST	A list of nodes required for current job (useful for MPI).
OpenMP	OMP_NUM_THREADS	Number of threads per process for OpenMP.
...		

e) Quotations & variables

Quotation	Description	Example
""	Allows variable expansion (“\$”) and command substitution (“`”) within quotes, and preserves literal values of all other characters .	\$ echo "echo \$USER" echo jasonli3
' '	Preserves the literal value of ALL CHARACTERS within the quotes.	\$ echo 'echo \$USER' echo \$USER
` `	Command substitute : Execute the command(s) inside the quotation and use its output to replace the quotation.	\$ echo `echo \$USER` jasonli3

1. Introduction

- 1) What's Shell?
- 2) What can Shell do?

2. Basic Knowledge

- 1) Interactive vs Non-interactive (Shell Script)
- 2) Basic Commands & Syntax
- 3) Variables
- 4) Arrays
- 5) Arithmetic Operations

3. Beyond Basics

- 1) Subshells
- 2) Flow Control
- 3) Advanced Text Processing Commands

4. BONUS: Where to Get Help

- **A collection of multiple values**
 - Basic logic very similar to “arrays” in any other language, **with some twists!**
 - Each element is accessed by **index**
 - Index starts with **0**

	To assign	To delete	To access
Entire array	\$ myAry=("Alice" "Bob" "Charlie")	\$ unset myAry	\$ echo \${myAry[@]}
One element	\$ myAry[1]="Brian"	\$ unset myAry[1]	\$ echo \${myAry[1]}

- **Bonus:** Get length of array - `${#myAry[@]}`

- Question:
 - I am not using Shell for heavy calculation anyways! What can I possibly need arrays for?



Introduction to GNU Parallel - Parallelizing Massive Individual Tasks

Siva Prasad Kasetti
HPC User Services
LSU HPC & LONI
sys-help@loni.org

Louisiana State University
Baton Rouge

Apr 9, 2025

```
$ parallel myexec ::: ${inputParams[@]}
```

[1] <https://www.hpc.lsu.edu/training/tutorials.php#upcoming>



1. Introduction

- 1) What's Shell?
- 2) What can Shell do?

2. Basic Knowledge

- 1) Interactive vs Non-interactive (Shell Script)
- 2) Basic Commands & Syntax
- 3) Variables
- 4) Arrays
- 5) Arithmetic Operations

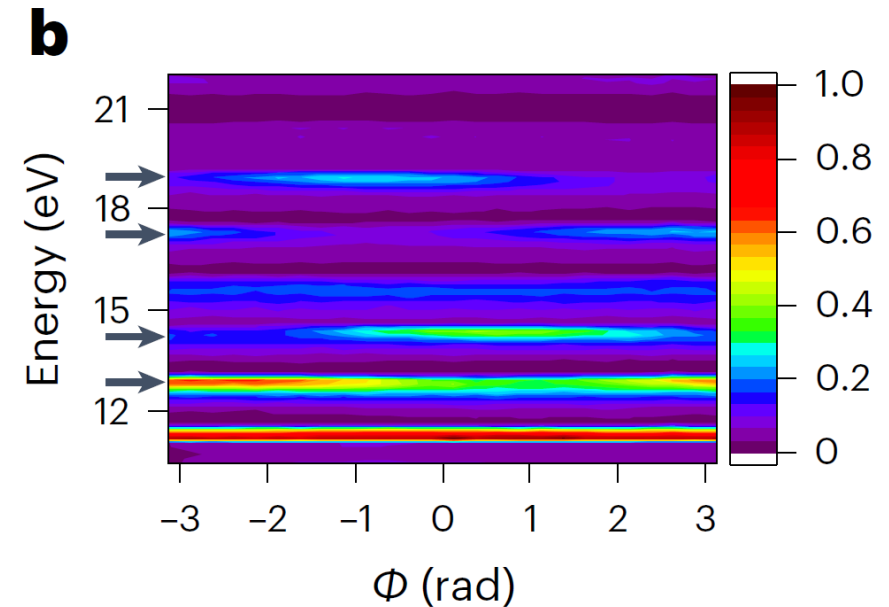
3. Beyond Basics

- 1) Subshells
- 2) Flow Control
- 3) Advanced Text Processing Commands

4. BONUS: Where to Get Help

5) Arithmetic Operations

- **Wait a minute!**
 - Didn't you say Shell **does not** support number type, and we **should not** use it for heavy calculation?
 - Correct!
 - But! Sometimes arithmetic is still needed.
 - **Example:** Parallelizing a photoelectron spectrum calculation (my actual research!)
 - Each parallel process labeled w/ an **integer index** [-100, 100]
 - But! Need to pass on a delay parameter to each process, a **float number** calculated from the integer index



5) Arithmetic Operations

- What does **NOT** work:

```
$ a=10
$ b=$a/3+2
$ echo $b      # Guess what you get?
10/3+2
```

5) Arithmetic Operations

- What **DOES** work (assuming **a=10**):

Method		Example	Remarks
1	<code>\$((...))</code> (Most common)	<code>\$ echo <code>\$(((\$a/3+2))</code></code>	- Evaluate everything inside the braces. Integers only!
2	<code>let</code> (Slightly more advanced)	<code>\$ <code>let</code> b=\$a/3+2</code> <code>\$ <code>let</code> b=a/3+2</code> <code>\$ <code>let</code> b++</code>	- Evaluate assignment w/ arithmetic calculation. "\$" can be emitted. Integers only!
3	<code>expr</code> (Legacy, most limited)	<code>\$ <code>expr</code> \$a / 3 + 2</code>	- Strictly limited to "ARG1 OPERATION ARG2" format. Integers only!
4	<code>bc</code> (Most powerful)	<code>\$ <code>bc</code></code> <code>scale=3</code> <code>a=10;a/3+2</code> <code>\$ <code>bc</code> < bcExample.txt</code> <code>\$ echo "\$a/2+3" <code>bc</code></code>	Interactive and non-interactive mode. - Does NOT support Shell syntax (namely, "\$" for variables). - Unassigned variables treated as 0. scale variable determines number of decimals. - Supports float number!

- **In this section, we talked about:**
 - 1) Interactive vs Non-interactive (Shell Script)
 - 2) Basic Commands & Syntax
 - 3) Variables
 - 4) Arrays
 - 5) Arithmetic Operations

- Get some water
- Use restroom
- Ask questions
- Don't forget, the examples are at:
 - <http://www.hpc.lsu.edu/training/weekly-materials/Downloads/ShellScripting.zip>

1. Introduction

- 1) What's Shell?
- 2) What can Shell do?

2. Basic Knowledge

- 1) Interactive vs Non-interactive (Shell Script)
- 2) Basic Commands & Syntax
- 3) Variables
- 4) Arrays
- 5) Arithmetic Operations

3. Beyond Basics

- 1) Subshells
- 2) Flow Control
- 3) Advanced Text Processing Commands

4. BONUS: Where to Get Help

1. Introduction

- 1) What's Shell?
- 2) What can Shell do?

2. Basic Knowledge

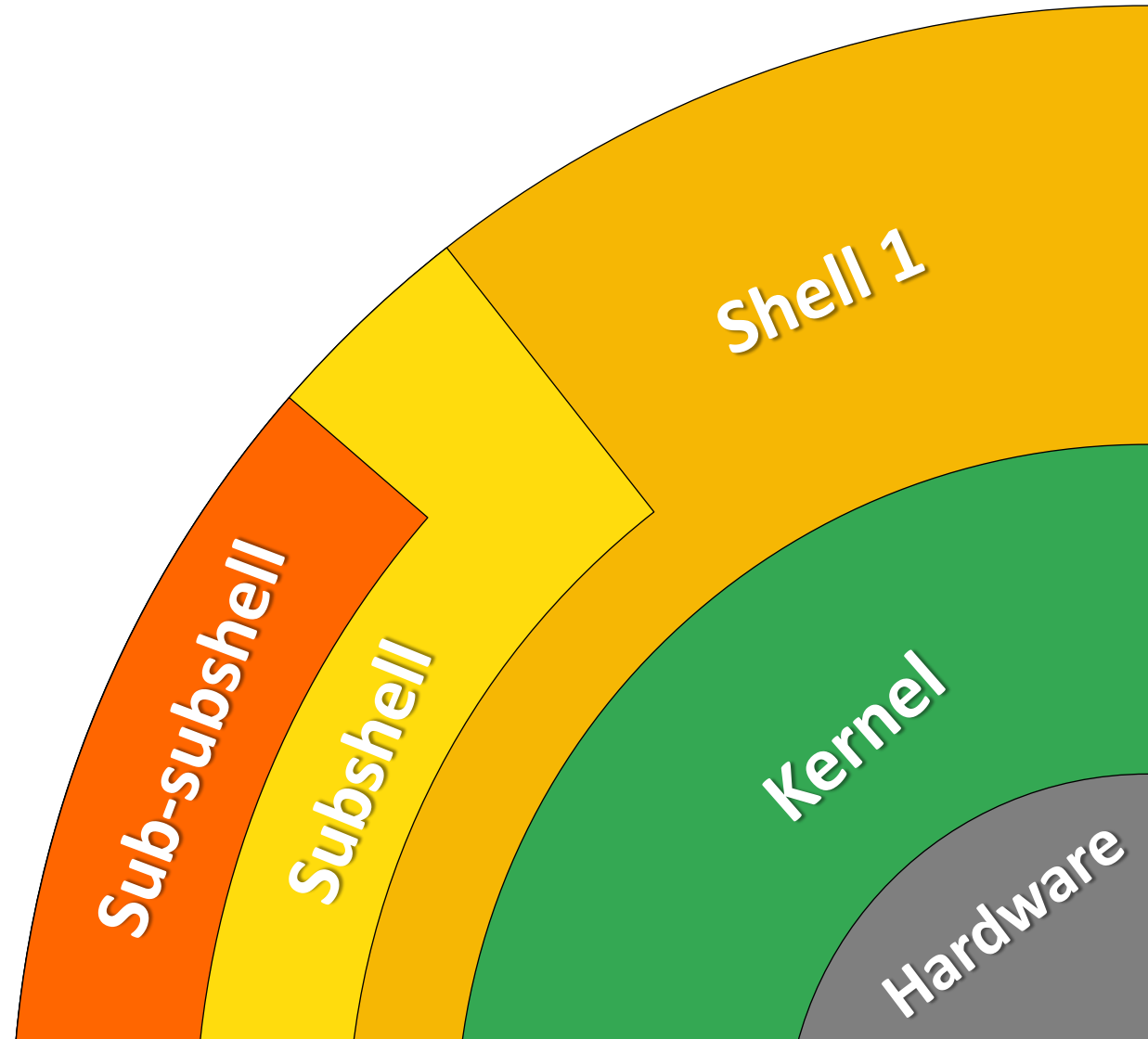
- 1) Interactive vs Non-interactive (Shell Script)
- 2) Basic Commands & Syntax
- 3) Variables
- 4) Arrays
- 5) Arithmetic Operations

3. Beyond Basics

- 1) Subshells
- 2) Flow Control
- 3) Advanced Text Processing Commands

4. BONUS: Where to Get Help

- **Definition:**
 - A **child process** of launched by an existing shell.
- **Similarity:**
 - **Still a Shell!**
(Everything we talked about works the same way!)
- **Difference:**
 - An **isolated** environment from its parent
(A “sandbox” Shell)

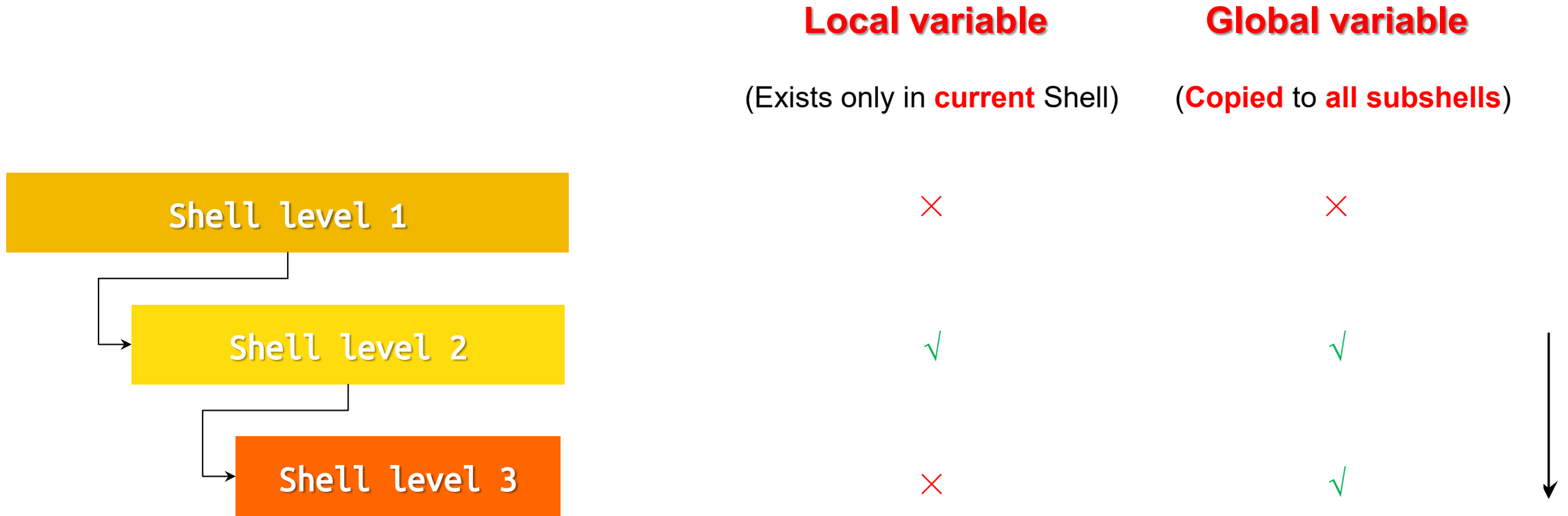


a) Launch a subshell

Method		Example	Remarks
1	Run a Shell script	<pre>\$./subshell.sh \$ bash subshell.sh</pre>	- Can launch different Shell types - Check subshell level: \$SHLVL
2	Explicitly launch an interactive subshell	<pre>\$ bash</pre>	
3	Use command grouping "(...)"	<pre>\$ (echo "I am in subshell!")</pre>	- Launches the same Shell type - Does NOT change \$SHLVL

- What does **NOT** launch a subshell?
 - **source** subshell.sh
 - Commonly used for **environment setting** scripts (You WANT it to set up current Shell)
 - source setenv.sh

b) Scope of variables



1. Introduction

- 1) What's Shell?
- 2) What can Shell do?

2. Basic Knowledge

- 1) Interactive vs Non-interactive (Shell Script)
- 2) Basic Commands & Syntax
- 3) Variables
- 4) Arrays
- 5) Arithmetic Operations

3. Beyond Basics

- 1) Subshells
- 2) Flow Control
- 3) Advanced Text Processing Commands

4. BONUS: Where to Get Help

2) Flow control

- a) Condition – **if** statement
- b) Loop – **for** loop
- c) Loop – **while** loop
- d) Functions

```
#!/bin/bash

# URL and file name
URL=$1
FILENAME=`basename $URL`

# Number of parts
PARTS=$SLURM_NTASKS
echo "Downloading in $PARTS parts..."

# Create temporary directory for parts
TEMP_DIR=$(mktemp -d)

# Calculate the range for each part
FILE_SIZE=$(curl -sIk $URL | awk '/Content-Length/ {print $2}' | tr -d '\r') # Get file size
PART_SIZE=$((FILE_SIZE / PARTS)) # Calculate part size
LAST_PART_SIZE=$((FILE_SIZE - PART_SIZE * (PARTS - 1))) # Calculate size of the last part

# Download each part concurrently
for ((i = 0; i < PARTS; i++)); do
    start=$((i * PART_SIZE))
    end=$((start + PART_SIZE - 1))

    if [[ $i -eq $((PARTS - 1)) ]]; then
        end=$((start + LAST_PART_SIZE - 1))
    fi

    curl -ks -o "$TEMP_DIR/part$i" --range "$start-$end" "$URL" &
done

# Wait for all downloads to finish
wait

# Merge the parts into a single file
for ((i = 0; i < PARTS; i++)); do
    cat "$TEMP_DIR/part$i" >> "data/$FILENAME"
done

# Clean up temporary directory
rm -rf "$TEMP_DIR"

echo "Download completed!"
```

a) Condition – **if** statement

- Optional: **elif** and **else**
- **Strict spaces** between "[]" and conditions
- Use double braces "[[]]" : More modern features (regular expressions, logic operators, etc.)

Syntax

```
if [ condition ]; then
    # Do something
elif [ condition 2 ] ; then
    # Do something
else
    # Do something else
fi
```

2) Flow control

a) Condition – **if** statement

Condition	Syntax
Equal to	[\$a -eq 0] # Integer [\$a == \$b] # String
Not equal to	[\$a -ne 0] # Integer [\$a != \$b] # String
Greater than	[\$a -gt 0] # Integer
Greater than or equal to	[\$a -ge 0] # Integer
Less than	[\$a -lt 0] # Integer
Less than or equal to	[\$a -le 0] # Integer
Zero length or null	[-z \$a] # String
Non zero length	[-n \$a] # String

2) Flow control

a) Condition – **if** statement

Condition	Syntax
File exists	[-e myfile]
File is a regular file	[-f myfile]
File is a directory	[-d /home/\$USER]
File is not zero size	[-s myfile]
File has read permission	[-r myfile]
File has write permission	[-w myfile]
File has execute permission	[-x myfile]

a) Condition – **if** statement

Condition	[]	[[]]
! (NOT)	[! -e myfile]	
&& (AND)	[-f myfile] && [-s myfile]	[[-f myfile && -s myfile]]
(OR)	[-f myfile1] [-f myfile2]	[[-f myfile1 -f myfile2]]

- Supported by **more Shells**.
- Use if you need **compatibility**.
- Best supported by **Bash**.
- Use if you need **versatility**.

b) Loop – **for** loop

- Do something for each element in an array.

Syntax

```
for arg in ${myAry[@]}  
do  
    # Do something  
done
```

b) Loop – **for** loop

Array	Example
User defined array	<pre>\$ myAry=("Alice" "Bob" "Charlie") \$ for arg in \${myAry[@]} ...</pre>
Shell generated sequence	<pre>\$ for arg in `seq 1 4` ...</pre>
Output of commands	<pre>\$ for arg in `ls \$HOME` ...</pre>

c) Loop – **while** loop

- Loop as long as **condition** is satisfied.
- Make sure there is an **escape condition** !
 - Otherwise the loop is doomed!

Syntax

```
while [ condition ]  
do  
    # Do something  
done
```

c) Loop – **while** loop

Example

```
$ counter=0
$ while [ $counter -lt 10 ]
do
    echo "Counter is now $counter"
    let counter++    # <- What does this do?
done
```

d) Functions

- A block of pre-defined code that can be reused.
- Passed **arguments** are accessed by:
 - **\$1, \$2, ... \$9, \${10}, ...**
 - **\$@** (All arguments)

Syntax

```
# Define
function_name () {
    # Do something
}

# Call, no "()"
function_name [ARG1] [ARG2]
```

d) Functions

Remarks	Example
All variables are global by default	<pre>\$ myFunc1 () { var="Bob" } \$ var="Alice"; myFunc1 ; echo \$var Bob</pre>
Local variables must be explicitly declared	<pre>\$ myFunc2 () { local var="Bob" } \$ var="Alice"; myFunc2 ; echo \$var Alice</pre>
Does NOT support return (Use global variable if needed)	<pre>\$ myAdd () { result=\$((\$1+\$2)) } \$ myAdd 10 20 ; echo \$result 30</pre>

- **Summary**
 - a) Condition – **if** statement
 - b) Loop – **for** loop
 - c) Loop – **while** loop
 - d) Functions

1. Introduction

- 1) What's Shell?
- 2) What can Shell do?

2. Basic Knowledge

- 1) Interactive vs Non-interactive (Shell Script)
- 2) Basic Commands & Syntax
- 3) Variables
- 4) Arrays
- 5) Arithmetic Operations

3. Beyond Basics

- 1) Subshells
- 2) Flow Control
- 3) Advanced Text Processing Commands

4. BONUS: Where to Get Help

3) Advanced Text Processing Commands

a) `grep` → search

b) `sed` → edit

3) Advanced Text Processing Commands

a) grep

- **Search** for patterns (formatted strings) in **input stream** (**files** & **pipe**)

Syntax

```
$ grep <options> <search pattern> <files>
```

3) Advanced Text Processing Commands

a) grep

- i. Basic functionality - Search for a string

Description	Example
Search for lines contain given string in a file	\$ grep "Sales" employee1.txt
Search for lines do NOT contain given string in a file	\$ grep -v "Sales" employee1.txt
Search all files for lines contain given string in the directory	\$ grep "Sales" *
List files that do NOT contain given string in the directory	\$ grep -L "Sales" *
Search for strings in a pipe	\$ squeue grep \$USER

3) Advanced Text Processing Commands

a) grep

ii. Useful options

Option	Description
-i	Ignore cases.
-r, -R	Search recursively.
-v	Invert match (return those do NOT match pattern)
-l	List names of the files that match the pattern.
-L	List names of the files that do NOT match the pattern.
-n	Print line number with output lines.
...	

3) Advanced Text Processing Commands

a) grep

iii. Pattern

- Can be as simple as strings.
- Can be **Regular Expression** (formatted strings to match beyond fixed strings).

a) grep

iii. Pattern

Metacharacter		Matches	Example
Anchor	^	Beginning of a line.	^Name (Beginning of a line followed by "Name")
	\$	End of a line.	Salary\$ ("Salary" followed by end of a line)
Substitution	.	Any single character	a.e (E.g., "age", "ame", "a#e", "a1e",...)
Repetition	*	Preceding char. repeats 0 or more times	50* (E.g., "5", "50", "500",...)
	+	Preceding char. repeats 1 or more times	50+ (E.g., "50", "500",...)
	?	Preceding char. repeats 0 or 1 times	50? (E.g., "5", "50")
	{n,m}	Preceding char. repeats n to m times	50{1,3} (E.g., "50", "500", "5000")
Or	[]	Any single character inside	[0-9] (E.g., any single number character)
	[^]	Any single character NOT inside	[^0-9] (E.g., any single character but a number)
		Either pattern	Sales Technology (E.g., "Sales" or "Technology")
...			

3) Advanced Text Processing Commands

b) sed

- A powerful “Stream editor” for **text transformation** on input stream (**files** & **pipe**)

Syntax

```
$ sed <options> <script> <files>
```

b) sed

i. Basic functionality (all patterns support regular expression)

Function	Usage	Description
Substitution	\$ sed 's/pattern/replacement/flags' file	For each line, replace matched “pattern” with “replacement”, and print out results.
	\$ sed 's/\$[0-9]*/\$9000/' employee2.txt	Replace only the first match of each line.
	\$ sed 's/\$[0-9]*/\$9000/g' employee2.txt	“Greedy” mode, replace all matches of each line.
Deletion	\$ sed '/pattern/d' file	Delete lines with matched pattern, and print results.
	\$ sed '/Sales/d' employee2.txt	Delete all lines matches “Sales”.
	\$ sed '2,4d' employee2.txt	Remove line 2 through 4.
Insertion	\$ sed '/pattern/ i\nnewline' file # Insert before \$ sed '/pattern/ a\nnewline' file # Insert after	Insert / Append new line at specific location, and print results.
	\$ sed '/Alice/ i\nnewline'	Insert before lines matches “Alice”.
	\$ sed '3 a\nnewline'	Append to line 3.

...

b) sed

ii. Other common usage

Usage	Example	Description
\$ sed -i <script> file	\$ sed -i 's/\${0-9}*/\$9000/' employee2.txt	Change file in-place instead of printing results.
\$ sed -e <script1> -e <script2> file	\$ sed -e 's/\${0-9}*/\$9000/' \ -e 's/Rep/Assistant/' employee2.txt	Execute multiple scripts.
\$ cmd sed <options> <script>	\$ conda env list sed '/^#/d'	Parsing piped output instead of file.

3) Advanced Text Processing Commands

- **Summary**
 - “**grep** searches, **sed** edits.”

1. Introduction

- 1) What's Shell?
- 2) What can Shell do?

2. Basic Knowledge

- 1) Interactive vs Non-interactive (Shell Script)
- 2) Basic Commands & Syntax
- 3) Variables
- 4) Arrays
- 5) Arithmetic Operations

3. Beyond Basics

- 1) Subshells
- 2) Flow Control
- 3) Advanced Text Processing Commands

4. **BONUS: Where to Get Help**

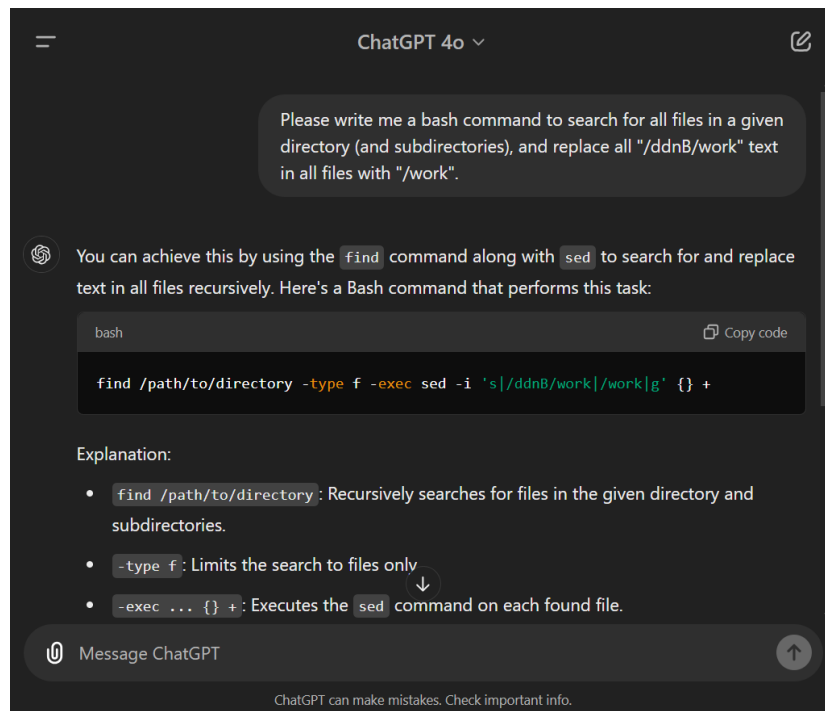
- I need more help with Shell scripting. Where do I get help?

1) Contact HPC User Services

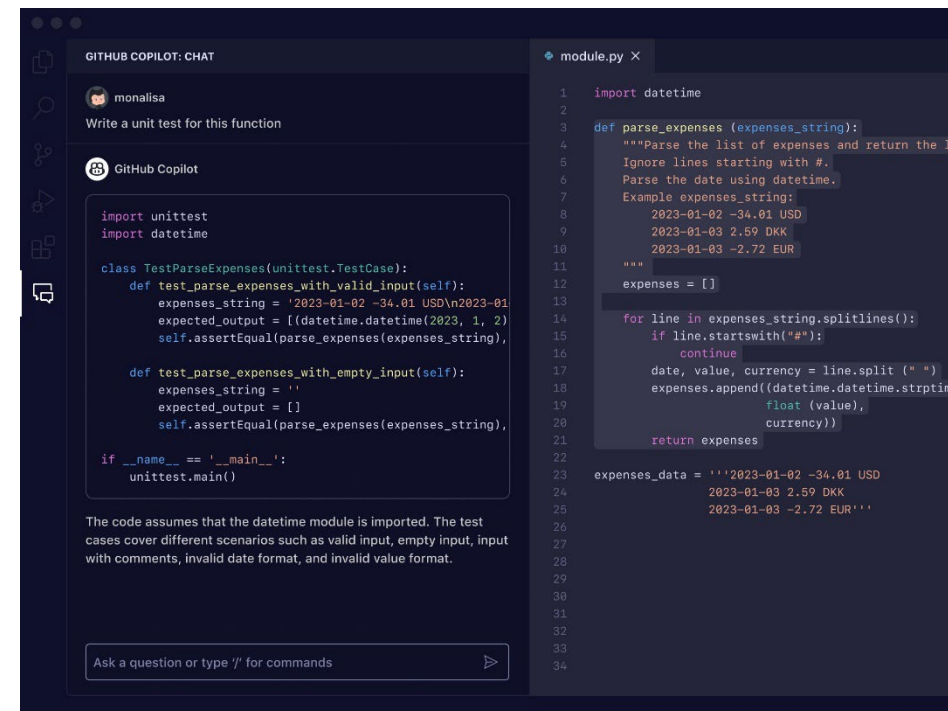
- Email Help Ticket: sys-help@loni.org
- Telephone Help Desk: +1 (225) 578-0900

- I need more help with Shell scripting. Where do I get help?

2) Generative AI



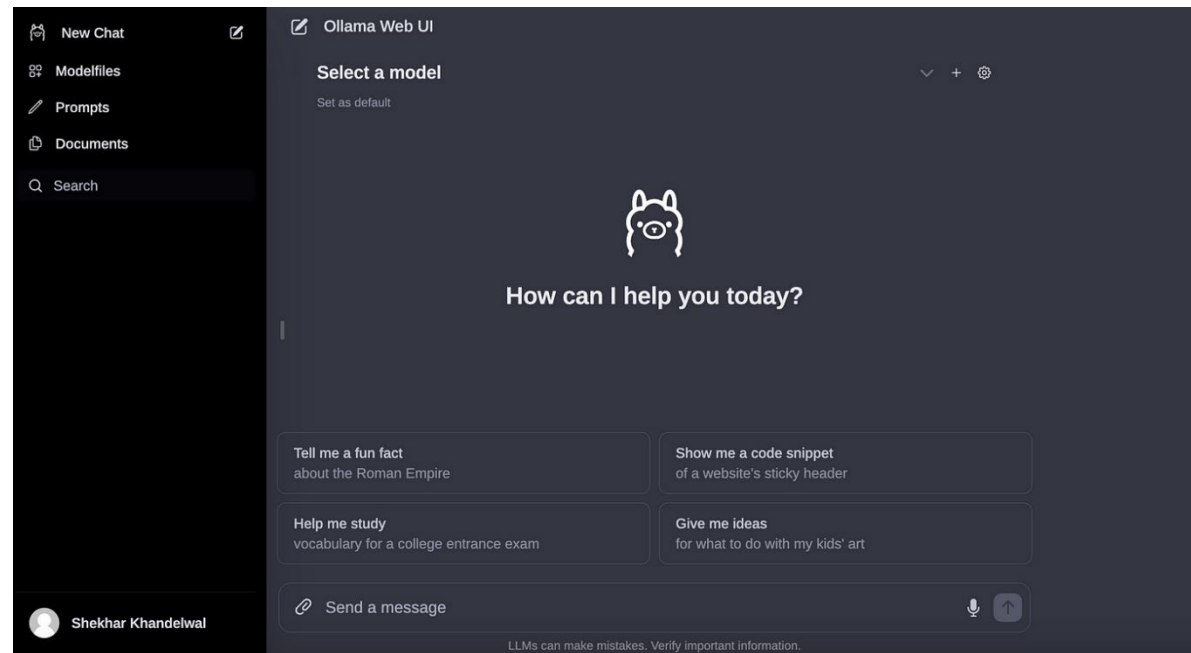
ChatGPT



GitHub Copilot

- I need more help with Shell scripting. Where do I get help?

2) Generative AI

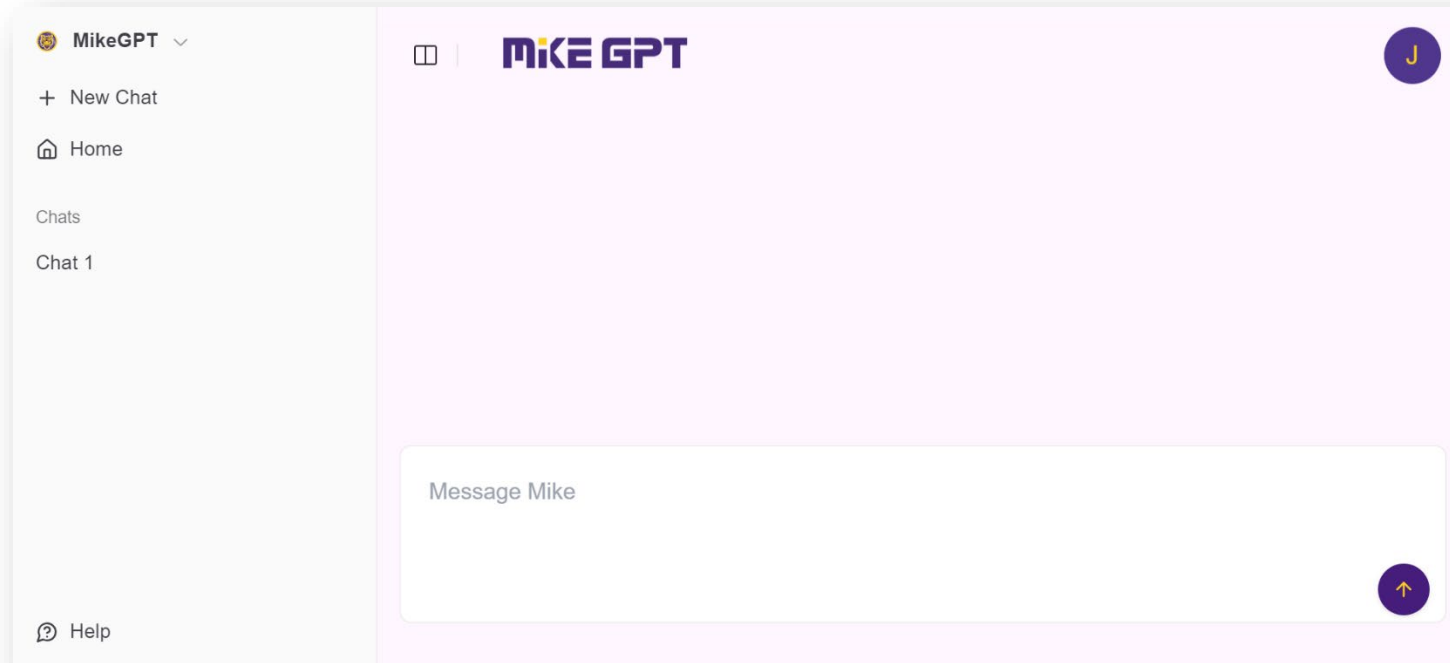


Ollama

(Deploy open-source LLMs on local machine)

- I need more help with Shell scripting. Where do I get help?

2) Generative AI



MikeGPT

(LSU's own GPT chat box! Trained with LSU public realm data)

[1] <https://mikegpt.lsu.edu/>



- Why I recommend generative AI for Shell scripting?

Shell scripting

- Something you may not be too familiar with, but have to work with on a daily basis for HPC jobs.
- Just need a **quick and dirty** solution!
- I don't really care about reliable sources.

Generative AI

- Easy to get an answer without extensive knowledge.
- Good at **giving quick and dirty answers**.
- Bad at giving reliable sources.

- **Steps**

- 1) Find out what you want to do and ask AI the right questions

- Try these examples (think about how to do it first, then ask AI):

- a) Change all text `"/ddnB/work"` to `"/work"` in all files in folder `"~/mycode/"` and subfolders.
- b) In a `","` separated .csv database, delete all columns starting from the 10th, and add an index column as the first column.
- c) Run executable `"myexec"` with `"input.txt"` as standard input, but replacing all `"TIME"` text in `"input.txt"` with current timestamp generated by `"date"`.

- **Steps**

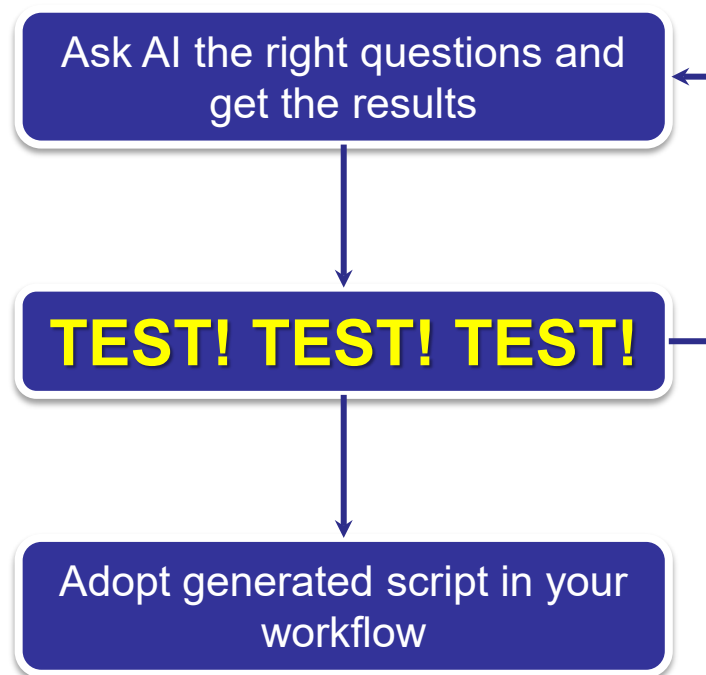
- 2) **TEST! TEST! TEST!**

- AI generated scripts may not work right away!
 - Test it in a **safe** & **isolated** environment (a sandbox) first, especially your script is something destructive!
 - You may need to come back and ask AI to revise your script.

- **Steps**

- 3) Adopt in your workflow

- **Steps**



1. Introduction

- 1) What's Shell?
- 2) What can Shell do?

2. Basic Knowledge

- 1) Interactive vs Non-interactive (Shell Script)
- 2) Basic Commands & Syntax
- 3) Variables
- 4) Arrays
- 5) Arithmetic Operations

3. Beyond Basics

- 1) Subshells
- 2) Flow Control
- 3) Advanced Text Processing Commands

4. BONUS: Where to Get Help

- Take-home message:

Anything you wish to run faster, you should not use shell!

When **NOT to use Shell scripting?**

- Heavy calculation!

When **TO use Shell scripting?**

- Automating job workflow
- Pre-processing / Post-processing
- ...

- **Contact user services**
 - Email Help Ticket: sys-help@loni.org
 - Telephone Help Desk: +1 (225) 578-0900