Lecture 4

- Lies from the previous lecture
- Python
  - File operations
  - Functions & methods
- Command line
  - Remote access with `ssh`
  - Persistent connections
  - Remote Jupyter notebooks
  - Brief: CUDA and GPUs with Python
  - Brief: Submitting jobs to the university HPC cluster

Lies!

- Persistent environment variables
- Setting environment variables from `.env` in the shell

Persistent Environment Variables:

Setting default editor to nano

1. Open your shell configuration file: `nano ~/.bashrc`
2. Add this line at the end of the file: `export EDITOR=nano`
3. Save and exit (Ctrl+X, then Y, then Enter)
4. Reload the configuration: `source ~/.bashrc`

THAT DIDN'T WORK! Why?

Modifying only `.bashrc` won't work for all scenarios because:

1. Different shells use different configuration files
2. Some programs may not read `.bashrc`
3. Operating systems may have different default behaviors

For example, if a user is using Zsh (default on macOS since Catalina) instead of Bash, changes in `.bashrc` won't affect their environment.

Find out which shell you're using with `echo $SHELL`

Configuration Files: `bash` (most common)

- `.bashrc`: Executed for interactive non-login shells
- `.bash_profile`: Executed for login shells
- `.bash_login`: Executed for login shells if `.bash_profile` doesn't exist
- `.profile`: Executed for login shells if neither `.bash_profile` nor `.bash_login` exist

I cheat put everything in `.bashrc` and add `source .bashrc` to `.profile`

Configuration Files: `zsh` (MacOS default)

- `.zshenv` : Executed for all shells (login, interactive, or script)
- `.zprofile` : Executed for login shells
- `.zshrc` : Executed for interactive shells
- `.zlogin` : Executed for login shells, after `.zshrc`
- `.zlogout` : Executed when a login shell exits

Configuration Files: Others

- `fish`
  - `config.fish`: Executed for all shells
  - `fish_variables`: Stores universal variables
- `tcsh`
  - `.tcshrc`: Executed for all shells
  - `.login`: Executed for login shells, after `.tcshrc`
- `ksh` (Korn Shell)
  - `.kshrc`: Executed for interactive shells
  - `.profile`: Executed for login shells

Configuration File Takeaways

To ensure changes apply across different shells and scenarios:

- For `bash` users: Modify both `.bashrc` and `.profile`
- For `zsh` users (e.g, macOS): Focus on `.zshenv`, `.zshrc`, or `.zprofile`
- For cross-shell compatibility use shell-specific files to source a common configuration

Persistent Environment Variables (again)

Setting default editor to nano

1. Use `echo $SHELL` to learn which config file to change
2. Open your shell configuration file: `nano ~/<CONFIG_FILE>`
3. Add this line at the end of the file: `export EDITOR=nano`
4. Save and exit (Ctrl+X, then Y, then Enter)
5. Reload the configuration: `source ~/<CONFIG_FILE>`

Setting Variables from `.env` in the Shell

There is **NOT** a single command to load a `.env` file, so let's define one in our shell config using `set`'s `allexport` option:

```
# Add this to the shell configuration file, e.g., .bashrc for bash
load_env () {
    set -o allexport # enable the "allexport" option
    source $1        # set env var's from .env file
    set +o allexport # disable the "allexport" option
}

# Usage
load_env /path/to/.env
```

LIVE DEMO

Python: Files & Functions

- Interacting with files
- Python functions, modules
- Common file operations
- Reading a file line-by-line
- Splitting lines into arrays

Interacting with Files

Basic file operations:

- Opening a file: `open(filename, mode)`
- Reading from a file: `file.read()`, `file.readline()`, `file.readlines()`
- Writing to a file: `file.write()`, `file.writelines()`
- Closing a file: `file.close()`

Always use the `with` statement for automatic file closing:

```
with open('example.txt', 'r') as file:
    content = file.read()
```

File Modes

Common file modes:

- `'r'`: Read (default)
- `'w'`: Write (overwrites existing content)
- `'a'`: Append
- `'r+'`: Read and write
- `'b'`: Binary mode (e.g., `'rb'`, `'wb'`)

```
with open('example.txt', 'w') as file:
    file.write('Hello, World!')
```

Reading a File Line-by-Line

Method 1: Using a for loop

```
with open('example.txt', 'r') as file:
    for line in file:
        print(line.strip())
```

Method 2: Using `readline()`

```
with open('example.txt', 'r') as file:
    while True:
        line = file.readline()
        if not line:
            break
        print(line.strip())
```

Splitting Lines into Arrays

Using the `split()` method:

```
with open('data.txt', 'r') as file:
    for line in file:
        # Split by whitespace (default)
        items = line.split()

        # Split by specific delimiter
        items = line.split(',')

        print(items)
```

Common File Operations

- Check if a file exists:

```
import os # Need this for all examples
os.path.exists('file.txt')
```

- Delete a file:

```
os.remove('file.txt')
```

- Rename a file:

```
os.rename('old_name.txt', 'new_name.txt')
```

Printing to a File

- `print()` can redirect the output to a file using the `file` parameter
- `write()` is a built-in function specifically for writing to a file

```
out_file = "output_filename.txt"
with open(out_file, 'w') as f:
    print(f"This will be written to {out_file}", file=f)
    print("This is another line", file=f)
    f.write("write() needs you to specify new lines\n")
    # write() also only accepts strings
```

Common Directory Operations

- Create a new directory:

```
import os
# Create a new directory in the current working directory
os.mkdir('new_directory')
```

- Create nested directories:

```
import os
# Create new directory and all necessary parent directories
os.makedirs('path/to/new/directory')

# Can also allow the directory to already exist
os.makedirs('path/to/new/directory', exist_ok = True)
```

Working with Directories

```
# Get current working directory:
  current_dir = os.getcwd()

# Change current working directory:
  os.chdir('/path/to/new/directory')

# List contents of a directory:
  contents = os.listdir('/path/to/directory')

# Check if a path is a directory:
  is_dir = os.path.isdir('/path/to/check')
```

Python Functions

```
def greet(name):
    return f"Hello, {name}!"

# Calling the function
message = greet("Alice") # Hello, Alice!
```

Function with default parameters:

```
def greet(name="World"):
    return f"Hello, {name}!"

print(greet())  # Output: Hello, World!
print(greet("Bob"))  # Output: Hello, Bob!
```

Function Arguments

Positional arguments:

```
def add(a, b):
    return a + b

result = add(3, 5)  # result = 8
```

Keyword arguments:

```
def greet(first_name, last_name):
    return f"Hello, {first_name} {last_name}!"

message = greet(last_name="Doe", first_name="John")
print(message)  # Output: Hello, John Doe!
```

`*args` and `**kwargs` (uncommon)

`*args` : Variable number of positional arguments

```python
def sum_all(*args):
    return sum(args)

result = sum_all(1, 2, 3, 4)  # result = 10
```

`**kwargs` : Variable number of keyword arguments

```python
def print_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

print_info(name="Alice", age=30, city="New York")
```

Command Line Arguments in Python

You can pass arguments to python just like any other command

Two main methods:

1. `sys.argv` : Argument order matters
   `python script.py arg1 arg2`
2. `argparse` : Arguments are explicitly named
   `python script.py -two arg2 -one arg1`

Using `sys.argv` (order is important)

```python
import sys

script_name = sys.argv[0]
arguments = sys.argv[1:]

print(f"Script: {script_name}")
print(f"Args: {arguments}")
```

Usage: `python script.py arg1 arg2`

Using `argparse` (tell me about the argument)

```python
import argparse

parser = argparse.ArgumentParser()
parser.add_argument("name", help="Name to greet")
parser.add_argument("-c", "--count", type=int, default=1)

args = parser.parse_args()

for _ in range(args.count):
    print(f"Hello, {args.name}!")
```

Usage: `python script.py Alice -c 3`

Key Benefits of argparse

- Automatic help messages
- Type conversion
- Optional and positional arguments
- Default values

Example: `python script.py -h`

Python Modules

Importing modules:

```
import math
print(math.pi)  # Output: 3.141592653589793

from math import sqrt
print(sqrt(16))  # Output: 4.0

from math import *  # Import all (use cautiously)
```

Modules are just `.py` files!

Creating your own module:

1. Create a file `mymodule.py`
2. Define functions in the file
3. Import and use in another file:

```
import mymodule
mymodule.my_function()
```

Preparing a Script to be a Module

Whenever the Python interpreter reads a source file, it sets a few special variables like `__name__`, and then it executes all of the code found in the file (not wrapped up in functions/classes).

```
# Make this available as a function & module
def my_function(stuff):
  ...

# Do this if running the script
if __name__ == "__main__":
    my_function('stuff')
```

Summary

- File operations: open, read, write, close
- Reading files line-by-line
- Splitting lines into arrays
- Defining and using functions
- Function arguments: positional, keyword, *args, **kwargs
- Working with modules

LIVE DEMO!!!

Jupyter Notebooks

- Jupyter basics
- Remote Jupyter
  - No longer supported at Wynton
  - Paperspace - free option
  - $$$ (advanced) AWS and GCP

What is Jupyter Notebook?

- Interactive computing environment for Python, R, Julia, ...
- Combines code execution, rich text, mathematics, plots and rich media
- File format: `.ipynb` (IPython Notebook)
- Key features:
  - Interactive, in-line code execution
  - Markdown support
  - Code and output in the same document
  - Easy sharing and collaboration

Creating a Jupyter Notebook

From the Terminal:

1. Install Jupyter: `pip install jupyter`
2. Start Jupyter: `jupyter notebook`
3. In the browser interface click "New" > "Python 3"

From VS Code:

1. Install "Jupyter" extension
2. Command palette: "Jupyter: Create New Blank Notebook"
3. Select Python kernel when prompted

Remote Jupyter Notebook with VS Code

1. Start Jupyter on remote server:

```
jupyter notebook --no-browser --port=PORTNUMBER # Often 8888
```

2. In VS Code:
    - Command : "Jupyter: Specify local or remote Jupyter server"
    - Enter the remote server's URL (e.g., `http://localhost:8888`)
    - Provide the token or password if prompted

Remote Jupyter Notebook Notes

- The Jupyter notebook is not on the remote server
- All code will run on the remote server
- Any files or artifacts the code interacts with also have to be on the remote server

LIVE DEMO!!!

Spooky Action at a Distance

- `ssh`
- `scp`

SSH

- UCSF "Wynton" HPC (IT approval required)
- Super Dimension Fortress Remote Learning Lab
- Google Cloud Shell
- GitHub Codespaces
- Your own machine! (easiest with macOS & Linux)

```
ssh user@host.address
# Then enter your password or
# connect with a pre-shared key
```

Super Dimension Fortress Learning Lab

Offers basic access to a learning environment for free.

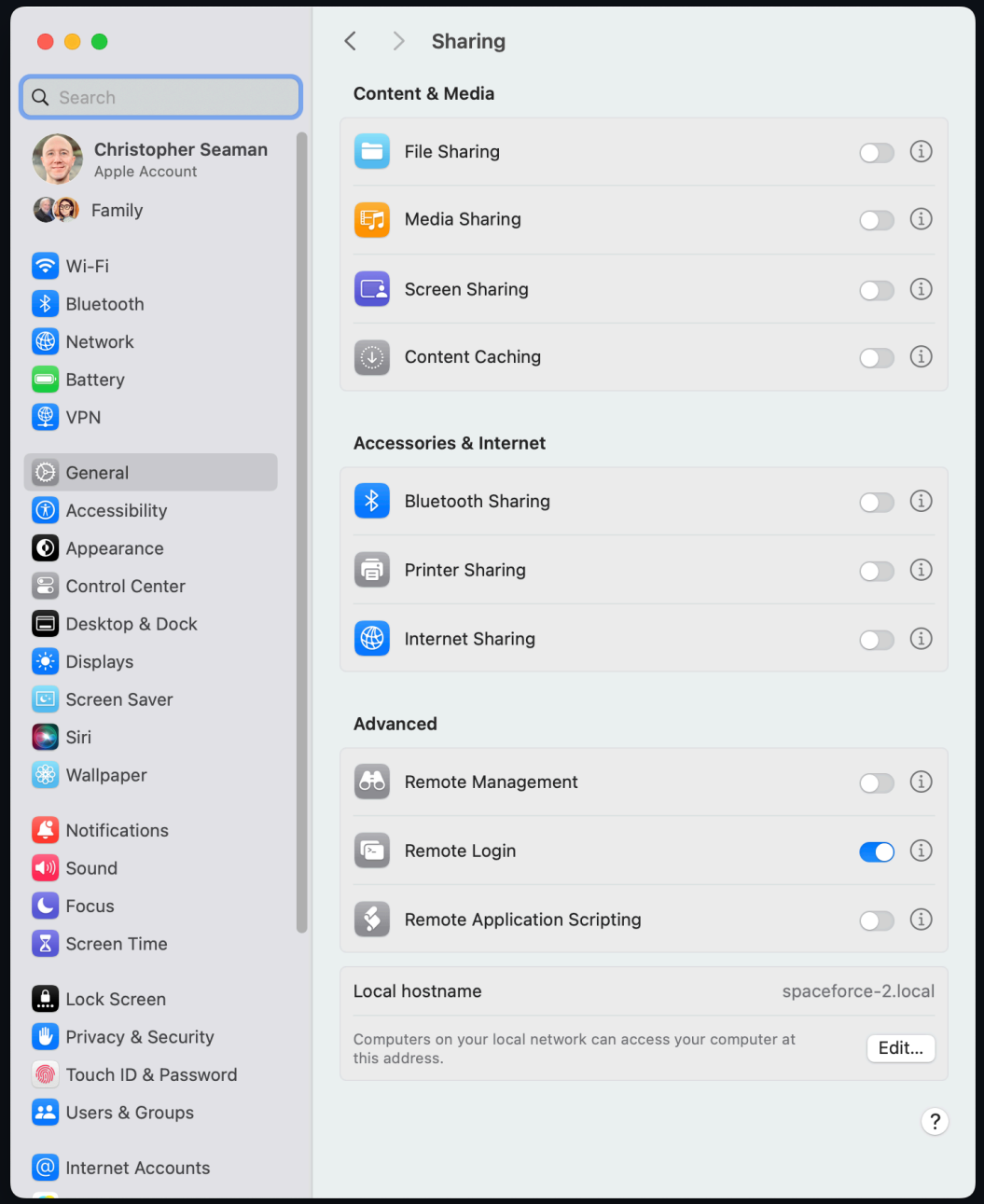
Open command line:

```
ssh new@sdf.org
# Follow the instructions
```

Your

Own

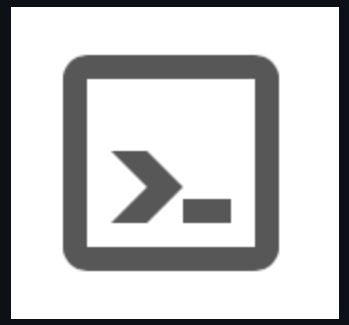
Machine



Google Cloud Shell

- Free temporary virtual machine
- Persistent 5gb storage

1. Open Google Cloud Console
2. Click the button at the top right that looks like a shell



Google Cloud Free Tier (advanced)

If you want an always-on option, Google Cloud offers a free tier for their Compute VM service:

- One instance: `e2-micro`
- Region: `us-west1`, `us-central1`, or `us-east1`
- Storage: 30gb persistent
- Always-on vs. Cloud Shell only active when you are

GitHub `gh` CLI to SSH into Codespaces

GitHub offers a Command Line Interface, which includes many git commands as well as `ssh` access to Codespaces

1. Install GitHub CLI
2. Authenticate with GitHub
3. Create or select a Codespace
4. Connect via SSH

`gh`: Install and Authenticate CLI

```
# Install GitHub CLI (example for macOS with Homebrew)
brew install gh
winget install --id GitHub.cli

# Authenticate
gh auth login
```

Follow the prompts to complete authentication.

`gh`: Create and Connect Codespace

```
# Create a new Codespace
gh codespace create

# List available Codespaces
gh codespace list

# SSH into a Codespace
gh codespace ssh -c CODESPACE_NAME
```

Replace `CODESPACE_NAME` with your Codespace's name.

`scp` (securely) Moving Files Over SSH

SCP (Secure Copy Protocol)

- Secure file transfer between hosts
- Based on SSH protocol
- Encrypted and authenticated

Key Features:

- File encryption
- SSH authentication
- Preserves file attributes

Using SCP

Basic Syntax:

```
scp [options] source destination
# Tip: Use `-r` for directories
```

1. Local to Remote:
   `scp file.txt user@host:/path/`
2. Remote to Local:
   `scp user@host:/file.txt /local/path/`
3. Between Remote Hosts:
   `scp user1@host1:/file.txt user2@host2:/path/`

A brief aside...

Wynton High Performance Computing

(very briefly)

- Uses Son of Grid Engine (SGE) as its job scheduler
- Consists of many compute nodes with identical configurations
- Allows fair sharing of resources among users
- Jobs are submitted to a queue and distributed across nodes

Running Jobs on Wynton HPC

1. Submit the script using `qsub` command
    - Example: `qsub -cwd -j yes COMMAND_YOU_WANT_TO_RUN`
2. Check job status with `qstat` command
3. Retrieve results from output files (e.g., `hello_world.o<job_id>`)

Key commands:

- `qsub`: Submit jobs
- `qstat`: Check job status

CUDA and GPU Computing

We'll hopefully get work with this more later in the course

- CUDA (Compute Unified Device Architecture)
  - NVIDIA's parallel computing platform and API model
  - Enables general-purpose computing on GPUs (GPGPU)
- GPU advantages:
  - Massive parallelism for data-intensive tasks
  - Significantly faster than CPUs for certain operations
- Wynton has dedicated GPU servers in their cluster

GPU Computing with Python

Common packages for using GPU computing:

- PyTorch: Deep learning framework with GPU acceleration
- TensorFlow: Machine learning platform with GPU support
- Numba: JIT compiler that can target NVIDIA GPUs

Steps:

1. Install necessary CUDA drivers and toolkit
2. Use GPU-enabled Python libraries
3. Specify device (CPU/GPU) in your code

Persistent Sessions on Remote Machines

- Challenge: SSH connections can drop unexpectedly
- Solution: Tools for maintaining persistent sessions
  - Screen
  - Tmux
  - Mosh (Mobile Shell)

NOTE: None of these will persist across machine restarts

`screen`

- Basic usage:

```
screen                # Start a new session
screen -S name        # Start a named session
screen -ls            # List sessions
screen -r [name]      # Reattach to a session
```

- Within a screen session:
  - `Ctrl-a d`: Detach from session
  - `Ctrl-a c`: Create a new window
  - `Ctrl-a n`: Next window
  - `Ctrl-a p`: Previous window

`tmux`

Terminal **Mu**ltiplexer: Similar to `screen`, but with more features

```
tmux                # Start a new session
tmux new -s name    # Start a named session
tmux ls             # List sessions
tmux attach -t name # Attach to a session
```

- Within a tmux session:
  - `Ctrl-b d`: Detach from session
  - `Ctrl-b c`: Create a new window
  - `Ctrl-b %`: Split pane vertically
  - `Ctrl-b "`: Split pane horizontally

`mosh` (Mobile Shell)

- Alternative to SSH, more resilient to network issues
- Maintains connection despite IP changes or sleep/wake
- Basic usage:

```
mosh username@remote-server
```

- Requires installation on both client and server (advanced)
- Uses SSH for initial authentication

Comparison

Feature	Screen	Tmux	Mosh
Persistence	Yes	Yes	Yes
Split panes	Limited	Yes	No
Network resilience	No	No	Yes
Scroll back	Yes	Yes	Limited
Learning curve	Moderate	Steeper	Easy

LIVE DEMO!!!