Week 2-4: Unix, Git, Containers & Scientific Python

Chapter 2: The Unix Operating System

What is Unix?

- Fundamental operating system used to control how computers execute programs
- Originated in the 60s
- Important for data analysis, programming and system management
- Designed to produce output that can be used as input to other programs
- Enables the creation of pipelines using small tools

The Shell

- A command line interface (CLI)
- Allows users to:
    - Run programs
    - Control files and folders
    - Receive output in text form
- Operated as a RELP
    - Read Evaluate Print Loop
- Common in modern system (macOS, Linux, Windows via Git Bash)

Basic Commands

Command	Description
`ls`	List the contents of the current directory.
`cd`	Change the current directory.
`pwd`	Print the path of the current directory.
`mkdir`	Create a new directory.
`touch`	Create a new file.
`cp`	Copy a file or directory.
`mv`	Move or rename a file or directory.
`rm`	Remove a file or directory.

`cat`	Print the contents of a file to the terminal.
`less`	View the contents of a file one page at a time.
`grep`	Search for a pattern in a file or files.
`sort`	Sort the lines of a file.
`find`	Search for files based on their name, size, or other attributes.
`wc`	Print the number of lines, words, and bytes in a file.
`chmod`	Change the permissions of a file or directory.
`chown`	Change the ownership of a file or directory.
`head`	Print the first few lines of a file.
`tail`	Print the last few lines of a file.
`diff`	Compare two files and show the differences between them.

Exploring the Filesystem

- When shell starts, it begins in home directory
- Adding flags changes command behavior
    - Example: ls -F adds / to folders
- Absolute path:
    - Always starts from root /

    - Example: cd /Users/arokem
- Relative path:
    - Depends on current directory
    - Example: cd Documents

Path Shortcuts

- .. – parent directory
    - Example: cd .. goes up one level
- ~ - home directory
    - Example: cd ~
    - Example: cd ~/Documents

Pipe Operator (|)

- Connects commands so output of one becomes input of another

Why Unix Matters

- Provides powerful control over:
    - Files and folders
    - Program execution
    - Automation through pipelines
- Becoming comfortable with Unix makes data work faster and more efficient

Chapter 3: Version Control (Git)

What is Git?

- A widely-used version control tool
- Works via a command line

Initialize a Repository

- Creating a Project
    - Mkdir my_project
    - Cd my_project
    - Get init
- Add a file
    - Touch my_file.txt.
    - Git add my_file.txt
- Check status

    - Git status
- Commit Changes
    - Git commit -m “Statement here”
- Check commit history
    - Git log
- Important concepts
    - Commit: saves a snapshot of your project
    - SHA: unique identifier for each commit
    - HEAD: current state of the repository

<u>Tracking Changes</u>

- Stages
    - Unstaged
    - Stages
    - Committed changes
- View changes
    - Git diff
- Workflow
    - Make changes
    - Git add
    - Git commit
- Undoing changes
    - To revert a file to a previous commit
        - Git checkout <SHA> myfile.txt

<u>Branching and Merging</u>

- Enables experimenting without affecting main code
- Keep main branch stable
- Merge only when readu
- Create and switch branches
    - Git branch feature_x
    - Git checkout feature_x
- Merge branch into main
    - Git checcnkout main
    - Git merge feature_x
- Delete branch
    - Git branch -d feature_x

Collaborating with Github

- Remote repository
  - A copy of your respo stored online
  - GitHub is the most common remote
- Add remote and push
  - Git remote add origin <URL>
  - Git push -u origin main
- Authentication
  - GitHub requires
    - Personal Access Token (PAT) OR
    - SSH Keys

Collaboration Workflow

- Cloning
  - Git clone <URL>
- Pulling changes
  - Git pull origin main
- Pushing changes
  - Git push origin main

Common Issues

- Push rejected
  - Caused by remote has changes you don't have locally
- Merge conflict
  - Occurs when two people edit the same lines

Pull Requests

- Used for review before merging into main
- Create a PR after pushing a feature branch
- Allows collaborators to:
  - Review changes
  - Comment
  - Approve merge

Advanced Collaboration

- Larger projects use branches & PRs & code review
- Helps prevents bugs and maintains project history

Version Control for Data: Datalad

- Git is not ideal for large data files
- Datalad is a git-like tool for data visioning
- Useful for tracking:
    - Code
    - Derived datasets
    - Analysis outputs

Chapter 4: Computational Environments & Containers

Why Comp Enviro Matter

- Data Science projects combine many software components
    - Python
    - Multiple Python libraries
    - OS-level dependencies
- Problems arise when:
    - Different projects require different library versions
    - You move work between computers, collaborators, clusters, or cloud
- Goal: reproducibility and portability
- Two main solutions:
    - Virtual environments (conda)
    - Containerization (Docker)

Virtual Environments with Conda

- Virtual Environment: a directory containing:
    - A specific Python version
    - Project-specific libraries and dependencies
- Keeps projects isolated from each other
- Why Conda
    - Popular package manager for scientific Python
    - Manages
        - Python versions
        - Libraries
        - Virtual environments
    - Works on Mac, Linux, Windows
- Base Environment Rule
    - Conda starts in the base environment

    - Best Practice: Do NOT work in base
    - Create one environment per project
- Creating an Environment
    - Conda create -n my_env python=3.8
    - Add packages at creation (jupyter)
- Activating/deactivating
    - Conda activate my_env
    - Conda deactivate
    - Prompt changes to show active enviorment
- Installing Packages
    - Conda install numpy
    - Installs into currently active environment
- Exporting & Sharing Enviroments
    - Exporting dependencies
        - Conda env export > environment.yml
            - Environment.yml
                - Lists exact package versions
                - Can be shared via GitHub
- Limitations of Conda
    - Does not capture:
        - Operating system
        - System level software
        - File system state
    - Leads to containerization

<u>Containerization with Docker</u>

- What is it?
    - Packages
        - OS
        - Software
        - Libraries
        - Data
    - Produces fully reproducible environments
- Docker Concepts
    - Image: Blueprint/recipe
    - Container: Running instance of an image
    - Host: Your local machine
    - Registry: Collection of images (DockerHub)

- Docker Images
    - Identified by:
        - SHA hashes
        - Tags (latest)
    - Latest changes over time -> not full reproducible

Getting Started with Docker

- Pull an Image
    - Docker pull hello-world
- Run a Container
    - Docker run hello-world
- Confirms Docker is working
- Container exists immediately
- Interactive Containers
    - Docker run -it ubuntu bash
    - Flags:
        - -i -> interactive
        - -t -> terminal
        - Bash -> Unix shell inside container
    - Exit container
        - Exit
- Persistence with Volumes
    - Without volumes -> files deleted when container stops

Creating Docker Images (Dockerfile)

- Basics
    - Text file name Dockerfile
    - Defines how to build an image
- Commands
    - FROM: base image
    - RUN: execute shell commands
    - COPY: add files into image
- Building an Image
    - Docker build -t arokem/niabel-notebook:0.1 .
    - Naming convention:
        - <username>/<image-name>:<tag>

Sharing Docker Images

- NeuroDocker
    - Tool for building neuroscience containers
    - Supports:
        - FreeSurfer
        - AFNI
        - FSL
        - ANTs
    - Simplifies complex neuroimaging installs

Chapter 5: A brief introduction to Python

Core Characteristics

- High level, interpreted lanagage
    - No need to manage memory manually
    - Code is executed line by line
- Readable and intuitive syntax
    - Uses Enlgih like words
    - Enforced structure (indentation), improving readability
    - Math translates cleanly into code
- General purpose
    - Used in data science, neuroimaging, web dev, etc.
    - Massive standard library
- Huge community
    - Easier to find libraries, documentation, collaborators, and help
- Strong neuroimaging ecosystem
    - Open source tools widely used in research
- Industry relevance
    - Common requirement for data science roles outside academia

Variable and Basic Types

- Variables
    - Stores data are assigned using “=”
    - No need to declare types
    - Variable can change during execution
- Printing and Comments

  - Use “print()” to display variables
  - In interactive env like Jupyter, last line auto displays
  - “#” = comment

Built in Types

- Integers (int)
  - Whole numbers only
  - Support standards arithmetic
  - Division (/) returns a float, even when dividing ints
- Floats (float)
  - Real number with decimals
  - Mixing int and float = float
- Strings (str)
  - Sequence of characters inside (‘ ‘) or (“ “)
  - Common operations:
    - Len() = length
    - .upper(), .lower(), .capitalize()
    - .count(), .replace()
  - Strings are objects with many built in methods
- Booleans (bool)
  - Only two values: True and False
  - Often produced via
    - Comparisons (>, ==)
    - Logical operators (and, or)
  - Some integers map to Booleans:
    - 0 == False
    - 1 == True
- None
  - Represents absence of a value
  - Different from False
  - Variable exists but has no meaningful value

Collections

- Lists (list)
  - Ordered, mutable, heterogenous collections
- Key Features
  - 0-based indexing
  - Negatice indexing

- Slicing: list[start:end]
- Mutable: elements can be reassigned
- Common methods:
  - .append() -> add one element

- Dictionaries (dict)
  - Key -> value mappings
  - Keys must be unique
  - Keys and values can be different types
- Key points
  - Access values using keys, not positions
  - Assignment updates or add entries
  - Lists can be values, but not keys
- Tuples (tuple)
  - Like lists, but immutable
  - Created with ()
  - Cannot modify elements after creation
  - Can be used as dictionary keys
  - Can convert to list if mutation is needed

Everything in Python is an Object

- No primitive data types, everything is an object
  - Integers, strings, Booleans, lists, dict, etc.
- Objects contain:
  - Data
  - Methods (functions attached the object)

Dot Notation

- Used to access an object's methods or attributes
- "Call the idea inside the object"
- Methods are type specific
  - Strings have .lower()
  - Integers do not

Inspecting Objects

- Type(obj) = returns object type
- Dir(obj) = lists all attributes and methods
  - Names with ___name___ are special/internal

Control Flow

- Determines which code runs and in what order
- Core constructs:
  - Conditional (if/elif/else)
  - Loops (for loops)

Conditional (if-then statements)

- Allow code to branch based on conditions
- Conditions evaluate to Boolean values (True/False)
- Structure:
  - If = required
  - Elif = optional, can have many
  - Else = optional, fallback case
- Only one branch executes
- Rules
  - Conditions are checked top to bottom
  - First condition that evaluates to True runs
  - Indentation defines which code belongs to each branch

Loops

- For Loops
  - Used to iterate over collections (lists, strings, etc)
  - Loop variable exists only inside the loop
- Looping over a range
  - Use range() to loop over integers
  - Range(N):
    - Starts at 0
    - Stops before n
- F-strings
  - Strings prefixed with f
  - Embed variables or expression inside {}

Nested Control Flow

- Control structures can be nested
  - If inside for
  - For inside if
- Common use case: filtering data

Comprehensions

- Compact syntax for loops
- Do not change meaning, just shorted
- Can replace many simple loops
- Nested comprehensions exist but reduce readability
- Prioritize clarity over cleverness

Whitespace is syntactically significant

- Indentation is mandatory
- Used to define code blocks
- Entering a compound statement -> indent
- Exiting -> de-dent

Namespaces and Imports

- A namespace = a mapping of names to objects
- Python keeps namespaces small and explicit
- Prevents name conflict

Importing a Module

- Modules must be explicitly imported
- If not imported -> NameError

Importing From a Module

- Import specific objects
- “from module import object”
- Makes object directly accessible
- Module itself is not imported unless explicity done

Renaming Imports

- Use as to alias long names
- “Import numpy as np”
- Follow community conventions for readability

Functions

- Reusable blocks of code
- Run only when called
- Defined using def

Arguments

- Inputs that change function behavior
- Required unless defaults are provided

Return values

- Explicit return sends output back
- No return -> returns None

Docstrings

- Describe function behavior
- Numpy style docstrings are standard in scientific Python

Positional Arguments

- Assigned by order
- Required unless defaults exisits
- Wrong number -> TypeError

Keywords Arguments

- Have defaults values
- Optional
- Can be passed in any order if named

*args and **kwargs

- Used when a function must accept:
  - Unknown number of positional arugemtns - *args
  - Unknown number of keywords arguments - **kwargs
- Purpose
  - Writing wrapper functions
  - Passing arguments through to other function
- Important Concept
  - Functions are objects
  - Can be passed as arguments like any other value

Classes (object oriented programming)

- A class is a blueprint
- An instance is a concrete object created from a class
- Defines

    - Attributes (data0
    - Methods (behavior)
- Closely related to the idea of a type
- Pass = placeholder
- Creating instances
    - Obj = MyClass()
        - Instance belongs to the class
        - Verified using type()
- Naming conventions
    - Variables and functions -> snake_case
    - Classes -> CamelCase

<u>Instance Methods & "self"</u>

- Methods defined inside a class
- First parameter is always "self"
- "self" refers to the current instance
- Used to store and access instance state
- __init__ (initializer)
    - Called automatically when instance is created
    - Used to initialize attributes

<u>Magic Methods</u>

- Methods with double underscores
- Call implicitly
- Define how objects behave with operators and built-ins

Operators are Method Calls

- Behavior depends on the left-hand object's class
- Explains why
    - Strings repeat
    - Lists repeat
    - Dictionaries fail
- + -> __add__
- < -> __lt__
- Len(obj) -> obj.__len__()
- Used mainly when writing custom classes

<u>Custom Operator Behavior</u>

- Classes can redefine operators

<u>Chapter 6: The Python environment</u>

<u>Development Environment</u>

- The software tools you use to write, run, debug, and test code
- Python is plain text
- Code specific editors dramatically improve productivity

<u>What Good Editors Provide</u>

- Syntax & error highlighting
- Automatic code completion
- Code formatting
- Integrated execution
- Built-in debugging

<u>Debugging</u>

- Finding out why code fails or behaves incorrectly
- Even expert programmers debug constantly
- You cannot avoid debugging

Debugging with print() and assert

- Print()
  - Inspect values during execution
  - Useful for checking assumptions
  - Simple but powerful
- Assert
  - Verifies conditions that must be true
  - Syntax
    - “assert condition, “error message”
  - Helpful for:
    - Catching edge cases early
    - Making errors more informative
  - Fails fast
  - Replace vague crashed with meaningful error messages
  - Useful during development and testing

Debugging with pdb

- Python's built-in debugger
- Pauses execution and inspect state
- Pdb.set_trace()
    - Freezes program at that line
    - Drops you into an interactive prompt
    - You can:
        - Inspect variable
        - Run Python commands
        - Modify variables
        - Continue execution
- Why use a debugger
    - Avoid rerunning code repeatedly
    - Inspect everything at once
    - Especially useful for complex bugs

Testing

- Why it matters
    - Scientists often test informally
    - Soft dev rely on automated, repeatable tests
    - Automated tests:
        - Catch bugs early
        - Prevent regressions
        - Improve confidence in refactoring (restructure without changing behavior)

Writing Test Functions (Unit Testing)

- Unit tests
    - Test one function at a time
    - Verify expected outputs for known inputs
    - Written using assert
- Test Function Conventions
    - Names starts with test_
    - Contains multiple assertions
    - Covers:
        - Normal cases
        - Edge cases

        - Different data types
- Why Unit Tests are powerful
    - Failures are localized
    - Easy to identify where code breaks
    - Can be ran repeatedly with zero effort
    - Enable safe refactoring

Profiling Code

- What is profiling?
    - Measuring performance (runtime, scaling)
    - Especially important for large datasets
    - Enable optimization without breaking functionality

%timeit (Jupyter Magic)

- Measures execution time
- Runs code multiple times
- Reports:
    - Mean runtime
    - Variability

Scaling performance

- Test runtime as input size increases
- Helps predict performance on large datasets
- Critical for neuroscience/neuroimaging workflows