MESA Course Notes

Author: Joshua Aiken

2025

Contents

1 Linux

1.1 Absolute Basics

The command line, also known as the terminal or shell, is a powerful way to interact with a Linux system. It allows users to execute commands, manipulate files, and run programs with greater control than a graphical user interface.

Opening the Terminal

On a *native* Linux system, you can open the terminal via the applications menu, or by pressing `Ctrl + Alt + T`. We will not be running Linux naively, however. Instead, we `ssh` onto a server that runs Linux. To do this, we will use Putty (Windows) or use an `ssh` command from some other command line native to the OS that we are using (Command Prompt or WSL (Windows Subsystem for Linux) for Windows, Terminal for MacOS, or even just the regular terminal from Linux).

Basic Command Structure

A typical command has the following structure:

```
command [options] [arguments]
```

For example:

```
ls -l /home/user
```

This lists the contents of the `/home/user` directory in long format.

Essential Commands

Here are a few commands that are essential for getting started:

- `pwd` – Print the current working directory.
- `ls` – List files and directories in the current directory.
  - `-l` (long listing), `-a` (include hidden files)
- `cd [directory]` – Change the current directory.
  - `cd ..` moves up one directory.
  - `cd` with no arguments returns to the home directory.
- `mkdir [name]` – Create a new directory.
- `touch [file]` – Create a new empty file.
- `cp [source] [destination]` – Copy files or directories.
- `mv [source] [destination]` – Move or rename files or directories.
- `rm [file]` – Remove (delete) a file.

  - Use `rm -rf [directory]` to remove a directory and its contents.
  - **Be careful**: deleted files are not moved to a trash bin.
- `man [command]` – Show the manual page for a command (e.g., `man ls`).

Tab Completion and Command History

- Press `Tab` to auto-complete file and directory names.
- Use the `Up/Down` arrow keys to scroll through your command history.

File Paths

- Absolute paths start from the root directory, e.g., `/home/user/docs`.
- Relative paths are based on your current directory, e.g., `../` refers to the parent directory.
- The tilde `~` refers to the home directory.

Getting Help

- `man [command]` – Displays the manual for a command.
- `[command] --help` – Shows (typically) usage instructions for many commands.

Examples and Environment Setup

1. Copy the text editor program `micro` into your `~/bin/` (binaries - or executable programs) directory:

   ```
   cp ~/projects/def-lnelson/jaiken/public/micro/micro ~/bin/micro
   ```

   This will give us a nicer command line text editor to use (you can still use `nano`, `vim`, or `emacs` if you would prefer). Placing since this program is placed in the binaries directory, we can invoke it without needing to specify it's path (note that this will *create* the file `filename.txt` if it doesn't exist):

   ```
   micro filename.txt
   ```

2. Copy the full MESA tutorials directory in the projects directory into your home directory:

   ```
   cp -r ~/projects/def-lnelson/jaiken/public/MESA_tutorial ~/.
   ```

   This will be where we run all of our examples and practice.

3. Change your directory to the `MESA_tutorial` directory and verify the contents:

```
cd ~/MESA_tutorial
ls
```

There should be four directories: `bash/`, `python/`, `MESA/`, and `SLURM/`.

4. Now enter the `bash/` directory:

   ```
   cd bash
   ```

   This directory contains two directories: `practice/` and `examples/`. The `practice/` directory is where you will write your own scripts and the `examples/` directory holds the example scripts that I have written.

5. Finally, enter the `examples/` directory and run the following command:

   ```
   ./hello_world.sh
   ```

   If you see the following printed to the screen `Hello World!` then you have successfully ran a bash script.

Practice Questions

1. - Use the command line to create a directory called `linux_practice` in your bash practice directory.
   - Enter that directory and create a new directory called `inside`.
   - Enter that directory and enter `pwd` into the terminal. The output should be

     `/home/username/MESA_tutorial/bash/practice/linux_practice/inside/`

2. - Inside the directory created in the previous question (`linux_practice`), create an empty file called `test1`.
   - Write `hello world!` in this file with the `echo` command.
   - Show the file's contents with the `cat` command.

3. - Create a new directory called `nest1`, then create a second directory `nest2` within it (without entering the directory).
   - Copy the `test1` file into the `nest1` directory (keep the same name), and create a new, empty file `test2` in `nest2`.
   - Make a copy of the directory `test1` (and all of its contents) called `test1_copy`.

4. - Create a new empty file called `hello_world.py`.
   - Using `micro`, type the following Python script into the previous file

     ```python
     print("Hello World!")
     ```

     Listing 1: `hello_world.py`

   - Save and exit the file (`ctrl+s` to save and `ctrl+q` to quit).
   - Run the file by typing `python3 hello_world.py` into the command line.

1.2 Bash

This section introduces foundational concepts needed to write basic Bash scripts. Bash is a Unix shell commonly used for automating tasks, managing files, and processing data on Linux systems. These topics will help students understand how to create and run simple yet powerful scripts.

What is Bash?

Bash (Bourne Again SHell) is a command processor that typically runs in a terminal window. It allows users to interact with the system by typing commands. Bash scripts are plain text files that contain a sequence of commands to be executed automatically. They are widely used for automation and task management on Unix-like systems.

Shebang Line

A Bash script should begin with a special line called the *shebang* to indicate which interpreter should execute the file:

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

This ensures that the system uses the correct shell to interpret the script, regardless of the user's default shell.

Script Permissions

Most likely, after writing and saving a Bash script, you will get an error when you try to run it. This is because it is not *executable* yet. Before a Bash script can be executed, it must have the appropriate permissions. Use the following command to make a script executable:

```
chmod +x script.sh
```

Then, run the script using:

```
./script.sh
```

Variables

Variables store values that can be reused throughout a script. To define a variable, use the syntax:

```
name="Alice"
```

To access the variable's value, prefix it with a dollar sign:

```
echo $name
```

Note that there should be no spaces around the equals sign when assigning a value.

Quoting

Quoting controls how the shell interprets text. Single quotes (`'...'`) preserve the literal value of characters, including variables, while double quotes (`"..."`) allow variable and command substitution. For example:

```
echo '$name'   # prints: $name
echo "$name"   # prints: Alice
```

Comments

Comments begin with a hash symbol (`#`) and are ignored by the shell. They are useful for explaining code:

```
# This is a comment
```

Try it out:

1. Enter the following commands into the terminal:

   ```
   hello='Hello'
   world='world!'
   echo "$hello $world"
   ```

   The output should be

   ```
   Hello world!
   ```

2. Write a Bash script `variables.sh` that performs these three commands and verify that the output is the same.
3. Comment out the `world='world!'` line. What do you expect the script will output? Run the script and verify.

Conditionals

Bash provides `if` statements to execute code based on conditions. The general syntax is:

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

Conditions can test file properties, string comparisons, or numeric relationships.

Loops

Loops allow repetition of commands. A *for loop* iterates over a list of values:

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

A *while loop* continues executing as long as a condition is true:

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

Command Substitution

Command substitution allows the output of a command to be used as part of another command or stored in a variable:

```
today=$(date)
echo "Today is $today"
```

This is particularly useful for capturing dynamic values within a script.

Arithmetic Operations

Bash supports simple integer arithmetic using double parentheses:

```
count=5
((count = count + 1))
echo $count
```

You can also use the `let` command or the `expr` utility for arithmetic, though `((...))` is more commonly used.

Try it out:

1. Create a directory `some_files` and three files `file1.txt`, `file2.txt`, and `file3.txt` within the directory. Then, enter the following into a Bash script:

```
#!/bin/bash

for file in some_files/*.txt; do
    echo $file
done
```

What should the output of this script be? Check it by running it (it needs to be ran in the same directory that `some_files` is in)

2. Write a script that counts the number of text files in the `some_files` directory created in the previous exercise.

Input and Output

Bash provides several mechanisms for handling input and output:

- Use `read` to get input from the user:

```
read name
```

- Redirect output to a file with > (overwrite) or >> (append).
- Redirect input from a file using <.
- Use a pipe (|) to send the output of one command into the input of another:

```
ls | grep ".txt"
```

Practice Questions

1. Write a bash script that outputs `Hello World!` to the screen (no need to use variables).
2. Write a bash script that creates ten directories with indexed names: `dir1`, `dir2`, ..., `dir10`.
3. What does the following bash script do?

```
#!/bin/bash

CURRENT_DIR=$(pwd)

DIR1=funny_dir
DIR2=funny_dir_2

mkdir $DIR1
cd $DIR1
mkdir $DIR2
cd $DIR2

pwd

cd $CURRENT_DIR

rm -d $DIR1/$DIR2
rm -d $DIR1

cd /dev
pwd
```

Listing 2: `cd_script.sh`

On line 20, the script changes the directory to the `/dev` directory. When the script ends, is that the directory that the terminal is now in? Check it with the `pwd` command.

4. Use Google to figure out what the following script does (look up any commands you haven't seen before):

```
1 #!/bin/bash
2
3 sleep_time=5
4
5 echo "I am tired, time for bed."
6 sleep $sleep_time
7 echo "What a great $sleep_time seconds rest!"
```

Listing 3: `sleep_script.sh`

Research how to use the `time` command and use it to check your understanding of the script.

```
1 #!/bin/bash
2
3 echo 'hello world!'
```

Listing 4: `hello_world.sh`

2 Python

2.1 What is Python?

Python is a high-level, general-purpose programming language that is widely used for scripting, automation, data analysis, and scientific computing. It is designed to be readable and easy to learn, making it an excellent choice for beginners as well as professionals.

Interpreter vs Compiler

Python is an *interpreted* language, meaning that code is executed line-by-line by an interpreter, rather than being fully compiled into machine code before execution (as is the case for languages like C or Fortran). This allows for interactive development and rapid prototyping, but often comes at the cost of slower execution compared to compiled programs.

The standard Python interpreter (commonly referred to as `CPython`) reads and executes Python source files directly. Other implementations exist, such as `PyPy` (a just-in-time compiling interpreter), `Jython` (for the Java Virtual Machine), and `IronPython` (for .NET), but `CPython` is by far the most commonly used. Note that the name "`CPython`" refers to the fact that that the interpreter is written in C - you do not need to know anything about C to use it.

Why Use Python?

Python is widely appreciated for its clear syntax, vast ecosystem of third-party libraries, and broad community support. It is especially well-suited for:

- Rapid prototyping and scripting
- Data analysis and scientific computing* (with libraries such as NumPy, SciPy, and pandas)
- Automation of repetitive tasks and workflows
- Teaching and learning programming concepts

Its versatility and ease of use make it a popular language in many fields, including physics, astronomy, biology, engineering, and finance.

When *Not* to Use Python

Despite its strengths, Python is not always the right tool for every job. You might want to choose another language if:

- Performance is critical and the application is CPU-bound or GPU-bound (e.g., in hardcore numerical computations)
- You need tight control over memory usage or hardware (e.g., embedded systems)
- You are deploying to a system where the Python interpreter is not available or not permitted

In such cases, compiled languages like (modern) Fortran, C, or C++ (maybe even Rust) may be more appropriate. However, Python can often be combined with these compiled languages using interfaces such as `F2PY` or `Cython`.

Ways to Run Python Code

Python code can be run in several different ways, depending on the use case and development environment:

- **Interactive mode:** By launching the Python interpreter from the terminal using the command `python` or `python3`. This allows execution of individual commands in a read-eval-print loop (REPL).
- **Script mode:** By writing code in a `.py` file and running it with `python filename.py`. This is the most common method for running reusable programs.
- **Notebooks:** Using tools like Jupyter Notebook or JupyterLab, which allow code to be run in cells with rich output, including plots and formatted text. This is widely used in data science and research.
- **Integrated Development Environments (IDEs):** Applications like VS Code, PyCharm, and Thonny provide graphical environments for writing, running, and debugging Python code.

These flexible execution options make Python highly adaptable for a variety of workflows and development styles.

2.2 The Basic Stuff

Before working with more complex programs, it is essential to understand some of the basic building blocks of Python code. These include how to define and use variables, how to write comments, how Python handles the end of statements, and how to display output using the `print()` function.

Variables

In Python, variables are created by simply assigning a value to a name using the = operator. There is no need to declare the type of the variable beforehand — Python figures it out automatically based on the value assigned (This is an additional aspect of Python that makes it slow).

```
x = 10
name = "Alice"
pi = 3.14159
```

Variable names can contain letters, numbers, and underscores, but must not begin with a number. They are also case-sensitive, so `score` and `Score` would refer to different variables.

Comments

Comments are used to add notes or explanations to your code. In Python, anything following a hash mark # on a line is ignored by the interpreter:

```
# This is a comment
x = 42  # This is also a comment
```

Comments are helpful for documenting what your code does, especially in more complex scripts.

Statement Termination

Unlike some programming languages (such as C, Java, and Rust) where each statement must end with a semicolon, Python uses newlines to mark the end of a statement. In most cases, each statement simply ends at the end of the line:

```
x = 3
y = 4
z = x + y
```

Semicolons are technically allowed to separate multiple statements on a single line, but their use is discouraged in idiomatic Python (i.e. multiple statements on a single line makes code harder to read and thus is discouraged):

```
x = 3; y = 4; print(x + y)  # This works, but it's not recommended
```

The `print()` Function

The built-in `print()` function is used to display output in the terminal. It can take one or more arguments and prints them with spaces in between:

```
print("Hello, world!")
print("The value of x is", x)
```

The `print()` function automatically adds a newline at the end of its output. You can suppress this by adding the `end` keyword argument:

```
print("Hello", end=" ")
print("world!")
# Output: Hello world!
```

You can also use formatted strings (f-strings) to embed variable values inside strings:

```
name = "Alice"
age = 30
print(f"{name} is {age} years old")
```

This makes it easy to combine variables and text in readable output.

Special Characters in Strings

When using the `print()` function, you can include special characters inside string literals by using escape sequences. These are preceded by a backslash (\) and represent characters that are not easy to type directly or have special meaning.

Common examples include:

```
print("Line 1\nLine 2")     # \n starts a new line
print("Tab\tseparated")     # \t inserts a tab
print("A backslash: \\")    # \\ prints a single backslash
print("He said: \"Hello\"") # \" allows quotes inside strings
```

These sequences help format output more clearly, especially when printing multi-line or structured text.

Calling Functions

In Python, a function is a reusable block of code that performs a specific task. To *call* a function means to execute it. Functions may take input arguments and may return output values.

A function call consists of the function's name followed by parentheses. If the function takes arguments, they are placed inside the parentheses and separated by commas.

```
print("Hello")           # Calling the built-in print() function
len("Python")            # Returns the length of the string
max(3, 7, 2)             # Returns the maximum of the given values
```

Python provides many built-in functions like `print()`, `len()`, `type()`, `max()`, and `abs()`. You can also use functions from imported modules (this will be discussed more later):

```
import math
math.sqrt(16)            # Calls the sqrt function from the math module
```

For a complete list of Python's built-in functions, see the official documentation at https://docs.python.org/3/library/functions.html.

Lists and Tuples

Lists and tuples are two of the most commonly used data structures in Python for storing collections of items.

Lists are ordered, mutable (changeable) sequences of elements. They are defined using square brackets:

```
fruits = ["apple", "banana", "cherry"]
numbers = [1, 2, 3, 4, 5]
```

You can modify the contents of a list after it is created:

```
fruits[0] = "orange"  # Replace "apple" with "orange"
```

Tuples are similar to lists, but they are immutable — once created, their contents cannot be changed. Tuples are defined using parentheses:

```
coordinates = (3, 4)
colors = ("red", "green", "blue")
```

Tuples are often used when you want to ensure that a collection of values should not be altered, such as fixed coordinates or configuration settings.

Indexing

Both lists are *indexed*, meaning you can access individual elements by their position in the sequence. **Python uses zero-based indexing**, so the first element is at index `0`, the second at `1`, and so on:

```
print(fruits[0])   # Outputs: orange
print(colors[2])   # Outputs: blue
```

You can also use negative indices to count from the end of the sequence:

```
print(fruits[-1])  # Outputs the last element: cherry
print(fruits[-2])  # Outputs the second-to-last element: banana
```

You can also use index *slices* to retrieve *parts* of a list:

```
x = [1,2,3,4,5]
print(x[0:3])   # Outputs [1,2,3]. Note the slice is from 0 to 3, excluding 3
```

Attempting to access an index that is out of range will result in an `IndexError`.

Control Flow: `if` Statements and Loops

Control flow allows your program to make decisions and repeat actions. This is essential for writing dynamic and responsive code.

Conditional Statements (`if`)

The `if` statement allows you to execute code only when a certain condition is true. It can be extended using `elif` (else if) and `else` for alternative paths:

```
x = 10

if x > 0:
    print("x is positive")
elif x == 0:
    print("x is zero")
else:
    print("x is negative")
```

Note that the condition is followed by a colon, and the indented block underneath it defines the code that will be executed if the condition is true. Python uses indentation (usually 4 spaces) to mark blocks of code, rather than braces or keywords.

`for` Loops

A `for` loop is used to iterate over a sequence such as a list, tuple, or string. It repeats a block of code once for each item in the sequence:

```
fruits = ["apple", "banana", "cherry"]

for fruit in fruits:
    print(fruit)
```

You can also use `range(n)` to loop over a sequence of numbers:

```
for i in range(5):
    print(i)  # Prints numbers 0 to 4
```

`while` Loops

A `while` loop keeps executing as long as a specified condition remains true:

```
x = 0
while x < 5:
    print(x)
    x += 1  # Same as x = x + 1
```

This means that `while` can lead to *infinite* loops:

```
x = 0
while x < 5:
    print(x)
    x -= 1  # x will always be less than 5 !
```

On the other hand, `for` loops should *never* lead to infinite loops.

Breaking Out of Loops

You can use the `break` statement to exit a loop early, and `continue` to skip the rest of the current iteration and move to the next one:

```
for i in range(10):
    if i == 5:
        break    # Exit the loop when i is 5
    print(i)
```

```
for i in range(5):
    if i == 2:
        continue  # Skip printing 2
    print(i)
```

These control flow tools are the foundation for making decisions and writing logic in Python programs.

Defining Functions

Functions allow you to group reusable blocks of code under a single name. You can define your own functions in Python using the `def` keyword, followed by the function name, parentheses (which may include parameters), and a colon. The function body is indented beneath the definition.

```
def greet():
    print("Hello!")
```

This function can be called later using its name followed by parentheses:

```
greet()
```

Functions can also take arguments and return values:

```
def square(x):
    return x * x

result = square(4)  # result is now 16
```

Using Type Hints

Python is a dynamically typed language, which means that types do not need to be declared. However, it is considered good practice—especially in scientific or collaborative code—to use *type hints* to indicate the expected types of function arguments and return values. This makes code easier to read, maintain, and debug.

As an example, take this code:

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

This function is obviously meant to be used with numeric data, but what happens when you call this function with `add("hello","world")`? The code doesn't crash, but it returns the value `"helloworld"` (The '+' operator in Python is addition for numeric types and concatenation for strings).

Here is the same `square` function with type hints:

```
def square(x: float) -> float:
    return x * x
```

Note that Python **does not enforce** these types at runtime—they are primarily for the benefit of developers and tools such as linters and editors.

Importing Modules and Functions

Python comes with a large standard library of modules that provide useful functions and constants. Python also has a large array of external libraries that will also be useful and so it is important to know how to access the code in these libraries (installation of external libraries will not be covered as it is complicated to do so on the Digital Research Alliance of Canada clusters without causing problems). To access code from another module, you use the `import` statement.

One of the most commonly used standard modules is `math`, which includes mathematical functions and constants. We will use this module as the primary example (the documentation for this module can be found at https://docs.python.org/3/library/math.html.

Importing the Whole Module You can import the entire module using the `import` keyword. Then, you access its contents using dot notation:

```
import math

print(math.sqrt(16))        # Square root of 16
print(math.pi)              # The value of pi
print(math.sin(math.pi/2))  # Sine of 90 degrees (in radians)
```

Importing Specific Functions Alternatively, you can import specific functions or constants directly using the `from ... import ...` syntax:

```
from math import sqrt, pi

print(sqrt(25))    # No need to prefix with math.
print(pi)
```

This can make your code shorter, but it may reduce clarity if many names are imported from different modules.

Using Aliases You can also assign an alias to a module or function using the `as` keyword. This is useful to shorten long module names or avoid name conflicts:

```
import math as m

print(m.cos(m.pi))  # Cosine of pi
```

Why Use Modules? Modules help organize code into reusable components and give you access to tools without having to write them from scratch. Python includes many built-in modules (like `math`, `os`, and `random`), and you can also install third-party modules using tools like `pip`.

Understanding how to import and use modules is essential for leveraging the full power of Python's ecosystem.

Reading and Writing to Files

Reading from and writing to files is a common task in many Python programs. The built-in `open()` function is used to work with files. It takes the filename and the mode as arguments. Common modes include:

- `'r'` – read (default mode)
- `'w'` – write (creates a new file or overwrites an existing one)
- `'a'` – append (writes to the end of the file if it exists)

Writing to a File To write to a file, open it in `'w'` or `'a'` mode. Use the `write()` method to write text to the file.

```
with open("example.txt", "w") as file:
    file.write("This is a line of text.\n")
    file.write("Here is another line.")
```

The `with` statement ensures the file is properly closed after writing, even if an error occurs.

Reading from a File To read the contents of a file, open it in `'r'` mode and use methods such as `read()`, `readline()`, or `readlines()`:

```
with open("example.txt", "r") as file:
    contents = file.read()
    print(contents)
```

This reads the entire file as a single string. You can also read the file line by line:

```
with open("example.txt", "r") as file:
    for line in file:
        print(line.strip())  # .strip() removes the newline character
```

2.3 Some Useful Python Modules

To make the most of Python, we often rely on additional modules that provide specialized functionality for tasks such as numerical computing, data analysis, and plotting.

On the cluster, we do not have permission to install Python packages system-wide using `pip` (we can install packages for just our user using pip, but this will likely lead to problems). Instead,

we must either use the modules that are already provided in the system environment, or set up isolated installations using virtual environments.

For this course, we will make use of a pre-configured environment module called the **SciPy stack**, which includes a wide range of commonly used scientific Python packages. These include `numpy` for numerical computing, `scipy` for scientific computing, `matplotlib` for plotting, `pandas` for data manipulation, and others.

To access these modules on the cluster, you must load the SciPy stack environment with the following command:

```
module load scipy-stack
```

Once the module is loaded, you can run Python as usual, and these additional libraries will be available for import in your scripts.

If you need to use a Python package that is not included in the SciPy stack, you **cannot** simply run `pip install` as you would on your personal machine (even on your personal machine this will lead to dependency issues - the Python module versioning and dependency requirements are a complete nightmare). Instead, you must create a virtual environment in your user space and install the package there. While we will not go into detail about virtual environments in this course, it's important to be aware that this is the proper method for working with additional Python modules on the cluster (and is good practice in general).

For our purposes, the SciPy stack provides everything we need.

Working with the `os` Module

The `os` module is part of Python's standard library and provides functions for interacting with the operating system in a way that works across different platforms (Linux, macOS, Windows). It is particularly useful when working with file paths, directories, and environment variables—especially when automating tasks or integrating Python with the command line.

Accessing Environment Variables

You can use `os.environ` to access environment variables:

```
import os

home_dir = os.environ["HOME"]
print("My home directory is:", home_dir)
```

If you are unsure whether a variable exists, it's safer to use the `get` method:

```
username = os.environ.get("USER", "unknown")
```

Working with Directories

You can get the current working directory using:

```
cwd = os.getcwd()
print("Current directory:", cwd)
```

You can change directories using:

```
os.chdir("/path/to/another/directory")
```

You can also list the contents of a directory:

```
files = os.listdir(".")
print("Files in current directory:", files)
```

Creating and Removing Directories

To create or remove directories:

```
os.mkdir("new_folder")       # Create a single folder
os.makedirs("a/b/c")         # Create nested directories
os.rmdir("new_folder")       # Remove an empty folder
```

For removing non-empty directories, use the `shutil` module instead.

Working with File Paths

The `os.path` submodule contains tools for safely constructing and manipulating file paths:

```
path = os.path.join("data", "file.txt")
print("Constructed path:", path)

exists = os.path.exists(path)
is_file = os.path.isfile(path)
is_dir = os.path.isdir(path)
```

Using `os.path.join` is better than manually combining paths with slashes, because it works correctly on all operating systems.

The `os` module is an essential tool for writing portable and flexible Python scripts that interact with the file system or depend on the runtime environment.

Working with the `pandas` Module

The `pandas` module is a powerful library for data manipulation and analysis. It is widely used in scientific computing, data science, and machine learning. The core data structures in `pandas` are the `Series` (1D) and `DataFrame` (2D), which allow you to store, manipulate, and analyze structured data efficiently.

To use `pandas`, you must first import it (commonly using the alias `pd`):

```
import pandas as pd
```

Creating DataFrames

You can create a `DataFrame` from a dictionary or a list of lists:

```
data = {
    "name": ["Einstein", "Eddington", "Kepler"],
    "age": [146, 142, 453]
}

df = pd.DataFrame(data)
print(df)
```

This will produce a table-like structure with columns labeled `name` and `age`.

Reading and Writing Data Files

A major strength of `pandas` is its ability to read and write various data formats, especially CSV (comma-separated values) files:

```
df.to_csv("output.csv", index=False) # Write DataFrame to CSV
df_new = pd.read_csv("output.csv")         # Read CSV into a DataFrame
```

The `read_csv` function automatically parses headers and infers data types.

Accessing and Modifying Data

You can inspect the top of the dataset using:

```
print(df.head())   # First 5 rows
```

Accessing a column is straightforward:

```
print(df["name"])
```

You can also filter rows based on conditions:

```
pre_Newton = df[df["age"] > 382]
```

New columns can be created or modified as follows:

```
df["age_plus_one"] = df["age"] + 1
```

Summary Statistics

pandas makes it easy to get a quick statistical summary of your data:

```
print(df.describe())
```

You can also compute specific statistics:

```
mean_age = df["age"].mean()
```

The pandas library is an essential tool for any kind of structured data analysis in Python. It enables you to work with tabular data efficiently and is particularly well-suited to tasks involving reading, filtering, transforming, and analyzing data from external sources. There is much more to the pandas library than I have described here, but this should be a good starting point for anyone who is interested.

Working with the matplotlib Library

matplotlib is Python's most widely used library for creating static, animated, and interactive visualizations. It is highly customizable and integrates well with other libraries like numpy and pandas. For basic plotting, we use the pyplot module, which is typically imported using the alias plt:

```
import matplotlib.pyplot as plt
```

Plotting Data

The most basic type of plot is a line plot, which can be created from lists or numpy arrays:

```
x = [0, 1, 2, 3, 4]
y = [0, 1, 4, 9, 16]

plt.plot(x, y)
plt.title("Example Plot")
plt.xlabel("x values")
plt.ylabel("y values")
plt.show()
```

The plt.show() function opens a window or displays the plot inline (e.g., in a Jupyter notebook).

Scatter Plots and Histograms

You can also easily create other types of plots:

```
# Scatter plot
plt.scatter(x, y)
plt.show()

# Histogram
import numpy as np
data = np.random.randn(1000)
plt.hist(data, bins=30)
plt.show()
```

Saving Figures

You can save plots to image files (PNG, PDF, etc.) using:

```
# ... create plot
plt.savefig("myplot.png")
```

This is useful when running scripts on a remote cluster or when generating plots for a report.

Multiple Plots and Subplots

You can overlay multiple plots or create subplots:

```
# Overlaying multiple lines
plt.plot(x, y, label="squares")
plt.plot(x, [i**3 for i in x], label="cubes")
plt.legend()
plt.show()

# Creating subplots
fig, (ax1, ax2) = plt.subplots(1, 2)
ax1.plot(x, y)
ax2.plot(x, [i**3 for i in x])
plt.show()
```

`matplotlib` is an essential tool for visualizing data. Whether you're exploring datasets, debugging computations, or preparing publication-quality figures, `matplotlib` provides the flexibility and control to produce a wide range of visualizations.

Other Scientific Modules

In addition to `pandas`, `matplotlib`, and `os`, there are several other Python modules that are widely used in scientific computing. While we will not cover these in detail during this course, it is helpful to be aware of them and their typical uses:

- `numpy` – The foundational library for numerical computing in Python. It provides efficient array and matrix operations, mathematical functions, and tools for working with large datasets in a vectorized way. Many other scientific libraries depend on `numpy`.
- `scipy` – Builds on `numpy` to provide advanced numerical routines for tasks such as integration, optimization, root-finding, signal processing, and linear algebra.
- `sympy` – A symbolic mathematics library useful for algebraic manipulation, calculus, equation solving, and working with mathematical expressions analytically.
- `astropy` – A library developed for astronomy and astrophysics. It includes tools for handling units and constants, celestial coordinates, time conversions, and common data formats used in astronomy.
- `seaborn` – A statistical data visualization library built on top of `matplotlib`. It offers high-level interfaces for creating attractive and informative plots, especially with `pandas` DataFrames.
- `numba` – A just-in-time (JIT) compiler for Python that speeds up numerical code by compiling annotated functions to machine code. It is especially useful for accelerating loops and array operations.
- `cython` – A superset of Python that allows writing C extensions for Python. It can significantly speed up Python code by compiling it to C, especially in performance-critical applications.
- `F2PY` – A tool included with `numpy` that allows you to call Fortran code directly from Python. It can be used to incorporate modern Fortran routines into Python programs, enabling significant speed improvements for numerically intensive computations.

These modules, along with those we have covered more fully, form the backbone of the Python scientific computing ecosystem. Note that not all of them are included in the SciPy stack module available on the cluster.

3 MESA

4 SLURM

[heading=bibintoc]