Basic Shell Scripting

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

Louisiana State University
Baton Rouge
11 February 2026

Outlines

- **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

Outlines

Outlines

- **Example and exercises:**
  - http://www.hpc.lsu.edu/training/weekly-materials/Downloads/ShellScripting.zip

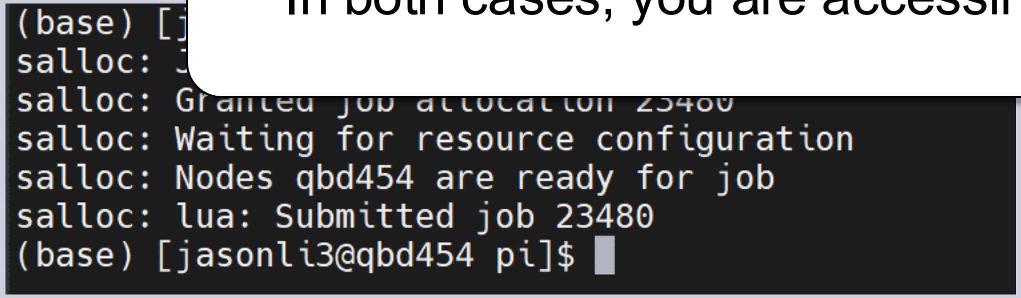
Outlines

INFORMATION TECHNOLOGY SERVICES

1) What's Shell?

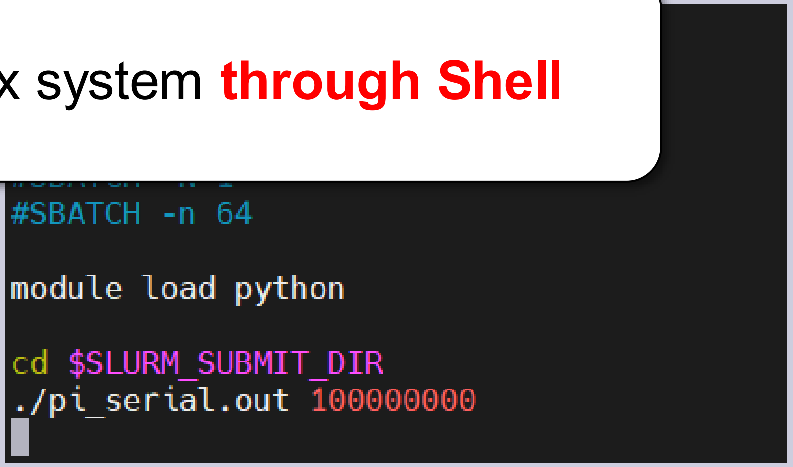
- **Previously in HPC User Environment 2…**
  - Two types of jobs

1) Interactive job



```
(base) [j
salloc: J
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]$ █
```

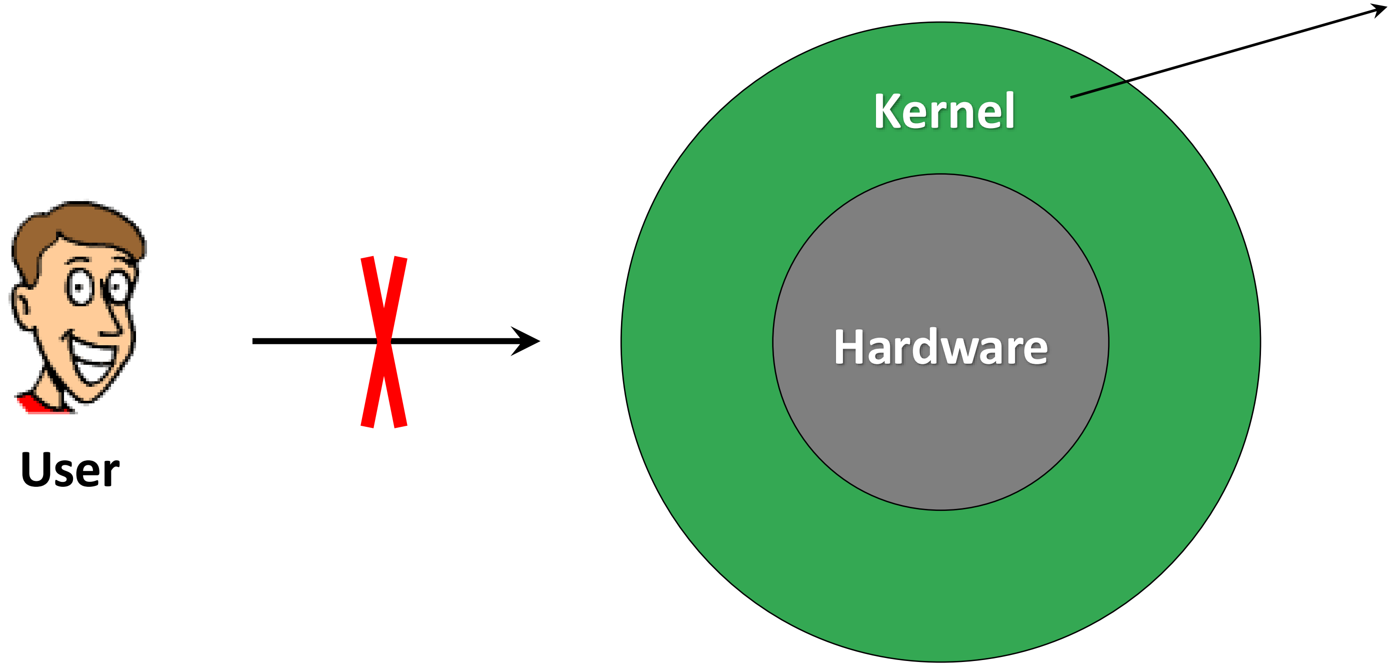
2) Batch job



```
#SBATCH -N 1
#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**

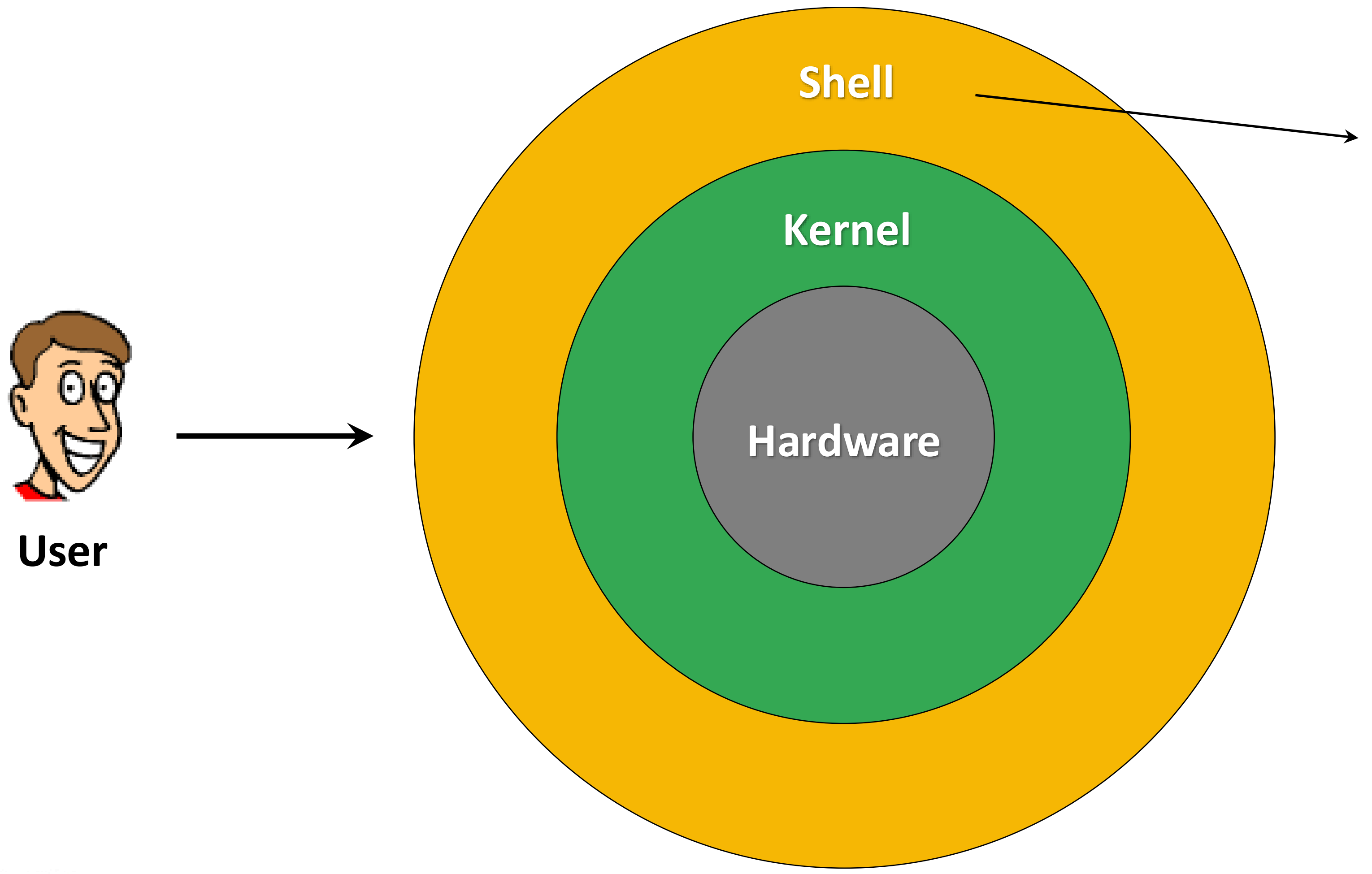
1) What's Shell?



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

→ **Speaks: Machine**

1) What's Shell?

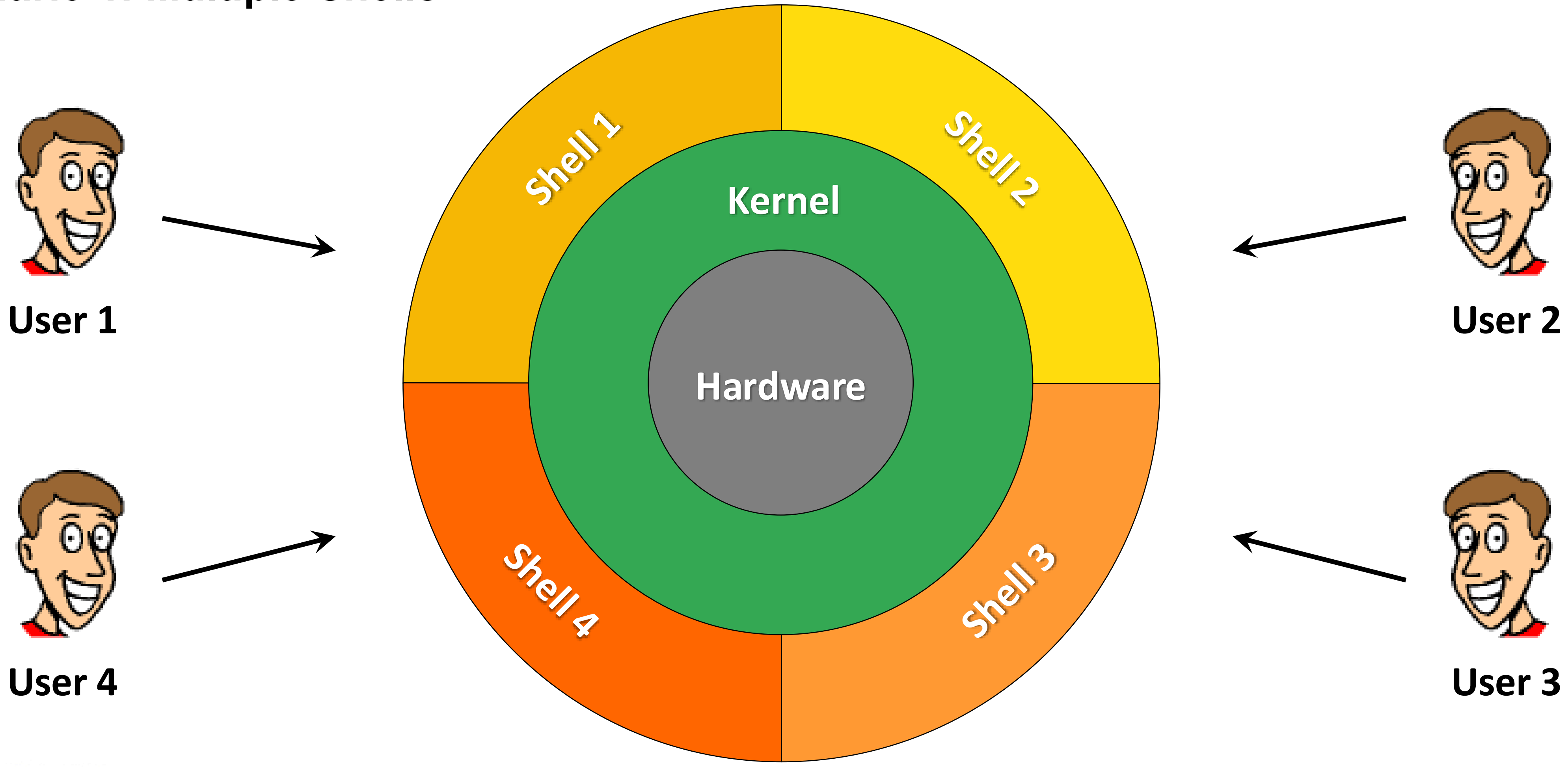


- Command execution
- Scripting
- …

→ **Speaks: Human**

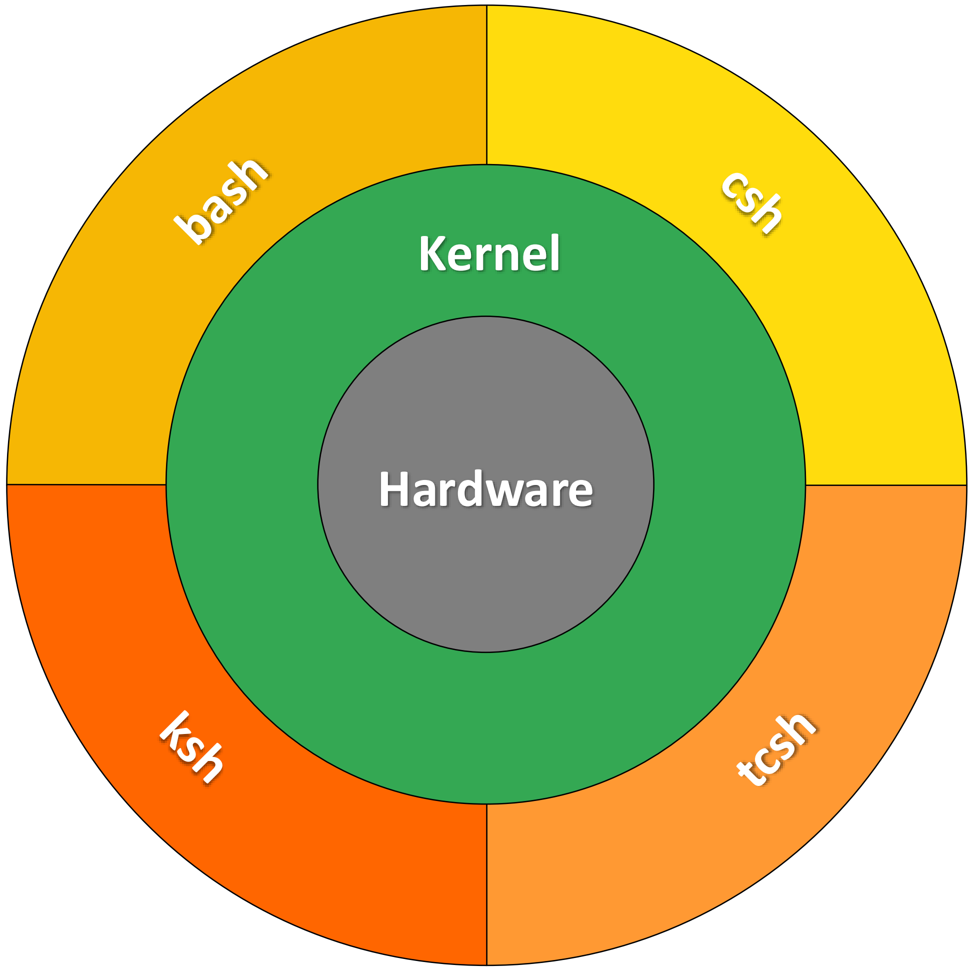
1) What's Shell?

- **Scenario 1: Multiple Shells**



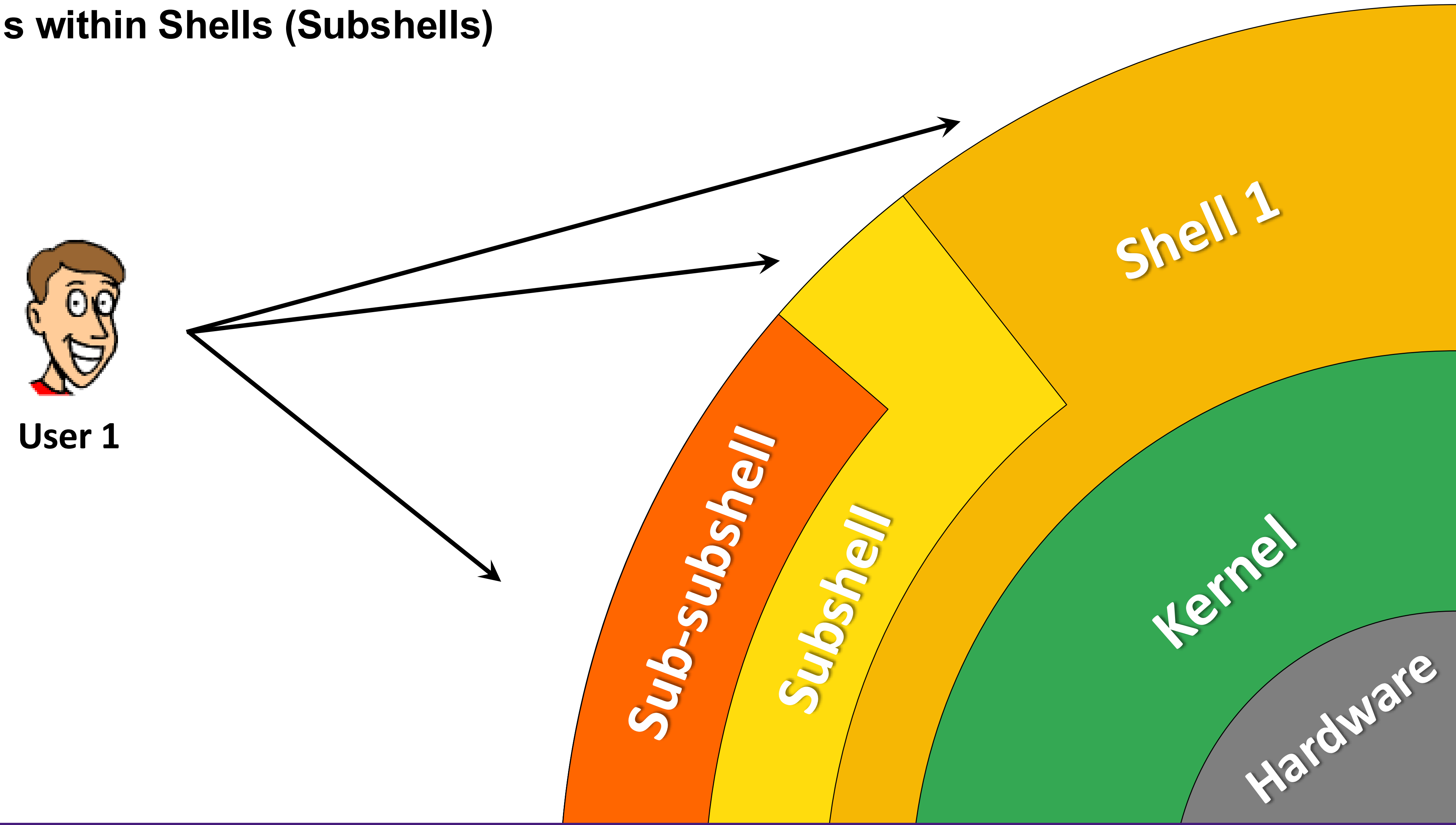
1) What's Shell?

- **Scenario 1: Multiple Shells**



1) What's Shell?

- **Scenario 2: Shells within Shells (Subshells)**



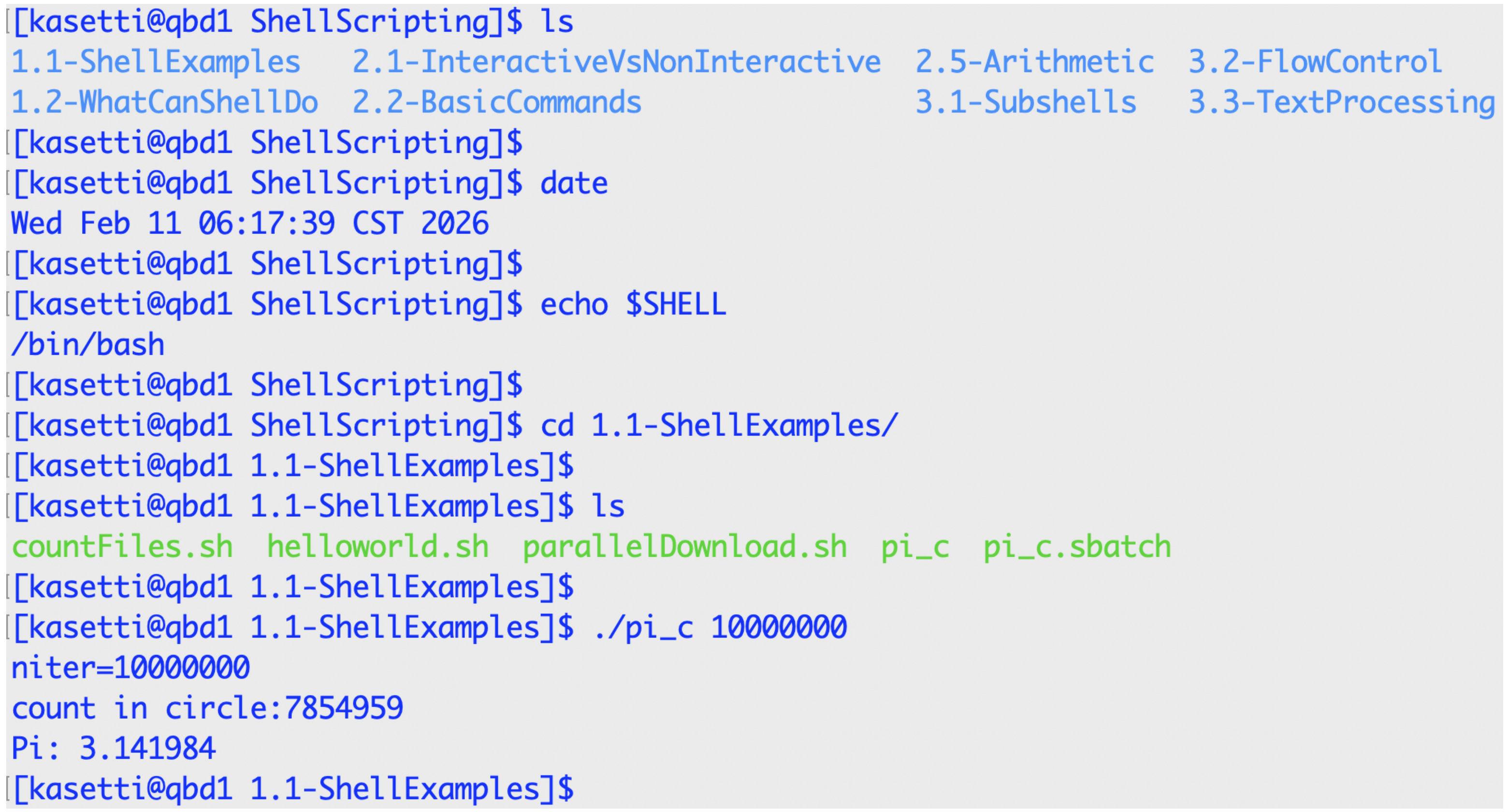
1) What's Shell?

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

Outlines

2) What can Shell do?

- **Shell can do this …**
  - Typing commands one by one



```
[kasetti@qbd1 ShellScripting]$ ls
1.1-ShellExamples   2.1-InteractiveVsNonInteractive  2.5-Arithmetic  3.2-FlowControl
1.2-WhatCanShellDo  2.2-BasicCommands                3.1-Subshells   3.3-TextProcessing
[kasetti@qbd1 ShellScripting]$
[kasetti@qbd1 ShellScripting]$ date
Wed Feb 11 06:17:39 CST 2026
[kasetti@qbd1 ShellScripting]$
[kasetti@qbd1 ShellScripting]$ echo $SHELL
/bin/bash
[kasetti@qbd1 ShellScripting]$
[kasetti@qbd1 ShellScripting]$ cd 1.1-ShellExamples/
[kasetti@qbd1 1.1-ShellExamples]$
[kasetti@qbd1 1.1-ShellExamples]$ ls
countFiles.sh  helloworld.sh  parallelDownload.sh  pi_c  pi_c.sbatch
[kasetti@qbd1 1.1-ShellExamples]$
[kasetti@qbd1 1.1-ShellExamples]$ ./pi_c 10000000
niter=10000000
count in circle:7854959
Pi: 3.141984
[kasetti@qbd1 1.1-ShellExamples]$
```

2) What can Shell do?

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

```
#!/bin/bash

# Variables
DATA_DIR=$1
MAX_FILES=3
count=0

echo "Starting the script..."
echo "Looking for files in $DATA_DIR"

# Check if directory exists
if [[ ! -d $DATA_DIR ]]; then
  echo "Directory $DATA_DIR does not exist"
  exit 1
fi

# Check if any .txt files exist
if [[ -z $(find "$DATA_DIR" -maxdepth 1 -name "*.txt" -print -quit) ]]; then
  echo "No .txt files found"
  exit 0
fi

# Loop over files
for file in "$DATA_DIR"/*.txt; do
  echo "Found file: $file"

  count=$((count + 1))

  # Stop after a few files
  if [[ $count -ge $MAX_FILES ]]; then
    echo "Reached the limit of $MAX_FILES files"
    break
  fi
done

echo "Script finished. Processed $count files."
```

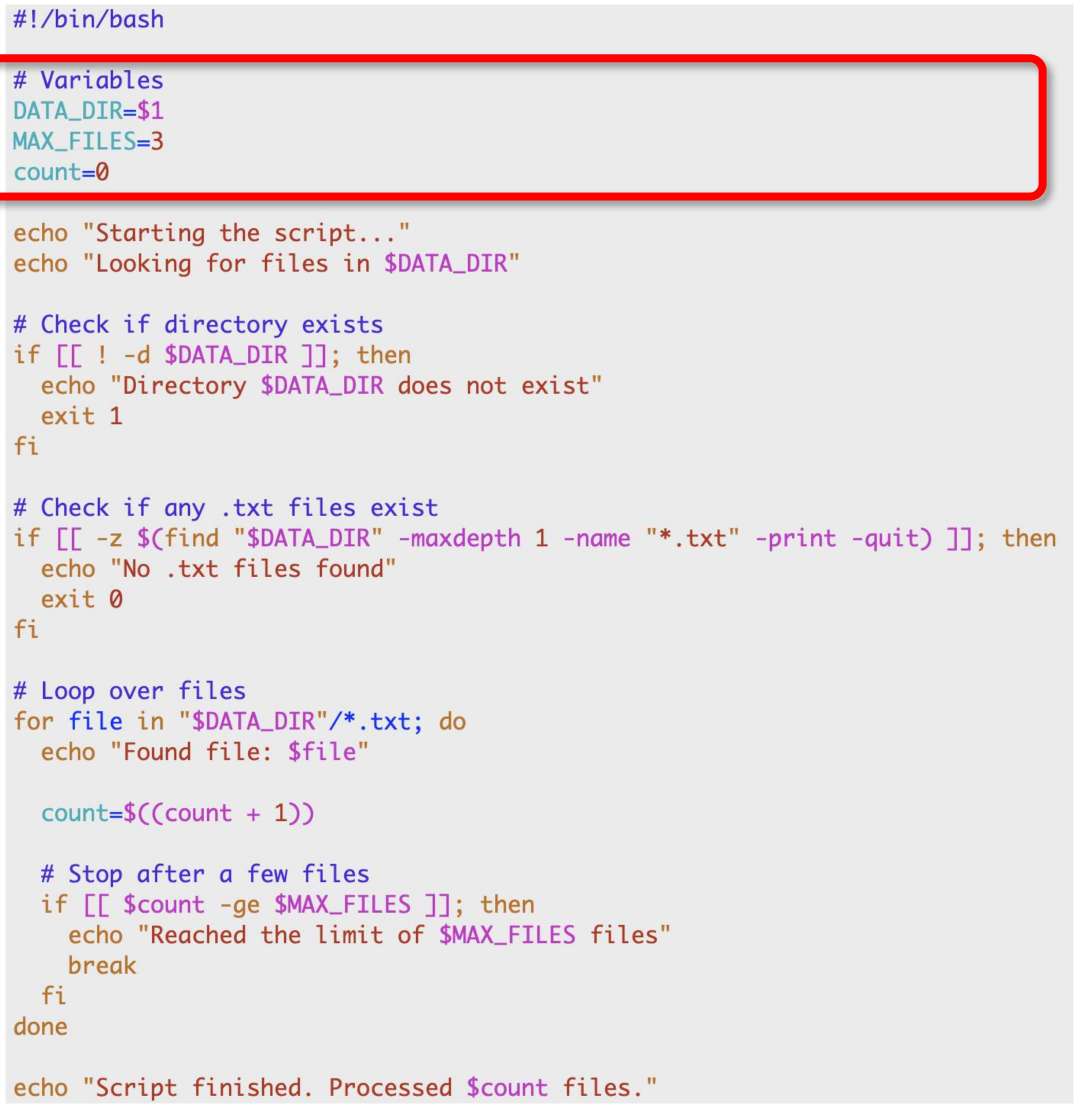
[1] *ShellScripting/1.2-WhatCanShellDo/parallelDownload.sh*

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

# Variables
DATA_DIR=$1
MAX_FILES=3
count=0

echo "Starting the script..."
echo "Looking for files in $DATA_DIR"

# Check if directory exists
if [[ ! -d $DATA_DIR ]]; then
  echo "Directory $DATA_DIR does not exist"
  exit 1
fi

# Check if any .txt files exist
if [[ -z $(find "$DATA_DIR" -maxdepth 1 -name "*.txt" -print -quit) ]]; then
  echo "No .txt files found"
  exit 0
fi

# Loop over files
for file in "$DATA_DIR"/*.txt; do
  echo "Found file: $file"

  count=$((count + 1))

  # Stop after a few files
  if [[ $count -ge $MAX_FILES ]]; then
    echo "Reached the limit of $MAX_FILES files"
    break
  fi
done

echo "Script finished. Processed $count files."
```

[1] *ShellScripting/1.2-WhatCanShellDo/parallelDownload.sh*

2) What can Shell do?

- **Take a closer look at this:**

```
#!/bin/bash

# Variables
DATA_DIR=$1
MAX_FILES=3
count=0

echo "Starting the script..."
echo "Looking for files in $DATA_DIR"

# Check if directory exists
if [[ ! -d $DATA_DIR ]]; then
  echo "Directory $DATA_DIR does not exist"
  exit 1
fi

# Check if any .txt files exist
if [[ -z $(find "$DATA_DIR" -maxdepth 1 -name "*.txt" -print -quit) ]]; then
  echo "No .txt files found"
  exit 0
fi

# Loop over files
for file in "$DATA_DIR"/*.txt; do
  echo "Found file: $file"

  count=$((count + 1))

  # Stop after a few files
  if [[ $count -ge $MAX_FILES ]]; then
    echo "Reached the limit of $MAX_FILES files"
    break
  fi
done

echo "Script finished. Processed $count files."
```

[1] *ShellScripting/1.2-WhatCanShellDo/parallelDownload.sh*

2) What can Shell do?

- **Take a closer look at this:**

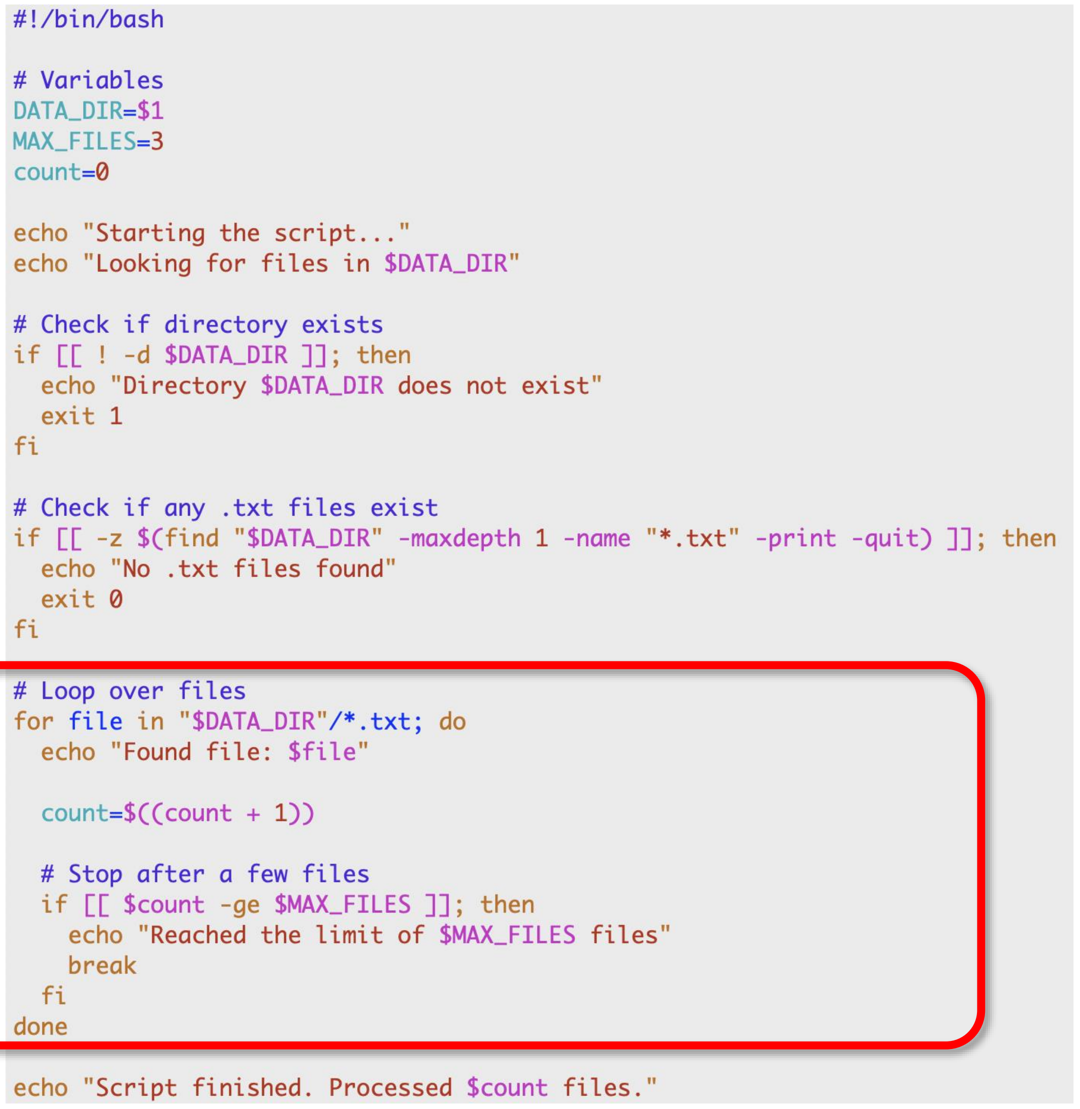
```
#!/bin/bash

# Variables
DATA_DIR=$1
MAX_FILES=3
count=0

echo "Starting the script..."
echo "Looking for files in $DATA_DIR"

# Check if directory exists
if [[ ! -d $DATA_DIR ]]; then
  echo "Directory $DATA_DIR does not exist"
  exit 1
fi

# Check if any .txt files exist
if [[ -z $(find "$DATA_DIR" -maxdepth 1 -name "*.txt" -print -quit) ]]; then
  echo "No .txt files found"
  exit 0
fi

# Loop over files
for file in "$DATA_DIR"/*.txt; do
  echo "Found file: $file"

  count=$((count + 1))

  # Stop after a few files
  if [[ $count -ge $MAX_FILES ]]; then
    echo "Reached the limit of $MAX_FILES files"
    break
  fi
done

echo "Script finished. Processed $count files."
```

[1] *ShellScripting/1.2-WhatCanShellDo/parallelDownload.sh*

2) What can Shell do?

- **Take a closer look at this:**

Isn't it basically a programming language?



```bash
#!/bin/bash

# Variables
DATA_DIR=$1
MAX_FILES=3
count=0

echo "Starting the script..."
echo "Looking for files in $DATA_DIR"

# Check if directory exists
if [[ ! -d $DATA_DIR ]]; then
  echo "Directory $DATA_DIR does not exist"
  exit 1
fi

# Check if any .txt files exist
if [[ -z $(find "$DATA_DIR" -maxdepth 1 -name "*.txt" -print -quit) ]]; then
  echo "No .txt files found"
  exit 0
fi

# Loop over files
for file in "$DATA_DIR"/*.txt; do
  echo "Found file: $file"

  count=$((count + 1))

  # Stop after a few files
  if [[ $count -ge $MAX_FILES ]]; then
    echo "Reached the limit of $MAX_FILES files"
    break
  fi
done

echo "Script finished. Processed $count files."
```

[1] *ShellScripting/1.2-WhatCanShellDo/parallelDownload.sh*

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

```python
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)
```

Shell

```bash
find ~/mycode/ -type f -exec sed -i 's|/ddnB/work|/work|g' {} +
```

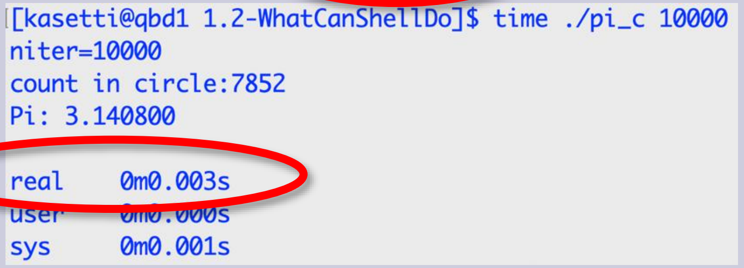
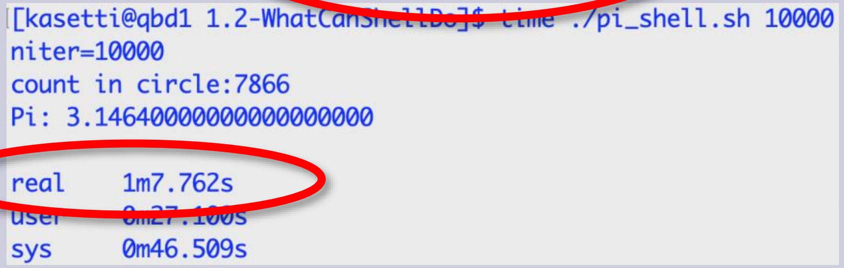
[1] *ShellScripting/1.2-WhatCanShellDo/pathSwap.py*
[2] *ShellScripting/1.2-WhatCanShellDo/pathSwap.sh*

2) What can Shell do?

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

– Shell is **highly inefficient** for heavy calculation!

- *Example*: Try the pi calculation codes in folder "`ShellScripting/1.2-WhatCanShellDo/`":

C	Shell
`$ ./pi_c 10000`	`$ ./pi_shell.sh 10000`
	

```
[kasetti@qbd1 1.2-WhatCanShellDo]$ time ./pi_c 10000
niter=10000
count in circle:7852
Pi: 3.140800

real    0m0.003s
user    0m0.000s
sys     0m0.001s
```

```
[kasetti@qbd1 1.2-WhatCanShellDo]$ time ./pi_shell.sh 10000
niter=10000
count in circle:7866
Pi: 3.14640000000000000000

real    1m7.762s
user    0m27.100s
sys     0m46.509s
```

[1] *ShellScripting/1.2-WhatCanShellDo/pi_c*
[2] *ShellScripting/1.2-WhatCanShellDo/pi_shell.sh*

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.)

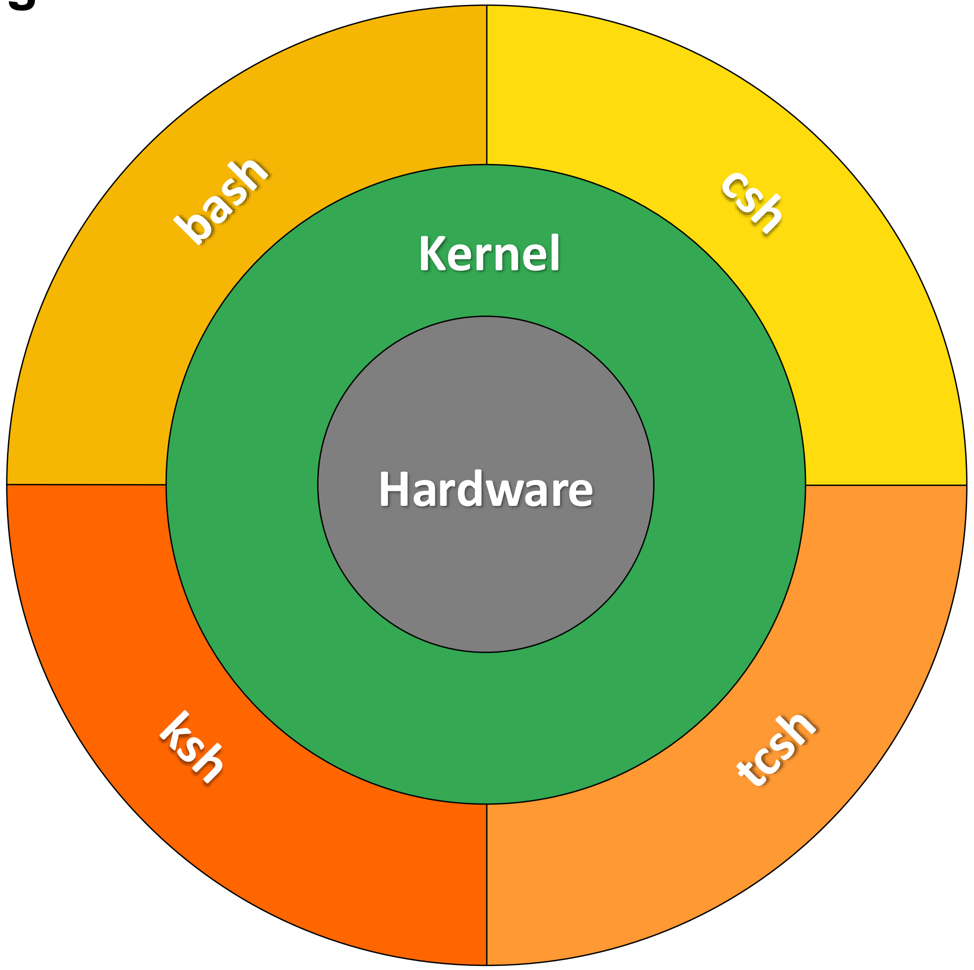
- **Goal of this training:**

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**.

Outlines

Before we continue…

- Remember we had this figure…



Before we continue…

- **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

Before we continue…

- **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)
  - …

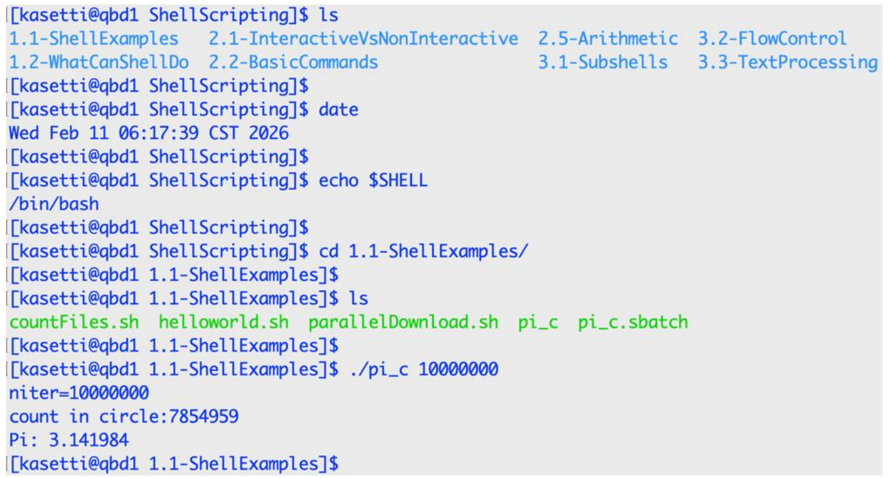
- Default Shell on all clusters
- Will only talk about it today

Outlines

1) Interactive vs Non-interactive Shell

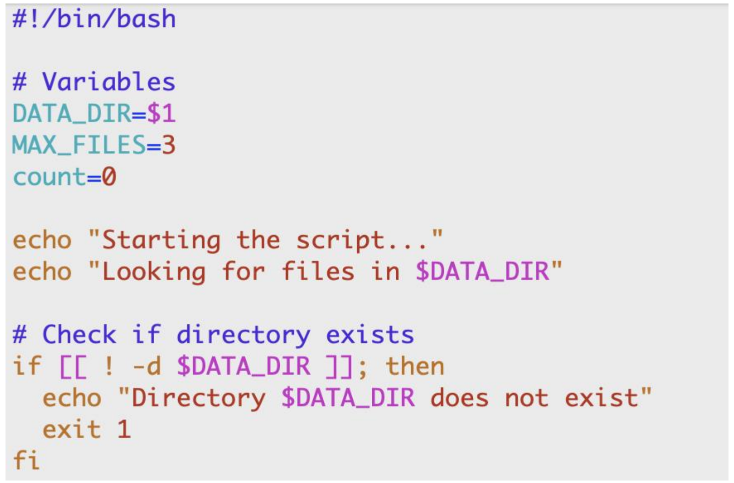
a) Two ways to access Shell

Interactive



```
[kasetti@qbd1 ShellScripting]$ ls
1.1-ShellExamples   2.1-InteractiveVsNonInteractive  2.5-Arithmetic  3.2-FlowControl
1.2-WhatCanShellDo  2.2-BasicCommands                3.1-Subshells   3.3-TextProcessing
[kasetti@qbd1 ShellScripting]$
[kasetti@qbd1 ShellScripting]$ date
Wed Feb 11 06:17:39 CST 2026
[kasetti@qbd1 ShellScripting]$
[kasetti@qbd1 ShellScripting]$ echo $SHELL
/bin/bash
[kasetti@qbd1 ShellScripting]$
[kasetti@qbd1 ShellScripting]$ cd 1.1-ShellExamples/
[kasetti@qbd1 1.1-ShellExamples]$
[kasetti@qbd1 1.1-ShellExamples]$ ls
countFiles.sh  helloworld.sh  parallelDownload.sh  pi_c  pi_c.sbatch
[kasetti@qbd1 1.1-ShellExamples]$
[kasetti@qbd1 1.1-ShellExamples]$ ./pi_c 10000000
niter=10000000
count in circle:7854959
Pi: 3.141984
[kasetti@qbd1 1.1-ShellExamples]$
```

Non-interactive



```
#!/bin/bash

# Variables
DATA_DIR=$1
MAX_FILES=3
count=0

echo "Starting the script..."
echo "Looking for files in $DATA_DIR"

# Check if directory exists
if [[ ! -d $DATA_DIR ]]; then
  echo "Directory $DATA_DIR does not exist"
  exit 1
fi
```

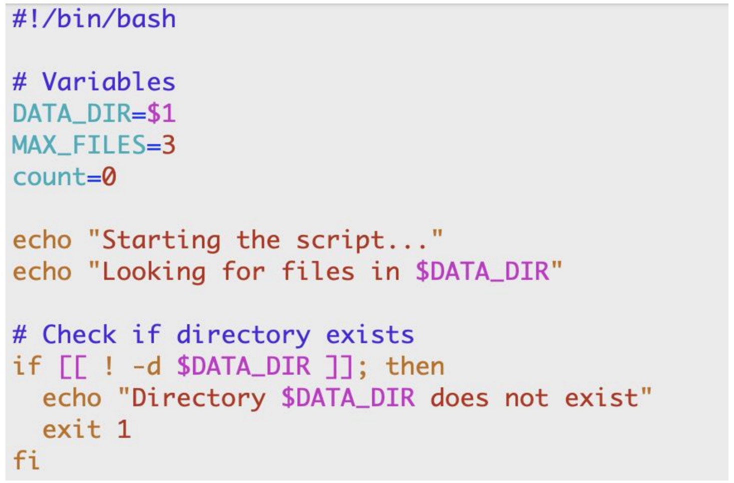
1) Interactive vs Non-interactive Shell

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

# Variables
DATA_DIR=$1
MAX_FILES=3
count=0

echo "Starting the script..."
echo "Looking for files in $DATA_DIR"

# Check if directory exists
if [[ ! -d $DATA_DIR ]]; then
  echo "Directory $DATA_DIR does not exist"
  exit 1
fi
```

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

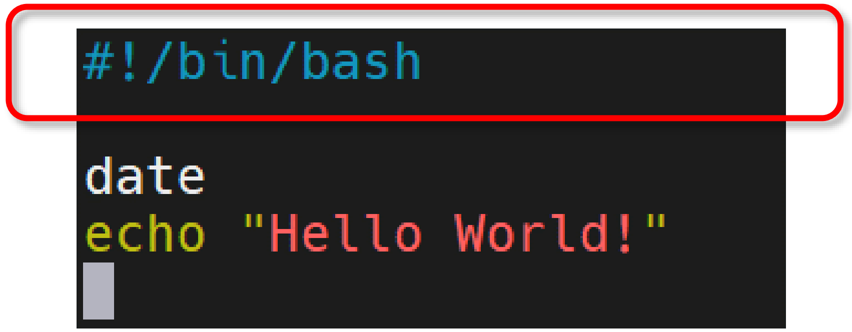
a) Two ways to access Shell

$$\begin{tikzcd}
\text{Interactive} & & \text{Non-interactive} \\
& \text{Shell scripting works the same way in both! *} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

* 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!"
```

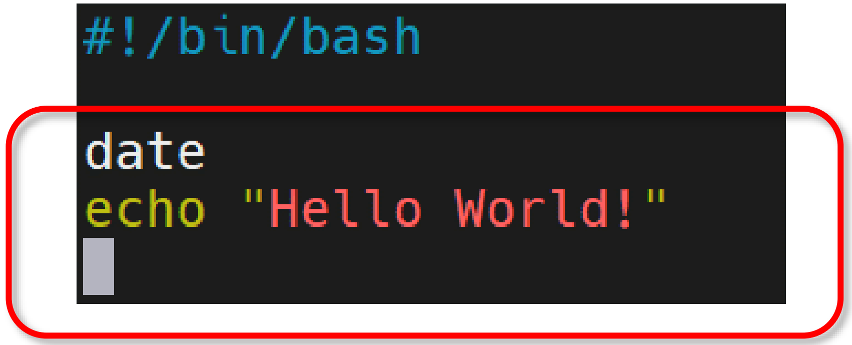
" ***Shebang*** " -

- Shell to run this script with

[1] *ShellScripting/2.1-InteractiveVsNonInteractive/helloworld.sh*

1) Interactive vs Non-interactive Shell

b) How to write a Shell script



```
#!/bin/bash

date
echo "Hello World!"
```

Commands to run

[1] *ShellScripting/2.1-InteractiveVsNonInteractive/helloworld.sh*

1) Interactive vs Non-interactive Shell

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`
		 `$ csh 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

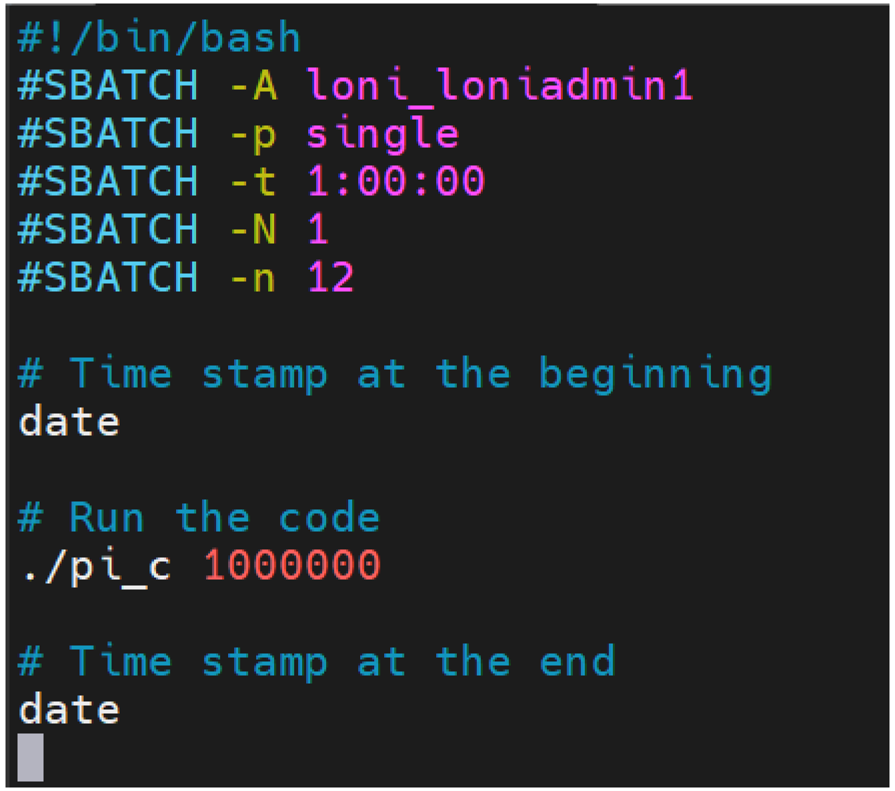
Run "`chmod u+x [filename]`" if file is not executable

[1] *ShellScripting/2.1-InteractiveVsNonInteractive/helloworld.sh*

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] *ShellScripting/2.1-InteractiveVsNonInteractive/pi_c.sbatch*

Outlines

2) Basic Commands & Syntax

a) Basic commands

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

[1] https://www.hpc.lsu.edu/training/archive/tutorials.php

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.	`$ squeue -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] *ShellScripting/2.2-BasicCommands/testoutput*
[2] *ShellScripting/2.2-BasicCommands/testinput*

Outlines

3) Variables

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"`
	`$ myout="/work/jasonli3/out.txt"`
		`$ $mycmd > $myout`	

– **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?

3) Variables

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!

3) 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

3) Variables

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 …)

3) Variables

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.**
…		

[1] https://www.hpc.lsu.edu/docs/slurm.php

3) Variables

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`

Outlines

4) Arrays

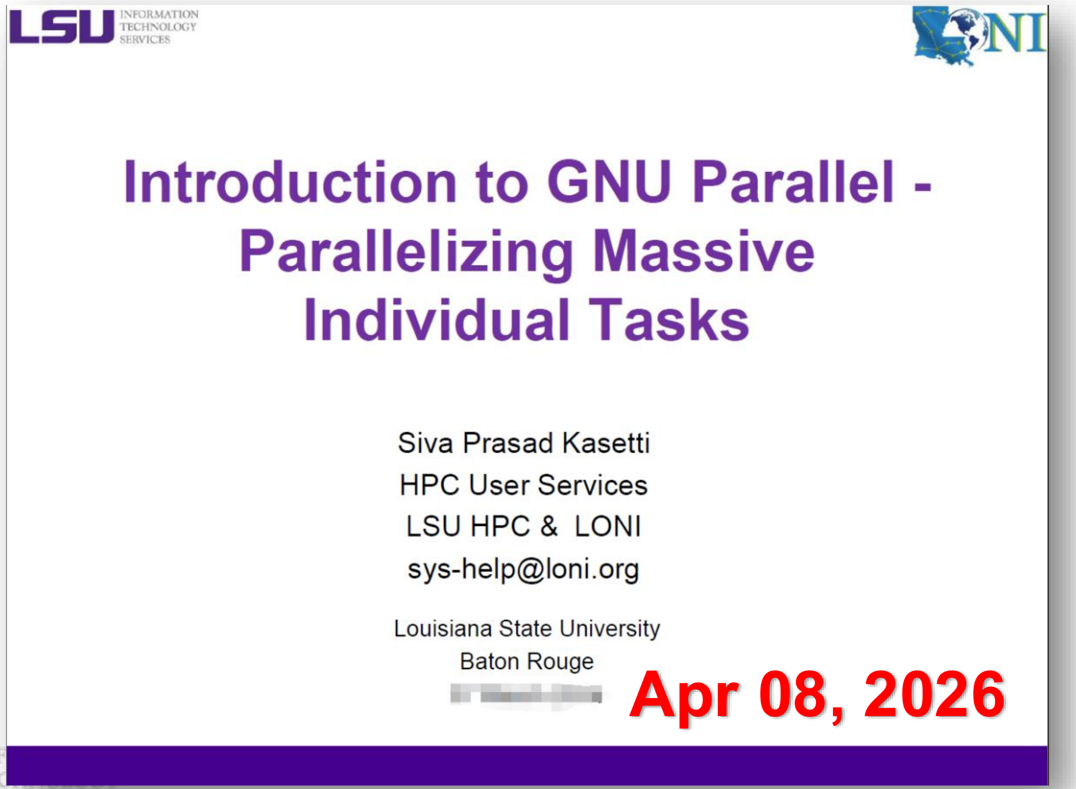
- **A collection of multiple values**
  - Basic logic very similar to "arrays" in any other language, **with some twists**!
    - Each element is accessed by **index**
    - Index starts with **0**

	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[@]}`

4) Arrays

- **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
[redacted] Apr 08, 2026

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

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

Outlines

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.

5) Arithmetic Operations

- **What does NOT work:**

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

[1] *ShellScripting/2.5-Arithmetic/bcExample.txt*

5) Arithmetic Operations

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

	Method	Example	Remarks
1	`$((…))` (Most common)	`$ echo $(($a/3+2))`	• Evaluate **everything inside** the braces. • **Integers** only!

2	`let` (Slightly more advanced)	`$ let b=$a/3+2`
		`$ let b=a/3+2`
		`$ let b++`
			• Evaluate **assignment** w/ arithmetic calculation.
			• "`$`" can be emitted.
			• **Integers** only!

3	`expr` (Legacy, most limited)	`$ expr $a / 3 + 2`	• Strictly limited to "`ARG1 OPERATION ARG2`" format. • **Integers** only!

4	`bc` (Most powerful)	`$ bc`
		`scale=3`
		`a=10;a/3+2`
		`$ bc < bcExample.txt`
		`$ echo "$a/2+3" | bc`
			• **Interactive** and **non-interactive** mode.
			• Does **NOT** support Shell syntax (namely, "`$`" for variables).
			• Unassigned variables treated as `0`.
			• `scale` variable determines number of decimals.
			• Supports **float number**!

[1] *ShellScripting/2.5-Arithmetic/bcExample.txt*

Summary

- **In this section, we talked about:**

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

Break

- Get some water
- Use restroom
- Ask questions

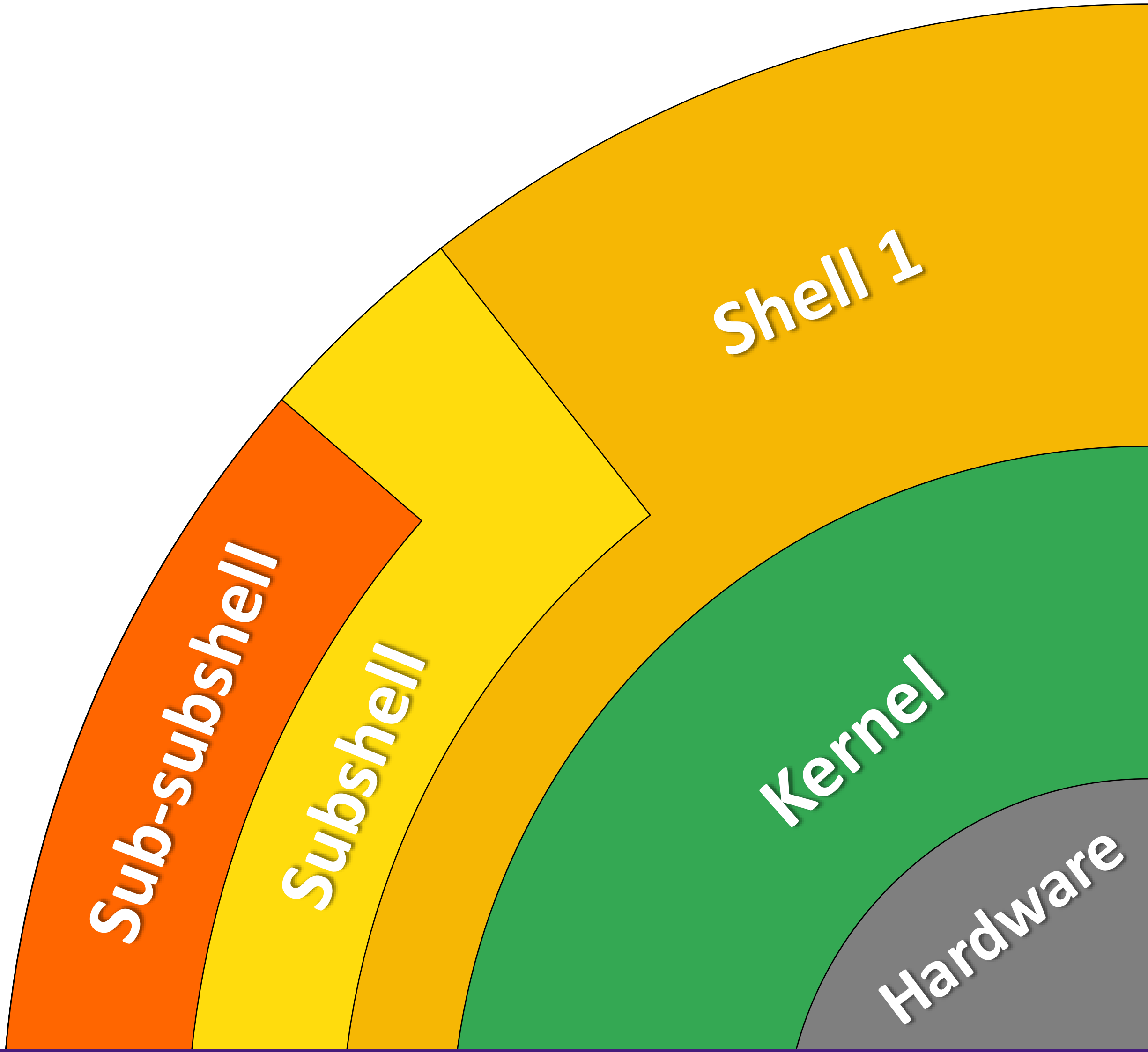
- Don’t forget, the examples are at:
  - http://www.hpc.lsu.edu/training/weekly-materials/Downloads/ShellScripting.zip

Outlines

Outlines

1) Subshells

- **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)



1) Subshells

a) Launch a subshell

	Method	Example	Remarks

1	Run a **Shell script**	`$ ./subshell.sh`
		`$ bash subshell.sh`
			• Can launch **different** Shell types
			• Check subshell level: **`$SHLVL`**

2	Explicitly launch an **interactive** subshell	`$ bash`	
3	Use **command grouping "(…)"**	`$ (echo "I am in subshell!")`	• Launches the **same** Shell type • Does **NOT** change `$SHLVL`

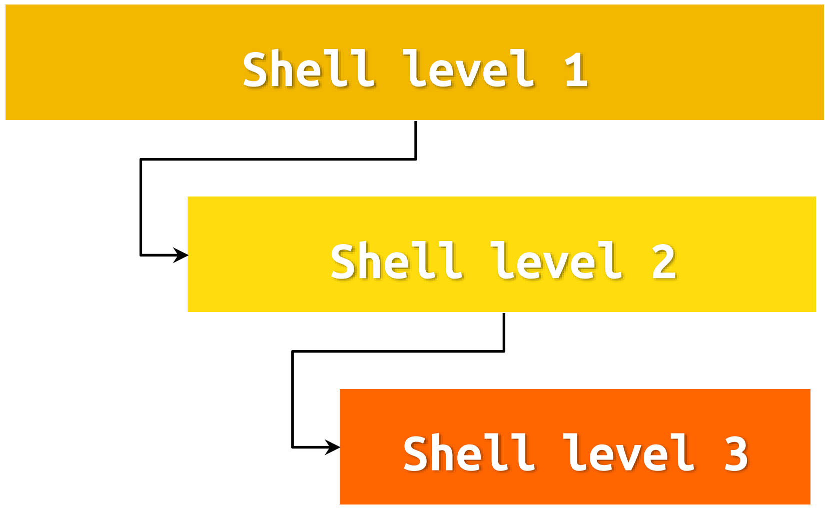
– What does **NOT** launch a subshell?

- **`source`** `subshell.sh`
- Commonly used for **environment setting** scripts (You WANT it to set up current Shell)
  - `source setenv.sh`

[1] *ShellScripting/3.1-Subshells/subshell.sh*
[2] *ShellScripting/3.1-Subshells/setenv.sh*

1) Subshells

b) Scope of variables



	Local variable (Exists only in **current** Shell)	**Global variable** (**Copied** to **all subshells**)
Shell level 1	×	×
Shell level 2	√	√
Shell level 3	×	√

Outlines

INFORMATION TECHNOLOGY SERVICES

2) Flow control

a) Condition – `if` statement

b) Loop – `for` loop

c) Loop – `while` loop

d) Functions

```
#!/bin/bash

# Variables
DATA_DIR=$1
MAX_FILES=3
count=0

echo "Starting the script..."
echo "Looking for files in $DATA_DIR"

# Check if directory exists
if [[ ! -d $DATA_DIR ]]; then
  echo "Directory $DATA_DIR does not exist"
  exit 1
fi

# Check if any .txt files exist
if [[ -z $(find "$DATA_DIR" -maxdepth 1 -name "*.txt" -print -quit) ]]; then
  echo "No .txt files found"
  exit 0
fi

# Loop over files
for file in "$DATA_DIR"/*.txt; do
  echo "Found file: $file"

  count=$((count + 1))

  # Stop after a few files
  if [[ $count -ge $MAX_FILES ]]; then
    echo "Reached the limit of $MAX_FILES files"
    break
  fi
done

echo "Script finished. Processed $count files."
```

2) Flow control

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 ]`

2) Flow control

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**.

2) Flow control

b) Loop – for loop

– Do something for each element in an array.

Syntax

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

2) Flow control

b) Loop – for loop

Array	Example

User defined array	`$ myAry=("Alice" "Bob" "Charlie")`
	`$ for arg in ${myAry[@]}`
	`…`

Shell generated sequence	`$ for arg in `seq 1 4``
	`…`

Output of commands	`$ for arg in `ls $HOME``
	`…`

[1] *ShellScripting/3.2-FlowControl/for.sh*

2) Flow control

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
```

2) Flow control

c) Loop – while loop

Example

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

[1] *ShellScripting/3.2-FlowControl/while.sh*

2) Flow control

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]
```

2) Flow control

d) Functions

Remarks	Example

All variables are **global** by default	`$  myFunc1 () {`
	`    var="Bob"`
	`}`
	`$  var="Alice"; myFunc1 ; echo $var`
	`Bob`

Local variables must be explicitly declared	`$  myFunc2 () {`
	`    local var="Bob"`
	`}`
	`$  var="Alice"; myFunc2 ; echo $var`
	`Alice`

Does **NOT** support **return**
(Use global variable if needed)
	`$  myAdd () {`
	`    result=$(($1+$2))`
	`}`
	`$  myAdd 10 20 ; echo $result`
	`30`

[1] *ShellScripting/3.2-FlowControl/function.sh*

2) Flow control

- **Summary**

a) Condition – `if` statement

b) Loop – `for` loop

c) Loop – `while` loop

d) Functions

Outlines

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`

[1] *ShellScripting/3.3-TextProcessing/employee1.txt*
[2] *ShellScripting/3.3-TextProcessing/employee2.txt*

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.
…	

[1] https://man7.org/linux/man-pages/man1/grep.1.html

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).

3) Advanced Text Processing Commands

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>
```

3) Advanced Text Processing Commands

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\newline' file # Insert before`
	`$ sed '/pattern/ a\newline' file # Insert after`
		Insert / Append new line at specific location, and print results.

	`$ sed '/Alice/ i\newline'`	Insert before lines matches "`Alice`".
	`$ sed '3 a\newline'`	Append to line 3.
…		

[1] *ShellScripting/3.3-TextProcessing/employee2.txt*

3) Advanced Text Processing Commands

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.

[1] *ShellScripting/3.3-TextProcessing/employee2.txt*

3) Advanced Text Processing Commands

- **Summary**
  - "`grep` searches, `sed` edits."

Outlines

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**

BONUS: Where to Get Help

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

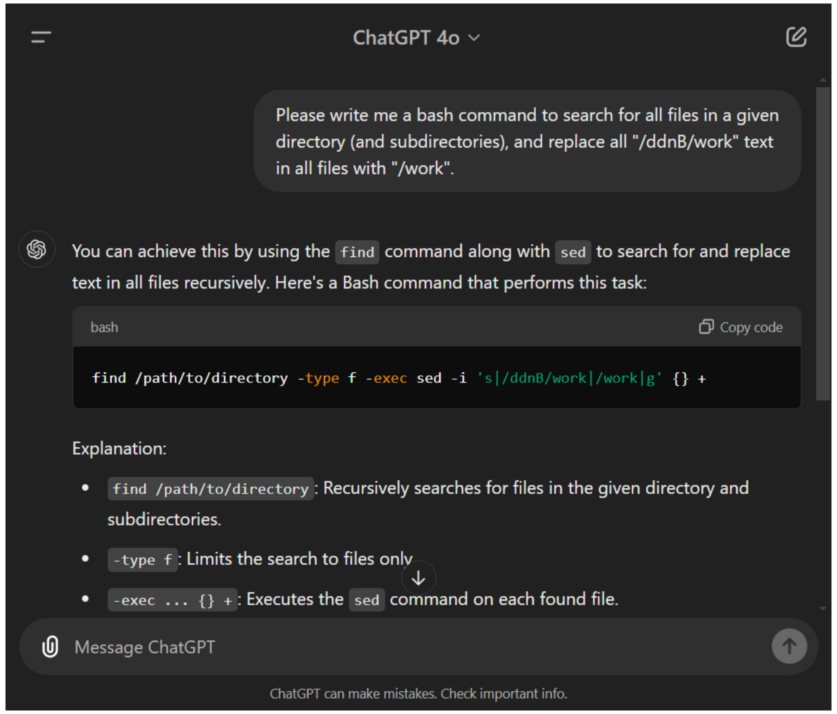
  1) Contact HPC User Services

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

BONUS: Where to Get Help

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

  2) Generative AI



Please write me a bash command to search for all files in a given directory (and subdirectories), and replace all "/ddnB/work" text in all files with "/work".

You can achieve this by using the `find` command along with `sed` to search for and replace text in all files recursively. Here's a Bash command that performs this task:

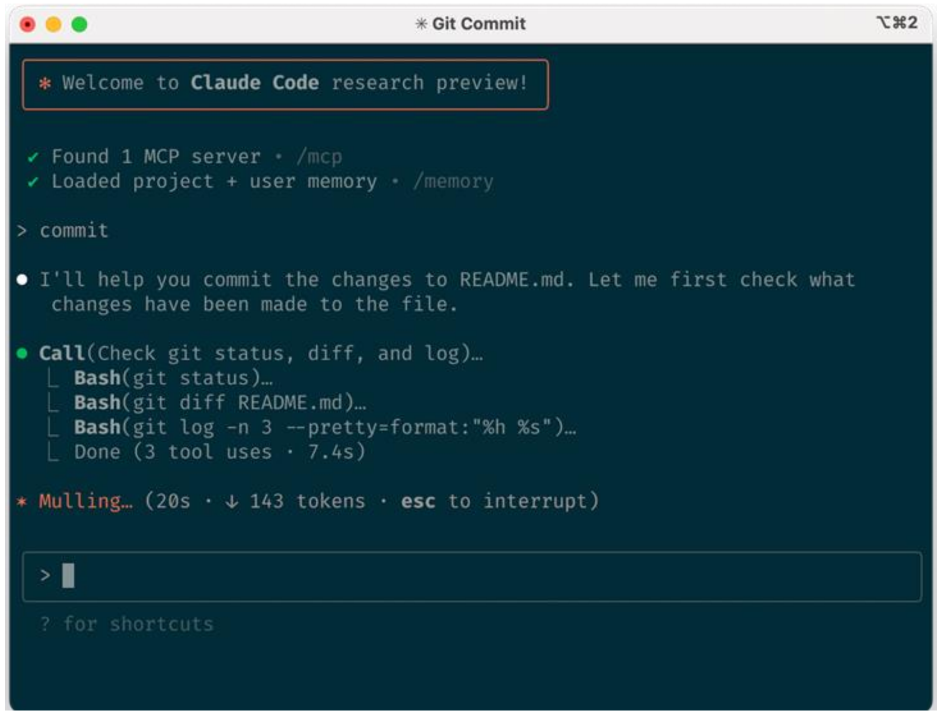
```bash
find /path/to/directory -type f -exec sed -i 's|/ddnB/work|/work|g' {} +
```

Explanation:

- `find /path/to/directory`: Recursively searches for files in the given directory and subdirectories.
- `-type f`: Limits the search to files only
- `-exec ... {} +`: Executes the `sed` command on each found file.

ChatGPT can make mistakes. Check important info.

ChatGPT



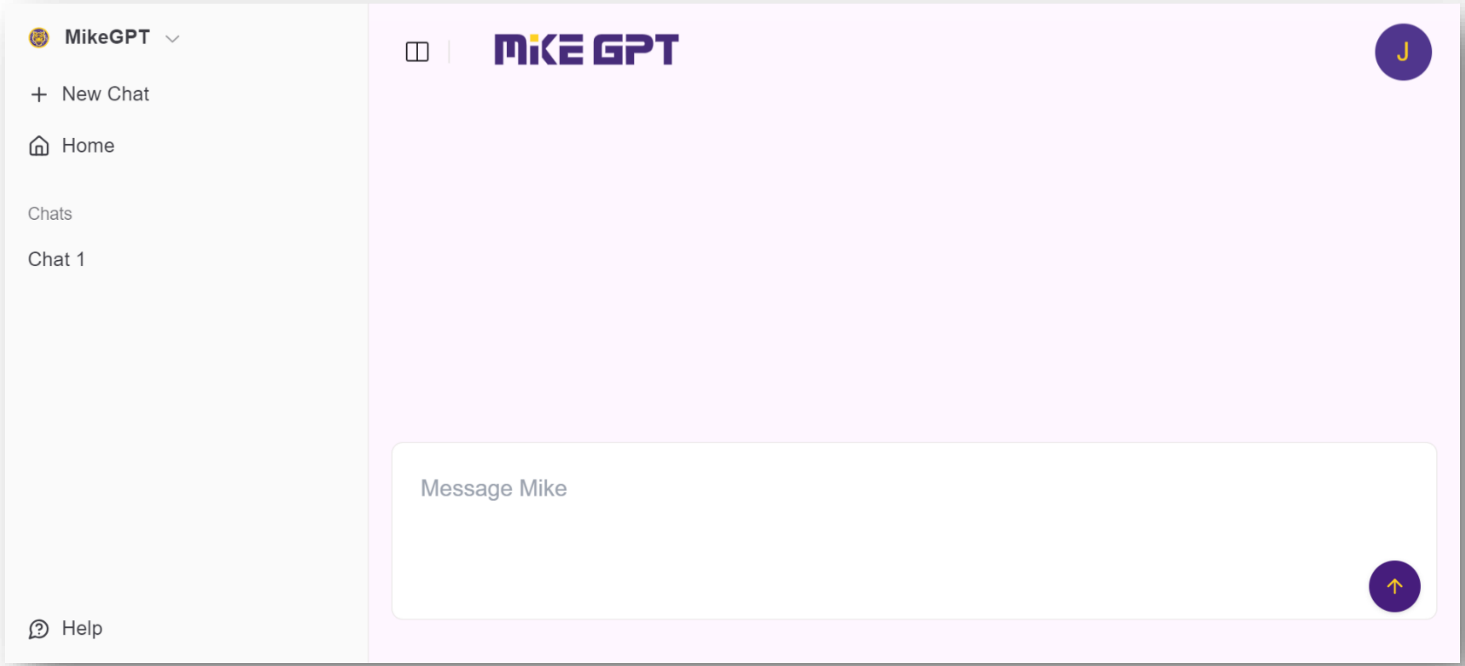
```
* Welcome to Claude Code research preview!

✓ Found 1 MCP server · /mcp
✓ Loaded project + user memory · /memory

> commit

● I'll help you commit the changes to README.md. Let me first check what
  changes have been made to the file.

● Call(Check git status, diff, and log)…
  ⎿ Bash(git status)…
  ⎿ Bash(git diff README.md)…
  ⎿ Bash(git log -n 3 --pretty=format:"%h %s")…
  ⎿ Done (3 tool uses · 7.4s)

* Mulling… (20s · ↓ 143 tokens · esc to interrupt)

> ▌

  ? for shortcuts
```

Claude Code

BONUS: Where to Get Help

- **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/

BONUS: Where to Get Help

- **Why recommend generative AI for Shell scripting?**

Generative AI	**Shell scripting**
• Good at giving **quick and dirty answers**. • Bad at giving **reliable sources**.	• **Quick and dirty** answer is all you need! • Don't really care about sources.
• Good at coding. • Bad when code is too long & complicated.	• **One line** is all you need

BONUS: Where to Get Help

- **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`".

BONUS: Where to Get Help

- **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.

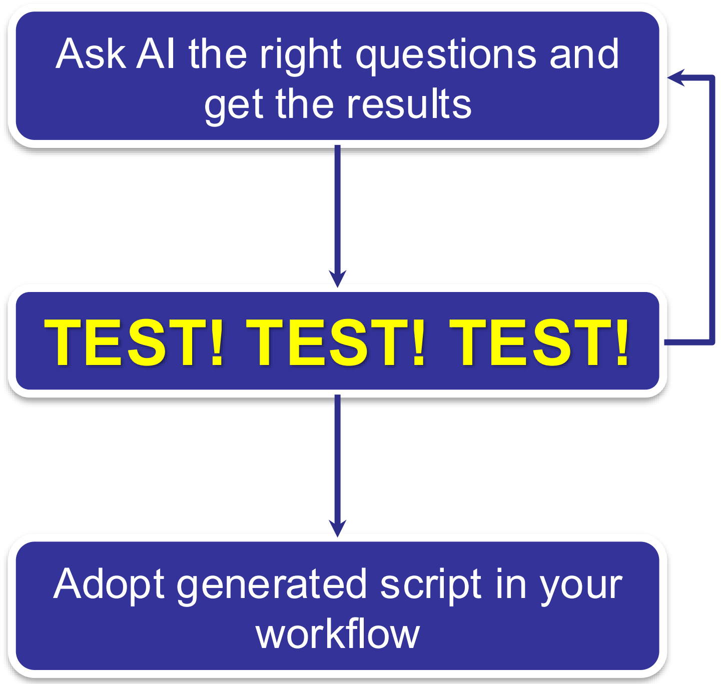
BONUS: Where to Get Help

- **Steps**

  3) Adopt in your workflow

BONUS: Where to Get Help

- **Steps**


Ask AI the right questions and get the results
TEST! TEST! TEST!
Adopt generated script in your workflow

Conclusion

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**

Conclusion

- 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 us

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