Exam 1

Basic Navigation and File Management

Key topics include understanding the current working directory, changing directories, creating directories, listing files, and symbolic links.

Current Working Directory (cwd)

The current working directory is the directory you are presently working in. Knowing your cwd is fundamental for file management and navigation in Linux.

Current Working Directory

The command 'pwd' (print working directory) displays the current working directory path.

- **Usage**: Simply type 'pwd' and press Enter.
- **Example**: If you are in '/CSPB/Courses/CSPB-1300/Materials', 'pwd' will return '/CSPB/Courses/CSPB-1300/Materials'.

Changing Directories

Changing directories allows you to navigate through the file system.

Changing Directories

The command 'cd' is used to change the current working directory.

- **Usage**: 'cd [directory]'
- **Relative Path**: 'cd ../../CS-2270/Instructors' moves you up two directories and then into 'CS-2270/Instructors'.
- **Special Characters**:
  - '.' (dot): current directory
  - '..' (double dot): parent directory
  - '~' (tilde): home directory

Creating Directories

Creating directories is essential for organizing files.

Creating Directories

The command 'mkdir' is used to create new directories.

- **Usage**: 'mkdir [directory_name]'
- **Example**: 'mkdir cs3308' creates a directory named 'cs3308' in the current working directory.

Listing Files

Listing files helps you view the contents of directories.

Listing Files

The 'ls' command is used to list files and directories.

- **Basic Usage**: 'ls'
- **Long Format**: 'ls -l' displays detailed information.
- **Including Hidden Files**: 'ls -a'
- **Sorted by Time**: 'ls -t'

- **Sorted by Size**: 'ls -lS'
- **Hidden Files**: Files starting with a dot (e.g., '.secret_file') are hidden.

Symbolic Links

Symbolic links are references to other files or directories.

Symbolic Links

The command 'ln -s' creates symbolic links.

- **Usage**: 'ln -s [target] [link_name]'
- **Example**: 'ln -s /path/to/file symlink_name' creates a symbolic link 'symlink_name' pointing to '/path/to/file'.

Advanced File Management and Environment Handling

Key topics include renaming and removing directories, environment variables, redirection, piping, aliases, permissions, and command types.

Renaming and Removing Directories

Efficiently managing directories involves renaming and removing them when necessary.

Renaming and Removing Directories

Use 'mv' to rename and 'rm -r' to remove directories.

- **Renaming**: 'mv wrong_name correct_name' renames 'wrong_name' to 'correct_name'.
- **Removing**: 'rm -r bigData' removes the directory 'bigData' and all its contents.

Environment Variables

Environment variables store information about the system configuration and user environment.

Environment Variables

Environment variables are crucial for configuring the behavior of processes and the shell.

- **Accessing Variables**: Use '$VARIABLE_NAME' to access the value of an environment variable.
- **Common Variables**:
  - 'PATH': Directories to search for executable files.
  - 'HOME': The current user's home directory.
  - 'USER': The current user's username.

Redirection and Piping

Redirecting output and piping commands are fundamental for effective command-line operations.

Redirection and Piping

Redirection changes where the input/output of commands are sent, and piping sends output from one command as input to another.

- **Redirect Output**: 'command > file' writes the output to 'file'.
- **Redirect Errors**: 'command 2> err.txt' captures errors.
- **Pipe Commands**: 'command1 | command2' uses the output of 'command1' as input for 'command2'.

Aliases

Aliases create shortcuts for commands, making complex or frequently used commands simpler to execute.

Aliases

The 'alias' command defines custom shortcuts for commands.

- **Creating Aliases**: 'alias name='command"
- **Example**: 'alias ll='ls -l" creates an alias 'll' for 'ls -l'.

Permissions

File and directory permissions control access and actions that users can perform.

Permissions

Permissions determine the actions that can be performed on files and directories.

- **Types**: read (r), write (w), execute (x)
- **Changing Permissions**: 'chmod [permissions] [file]'
- **Example**: 'chmod 755 myfile' sets read, write, and execute permissions for the owner, and read and execute for others.

Common Commands

Commonly used commands facilitate routine tasks like copying, moving, and removing files.

Common Commands

Basic commands for managing files and directories.

- **Copy Files**: 'cp source destination'
- **Move Files**: 'mv source destination'
- **Remove Files**: 'rm file'
- **Create Links**: 'ln -s target link_name'

Capturing Errors and Using Pipes

Managing command outputs and errors effectively enhances script and command reliability.

Capturing Errors and Using Pipes

Handling errors and chaining commands are essential skills.

- **Error Redirection**: 'command 2> err.txt' captures command errors in 'err.txt'.
- **Piping Commands**: 'command1 | command2' sends 'command1' output to 'command2' input.
- **Example**: 'ls -l | wc -l' lists files and counts lines.

Basics of Regular Expressions

Regular expressions are powerful tools used for pattern matching and text manipulation. Key concepts include special characters, character classes, and repetition operators.

Special Characters in Regular Expressions

Special characters have unique meanings and are used to define specific patterns.

Special Characters

Understanding special characters is essential for effective pattern matching.

- ^: Represents the beginning of a line.
- **$**: Represents the end of a line.
- **.**: Matches any single character.
- +: Matches one or more occurrences of the previous character.

Character Classes

Character classes allow for the matching of specific sets of characters.

Character Classes

Character classes simplify pattern definitions by grouping characters.

- [**:alnum:**]: Matches any alphanumeric character.
- [**:alpha:**]: Matches any alphabetic character.
- [**:digit:**]: Matches any numeral.
- [**:lower:**]: Matches any lowercase letter.
- [**:upper:**]: Matches any uppercase letter.
- [**:blank:**]: Includes the space and tab characters.
- [**:punct:**]: Matches punctuation characters.
- [**:space:**]: Matches whitespace characters including space, tab, and newline.

Common Patterns

Common patterns are used to match typical text formats.

Common Patterns

Some regular expressions are commonly used for matching standard text formats.

- **Valid Username**: ^[a-z0-9_-]3,16$ matches usernames with 3 to 16 characters, including lowercase letters, numbers, underscores, and hyphens.
- **Email Address**: ^[A-Z0-9._%+]+[A-Z0-9.]+.[A-Z]2,$ matches a standard email format.

Introduction to Scripting

Scripting involves writing small programs to automate tasks. Key concepts include using commands, handling variables, and controlling flow.

Basic Commands

Common commands are used to perform essential operations in scripts.

Basic Commands

Basic commands facilitate the execution of common tasks in scripts.

- **echo**: Used to print text to the output.
- **history**: Displays the list of previously executed commands.
- **chmod**: Changes the permissions of a file.

Permissions

File permissions determine the actions that can be performed on files.

Permissions

Permissions are crucial for security and proper file handling.

- **r**: Read permission.
- **w**: Write permission.
- **x**: Execute permission.
- **Changing Permissions**: Use 'chmod [permissions] [file]' to modify permissions.
- **Example**: 'chmod 755 myfile' sets read, write, and execute permissions for the owner, and read and execute for others.

Conditional Statements

Conditional statements control the flow of execution based on conditions.

Conditional Statements

IF statements and their required keywords manage decision-making in scripts.

- **Required Keywords**:
  - 'if': Begins the conditional statement.
  - 'then': Follows the condition.
  - 'else': Provides an alternative if the condition is false.
  - 'fi': Ends the conditional statement.

Positional Parameters

Positional parameters allow scripts to accept input arguments.

Positional Parameters

Positional parameters enable dynamic input handling in scripts.

- **$1, $2, etc.**: Represent the first, second, and subsequent arguments passed to the script.
- **Example**: In a script 'myscript', the command 'myscript first second' assigns 'first' to '$1' and 'second' to '$2'.
- **Output Example**: The script 'cd ..; echo $2 $1' will output 'second first'.

Key Concepts

This section encapsulates the main ideas and commands necessary for understanding regular expressions and scripting in Linux.

- **Regular Expressions**:
  - Special characters such as ^ (beginning of a line), $ (end of a line), . (any single character), and + (one or more occurrences of the previous character) are used for pattern matching.
  - Character classes like '[:alnum:]' (alphanumeric characters), '[:alpha:]' (alphabetic characters), '[:digit:]' (numerals), '[:lower:]' (lowercase letters), '[:upper:]' (uppercase letters), '[:blank:]' (space and tab characters), '[:punct:]' (punctuation characters), and '[:space:]' (whitespace characters) simplify pattern definitions.
  - Common patterns for matching typical text formats such as '^[a-z0-9_-]3,16$' for valid usernames and '^[A-Z0-9._%+]+[A-Z0-9.]+.[A-Z]2,$' for email addresses.
- **Scripting Basics**:

- Basic commands like 'echo' (print text to output), 'history' (display previously executed commands), and 'chmod' (change file permissions) are essential for script operations.
- File permissions ('r' for read, 'w' for write, 'x' for execute) control access to files and can be modified using 'chmod [permissions] [file]'.
- Conditional statements in scripts use keywords such as 'if', 'then', 'else', and 'fi' to manage decision-making processes.
- Positional parameters ('$1', '$2', etc.) allow scripts to handle input arguments dynamically. For example, in a script 'myscript', the command 'myscript first second' assigns 'first' to '$1' and 'second' to '$2'.

Key Roles in Agile/Scrum Teams

Agile/Scrum teams have specific roles, each with distinct responsibilities that contribute to the efficiency and effectiveness of the development process.

Product Owner

The Product Owner prioritizes the feature backlog and ensures the team is working on the most valuable features.

Product Owner

Responsibilities of the Product Owner include:

- Prioritizing the feature backlog.
- Ensuring the team understands the project goals and requirements.
- Maximizing the value of the product through effective backlog management.

Sprint Team

The Sprint Team is a self-organizing group of developers responsible for building features.

Sprint Team

Characteristics of the Sprint Team include:

- Self-organizing and cross-functional.
- Committed to delivering potentially shippable product increments.
- Collaborates closely to meet sprint goals.

Scrum Master

The Scrum Master acts as a coach to the team, ensuring adherence to Scrum practices.

Scrum Master

Responsibilities of the Scrum Master include:

- Facilitating Scrum ceremonies (daily stand-ups, sprint planning, reviews, and retrospectives).
- Removing impediments that hinder the team's progress.
- Ensuring the team follows Scrum processes and practices.

Agile Manifesto Principles

The Agile Manifesto outlines key principles that prioritize individuals, interactions, working software, and customer collaboration over processes and tools.

Agile Manifesto Principles

Key principles from the Agile Manifesto include:

- **Individuals and interactions over processes and tools.**
- **Working software over comprehensive documentation.**
- **Customer collaboration over contract negotiation.**
- **Responding to change over following a plan.**

Daily Scrum Meetings

Daily scrums are short, time-boxed meetings where team members discuss their progress and obstacles.

Daily Scrum Meetings

Daily scrums should:

- Be limited to 15 minutes.
- Include updates on what was completed, plans until the next scrum, and any obstacles.
- Facilitate communication and quick resolution of issues.

Sprint Duration

Sprints are fixed time periods during which specific work has to be completed and made ready for review.

Sprint Duration

Common characteristics of sprints include:

- Duration typically ranges from 1 to 4 weeks.
- Each sprint should produce fully functional features.
- Sprints provide regular opportunities for reassessment and adaptation.

Agile Metrics and Tools

Agile teams use various charts and metrics to track progress and performance.

Burn Down Chart

A Burn Down chart shows the amount of work remaining in a sprint or release.

Burn Down Chart

Characteristics of a Burn Down chart include:

- Tracks the progress of work to be completed.
- Helps in predicting when the work will be completed.
- Represents the percent (or number) of features not yet implemented.

Burn Up Chart

A Burn Up chart shows the amount of work completed.

Burn Up Chart

Characteristics of a Burn Up chart include:

- Tracks the progress of work completed.
- Helps in visualizing scope changes and completed work over time.

- Represents the percent (or number) of features implemented.

Pair Programming

Pair programming involves two developers working together on the same code.

Pair Programming

Benefits and practices of pair programming include:

- **Switching Roles**: Developers should switch roles every half-hour to maximize engagement and learning.
- **Cost Effectiveness**: Despite increased development time, defect counts are lowered, reducing overall development costs.
- **Collaboration**: Enhances code quality and knowledge sharing among team members.

Waterfall Model

The waterfall model is a traditional software development approach.

Waterfall Model

Characteristics of the waterfall model include:

- Sequential design process.
- Suitable when requirements are well-defined and stable.
- Less flexible in accommodating changes compared to Agile methodologies.

Key Concepts

This section encapsulates the main ideas and practices essential for Agile development.

- **Agile/Scrum Roles**:
  - **Product Owner**: Prioritizes the feature backlog.
  - **Sprint Team**: Self-organizing group of developers building features.
  - **Scrum Master**: Ensures adherence to Scrum practices and removes impediments.
- **Agile Principles**:
  - Individuals and interactions over processes and tools.
  - Working software over comprehensive documentation.
- **Agile Practices**:
  - Daily scrums limited to 15 minutes.
  - Sprints typically lasting 1-4 weeks.
- **Agile Metrics**:
  - Burn Down Chart: Tracks work remaining.
  - Burn Up Chart: Tracks work completed.
- **Pair Programming**:
  - Switching roles every half-hour.
  - Lowering defect counts to reduce development costs.
- **Waterfall Model**:
  - Sequential process suitable for well-defined requirements.

Git Commands and Their Functions

Understanding Git commands is crucial for efficient version control and collaboration.

Basic Git Commands

These commands are fundamental for initializing, managing, and navigating repositories.

Basic Git Commands

Essential Git commands and their functions include:

- **git init**: Creates an empty local repository.
- **git add**: Stages a file under Git tracking, making it ready for a commit.
- **git commit**: Copies a staged file into the local Git repository.
- **git checkout**: Moves the HEAD pointer to a different branch, making that branch active.
- **git diff**: Compares two versions of a Git-managed file.
- **git log**: Shows a record of recent commits.

Cloning and Connecting Repositories

Cloning creates a local copy of a remote repository, and understanding Git's distributed nature is important.

Cloning and Connecting Repositories

Key points about cloning and connecting repositories include:

- **git clone**: Creates a working directory and makes a local copy of the repository.
- Git does not need to connect to a server to commit changes; commits are made to the local repository and pushed to a remote server.

Commits and Tracking Changes

Git uses hashes to uniquely identify commits, ensuring integrity and traceability.

Commits and Tracking Changes

Important aspects of commits and tracking include:

- **Unique Identification**: Each commit is uniquely identified by a hash.
- **Manual Tracking**: After initializing a repository with "git init", files must be manually added to be tracked.

Branching and Merging

Branching allows isolation of work, and merging integrates changes back into the main codebase.

Branching and Merging

Benefits and uses of branching include:

- **Isolation of Work**: Branches allow changes without affecting the main codebase, facilitating experimentation and new feature development.
- **Collaboration**: Branches can be shared with others, who can then make changes and submit them back via pull requests.
- **Release Management**: Different branches manage various codebase versions, like development and release branches.
- **Bug Fixes**: Separate branches for fixing bugs that can be merged back without disrupting ongoing development.

- **Feature Development**: Develop new features in isolation and merge them back after completion and testing.

Pulling and Updating

Pulling updates the local repository with changes from the remote repository.

Pulling and Updating

Key points about pulling and updating include:

- **git pull**: Downloads changes from the remote repository into the local working copy.

Key Concepts

This section encapsulates the main ideas and practices essential for version control using Git.

- **Git Commands**:
  - "git init", "git add", "git commit", "git checkout", "git diff", "git log".
  - "git clone" for creating a local copy of a remote repository.
  - "git pull" for updating the local repository with remote changes.
- **Commits and Tracking**:
  - Commits are uniquely identified by hashes.
  - Files must be manually added to be tracked after initializing a repository.
- **Branching and Merging**:
  - Branches isolate work, facilitate collaboration, manage releases, and develop features independently.
  - Bug fixes and feature developments can be handled in separate branches and merged back into the main branch.
- **Cloning and Pulling**:
  - "git clone" creates a working directory and copies the repository locally.
  - "git pull" updates the local repository with changes from the remote repository.

Introduction To Software Testing

Software testing is the process of evaluating and verifying that a software application or system meets the specified requirements. The primary goals of testing are to identify defects, ensure quality, and validate that the software performs as expected.

Introduction to Software Testing

Key points about software testing include:

- **Definition**: The process of evaluating and verifying that a software application or system meets the specified requirements.
- **Goals**: Identifying defects, ensuring quality, validating performance.
- **Types of Testing**:
  - Manual Testing: Performed by humans, involves checking software functionality without automated tools.
  - Automated Testing: Uses scripts and tools to perform tests, increases efficiency and coverage.

Unit Testing

Unit testing involves testing individual components or functions of a software application in isolation. The primary goal is to ensure that each unit of the software performs as expected.

Unit Testing

Key points about unit testing include:

- **Scope**: Focuses on individual components or functions.
- **Tools**: Common tools include JUnit for Java, PyTest for Python.
- **Benefits**:
  - Early defect identification.
  - Simplifies debugging and maintenance.

Integration Testing

Integration testing involves testing the interactions between different components or modules of a software application. The goal is to identify issues that occur when units are combined.

Integration Testing

Key points about integration testing include:

- **Scope**: Focuses on interactions between components.
- **Types**:
  - Top-down Integration: Testing from top to bottom.
  - Bottom-up Integration: Testing from bottom to top.
  - Sandwich Testing: Combination of both top-down and bottom-up.
- **Benefits**:
  - Detects interface issues.
  - Ensures components work together as intended.

System Testing

System testing involves testing the complete and integrated software system to verify that it meets the specified requirements. It is performed in an environment that closely resembles the production environment.

System Testing

Key points about system testing include:

- **Scope**: Focuses on the entire system.
- **Types**:
  - Functional Testing: Verifies software functions as expected.
  - Non-functional Testing: Checks performance, usability, security, etc.
- **Benefits**:
  - Validates the complete system.
  - Ensures the system meets requirements and specifications.

Acceptance Testing

Acceptance testing is the final level of testing performed to determine whether the software is ready for release. It verifies that the software meets the business requirements and is acceptable to the end-users.

Acceptance Testing

Key points about acceptance testing include:

- **Scope**: Focuses on business requirements and user needs.

- **Types**:
  - User Acceptance Testing (UAT): Conducted by end-users.
  - Business Acceptance Testing (BAT): Ensures software meets business goals.
- **Benefits**:
  - Ensures software is ready for production.
  - Validates software against business requirements.

Test-Driven Development (TDD)

Test-Driven Development is a software development approach in which tests are written before writing the actual code. It emphasizes writing small, incremental tests for each functionality.

Test-Driven Development (TDD)

Key points about TDD include:

- **Process**:
  - Write a test for a new functionality.
  - Write code to pass the test.
  - Refactor the code while ensuring the test still passes.
- **Benefits**:
  - Ensures code quality.
  - Reduces the likelihood of defects.

Behavior-Driven Development (BDD)

Behavior-Driven Development extends TDD by writing tests in a more human-readable format. It emphasizes collaboration between developers, testers, and business stakeholders.

Behavior-Driven Development (BDD)

Key points about BDD include:

- **Process**:
  - Write a scenario in a language like Gherkin.
  - Implement steps to satisfy the scenario.
  - Ensure tests pass for the defined behavior.
- **Benefits**:
  - Enhances communication among stakeholders.
  - Aligns development with business requirements.

Key Concepts

This section encapsulates the main ideas and practices essential for effective software testing.

- **Types of Testing**:
  - Unit Testing: Tests individual components or functions.
  - Integration Testing: Tests interactions between components.
  - System Testing: Tests the complete and integrated system.
  - Acceptance Testing: Final level of testing to verify readiness for release.
- **Testing Strategies**:
  - Test-Driven Development (TDD): Writing tests before code to ensure quality.

  - Behavior-Driven Development (BDD): Writing human-readable tests to enhance collaboration.
- **Benefits of Testing**:
  - Early identification of defects.
  - Improved code quality and maintainability.
  - Ensures software meets requirements and performs as expected.

Introduction to Flask

Flask is a micro web framework written in Python. It is lightweight and modular, making it easy to scale up to complex applications. Flask is known for its simplicity and flexibility.

Introduction to Flask

Key points about Flask include:

- **Definition**: Flask is a micro web framework for Python.
- **Characteristics**:
  - Lightweight and modular.
  - Simple and flexible.
  - Suitable for both small and large applications.
- **Philosophy**: Flask is designed to be simple and extensible.

Installation and Setup

Installing and setting up Flask is straightforward and involves using pip, Python's package installer.

Installation and Setup

Steps to install and set up Flask include:

- **Install Flask**:
  - Run 'pip install Flask'.
- **Create a Flask App**:
  - Create a Python file (e.g., 'app.py').
  - Import Flask and create an instance of the Flask class.
  - Define routes and their corresponding request handler functions.
  - Run the application using 'app.run()'.

Routes and Views

Routes map URLs to functions in your Flask application, and views are the functions that handle the requests.

Routes and Views

Key points about routes and views include:

- **Routes**:
  - Defined using the 'app.route' decorator.
  - Map URLs to Python functions.
- **Views**:
  - Functions that return responses for the routes.
  - Can return HTML, JSON, or other types of responses.

Templates

Flask uses the Jinja2 template engine to render HTML templates. Templates allow you to separate the presentation logic from the business logic.

Templates

Key points about templates include:

- **Jinja2 Template Engine**:
  - Used to render HTML templates.
  - Supports template inheritance and reusable components.
- **Rendering Templates**:
  - Use 'render_template' function to render templates.
  - Pass variables to templates to dynamically generate content.

Forms and Input Handling

Flask provides tools for handling form data and user input, making it easy to process and validate input from users.

Forms and Input Handling

Key points about forms and input handling include:

- **Handling Form Data**:
  - Use 'request.form' to access form data.
  - Use 'request.args' to access query parameters.
- **Validation**:
  - Use libraries like WTForms for form validation.
  - Validate input to ensure data integrity and security.

Database Integration

Flask can integrate with databases using SQLAlchemy or other ORM libraries to manage database operations.

Database Integration

Key points about database integration include:

- **SQLAlchemy**:
  - A popular ORM library for Flask.
  - Simplifies database operations and management.
- **Setup**:
  - Install SQLAlchemy with 'pip install Flask-SQLAlchemy'.
  - Configure the database URI and initialize the SQLAlchemy instance.
- **Models**:
  - Define models as Python classes.
  - Use models to create, read, update, and delete database records.

Blueprints

Blueprints allow you to organize your Flask application into modular components, promoting better code organization and reusability.

Blueprints

Key points about blueprints include:

- **Purpose**:
  - Organize the application into modules.
  - Enable code reuse and better organization.
- **Creating a Blueprint**:
  - Create a Blueprint instance.
  - Define routes and views within the blueprint.
  - Register the blueprint with the main application instance.

RESTful APIs

Flask can be used to build RESTful APIs, allowing for the creation of web services that follow REST principles.

RESTful APIs

Key points about RESTful APIs include:

- **REST Principles**:
  - Use standard HTTP methods (GET, POST, PUT, DELETE).
  - Use URIs to identify resources.
  - Stateless communication.
- **Creating an API**:
  - Define API routes using 'app.route'.
  - Return JSON responses using 'jsonify'.
  - Handle different HTTP methods within route functions.

Key Concepts

This section encapsulates the main ideas and practices essential for developing applications using the Flask framework.

- **Basic Concepts**:
  - Flask setup and installation.
  - Defining routes and views.
  - Using templates for rendering HTML.
  - Handling forms and input validation.
- **Advanced Concepts**:
  - Integrating databases with SQLAlchemy.
  - Organizing code with blueprints.
  - Building RESTful APIs.
- **Benefits of Flask**:
  - Lightweight and flexible.
  - Easy to learn and use.
  - Scalable for both small and large applications.