CSE 374 Programming Concepts and Tools

Lecture 04 - Scripting

Shell Features

Quick Review

Input, Output, I/O Redirections,
Pipes, Combining Commands

Review: Jobs, Shell Features

Shell Job Control, Processes

- `fg, bg, Ctrl-Z, &, ps, top, Ctrl-C, kill, exit`

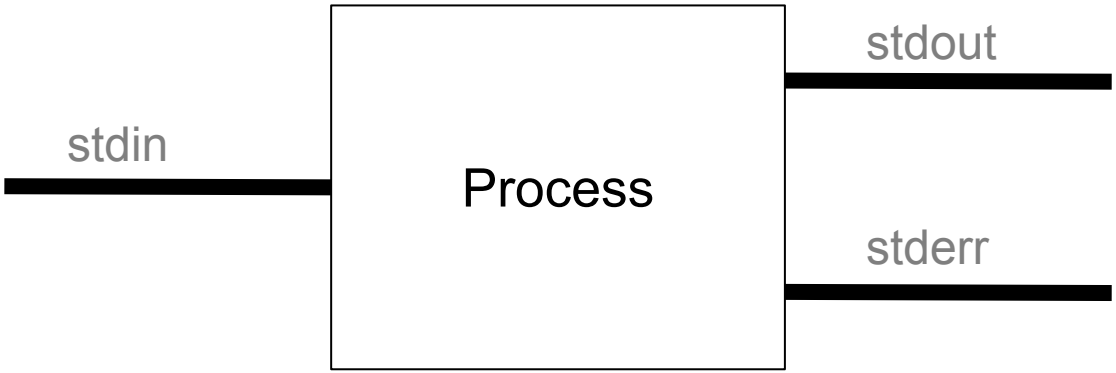
`job > program > process`

Shell Features

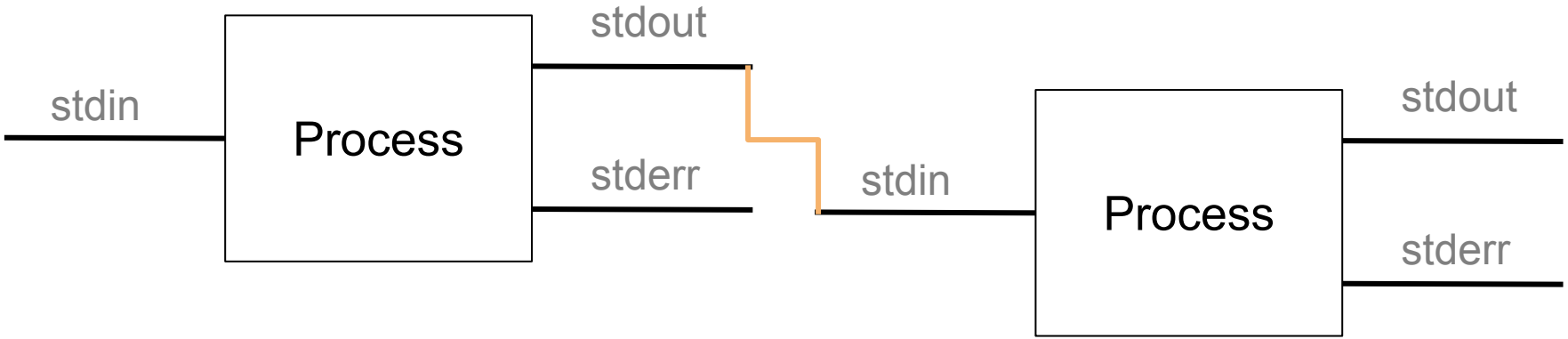
- Variables, Aliases, Quoting, Escaping, Math
- Input, Output, Redirections, Pipes, Combining commands

Process as a “circuit” component

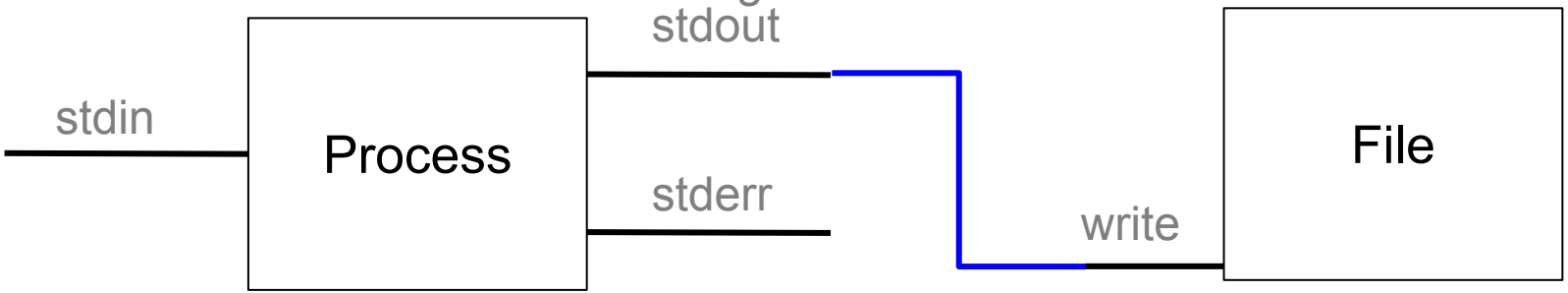
- Input/output streams are inherent properties of processes
  - Like in circuit components, there is a set number of inputs and also outputs
- Output streams (stdout/stderr) can also be combined, passed into the inputs of additional processes, etc.



We can combine processes together by connecting them in specific ways



If we connect a process to a file, though, the file has no outputs, and we cannot continue combining



Review: Input/Output Redirection

The shell can redirect standard input, standard output, and standard error.

In other words, any command that reads from standard input can have its input come from a **file** instead with the shell's **<** operator:

```
command < infile
```

Likewise, any command that writes to standard output can write to a file instead:

```
command > outfile
```
Create/overwrite outfile

```
command >> outfile
```
Append to outfile

Redirection Syntax

redirect stdout to a file	`command > output`
redirect stderr to a file	`command 2> output`
redirect stdout to stderr	`command 1>&2`
redirect stderr to stdout	`command 2>&1`
redirect stderr and stdout to a file	`command &> output`

Discussion

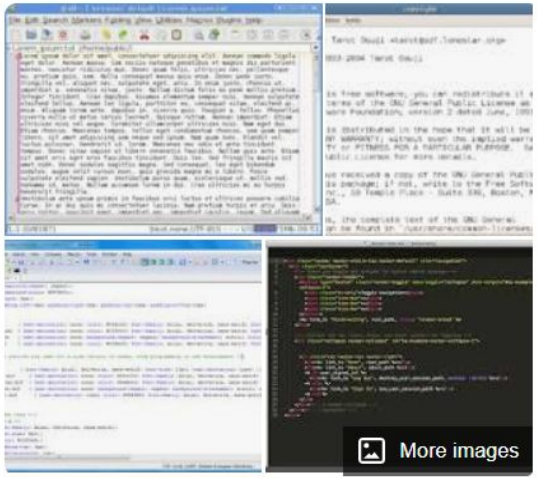
Turn and talk:

In the last lecture I shared an analogy about how the concept of foreground/background jobs is analogous to human multitasking (i.e, cooking dinner while watching a cooking tutorial video and while texting your friend).

What's a real-life analogy you can think of along the lines of piping and/or redirecting?

Text Editors

Text Editors



A text editor is a type of computer program that edits plain text. Such programs are sometimes known as "notepad" software, following the naming of Microsoft Notepad. Wikipedia

Vim

- Move around, make edits using letter keys
- Modes: '`i`' for editing mode, `ESC` to go back to command mode
- Get to menu by typing "`:`"
- Save and quit "`:wq`", no-save and quit: "`:q!`"

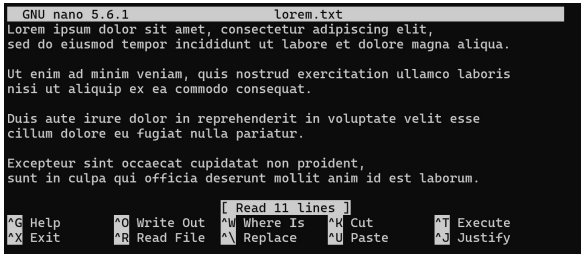
Emacs

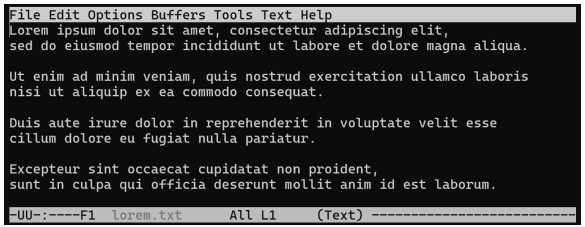
- Can use the arrows to move
- Menu commands use `C` (control) or `M` (meta/alt, esc on mac)
- Quit: `C-x C-c`
- Tutorial, `C-h` for help

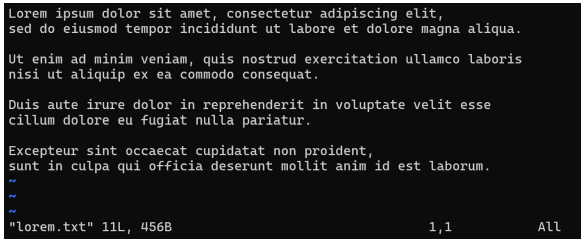
Terminal Text Editing

It is helpful to know how to use one of the terminal text editors that are installed on Calgary.

- There’s no required text editor in this course
- Editors are a matter of preference







nano Emacs Vim

Working in Vim

Basic Navigation:

- **`h`** or ←: Move cursor left
- **`j`** or ↓: Move cursor down
- **`k`** or ↑: Move cursor up
- **`l`** or →: Move cursor right
- **`0`**: Move to the beginning of the line
- **`$`**: Move to the end of the line

More in-depth: Vim quick reference

File Operations:

- **`:e <file>`** – Open a file
- **`:w`** – Save (write) the current file
- **`:w <file>`** – Save the current file with a new name
- **`:q`** – Quit Vim
- **`:q!`** – Quit Vim without saving changes
- **`:x`** – shortcut for **`:wq`** (does the same thing)

Working in Emacs

Basic Navigation:

- **`C-p`** or ↑: Move cursor up
- **`C-n`** or ↓: Move cursor down
- **`C-b`** or ←: Move cursor backward
- **`C-f`** or →: Move cursor forward
- **`C-a`**: Move to the beginning of the line
- **`C-e`**: Move to the end of the line

File Operations:

- **`C-x C-f`**: Open a file
- **`C-x C-s`**: Save the current buffer
- **`C-x C-w`**: Save the current buffer with a new name
- **`C-x C-c`**: Exit Emacs

More in-depth: Emacs quick reference

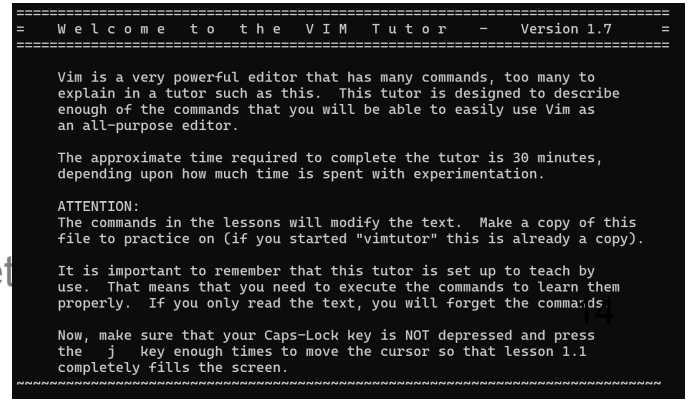
Configuring Emacs and Vim

Emacs config: **`~/.emacs`**

Vim config: **`~/.vimrc`**

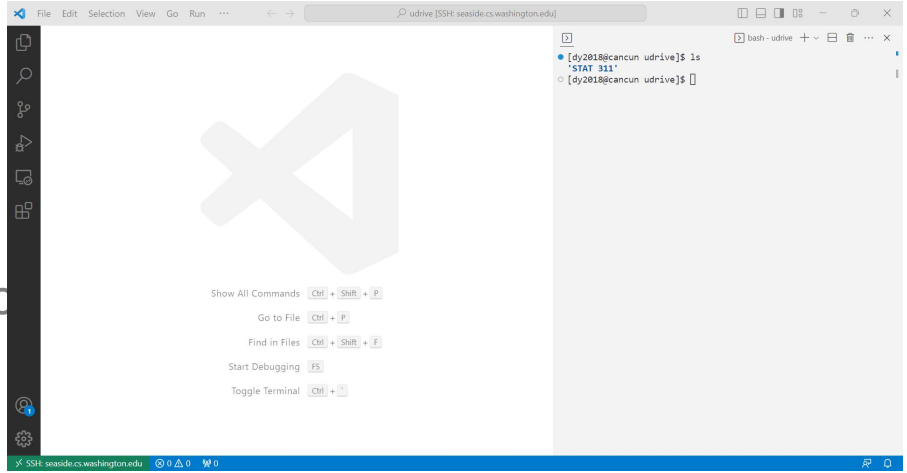
Both editors are highly configurable

- Can be made to act and feel like rich text editors like VSCode, IntelliJ, etc.
  - Autocompletion, syntax highlighting, code formatting, language support, etc.
- Tailored to fit your own tastes and preferences.
- Each has a learning curve
  - Emacs has a built-in tutorial
    - run **`emacs -nw`** and and then hit **`ctrl-h`**, then press the **`t`** key
  - Vim tutorial: run the **`vimtutor`** command
- People get very opinionated about both on the internet



Working in VS Code

1. Install VS Code
   a. It's free!
2. Ctrl + shift + P or F1 to open SSH dialog
   a. Enter in UW NetID account info fo
      Calgary
3. Open folder to see your files
4. Use terminal on bottom



Detailed steps in hw0 spec!

Demo: Text Editors

Scripting

What is a Scripting Language?

- A language which automates the execution of tasks which could alternatively be done manually, one at a time.
- Bash is a shell, but *also* a scripting language.
- Other languages used for scripting: Python, R, Ruby, Perl, etc.
- Great for automation and lightweight, commonly repeated tasks.
  - Developers often write short, small scripts in Bash!
- Bash is NOT a general purpose programming language
  - Python, R, Ruby, Perl are all general purpose

18

Writing Scripts

Instead of writing commands directly into terminal, we can save them in a file. Bash can run these files as executables.

Step 1: Add line at top of file to tell Linux to run the script using bash

`#!/bin/bash` OR `#!/usr/bin/env bash`

- `#!` is called the shebang
- `#` (with no `!` afterwards) begins a comment
- **Best practice: Include a header comment with usage instructions**

Step 2: Make the file executable, i.e., change the execution permissions

```
chmod u+x myscript
```

If the script is in your current directory, but the current directory "`.`" is not in your PATH, you'll need to prepend "`./`" so the shell finds the script and run it:

```
./myscript
```

Script Variables

We described shell variables earlier: **`MYVAR=6`**

When you refer to a variable's value in a shell script, it's a good idea to surround it with double quotes to prevent certain runtime errors:

```
$ FILENAME="My Document"
$ ls $FILENAME
ls: My: No such file or directory
ls: Document: No such file or directory
```

Space in the name
Try to list it
`ls` saw 2 arguments

```
$ ls "$FILENAME"
My Document
```

List it properly
`ls` saw 1 argument

Script Arguments

Scripts can take arguments just like any other program. The script can accept any number of arguments (no limit).

- Arguments are automatically named **`$1, $2,`** …
  - **`$1`** is the first argument after the filename
- **$0** first string entered - the command name
  - Ex: **`echo "$0 needs 1 argument"`** prints **`"<name of script> needs 1 argument"`**
- **$<N>** returns the Nth argument passed to script
  - Ex: **`sort $1`** passes first argument passed into script into **`sort`** command
- **$#** stores number of arguments entered
  - Ex: **`if [ $# -lt 1 ]`** tests if script was passed less than 1 argument

Dealing with All Arguments

- Use `$@` to get a space-separated list of all the arguments
- Use `"$@"` with quotes to make each argument is quoted
  - Prevents arguments which were originally quoted from being read as multiple arguments.
  - **This is usually what you want.**
- To remove the first argument from `$@`, use `shift`
  - `$1` goes away, `$2` becomes `$1`, `$3` becomes `$2`, etc.
  - `shift n` will apply `shift` n times

22

Booleans and Return Codes

Before we can describe conditionals and loops, we need to explain the concept of a Boolean (true/false) test.

- The value 0 means true or success
- Anything else means false or failure
- **Think of zero as “no error” and other values as error codes.**

Additionally, every Linux command returns an integer value, called a **return code** or **exit status**, to the shell when the command exits.

You can see this value in the special variable `$?` e.g. `echo $?`

Exiting with a Return Code

The command **`exit`** exits a shell and ends a shell-script program

When a process exits you can use a number to indicate either a success (0) or failure (1 or 2~255)

- Success: **`exit`** or **`exit 0`**
- Failure: **`exit 1`** or.. {1-255}

If your script doesn't call **`exit`**, the return code is automatically 0.

Polling Time!

Please answer in the LP4 assignment on Gradescope!

Assume we have a bash script called **`script`** :

```
#! /bin/bash
echo "$0"
echo "$2"
echo "$#"
echo "$?"
```

What will the output be if you run

`./script one two three`

(assuming each command succeeds)?

- `$0`
  `$2`
  `$#`
  `$?`
- `script`
  `2`
  `#`
  `?`
- `./script`
  `two`
  `3`
  `1`
- `./script`
  `two`
  `3`
  `0`

Making Your Own Script

1. First line of file is `#! /bin/bash`
2. Make file executable (`chmod u+x myscript`)
3. Try running your script: `./myscript`
4. Shell sees the shebang (#!) program (`/bin/bash`) and launches it to run the script

Demo: First Bash Script (myscript)

Boolean Tests and Conditionals

Test and “ [”

The test command (built into the shell) will evaluate simple Boolean expressions involving numbers and strings, setting its exit status (stored in `$?`) to 0 (true) or 1 (false):

```
$ test 10 -lt 5     # Is 10 less than 5?

$ echo $?

1                   # No, 10 is not less than 5
```

Equivalent syntax using just square brackets

```
[cf2024@calgary ~]$ [ 10 -lt 5 ]
[cf2024@calgary ~]$ echo $?
1
[cf2024@calgary ~]$ [ 10 -gt 5 ]
[cf2024@calgary ~]$ echo $?
0
```

Common Test Expressions

File tests

- `-d name` name is a directory
- `-f name` name is a regular file
- `-e name` name exists
- `-s name` name exists and its size is nonzero (i.e., it's not empty)

String tests

- `s1 == s2` String s1 equals string s2
- `s1 != s2` String s1 does not equal string s2

Numeric tests

- `a -eq b` Integers a and b are equal
- `a -ne b` Integers a and b are not equal
- `a -gt b` Integer a > b
- `a -ge b` Integer a >= b
- `a -lt b` Integer a < b
- `a -le b` Integer a <= b

Boolean Logic in Bash

```
[cf2024@calgary ~]$ ! [ 10 -lt 5 ]
[cf2024@calgary ~]$ echo $?
0
[cf2024@calgary ~]$ ! [ 10 -gt 5 ]
[cf2024@calgary ~]$ echo $?
1
```

Negation: ! not

Square brackets encase tests: `[ <test expression> ]`

- This “[<test expression>]”is a shorthand for use with conditionals and loops
- Remember that “[” is a command like any other, so it is followed by individual arguments **separated by whitespace**. So if you mistakenly forget some whitespace:

```
[ 5 -lt 4]
```

No space between 4 and]

```
bash: [: missing ']'
```

Conditionals

```
if [ test ]; then
    commands
fi

if [ test ]; then
    commands
else
    commands
fi
```

```
if [ test ]; then
    commands
elif [ test ]; then
    commands
elif [ test ]; then
    commands
else
    commands
fi
```

Common Conditionals Use Cases Examples

Check if passed the correct number of arguments. Good for a usage message.

```
if [ $# -ne 2 ]; then
  echo "$0 requires 2 arguments" >&2
  exit 1
fi
```

Check if file exists

```
if [ ! -f "$1" ]; then
  echo "File does not exist" >&2
  exit 1
fi
```

Double brackets

You may see double brackets sometimes. Inside of double brackets, you can do fancier stuff. You can use boolean operators like `&&` and `||` (these are not the shell command combinators)

```
if [[ 5 -gt 4 && 3 -gt 2 ]]; then
  echo "This statement is true!”
fi
```

Unfortunately, this syntax is less *portable* - it isn’t guaranteed to be available on every system. Prefer single brackets whenever possible!

Usage Message

- Tells someone the basics of how to use your script
- Good practice to include one at the top of your script
- Allows other people to make use of your scripts :)
- Also comes in handy when you're trying to use a script that you wrote months (or years) ago, and haven't used since 😆

```
if [ $# -ne 2 ]; then
    echo "Usage: $0 outfile infile"
    exit 1
fi
```


35

Polling Time!

Please answer in the LP4 assignment on Gradescope!

Which of these `if` statements uses correct syntax?

```
if ["hello world" == $msg]; then
  echo "what a day"
fi
```

```
if [ "hello world" == "$msg" ]; then
  echo "what a day"
fi
```

```
if "hello world" == "$msg"; then
  echo "what a day"
fi
```

```
if ("hello world" == $msg) then
  echo "what a day"
fi
```

Demo: Conditionals Demo (directory_check)

Loops

While Loops

```
while [ test ]; do
    commands
done
```

```
while read line; do
    commands
done < inputfile
```

```
# Loop to print numbers from 1 to 10
counter=1
while [ $counter -le 10 ]; do
    echo $counter
    ((counter++))
done
```

```
# Loop to process command-line arguments
while [ $# -gt 0 ]; do
    echo "$1"
    shift
done
```

For Loops

```
for variable in list; do
    commands
done
```

```
# Loop to print numbers from 1 to 10
#   Note: Can also use $(seq 1 10)
for counter in {1..10}; do
  echo $counter
done

# Loop to process command-line arguments
for arg in "$@"; do
  echo "$arg"
done
```

Iterate Over Files

Print every file in the current directory

```
for file in $(ls); do
  if [ -f “$file” ]; then
    echo "$(wc -c < $file) bytes in $file"
  fi
done
```

Demo: Loop Demo

Demo: Fibonacci sequence

Stuff to watch out for

Word Splitting: Use quotes for values that include spaces (e.g. “hello world”)

White space: Spacing of words and symbols matters

- Assign WITHOUT spaces around the equal, brackets are WITH SPACES
- `[ 5 -lt 4 ]`
- `name=amber`

Non number converted to number produces ‘0’

- `x=$((5 + A)) => x=$((5 + 0)) => x=5`

Scripting Style Guide

Scripts should generally be < 200 lines

Do one thing and do it well.

Always use spaces, not tabs (indent line with two spaces)

Comment code with '#'

Function and variable names: lowercase, with underscores to separate words.

- e.g. `my_var=5`

Constants: all caps, separated with underscores, declared at the top of the file.

- e.g. `MY_GLOBAL_VAR=5`

https://google.github.io/styleguide/shell.xml

Assignment Reminders

EX2 due ~~Monday~~ Wednesday 7/1 **@10:49 AM**
EX3 due Wednesday 7/1 **@10:49 AM**

HW1: Bash Commands is due **Thursday @ 11:59 PM**

- Become more familiar with running commands in the shell
- Relatively short assignment; try to submit on time and avoid using late days

Make sure to submit LP4 before the next class as well.

All of these assignments can be found on our course's Gradescope page.

NO Class on Friday, July 3rd. Enjoy the long weekend!

Bonus Slides 

Functions

Declaration

```
function_name() {
    commands
}
```

- semicolons not required at end of lines

single line version

```
function_name() { commands; }
```

- semicolon required

call by name

```
function_name
```

```
#!/bin/bash
hello_world () {
    echo “hello, world!”
}
hello_world
```

Functions

Returns

- By default, return value is status of last executed statement (0 for success, non-zero for fail)
- return and exit keywords both end function and send error code

Parameters

- Bash functions can accept parameters.
- You can define parameters within the function declaration, and these parameters can be accessed within the function body using the `$N` syntax

Function Parameters

```
#!/bin/bash

# Function to process input parameters


my_function() {
  echo "Function param: $1"
  echo "Function param: $2"
}

# Call the function with the provided arguments
my_function "function_arg1" "function_arg2"


echo "Script arg: $1"
echo "Script arg: $2"
```

Function Return Values

```
#!/bin/bash

function return_string() {

    local str="Hello, World!"

    echo $str

}

result=`return_string`

echo $result
```

Variables

```
#!/bin/bash
var1=”A”
var2=”B”
my_function () {
  local var1=”C”
  var2=”D”
  echo "Inside function: var1: $var1, var2: $var2"
}
echo "Before executing function: var1: $var1, var2: $var2"
my_function
echo "After executing function: var1: $var1, var2: $var2"
```

- All variables are by default global, even if declared inside a function.
- Local can be declared using the `local` keyword.

Output:

```
Before executing function: var1: A, var2: B
Inside function: var1: C, var2: D
After executing function: var1: A, var2: D
```

Extra notes