OpenEDG™ Python Institute
Certifying Python Skills for Real Careers
https://pythoninstitute.org

PCEA™ – Certified Entry-Level Automation Specialist with Python EXAM SYLLABUS

(Exam PCEA-30-01)

Last revised: September 2, 2025

Module 1. Fundamentals of Automation (6) (13%)

1.1 The Importance of Digitizing Tasks (1)

1.1.1 Identify examples of routine and repetitive tasks suitable for automation

A. Provide real-world examples from IT (file backups, log cleanup), business (report generation, data entry), and home contexts (organizing photos, scheduling reminders).
B. Distinguish between tasks that are effective to automate (repetitive, rule-based, time-consuming) and those that are not (creative, judgment-based).

1.2 Benefits and Limitations of Automation (2)

1.2.1 Explain the advantages of automation

A. Explain key benefits: consistency, accuracy, speed, scalability.
B. Identify cost savings, time savings, and productivity improvements achieved through automation.
C. Describe industry-wide benefits in IT (system monitoring), finance (data processing), and manufacturing (process control).
D. Describe how automation frees humans from routine tasks to focus on higher-value work.

1.2.2 Describe challenges and limitations of automation

A. Identify potential drawbacks: setup cost, script errors, dependency on systems, and maintenance overhead.
B. Explain why automation cannot fully replace human creativity, adaptability, and judgment.
C. Recognize situations where manual intervention remains necessary.

1.3 Levels of Automation (1)

1.3.1 Differentiate between scripting, process automation, and orchestration

A. Define *basic scripting* (e.g., Python scripts for file manipulation).
B. Describe *process automation* (workflow tools, task schedulers).
C. Explain *orchestration* (managing multiple processes and systems together).

1.4 Measuring the Value of Automation (1)

1.4.1 Apply basic methods to calculate ROI of automation

A. Identify metrics such as time saved, error reduction, and cost savings.
B. Apply formulas to simple ROI scenarios (e.g., hours saved × hourly cost).
C. Interpret whether an automation initiative delivers measurable value.

1.5 Python as a Tool for Automation (1)

1.3.1 Explain why Python is widely used for automation

A. Describe Python's strengths: readability, cross-platform compatibility, comprehensive standard library, and strong community support.
B. List popular Python libraries for automation: *subprocess*, *os*, *shutil*, *logging*, *requests*, and *schedule*.
C. Explain how Python can integrate with operating system commands, APIs, and external tools.

Module 2. Basic Command-Line Automation with Python (9) (19.5%)

2.1 Running Python Scripts from the Command Line (3)

2.1.1 Execute Python scripts using terminal/command prompt

A. Demonstrate how to run a script with *python script.py*.
B. Demonstrate running scripts from different directories.
C. Diagnose and resolve common errors (wrong path, missing interpreter).

2.1.2 Use script arguments with *sys.argv*

A. Demonstrate how to pass one or more arguments into a script at runtime (e.g., *python script.py input.txt*).
B. Access arguments in Python and apply them in tasks.
C. Analyze how argument values change program behavior (e.g., specifying input files, folder paths, or configuration options).
D. Implement a script that accepts a filename and prints its content or metadata.

2.1.3 Explain the role of virtual environments in automation

A. Define what a Python virtual environment is and why it is used.
B. Explain how virtual environments isolate dependencies and improve script portability.
C. Demonstrate creating (*python -m venv venv*), activating, and deactivating a virtual environment.
D. Identify scenarios where using a virtual environment is critical (e.g., when deploying automation on different systems).

2.2 Script Configuration and Execution Basics (3)

2.2.1 Use shebang lines for Unix/Linux/macOS automation

A. Explain the function of a shebang line in executable Python scripts.
B. Write the standard shebang *#!/usr/bin/env python3* in a script.
C. Demonstrate how to make a script executable with *chmod +x*.
D. Verify execution of a script without explicitly calling the Python interpreter.

2.2.2 Apply output and error redirection for Python scripts

A. Redirect standard output (*stdout*) from a script into a text file using >.

B. Redirect error messages (*stderr*) into a separate file using *2>*.
C. Apply combined redirection (&> or equivalents) to capture all script output.
D. Analyze script logs to separate successful runs from error traces.

2.2.3 Describe environment variables in automation

A. Define environment variables and explain their purpose in automation.
B. Identify common environment variables such as *PATH*, *HOME*, or *USERNAME*.
C. Demonstrate how to read environment variables in Python using *os.environ*.
D. Apply environment variables to customize script behavior (e.g., dynamic file paths or system-specific configurations).
E. Demonstrate how to create and assign values to environment variables.

2.3 Integration with System Tools (3)

2.3.1 Explain how automation scripts are combined with OS-level tools

A. Describe *Task Scheduler* (Windows) and *cron* (Linux/macOS) as scheduling tools.
B. Explain how Python scripts can be configured to run automatically using these tools.
C. Evaluate scenarios where OS-level scheduling is more appropriate than in-script scheduling.
D. Provide examples such as scheduling a daily cleanup or weekly backup.

2.3.2 Use *subprocess* to execute external commands

A. Execute shell commands from within Python using the *subprocess* module.
B. Capture command output and return codes to verify execution success.
C. Redirect external command output into files or Python variables for further processing.
D. Provide examples such as running *ls* (Linux/macOS) or *dir* (Windows) and analyzing results.

2.3.3 Identify use cases for command-line automation

A. Recognize tasks that benefit from command-line automation, such as batch file renaming, backups, and log cleanup.
B. Evaluate how command-line automation reduces manual effort in repetitive tasks.
C. Compare scenarios where Python automation is more effective than manual command entry.
D. Provide examples of combining multiple scripts into a single automated workflow.

Module 3. Logging and Monitoring Essentials (7) (15%)

3.1 Understanding the Role of Logging (2)

3.1.1 Explain why logging is essential in automation

A. Describe the role of logging for debugging, monitoring, and auditing automated tasks.
B. Explain the limitations of using *print()* functions compared to logging.
C. Provide examples of automation tasks where logs are essential (e.g., system monitoring, error tracking).
D. Explain why sensitive information (e.g., passwords, API keys) should never be logged.

3.1.2 Configure simple logging in Python

A. Demonstrate how to import and use the *logging* module.
B. Use *logging.basicConfig* to configure a simple logger.
C. Generate log messages with levels such as *INFO*, *WARNING*, and *ERROR*.
D. Write log outputs to both console and file.

3.2 Log Levels and Formats (3)

3.2.1 Differentiate between logging levels

A. Define standard log levels: *DEBUG*, *INFO*, *WARNING*, *ERROR*, and *CRITICAL*.
B. Provide examples of when to use each log level (e.g., *DEBUG* for troubleshooting, *ERROR* for critical issues).
C. Explain why consistent use of log levels helps organize and analyze automation logs.

3.2.2 Apply custom formatting in logs

A. Configure log messages to include timestamps, log levels, and message details.
B. Demonstrate structured logging with custom formats for better monitoring.
C. Compare simple vs formatted log outputs to highlight clarity improvements.

3.2.3 Implement logging during file operations

A. Record every time a file is read or written by the script.
B. Log the number of lines processed in a file.
C. Compare logs of successful operations vs failed attempts.

D. Explain how logging improves traceability in file automation.

3.3 Monitoring Automation Tasks (2)

3.3.1 Implement basic monitoring strategies in automation

A. Record script start and end times automatically in logs.
B. Track the number of processed files, records, or tasks.
C. Detect and record errors during execution for later review.

3.3.2 Use logs to debug automation workflows

A. Interpret log messages to identify causes of script failures.
B. Use log analysis to differentiate between normal and abnormal system behavior.
C. Provide scenarios such as diagnosing a failed backup or handling an API timeout.

Module 4. Basic File and Data Automation (8) (17.5%)

4.1 File Operations with Python (3)

4.1.1 Perform file and folder operations

A. List and verify files or directories using os functions such as *os.listdir()* and *os.path.exists()*.
B. Create and delete directories programmatically to manage file structures.
C. Automate simple housekeeping tasks such as cleaning a temporary folder.
D. Open, read, write, and append to text files in different modes (*r*, *w*, *a*), including handling encodings like UTF-8.

4.1.2 Use *shutil* for advanced operations

A. Copy files from one location to another using *shutil.copy()*.
B. Move and rename files with *shutil.move()*.
C. Apply directory-level operations for organizing large groups of files.
D. Implement a basic backup script to duplicate files into a backup folder.

4.1.3 Detect and handle file errors

A. Identify common file errors such as missing files or permission issues.
B. Handle these errors using *try/except* blocks in Python.

C. Record error messages in logs for troubleshooting.
D. Evaluate scenarios where error handling prevents data loss.

4.2 CSV and JSON Processing (3)

4.2.1 Differentiate between CSV and JSON formats

A. Recognize CSV use cases such as expense trackers and contact databases.
B. Recognize JSON use cases such as API responses and configuration files.
C. Evaluate the advantages and limitations of CSV and JSON formats.
D. Evaluate when to use each format in automation tasks.

4.2.2 Process CSV files with Python's *csv* module

A. Read CSV data into Python using *csv.reader()* and *csv.DictReader() for row-by-row access.*
B. Write CSV data using *csv.writer()* or *csv.DictWriter()* for flexible output.
C. Automate simple summaries such as totals, counts, or averages.
D. Demonstrate automation by combining data from multiple CSV files.

4.2.3 Parse and generate JSON files with the *json* module

A. Convert Python dictionaries or lists into JSON strings with *json.dumps()*.
B. Save JSON objects into files with *json.dump()*.
C. Parse JSON data into Python objects using *json.load()* or *json.loads()*.
D. Apply JSON automation to store structured data for reuse.

4.3 Professional Practices in Data Automation (2)

4.3.1 Apply safe and reliable practices in file handling

A. Detect and handle empty files gracefully.
B. Manage corrupted files without script failure.
C. Apply context managers (with *open()*) to ensure files are always closed safely.
D. Use logging to capture data-related errors.
E. Explain why error handling is critical for reliability in automation.

4.3.2 Explain ethical, security, and privacy considerations

A. Avoid overwriting or deleting important files by implementing safeguards.
B. Recognize the risks of storing or exposing sensitive data (e.g., passwords, medical data, financial data, etc.).
C. Describe best practices for file naming and version control to ensure traceability.

D. Evaluate responsible use of automation in handling confidential information.
E. Explain why temporary files pose privacy risks, and apply safeguards such as access control, secure deletion, and compliance with Privacy by Design principles (e.g., ISO 27701).

Module 5. Basic Web and API Automation (8) (17.5%)

5.1 Introduction to Web Automation (2)

5.1.1 Differentiate between web scraping and APIs

A. Explain the difference between retrieving information from raw HTML vs structured data from an API.
B. Compare web scraping (e.g., extracting headlines from a news site) with API-based data access (e.g., requesting weather information).
C. Identify which approach is more efficient or reliable in different scenarios.

5.1.2 Identify ethical and legal considerations in web scraping

A. Respect website rules defined in *robots.txt*.
B. Recognize the risks of overloading websites with frequent automated requests.
C. Identify common anti-scraping techniques such as captchas, IP rate limiting, and request throttling.
D. Explain why consent and responsible use are critical in automation.

5.2 Using *requests* for Web Content (2)

5.2.1 Fetch web pages with requests.get()

A. Import the *requests* library.
B. Retrieve the HTML content of a webpage using *requests.get()*.
C. Interpret response objects and status codes (*200*, *404*, *500*).
D. Apply basic error handling to detect failed requests.

5.2.2 Parse JSON responses from web services

A. Recognize JSON as a standard data format for web APIs.
B. Load JSON responses into Python dictionaries using *response.json()*.
C. Process and extract values from JSON objects.
D. Demonstrate saving API responses into a file (CSV or JSON) for later use.

5.3 Parsing HTML with *BeautifulSoup* (2)

5.3.1 Extract information from simple HTML pages

A. Use *BeautifulSoup* to parse HTML documents.
B. Find and extract specific elements such as titles, links, or paragraphs.
C. Loop through multiple elements to build lists of results (e.g., all headlines).

5.3.2 Apply logging in web automation

A. Record whether a request succeeded or failed.
B. Log the number of elements scraped from a webpage.
C. Analyze logs to detect unusual behavior (e.g., missing elements, errors).

5.4 Working with APIs (2)

5.4.1 Explain REST API basics

A. Define common HTTP methods: *GET*, *POST*.
B. Identify JSON as the most common format for REST API responses.
C. Recognize the difference between requesting data and sending data.
D. Identify common restrictions such as API keys, authentication, and request rate limits.
E. Differentiate between public APIs (freely accessible) and private APIs (restricted access).

5.4.2 Fetch and interpret data from a simple API

A. Perform a request to a simple API (e.g., weather, exchange rates, quotes).
B. Parse the returned JSON data into Python structures.
C. Store results in local files such as CSV or JSON for reporting.

Module 6. Scheduling, Notifications, and Reporting (8) (17.5%)

6.1 Scheduling Basics (2)

6.1.1 Explain the need for scheduling in automation

A. Identify common repetitive tasks suitable for scheduling (e.g., backups, report generation, log cleanup).

B. Distinguish between manual script execution and automated scheduled runs.
C. Explain how scheduling improves reliability, consistency, and efficiency.

6.1.2 Schedule scripts using Python's schedule module

A. Install and import the *schedule* library.
B. Demonstrate how to run a simple job at fixed intervals (e.g., every 10 minutes).
C. Use time-based logic to execute tasks daily or weekly.
D. Demonstrate combining *schedule* with *time.sleep()* for continuous execution.

6.2 System-Level Scheduling (2)

6.2.1 Describe cron jobs and Windows Task Scheduler

A. Define *cron* jobs in Linux/macOS and *Task Scheduler* in Windows.
B. Explain key differences between scheduling in Unix-like vs Windows environments.
C. Explain simple examples such as scheduling a Python script to run once per day.

6.2.2 Recognize advantages and limitations of system scheduling

A. Recognize strengths: flexibility, reliability, running scripts without user input.
B. Identify risks such as misconfigured jobs, overlapping executions, and missed triggers.
C. Evaluate when to use system-level scheduling instead of Python-based scheduling.

6.3 Notifications and Alerts (2)

6.3.1 Send email notifications with smtplib

A. Configure a simple SMTP connection in Python.
B. Automate sending plain-text emails (e.g., task completion or error alerts).
C. Log outgoing messages for record-keeping.
D. Recognize the need for secure handling of email credentials.

6.3.2 Describe desktop notification options

A. Identify cross-platform notification tools such as *plyer*.
B. Demonstrate creating a simple desktop notification.
C. Provide use cases such as reminders, process completions, or status updates.
D. Evaluate when desktop notifications are useful vs when email alerts are more appropriate.

6.4 Reporting in Automation (2)

6.4.1 Generate simple reports from automation tasks

A. Summarize task results in plain text files for record-keeping.
B. Generate basic HTML reports with headings, tables, or lists for easier readability.
C. Create simple daily or weekly reports with timestamps to track progress over time.
D. Automate the saving of generated reports into designated directories for organization and retrieval.

6.4.2 Apply logging to scheduled and reporting workflows

A. Record report generation steps in log files.
B. Log both successful and failed reporting tasks.
C. Analyze logs to identify scheduling or reporting errors.
D. Explain how logging improves auditability and troubleshooting in automated workflows.

MQC Profile

The Minimally Qualified Candidate (MQC) for the *PCEA™ – Certified Entry-Level Automation Specialist with Python* exam is an entry-level learner, student, or early-career professional with **foundational Python skills and introductory knowledge of practical automation**.

The MQC can apply Python to perform simple, well-defined automation tasks under guidance or within structured environments, focusing on command-line scripting, logging and monitoring, basic file/data handling, simple web/API interactions, and scheduled reporting.

Knowledge & Skills

The MQC is expected to:

- Explain core automation concepts: what to automate, when/why to automate, benefits vs. limitations, levels of automation (scripting, process automation, orchestration), and basic ROI reasoning.
- Run Python scripts from the command line; use script arguments, virtual environments, shebangs, and output/error redirection; work with environment variables.
- Use Python standard libraries for file/system tasks (e.g., os, *shutil*, *subprocess*, *time*, *datetime*) to organize files, perform backups, and automate housekeeping.

- Apply logging and basic monitoring with *logging*: levels, formatting, log-to-file, and using logs to trace successes/failures in automation.
- Process everyday data formats: read/write text, CSV (*csv.reader*/*DictReader*, *csv.writer*/*DictWriter*) and JSON *(json.load*/*loads*, *json.dump*/*dumps*); select CSV vs JSON appropriately.
- Fetch and parse simple web resources: use *requests* for HTML/JSON; extract basic elements from HTML with *BeautifulSoup*; understand ethical/legal considerations of scraping (*robots.txt*, rate limits).
- Interact with basic APIs: perform simple GET requests, parse JSON responses, and store results for later use.
- Schedule jobs using Python libraries (e.g., *schedule*) and describe OS-level schedulers (cron, Task Scheduler); send basic notifications (e.g., *smtplib*) and generate simple text/HTML reports.
- Follow professional practices in automation: safe file handling (context managers), error handling, logging, privacy/security awareness (e.g., handling sensitive data, temporary files, access control).

Abilities

The MQC is able to:

- Recognize suitable, repetitive, rule-based tasks for automation across IT/business/personal contexts.
- Develop and run small Python scripts that reduce manual effort and produce consistent results.
- Document and monitor automation runs with logs and simple status reporting.
- Communicate outcomes via notifications and basic reports understandable to non-technical stakeholders.
- Exercise ethical judgment in web/data automation, respecting site rules and privacy considerations.

Limitations

The MQC is not expected to:

- Design enterprise-grade automation frameworks, workflow engines, or orchestration platforms.
- Implement advanced API integrations (authentication flows, rate-limit backoff strategies, streaming/event-driven pipelines).
- Build production-level monitoring/observability stacks or CI/CD orchestrations.

- Optimize automation for high concurrency, distributed execution, or large-scale data processing.
- Replace professional RPA/ETL platforms; rather, they apply entry-level Python to well-scoped tasks.

Candidate Profile

The *PCEA™ – Certified Entry-Level Automation Specialist with Python* exam is designed for learners beginning their journey in automation. A successful candidate demonstrates the ability to apply foundational Python skills to automate routine tasks in well-defined, ethical contexts.

Background
Students, junior IT/operations staff, beginner programmers, or career changers with a foundation in Python programming (basic syntax, variables, loops, conditions, functions, imports).

Experience
Some exposure to scripting or simple data handling; no professional automation experience required. Recommended prior preparation: *PCEP™ – Certified Entry-Level Python Programmer* or equivalent knowledge.

Independence
Can perform simple automation tasks independently, but may require guidance when troubleshooting or composing multi-step workflows.

Passing Requirement

To pass the PCEA™ exam, a candidate must achieve a **cumulative average score of at least 75%** across all exam blocks.

PCEA-30-01 Exam Structure Summary

The *PCEA™ – Certified Entry-Level Automation Specialist with Python* exam consists of single-select and multiple-select items designed to evaluate a candidate's ability to understand core automation principles and apply Python to command-line scripting, logging/monitoring, file & data tasks, basic web/API interactions, and scheduled reporting. Each item is worth a maximum of 1 point. After the exam is completed, the candidate's raw score is normalized, and the final result is expressed as a percentage.

The exam is divided into six blocks, each covering a specific area of entry-level Python automation. The distribution of items and weights reflects their relative emphasis in beginner practice.

Block Number	**Block Name**	**Number of Items**	**Weight**
1	Fundamentals of Automation	6	13%
2	Basic Command-Line Automation with Python	9	19.5%
3	Logging and Monitoring Essentials	7	15%
4	Basic File and Data Automation	8	17.5%
5	Basic Web and API Automation	8	17.5%
6	Scheduling, Notifications, and Reporting	8	17.5%
		46	**100%**