Scripting & Automation in Linux

- By Anas Mohammad, Experimental Data Engineer @SESAME
- (28-04-2025) session start at 09:00 AM UTC

Topics:

- 10 mins — Introduction about scripting & automation in Linux
- 40 mins — Bash scripting basics
- 45 mins — Python scripting methods
- 25 mins — Scheduling tasks in Linux

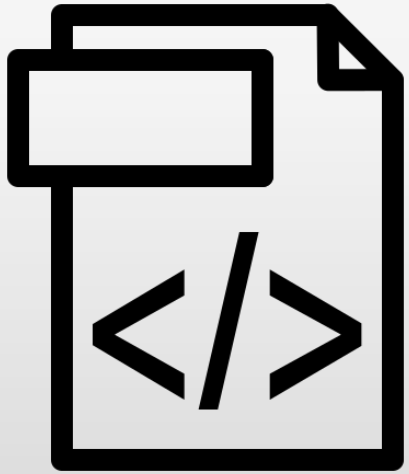
Bash scripting hands-on

Python scripting hands-on

Introduction about scripting & automation in Linux

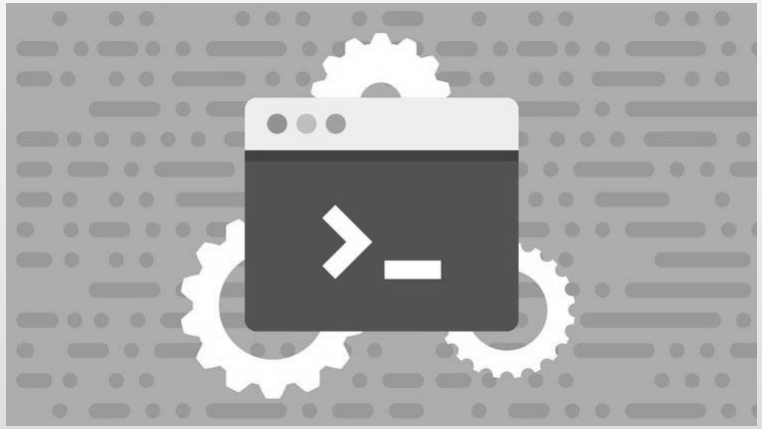
What is Scripting in Linux?

Scripting in Linux refers to writing and executing shell scripts, which are files containing a series of commands that the shell (command-line interpreter) executes in sequence.



What is Automation in Linux?

Automation means using scripts, tools, and scheduling methods to perform tasks automatically without manual intervention.



Why Use Scripting and Automation?

- ✔ Reduces manual effort and human errors
- ✔ Improves efficiency and consistency
- ✔ Saves time on repetitive tasks
- ✔ Helps in system administration and DevOps tasks

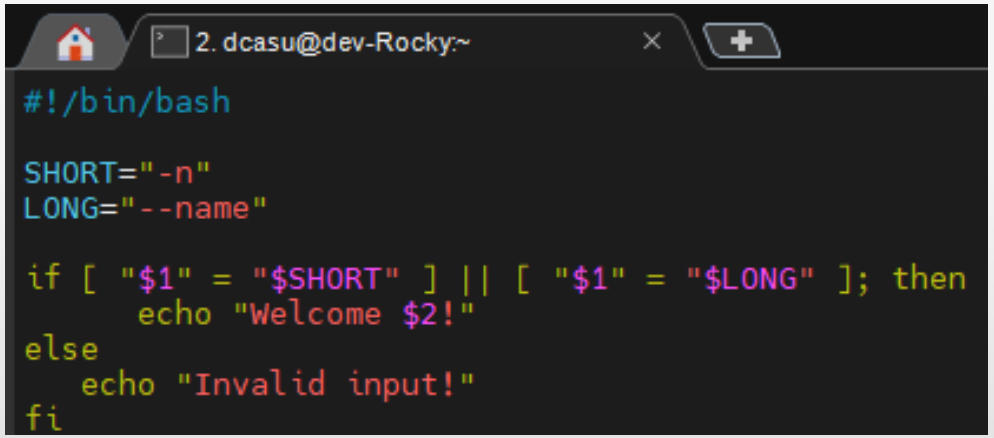
Key Aspects of Linux Scripting:

Bash Scripting

- Bash (Bourne Again Shell) is a command-line interpreter for Unix/Linux.
- It allows users to execute commands and write scripts to automate tasks.
- Common uses: System automation, server management, DevOps, etc.

Get Started with Bash Scripting

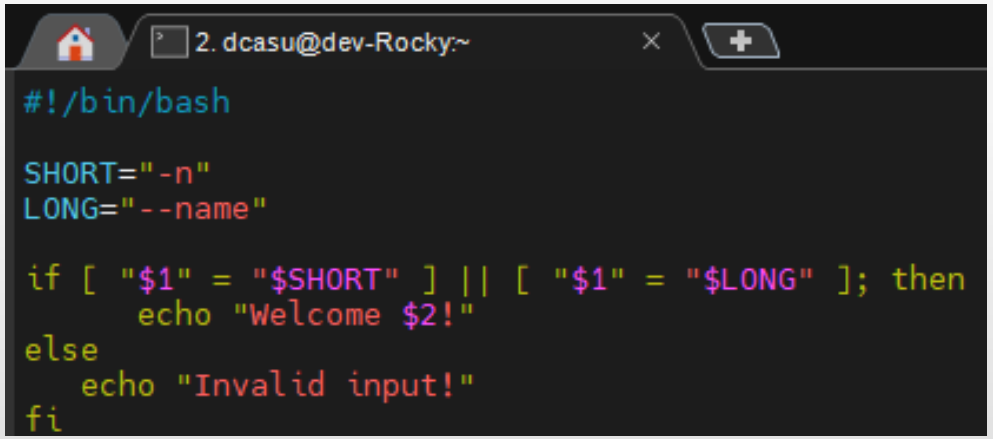
- Naming convention (`.sh`)
- Shebang (`#!/bin/bash`)
- System Commands (`mv`, `echo`, ..etc.)



Executing Bash Scripts

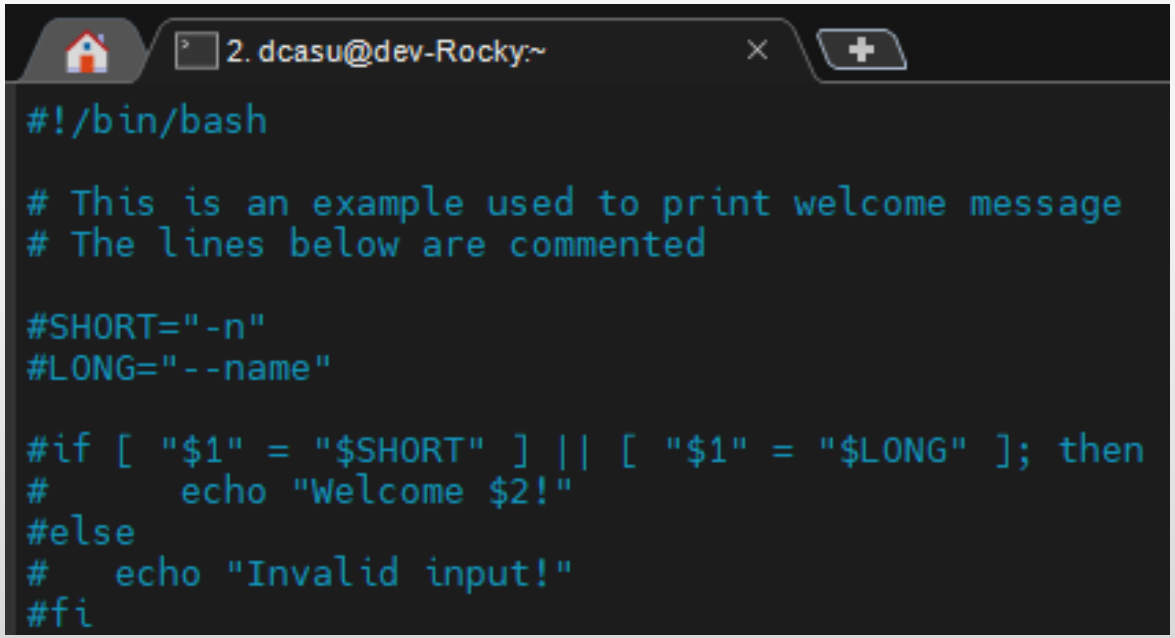
To make the script executable, assign execution rights to the user using this `chmod` command:

- Make it executable (`chmod +x test.sh`)
- run it using one of these commands:
  * bash test.sh
  *./test.sh



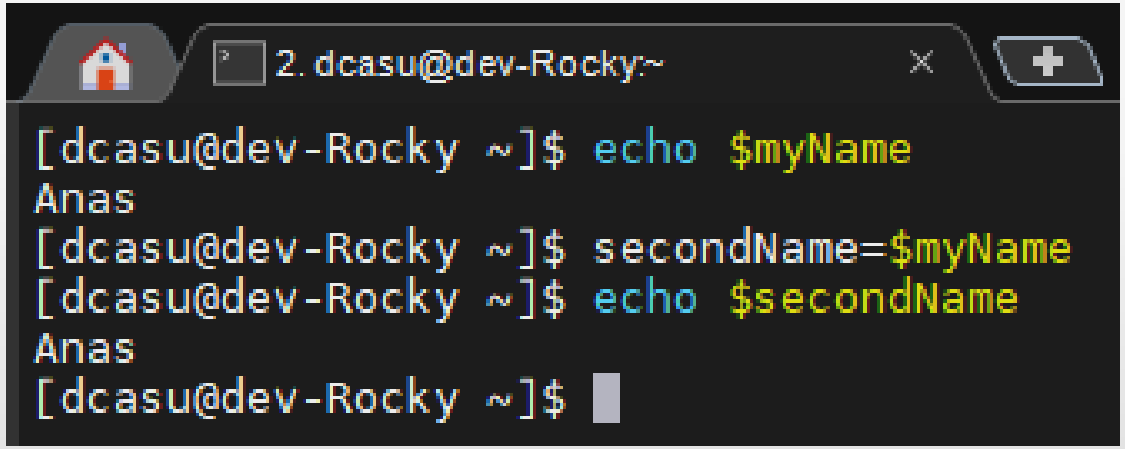
Bash Scripting Basics

1. Comments: comments start with a (#) in bash scripting, they are very helpful in documenting the code, in order to help others understand the code.



Bash Scripting Basics

2. Variables: variables are used to store data values without specifying data types. In Bash, a variable is capable of storing numeric values, strings of characters, ..etc.



Bash Scripting Basics

Variable naming conventions:

Rule	*Description*	*Example (Valid)*	*Example (Invalid)*
Start with a letter or underscore	Variable names must begin with a letter (a-z, A-Z) or an underscore (_)	name = “Anas" _name = “Anas"	1name = "Anas" ✗
Can contain letters, numbers, and underscores	Allowed characters: letters (a-z, A-Z), numbers (0-9), underscores (_)	user_age = 25	user-age = 25 ✗
Case-sensitive	`Count` and `count` are different variables	count = 10 Count = 20	-
No spaces or special characters	Variable names should not include spaces or special characters	userName = "Anas"	user name = "Anas" ✗ user@name = "Anas" ✗
Use descriptive names	Name variables based on their purpose	total_price = 99.99	x = 99.99 (Not descriptive)
Avoid reserved keywords	Don't use bash keywords (if, else, …etc.) as variable names	my_name = “Anas"	if = “Anas" ✗

Bash Scripting Basics

3. Input and output:

Concept	*Example*

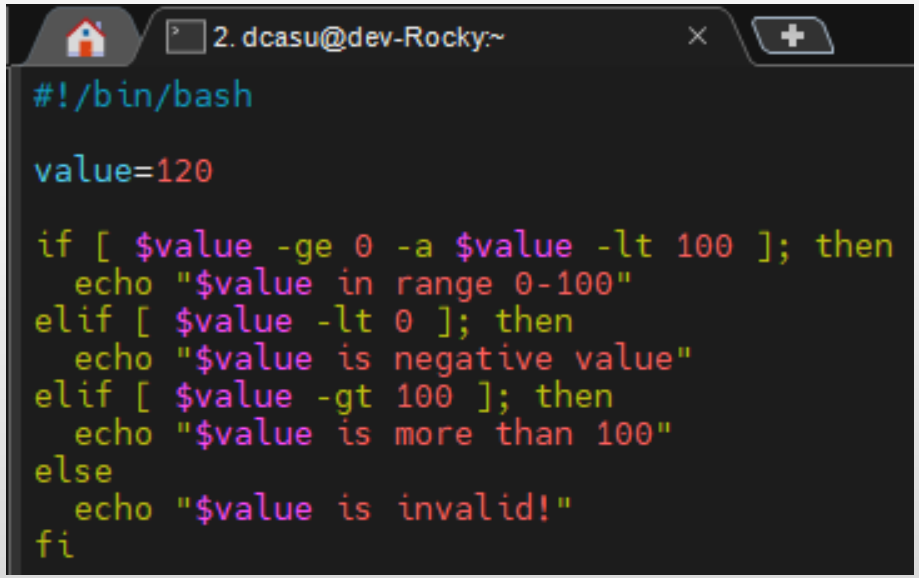
Reading user input (input)	`echo "What's your name?"`
	`read name`
	`echo "Welcome $name!"`

Reading from a file (input)	`while read line;`
	`do`
	`  echo $line`
	`done < test.txt`

Printing to terminal (output)	`echo "Hello, World!"`
Redirecting output (output)	`echo "output of testing script!" > output.txt`

Bash Scripting Basics

4. Conditional statements (if/else): used for decision-making, there are several ways to evaluate conditions, including if, if-else, if-elif-else, and nested conditionals



Bash Scripting Basics

List of conditions operators in Bash, including numeric comparisons, string comparisons, file checks, and logical operators:

1. Numeric Comparisons

Operator	Description	Example
-eq	Equal to (==)	[$a -eq $b]
-ne	Not equal to (!=)	[$a -ne $b]
-gt	Greater than (>)	[$a -gt $b]
-lt	Less than (<)	[$a -lt $b]
-ge	Greater than or equal to (>=)	[$a -ge $b]
-le	Less than or equal to (<=)	[$a -le $b]

Bash Scripting Basics

List of conditions operators in Bash, including numeric comparisons, string comparisons, file checks, and logical operators:

2. String Comparisons

Operator	Description	Example
= or ==	Strings are equal	["$a" = "$b"] or [["$a" == "$b"]]
!=	Strings are not equal	["$a" != "$b"]
-z	String is empty (None or Null)	[-z "$a"]
-n	String is not empty	[-n "$a"]

Bash Scripting Basics

List of conditions operators in Bash, including numeric comparisons, string comparisons, file checks, and logical operators:

3. File Checks

Operator	Description	Example
-e	File exists	[-e file.txt]
-f	File is a regular file	[-f file.txt]
-d	Directory exists	[-d /path/to/dir]
-r	File is readable	[-r file.txt]
-w	File is writable	[-w file.txt]
-x	File is executable	[-x script.sh]
-s	File is not empty	[-s file.txt]

Bash Scripting Basics

List of conditions operators in Bash, including numeric comparisons, string comparisons, file checks, and logical operators:

4. Logical Operators

Operator	Description	Example
-a	AND (inside [ ], less preferred)	[$a -gt 5 -a $b -lt 10]
-o	OR (inside [ ], less preferred)	[$a -gt 5 -o $b -lt 10]
&&	AND (inside [[ ]], preferred)	[[$a -gt 5 && $b -lt 10]]
!	NOT (negation)	[! -f file.txt]

Bash Scripting Basics

5. Looping and Branching: Loops repeat actions multiple times (for loop, while loop, case statement)

2. dcasu@dev-Rocky:~

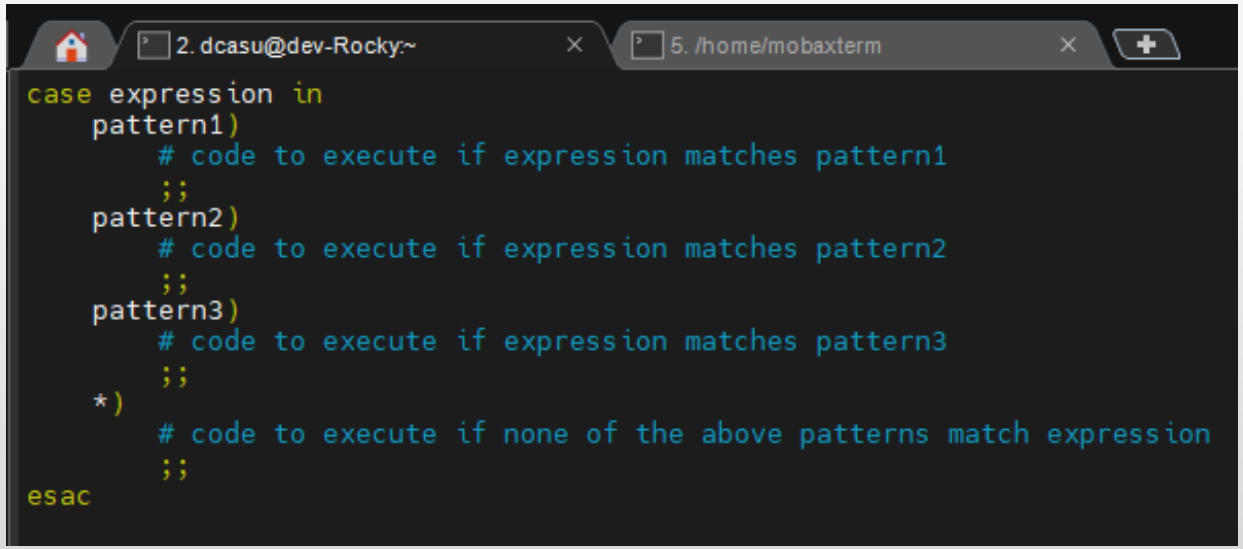
```
#!/bin/bash

for i in {1..5}
do
  echo "Number: $i"
done


#!/bin/bash
i=90
while [[ $i -le 100 ]] ; do
   echo "$i"
  (( i += 1 ))
done
```

For loop

while loop

Bash Scripting Basics

5. Looping and Branching: Loops repeat actions multiple times (for loop, while loop, case statement)




```
#!/bin/bash

fruit="apple"

case $fruit in
    "apple")
        echo "This is a red fruit."
        ;;
    "banana")
        echo "This is a yellow fruit."
        ;;
    "orange")
        echo "This is an orange fruit."
        ;;
    *)
        echo "Unknown fruit."
        ;;
esac
```

Bash Scripting Basics

6. Functions: Bash functions allow you to reuse code by grouping commands into a single unit. They help in making scripts more modular, readable, and maintainable.

Bash Scripting Basics

Feature	Description	Example
Defining a Function	A function is defined using curly braces {} or the function keyword.	`greet() { echo "Hello!"; }`
Calling a Function	Simply write the function name to execute it.	`greet`
Function with Arguments	Use $1, $2, … to access arguments passed to the function.	`greet_user() { echo "Hello, $1!"; } greet_user "Alice"`
Returning a Value	Use echo to return a value (Bash does not support return for values).	`add() { echo $(( $1 + $2 )); } result=$(add 5 10); echo "Sum: $result"`
Exit Status (Return Code)	Functions return **0** for success and **non-zero** for failure using return.	`check() { return 1; } check; echo $?`
Local Variables	Use local to restrict a variable's scope to the function.	`my_func() { local x=10; echo $x; }`
Conditional Function Calls	Use a function inside an if statement.	`is_admin() { [ "$1" == "admin" ]; } if is_admin "admin"; then echo "Allowed"; fi`
Recursive Function	A function can call itself (with a base condition).	`factorial() { [ $1 -le 1 ] && echo 1`
Error Handling	Use return or set -e to handle errors.	`check_file() {[ -f "$1" ]}`

Bash Task

Build a simple server/client system using Netcat (nc) and Bash

Description:

You will create two Bash scripts:

- server.sh — sets up a simple chat server.
- client.sh — connects to the server and allows a user to send/receive messages.

Details:

- **Server (server.sh)**:
  - Listens on a specified port.
  - Accepts incoming messages and displays them.
- **Client (client.sh)**:
  - Connects to the server IP and port.
  - Sends typed messages to the server.
  - Displays responses from the server.
- Log all messages to a file (on both client and server).

Tools/Concepts Involved:

- netcat (nc)
- read and echo
- loops
- redirection (>, <)
- background processes (&)

Python Scripting Methods in Linux

What is Python?
Python is a high-level, interpreted programming language. It emphasizes code readability with its notable use of significant whitespace.

Why use Python for scripting in Linux?
Python comes pre-installed on most Linux distributions. Its clear syntax makes automation scripts easy to write and maintain.

The extensive standard library provides ready solutions for most system tasks. Python scripts work across different Linux versions with minimal changes.

Python Modules, Methods, Design Patterns

1. Logging
2. os and sys
3. Time and Datetime
4. argparsing
5. Subprocess
6. File Operations (Txt, CSV, JSON)

Package Manager in Python

What is pip and Why Use It?

Package Management Simplified

- `pip` streamlines the installation and management of Python packages, enhancing development efficiency

Dependency Handling

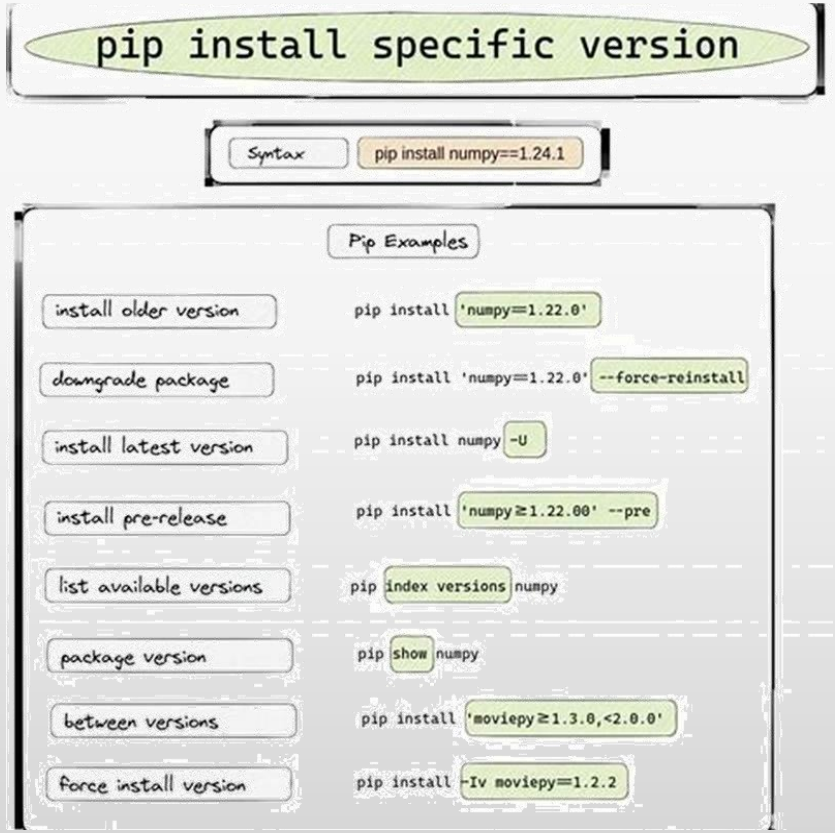
- Automatically resolves and installs package dependencies, reducing conflicts and ensuring compatibility

Environment Isolation

- Integrates with virtual environments, allowing separate project dependencies to prevent version conflicts

Installing Packages with pip

Basic Installation Command: The fundamental command for installing a package with pip is `pip install package_name`, which initiates the process of downloading and integrating the specified package into the current Python environment, ensuring access to its functionalities.



Pip basic commands

Command	Description	Example
pip install <package_name>	Install a package from PyPI (Python Package Index).	pip install requests
pip install <package_name>==<version>	Install a specific version of a package.	pip install requests==2.25.1
pip uninstall <package_name>	Uninstall a package.	pip uninstall requests
pip freeze	List installed packages with their versions.	pip freeze
pip show <package_name>	Show detailed information about a package.	pip show requests
pip list	List installed packages (without version details).	pip list
pip search <query>	Search for a package on PyPI (deprecated in newer versions).	pip search requests
pip install -r <requirements_file>	Install all the packages listed in a requirements file.	pip install -r requirements.txt
pip install --upgrade <package_name>	Upgrade an installed package to the latest version.	pip install --upgrade requests
pip install --user <package_name>	Install a package for the current user only (not globally).	pip install --user requests
pip check	Check installed packages for dependency conflicts.	pip check

Understanding requirements.txt

Each line specifies a package with optional version constraints:

- package==1.0.0 (exact version)
- package>=1.0.0 (minimum version)
- package>=1.0.0,<2.0.0 (version range)

Lines starting with # are treated as comments and ignored during installation.

requirements.txt

```
1  cryptography==1.1
2  brotlipy==1.1.1
3  idna==3.0.3
4  pyOpenSSL==2.0.4
5  PySocks==4.2.5
6  glob2==4.2.3
7  py==1.6.7
8  pytest==5.6.7
9  attrs==8.1.1
10 selenium==1.0.2
11 pandas==1.2.4
```

Setting Up a Virtual Environment (Only Python dependencies)

What is a Virtual Environment?

1 Isolated Project Dependencies: Virtual environments prevent conflicts by isolating libraries for each project.

2 Customizable Python Interpreter: Each environment can use a specific Python version tailored to project needs.

3 Simplified Collaboration: Sharing projects becomes easier with a self-contained environment, ensuring consistent setups.

How to create venv ?

Install venv

- `sudo pipX.X install virtualenv`

Create venv

- `pythonX.X -m venv myvenv`
- `virtualenv -p pythonX.X myvenv`

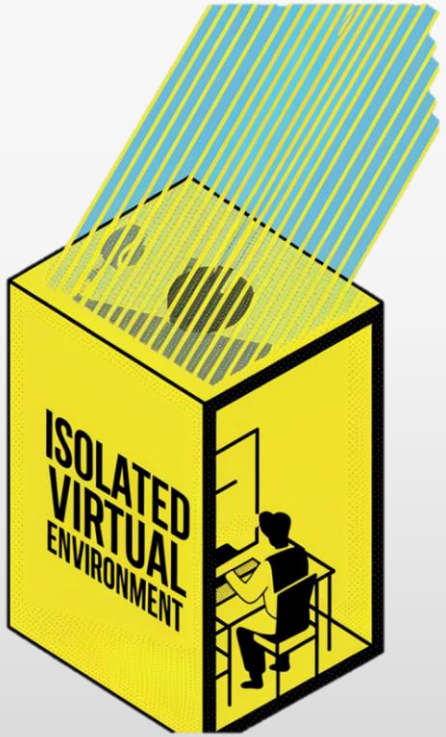
Activate venv

- `source myenv/bin/activate`

Deactivate venv

- `deactivate`

How the venv is being activated ?



- PATH is modified to prioritize the virtual environment
- Python and pip now point to the versions in your environment
- Shell prompt is updated as a visual indicator (`(myenv) user@host:~$`)
- Environment-specific packages become available

Install packages in venv

Activate First: Always ensure your virtual environment is activated

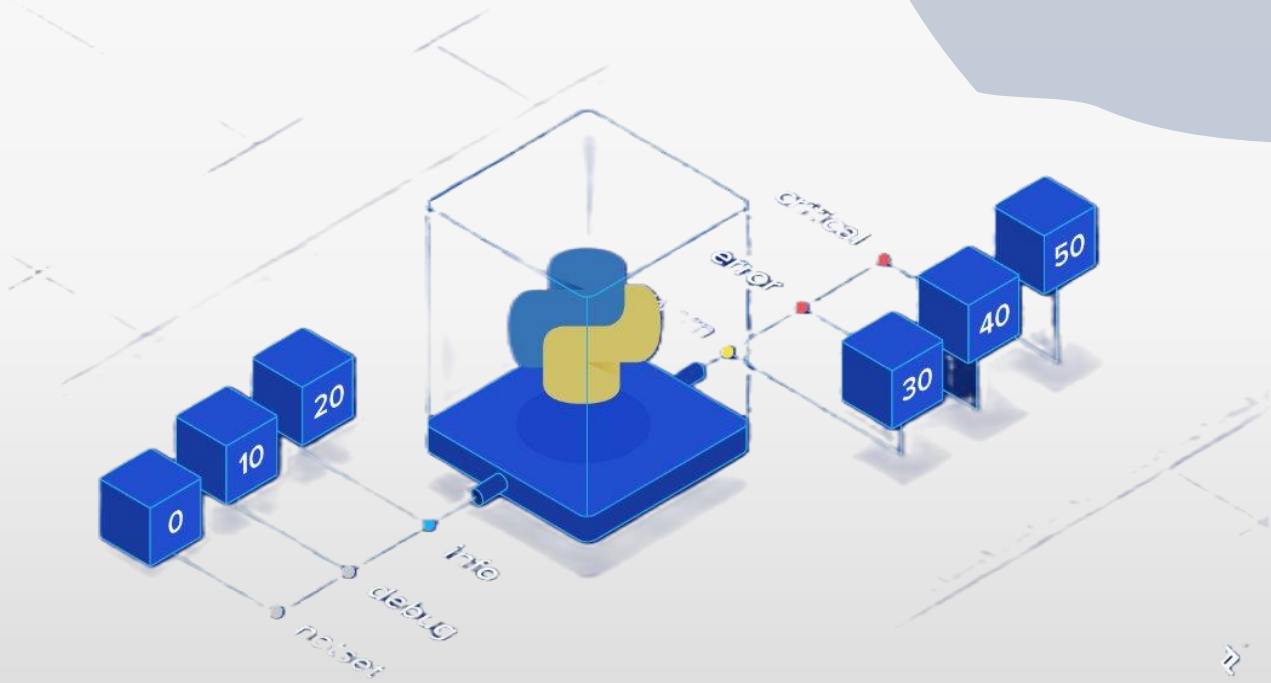
Use pip Normally: pip install package_name works as usual

Use Requirements File: pip install -r requirements.txt for multiple packages

Verify Installation: pip list shows packages installed in this environment only

Logging

- Logging is the process of recording events that happen during a program’s execution.
- It helps track bugs, monitor performance, and understand application flow.
- Preferred over print() for real-world applications.



Logging Levels

- DEBUG: Detailed info for diagnosing problems
- INFO: Confirmation that things are working as expected
- WARNING: Something unexpected happened, but the program still works
- ERROR: A more serious problem, the program may not work as expected
- CRITICAL: A serious error, program may not continue

OS and SYS

- Python has built-in modules to interact with the OS and system.
- os: Interact with the operating system (files, directories, paths).
- sys: Access Python runtime environment and system-specific parameters.

What is the os Module?

- Interface between Python and the OS.
- Useful for:
- File and directory operations
- Environment variables
- Process management

Common os Module Functions

- `os.name` – Name of the OS (`'posix'`, `'nt'`)
- `os.getcwd()` – Get current working directory
- `os.chdir(path)` – Change current directory
- `os.listdir(path)` – List files and directories
- `os.mkdir(path)` – Create a new directory
- `os.remove(filename)` – Delete a file
- `os.path` – Submodule for file paths

What is the sys Module?

Provides access to system-specific parameters and functions.

Useful for:

- Command-line arguments
- Exiting scripts
- Interacting with stdin, stdout, stderr
- Accessing module paths

Common sys Module Functions

```
sys.argv – List of command-line arguments
sys.exit() – Exit the program
sys.path – List of directories for module search
sys.version – Python version
sys.platform – Platform identifier
```

Time and datatime

Why Work With Time in Python?

- Logging timestamps
- Scheduling tasks
- Measuring execution time
- Parsing and formatting date/time strings
- Handling time zones and daylight saving

The time Module

- Provides access to time-related functions.
- Works with UNIX timestamps (seconds since Jan 1, 1970).

```
import time
time.time() # Current timestamp
time.sleep(2) # Sleep for 2 seconds
time.localtime() # Convert timestamp to struct_time
```

The datetime Module

- More powerful and object-oriented.
- Supports dates, times, time zones, and arithmetic.

Key Classes:

- datetime.date
- datetime.time
- datetime.datetime
- datetime.timedelta
- datetime.timezone

```
from datetime import datetime
now = datetime.now()
print(now) # 2025-04-25 12:34:56.789000
now.strftime("%Y-%m-%d %H:%M:%S")
datetime.strptime("2025-04-25", "%Y-%m-%d")
```

argparsing

What is argparse?

- argparse is a built-in Python module for parsing command-line arguments.
- Helps make Python scripts user-friendly and flexible.
- Replaces manual sys.argv parsing with a cleaner, more scalable approach.

Why Use argparse?

- Automatically generates help and usage messages.
- Handles both positional and optional arguments.
- Validates and converts argument types.
- Clean syntax with minimal boilerplate.

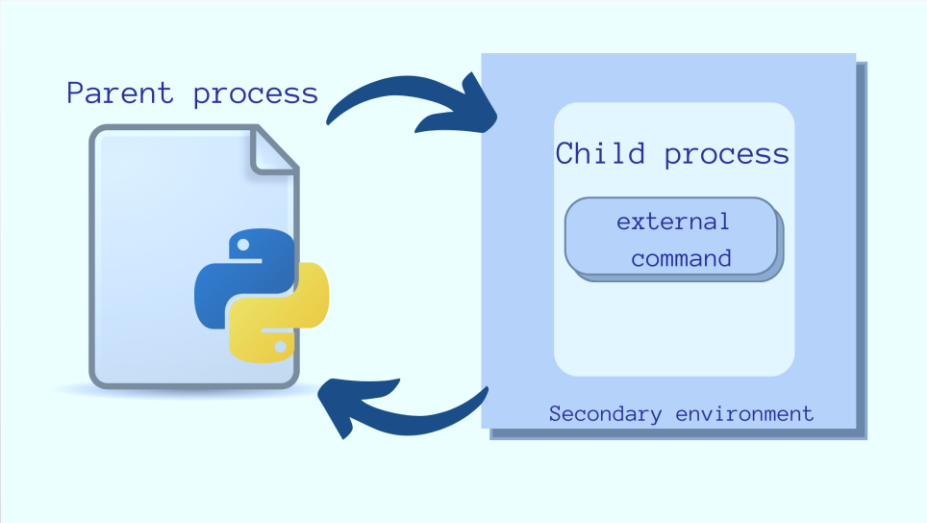
argparsing example

```
import argparse
parser = argparse.ArgumentParser()
args = parser.parse_args()
parser = argparse.ArgumentParser(description="My awesome script")
parser.add_argument("--mode", help="Choose the mode")
parser.add_argument("y", type=int)
```

Subprocess

The subprocess module in python is a built-in module that allows us to create new child processes. We can get exit codes, output, and error messages from these child processes. It is a valuable tool for executing external functions or commands in your Python code.

```
import subprocess

subprocess.run(["ls", "-l"])

result = subprocess.run(
    ["echo", "Hello from subprocess"],
    capture_output=True,
    text=True
)
print(result.stdout)

result = subprocess.run(
    ["false"],
    capture_output=True
)
print(result.returncode)
```



File operations (txt, csv, json)

Text file operation modes in python:

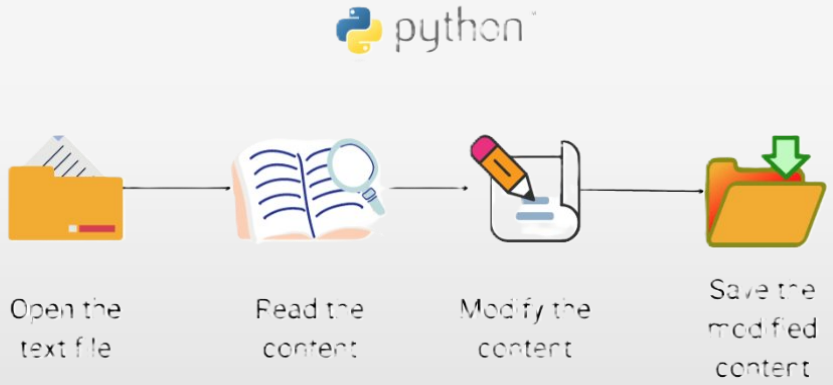
```
open("example.txt", Mode):
```

Mode	Read	Write	Create if Missing	Truncate	Append	Error if Exists	Description
r	☑	✗	✗	✗	✗	☑	Read a text file
r+	☑	☑	✗	✗	✗	☑	Read and write a text file
w	✗	☑	☑	☑	✗	✗	Write to a text file (overwrite)
w+	☑	☑	☑	☑	✗	✗	Read and write (overwrite) text
a	✗	☑	☑	✗	☑	✗	Append to a text file
a+	☑	☑	☑	✗	☑	✗	Read and append to text
x	✗	☑	☑	✗	✗	☑	Create new text file (error if exists)

File operations (txt, csv, json)

Working with .txt Files

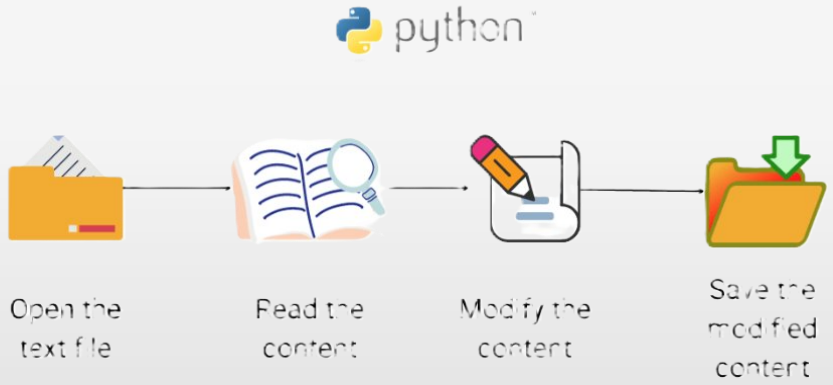
```
# Writing to a text file
with open("example.txt", "w") as file:
    file.write("Hello, World!")

# Reading from a text file
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
```



File operations (txt, csv, json)

Working with `.csv` Files

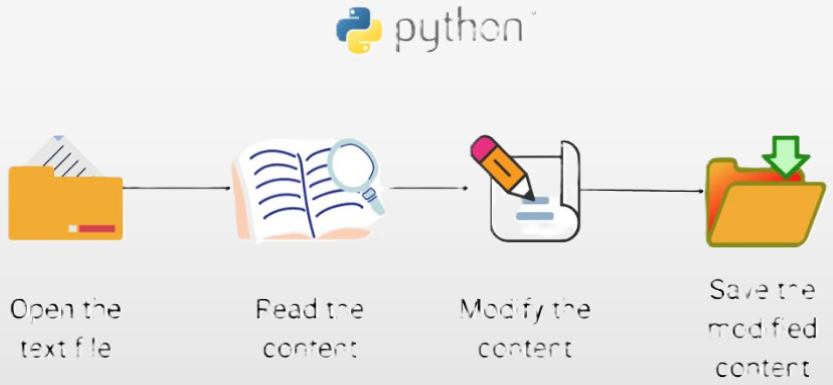
```
import csv

# Writing to CSV
with open("data.csv", "w", newline="")
as file:
    writer = csv.writer(file)
    writer.writerow(["Name", "Age"])
    writer.writerow(["Alice", 25])

# Reading from CSV
with open("data.csv", "r") as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```



File operations (txt, csv, json)

Working with `.json` Files

```
import json

# Writing JSON
data = {"name": "Alice", "age": 25}
with open("data.json", "w") as file:
    json.dump(data, file)

# Reading JSON
with open("data.json", "r") as file:
    content = json.load(file)
    print(content)
```



Python Task

- Accept a **folder path** via argparse
- Periodically **scan** the folder (every few seconds) for changes
- Use **subprocess** to *ls* Linux the folder
- Log **file events** with **logging** to a logfile
- Write a **summary report** in **TXT**, **CSV**, and **JSON** formats
- Use **os** and **sys** to handle paths, exits, etc.
- Use **time** and **datetime** for timestamps

Scheduling & Automation in Linux

Linux scheduling tools

Cron

The classic and most widely used Linux scheduler

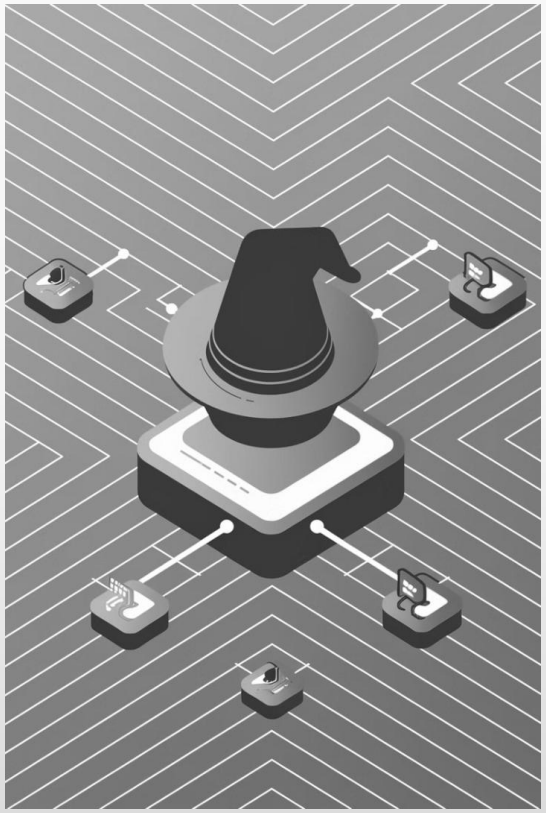
- Time-based job scheduler
- Built into all Linux distributions

At Command

For one-time scheduled tasks

- Simple interface
- Queue-based scheduling

Cron (Background Service)



Cron is a time-based job scheduler in Unix-like operating systems. It enables users to schedule commands or scripts to run at specific times or intervals.

The cron daemon (crond) runs continuously in the background. It starts automatically when the system boots up.

Automates system maintenance tasks, backups, notifications, and any process that needs to run regularly without human intervention.

Crontab Syntax

```
*  *  *  *  *  command_to_run
|  |  |  |  |
|  |  |  |  +----- Day of the week (0 - 7) (Sunday = 0 or 7)
|  |  |  +------- Month (1 - 12)
|  |  +---------- Day of the month (1 - 31)
|  +------------- Hour (0 - 23)
+---------------- Minute (0 - 59)
```

Each crontab entry follows this five-field time pattern. After these fields comes the command to execute.

- Asterisk (*): Matches all possible values for the field. Example: * in the hour field means "every hour".
- Comma (,): Separates multiple values for one field. Example: 1,3,5 in day field means 1st, 3rd, and 5th day.
- Hyphen (-): Defines a range of values. Example: 1-5 in day field means Monday through Friday.
- Forward slash (/): Specifies step values. Example: */4 in hour field means every 4th hour.

Crontab Commands

- **crontab –e:** Opens your crontab file in the default editor. Creates a new one if it doesn't exist.
- **crontab –l:** Displays the current user's crontab file. Shows all scheduled jobs.
- **crontab –r:** Deletes your entire crontab file. Use with caution!
- **crontab –u:** Specifies which user's crontab to modify. Only available to root.

Cron Generator: https://crontab.guru/

At Command

The "at" command schedules one-time tasks to run at a specified time.

Perfect for jobs that don't need to recur.

Syntax:

Time Specifications

- now + 5 minutes
- noon
- Midnight
- teatime (4pm)
- Tomorrow
- noon tomorrow
- next week
- now + 1 hour
- now + 30 minutes

Date Specifications

- 4:00 PM July 31
- July 31, 2025
- 31.7.2025
- 07/31/2025
- next Monday
- Friday

Task about cron and at

Very simple cron and at tasks to run a bash script after 5 min, the bash script redirect the output of // command to a file.

Learning References:

1. Python Course for Dr. Ghaith Abandah from University of Jordan: https://www.abandah.com/gheith/?page_id=13

2. Online web for hands on (Code Camp): https://www.freecodecamp.org/