Introduction

0.1 About the Syllabus

This syllabus serves as both a guidance document for participants and a requirement specification for Accredited Training Providers preparing candidates for the "Robot Framework® Certified Professional" (RFCP®) exam. It outlines the structure, learning objectives, and knowledge areas essential for certification.

This syllabus is not a training manual, tutorial, or comprehensive learning resource but instead defines the scope of knowledge that must be taught in a Robot Framework training and acquired by participants to meet the certification requirements.

> **NOTE**
>
> For further explanation, complementary examples, and practical exercises beyond the scope of this syllabus, participants are encouraged to consult additional resources, such as the official documentation at docs.robotframework.org.

The syllabus is divided into chapters that progress logically from basic concepts to more advanced topics of Robot Framework.

The learning objectives (LOs) specified within this document are binding, meaning they define the specific knowledge and skills participants are expected to acquire during the course in order to pass the exam. Therefore, trainers are required to effectively cover the syllabus within their course. Additionally, the recommended sequence of topics in this syllabus helps guide the order of learning, but the specific teaching methods, order and pace may be adapted by the instructor based on class dynamics or need.

0.2 About "Robot Framework® Certified Professional"

The Robot Framework® Certified Professional (RFCP®) certification represents the foundational level of expertise in Robot Framework. It provides participants with a strong understanding of the core principles, syntax, and basic control structures needed to develop effective automation scripts.

While the RFCP® includes an introduction to advanced features such as FOR-Loops and IF statements, the focus is primarily on awareness rather than in-depth mastery, leaving detailed exploration of these topics to the more advanced future certification levels.

RFCP® concentrates on essential concepts such as keyword-driven automation, script execution, and integrating external libraries. It is designed for those seeking proficiency in Robot Framework's core functionalities while gaining an overview of its broader capabilities. This certification does not require or teach domain-specific automation knowledge, such as web, API, or database automation.

0.3 Business Outcomes

Upon completing this course, participants will achieve the following capabilities:

- **Understand the architecture and mechanics of Robot Framework**: Gain a clear understanding of how Robot Framework® operates, including its core components, execution flow, and interaction with external libraries.
- **Develop and maintain stable automation scripts**: Learn how to create automation scripts that are robust, easy to maintain, and adaptable to different scenarios.

- **Develop user keywords and build keyword repositories for reuse**: Understand how to create reusable keywords and build keyword repositories to improve efficiency and maintainability in automation projects.
- **Write documentation**: Learn best practices for documenting keywords, suites and tests or tasks to ensure clarity and ease of use for future script maintenance or collaboration.
- **Integrate external automation libraries**: Leverage external libraries to enable Robot Framework® to interact with a wide range of technologies, such as APIs, user interfaces (Web, Mobile, others), databases, and many more.
- **Understand the flow of more complex automation scripts**: Gain insights into how to structure and manage automation scripts that involve flow control, conditional executions or more intricate workflows.
- **Run automated executions**: Develop skills in executing automation tasks efficiently.
- **Understand, analyze, and debug automation results/protocols**: Learn how to interpret automation execution results, identify issues, and debug scripts effectively.

0.4 About Learning Objectives and Knowledge Levels

The learning objectives (LOs) are a critical component of this syllabus, as they define what participants are expected to know and be able to do by the end of the course. To ensure a clear understanding of these objectives, we apply Knowledge Levels (K-Levels) as a framework for assessing learning progress. These levels are based on Bloom's Taxonomy of Educational Objectives. See Bloom's taxonomy

- **K1 (Remember)**: Basic knowledge of terminology and facts. At this level, participants are expected to recall essential terms, concepts, and definitions.
- **K2 (Understand)**: Comprehension of concepts. Participants should demonstrate an understanding of the principles behind Robot Framework, such as its mechanics, syntax and architecture.
- **K3 (Apply)**: Practical application of knowledge. Participants are expected to be able to write and execute automation scripts, develop keywords, interact with external libraries, and find errors in their automation scripts.

Throughout this syllabus, participants will progress through these knowledge levels—from basic recall (K1) to understanding and explaining concepts (K2), and ultimately applying their knowledge to practical automation tasks (K3). This structured approach ensures participants gain a comprehensive and practical understanding of Robot Framework fundamentals and their application in real-world scenarios.

0.5 About Accredited Training Providers

Accredited Training Providers are organizations officially accredited by the Robot Framework Foundation to offer certified training programs for a specific certification level. These partners shall deliver high-quality, structured courses designed to prepare candidates for the Robot Framework® Certified Professional (RFCP®) exam and other future Robot Framework certifications.

All training providers are members of the Robot Framework Foundation, and their training materials have been reviewed by independent Robot Framework experts chosen by the Robot Framework Foundation to ensure the Foundation's quality standards. Only these Accredited Training Providers are permitted to refer to their courses as "Robot Framework®" training or use the term "Robot Framework® Certified Professional" or "RFCP®" in connection with their programs, due to the trademark on these terms.

Trainings can be exclusively pursued through these partners, but obtaining a certificate is not dependent on completing their courses, allowing flexibility for candidates to self-study if desired.

0.6 About Exam Providers

Exam providers are independent organizations responsible for administering certification exams for the Robot Framework® certification program. These providers manage the entire examination process, from scheduling and conducting the exams to handling participant data and maintaining certification records.

An exam provider ensures that the certification process is handled professionally and securely. They are tasked with delivering a seamless exam experience, including remote proctoring services, technical support, and other logistical elements. In addition to overseeing the exam itself, they maintain strict confidentiality and compliance with data privacy regulations, ensuring the secure management of all participant information.

The exam provider is also responsible for storing and managing certification data. This includes tracking which participants have earned certifications, maintaining certification validity, and providing verification services if needed.

Global Association for Software Quality (GASQ)

Our current exclusive exam provider is the Global Association for Software Quality.

Global Association for Software Quality , abbreviated GASQ, is an international exam provider and a leading association in the software quality industry. GASQ was founded by experts from Europe, Asia and America as an independent, international non-profit association aiming to advocate and promote software quality in research, teaching and industry.

0.7 Acknowledgment of Contributors

The Robot Framework syllabus and the corresponding "Robot Framework Certified Professional®" (RFCP) certification would not have been possible without the efforts of its author and contributors. This chapter acknowledges their valuable contributions to the development of this syllabus.

The Author

The primary author of this syllabus is **René Rohner**.

Contributors

The following individuals have contributed to the development of this syllabus:

Alena Drebezgova, Alex Read, Christoph Singer, Elout van Leeuwen, Frank van der Kuur, Gerwin Laagland, Ilmari Salmela, Jörg Irle, Krzysztof Żminkowski, Lydia Peabody, Michael Biech, Miikka Solmela, Pekka Klärck, Pyry Hartman, Sami Pesonen, Simon Meggle, Tatu Kairi, and Tomáš Hák.

Special Mentions

Special recognition is given to **Gerwin Laagland**, **Simon Meggle**, and **Frank van der Kuur**, whose thorough reviews and insightful suggestions greatly enhanced the clarity, structure, and overall quality of the syllabus.

Krzysztof Żminkowski, and **Simon Meggle** contributed significantly to the creation of the exam by proposing thoughtful and challenging questions.

Sami Pesonen laid the groundwork for this syllabus by assembling the initial collection of topics to be covered, forming the foundation upon which the syllabus was built.

Acknowledgment

The creation of the "Robot Framework Certified Professional®" syllabus stands as a testament to the dedication and generosity of its contributors. Most of the work has been done pro bono, reflecting a deep commitment to the principles of open-source collaboration and knowledge sharing. Each contributor—from those who meticulously reviewed and refined the content to those who laid its very foundation—has left a lasting impact. Their combined efforts have ensured that this document serves as a meaningful and accessible resource. We extend our heartfelt gratitude to everyone involved for their invaluable contributions.

1 Introduction to Robot Framework

The upcoming chapters provide a concise overview of Robot Framework, including its core structure, use cases in test automation and Robotic Process Automation (RPA), and key specification styles like keyword-driven and behavior-driven testing. You'll learn about its architecture, syntax, and how test cases and tasks are organized. Additionally, the chapters explain the open-source licensing under Apache 2.0, the role of the Robot Framework Foundation in maintaining the ecosystem, and the foundational web resources available for further exploration and contributions.

1.1 Purpose / Use Cases

LEARNING OBJECTIVES

LO-1.1

Recall the two main use cases of Robot Framework

Robot Framework is a versatile, open-source automation framework that supports both **test automation** and **robotic process automation (RPA)**. Initially designed for acceptance testing, it has since evolved to cover other types of testing and various automation tasks in both IT and business environments. Its keyword-driven approach allows users to create reusable components, making it accessible even to those with minimal programming skills. Robot Framework can be extended through a vast array of third-party or custom made keyword libraries, allowing it to automate interactions with APIs, user interfaces, databases, and many more technologies.

1.1.1 Test Automation

LEARNING OBJECTIVES

LO-1.1.1

Recall the test levels Robot Framework is mostly used for

Robot Framework is widely used at various levels of testing, primarily focusing on:

- **System Testing**: Involves verifying the complete system's behavior and capabilities. It often includes both functional and non-functional aspects (e.g., accessibility, security) and may use simulated components.
- **System Integration Testing**: Focuses on the interaction between the system under test and external services, as well as on the integration of multiple systems into a larger system, ensuring that all integrated components communicate and function together as expected.
- **Acceptance Testing**: Aims to validate that the system meets business requirements and is ready for deployment or release. This often includes different forms of acceptance testing (e.g., user acceptance, operational acceptance, regulatory acceptance) and is frequently written or conducted by end-users or stakeholders to confirm the system's readiness for use. Acceptance tests, often defined by business stakeholders in approaches like Acceptance Test-Driven Development (ATDD), can be automated and executed earlier in the development process. This ensures that the solution aligns with business requirements from the start and provides immediate feedback, reducing costly changes later.
- **End-to-End Testing**: Verifies that a complete workflow or process within the system operates as intended, covering all interconnected subsystems, interfaces, and external components. End-to-end tests ensure the correct functioning of the application in real-world scenarios by simulating user interactions from start to finish.

Robot Framework's flexibility and support for external libraries make it an excellent tool for automating these comprehensive test cases, ensuring seamless interaction between components and validating the system's behavior also in production or production-like conditions.

Robot Framework is typically not used for **component testing** nor **integration testing** because its primary strength lies in higher-level testing, such as system, acceptance, and end-to-end testing, where behavior-driven and keyword-based approaches excel. Component testing requires low-level, granular tests focusing on individual units of code, often necessitating direct interaction

with the codebase, mocking, or stubbing, which are better handled by unit testing frameworks like JUnit, pytest, or NUnit. Similarly, integration testing at a low level often requires precise control over service interactions, such as API stubs or protocol-level testing, which may not align with Robot Framework's abstraction-oriented design. While Robot Framework can technically handle these cases through custom libraries, its overhead and design philosophy make it less efficient compared to tools specifically tailored for low-level and tightly scoped testing tasks.

1.1.1.1 Synthetic Monitoring

Beyond traditional test levels, **Synthetic Monitoring**, also referred to as **Active Monitoring** or **Proactive Monitoring**, is a proactive approach that simulates user interactions with live systems at regular intervals. It detects performance issues or downtime early with the goal of detecting such failure before they affect actual users.

1.1.2 Robotic Process Automation (RPA)

Robotic Process Automation (RPA) uses software bots to perform tasks and interactions normally performed by humans, without requiring changes to the underlying applications.

Robot Framework, with its keyword-driven approach, vast ecosystem of libraries, simplicity, and scalability, is widely adopted for RPA tasks. Robot Framework allows users to automate most workflows using ready-made keyword libraries that provide a wide range of functionalities. These libraries can be combined and reused in user-defined keywords, making automation simple and efficient. For custom functionalities or more complex tasks, Robot Framework also offers the flexibility to create custom keyword libraries using Python, enabling advanced use cases and seamless integration with unique systems.

Common use cases of RPA with Robot Framework include:

- **Data extraction and manipulation**: Automating data transfers and processing between systems.
- **Task / Process automation**: Automating tasks such as form submissions, clicks, and file operations across web or desktop applications.

1.2 Architecture of Robot Framework

Robot Framework is an open-source automation framework that allows you to build automation scripts for testing and RPA (Robotic Process Automation). It focuses on providing a keyword-driven or behavior-driven approach, making the automation easy to understand and maintain. However, it is not a full-stack solution that encompasses all layers of automation. Instead, it provides a flexible platform where different tools, libraries, and integrations handle specific tasks to implement a flexible automation solution.

1.2.1 Robot Framework and the gTAA (Generic Test Automation Architecture)

LEARNING OBJECTIVES

K1 LO-1.2.1

Recall the layers of the Generic Test Automation Architecture (gTAA) and their corresponding components in Robot Framework

The **Generic Test Automation Architecture (gTAA)** described in the ISTQB "Certified Tester Advanced Level Test Automation Engineering" offers a structured approach to test automation, dividing it into different layers for a clear separation of concerns:

- **Definition Layer**: This layer contains the "Test Data" (test cases, tasks, resource files which include user keywords and variables). In Robot Framework, the test data is written using the defined syntax and contains keyword calls and argument values that make the test case or task definitions structured in suites.
- **Execution Layer**: In Robot Framework, the execution layer consists of the framework itself, including its core components and APIs. It parses and interprets the test data syntax to build an execution model. The execution layer is responsible for processing this execution model to execute the library keywords with their argument values, logging results, and generating reports.
- **Adaptation Layer**: This layer provides the connection between Robot Framework and the system under test (SUT). In Robot Framework, this is where the keyword libraries, which contain code responsible for interacting with different technologies and interfaces, such as those for UI, API, database interactions, or others, are located. These keyword libraries enable interaction with different technologies and interfaces, ensuring the automation is flexible and adaptable to various environments.

Editors/IDEs that offer support for Robot Framework's syntax are tools that support or integrate in these layers. When writing tests|tasks or keywords, the editor supports the definition layer. When executing or debugging tests|tasks, the editor supports the execution layer. When writing keywords in e.g. Python for keyword libraries, the editor supports the adaptation layer. Therefore also other additional extensions of Robot Framework can be categorized into these layers.

1.2.2 What is Robot Framework & What It Is Not

LEARNING OBJECTIVES

K1 LO-1.2.2

Recall what is part of Robot Framework and what is not

Robot Framework itself focuses primarily on **test|task execution**. It includes:

- A parser to read test|task data and build an execution model.
- An execution engine to process model and execute the keywords.
- A result generation mechanism to provide logs and reports.
- A collection of generic standard libraries to process and handle data or interact with files and processes.
- Defined APIs for extensions and customizations.

However, Robot Framework **does not** include:

- Keyword libraries to control systems under test/RPA.

  Such as:

  - Web front-end automation libraries.
  - API interaction libraries.
  - Mobile automation libraries.
  - Database interaction libraries.
  - RPA libraries.
  - etc.

- Code editors or IDEs.

- CI/CD Integration.

Robot Framework defines the syntax for test|task data, but it is the role of external libraries and tools to extend its functionality for specific automation needs.

1.2.3 Technology & Prerequisites

LEARNING OBJECTIVES

LO-1.2.3

Recall the technology Robot Framework is built on and the prerequisites for running it

Robot Framework is built on **Python** but is adaptable to other languages and technologies through external libraries. To run Robot Framework, an officially supported version of the **Python interpreter** is required on the machine executing the tests|tasks. Typically, Robot Framework and its libraries are installed via the "package installer for Python" (`pip`) from PyPi.org, allowing for straightforward installation and setup. Robot Framework itself does not have any external dependencies, but additional third party tools or keyword libraries may require additional installations.

1.3 Basic Syntax & Structure

LEARNING OBJECTIVES

K1 LO-1.3

Recall the key attributes of the syntax that makes Robot Framework simple and human-readable

Robot Framework is a script-based interpreter for files that contain textual specifications. These files are typically organized into directories. The syntax of Robot Framework is designed to be simple and human-readable, allowing for quick learning and ease of use.

Key attributes of the syntax that improves the before mentioned:

- **Space-separated syntax**: Robot Framework uses two or more spaces as the primary separator (although one space is allowed as a character). A use of **FOUR (4)** spaces is recommended to ensure clarity and readability of the specification.
- **Indentation based blocks**: Code blocks like test, task or keyword bodies are defined by indentation. This makes the structure clear and easy to follow.
- **Reduced use of special characters**: Compared to programming languages the focus is on reducing special characters, making the syntax human-readable and user-friendly.
- **String first**: Unquoted strings are considered as strings, while variables need special syntax.
- **Single spaces are valid**: Single spaces are valid as a character in most elements and values without quotation.
- **Mostly case-insensitive**: Most elements like keyword or variable names are case insensitive. However, some syntax, like library imports is case-sensitive.

NOTE

This syllabus does NOT cover other formats like Pipe-Separated (|) Format or Restructured Text or JSON!

Example of test cases with their keyword calls written in Robot Framework:

```
*** Settings ***
Documentation     A test suite for valid login.
...
...               Keywords are imported from the resource file
Resource          keywords.resource
Suite Setup       Connect to Server
Test Teardown     Logout User
Suite Teardown    Disconnect


*** Test Cases ***
Access All Users With Admin Rights
    [Documentation]    Tests if all users can be accessed with Admin User.
    Login Admin
    Check All Users

Create User With Admin Rights
    [Documentation]    Tests if new users can be created with Admin User.
    Login Admin
    Create New User
    ...    name=Peter Parker
    ...    login=spider
    ...    password=123spiderman321
```

```
    ...    right=user
    Verify User Details    spider    Peter Parker
    Logout User
    Login User    spider    123spiderman321
```

1.3.1 What are Test Cases / Tasks?

In Robot Framework, **Test Cases** (**Tests**) or **Tasks** are executable entities that serve a specific purpose and are organized into suites. A **Test** is synonymous with a **Test Case**, while **Task**, technically being the same, is used in RPA mode, where the automation is not focused on testing but on automating business processes.

Tests or Tasks have a body made up of **keyword calls** and Robot Framework statements like **IF** or **VAR**, which represent the actions or steps executed during the test or task execution. These keywords make the automation modular, maintainable, reusable, and readable.

1.3.2 Files & Directories

Robot Framework organizes tests|tasks into **Suites**, which are either files or directories.

- `*.robot` files that contain test cases or tasks are suites.
- Each directory, starting from the top-level directory (the one executed by Robot Framework), and any sub-directory that contains a `*.robot` suite file, is considered a **Suite** as well. Suites can contain other suites, forming a hierarchical tree, which is by default alphabetically ordered. See 2.1 Suite File & Tree Structure for more details.

This structure allows for logical grouping and organization of tests and tasks, which can scale as needed.

1.3.3 What are Keywords?

LEARNING OBJECTIVES

K2 LO-1.3.3

Explain the difference between User Keywords and Library Keywords

Tests or Tasks are constructed using **Keywords**, which represent specific actions or sequences of actions to be performed.

Keywords in Robot Framework follow the concepts used in Behavior-Driven Development (BDD) and Keyword-Driven Testing.

Definition: one or more words used as a reference to a specific set of actions intended to be performed during the execution of one or more tests or tasks.

There are two types of keywords in Robot Framework:

1. **User Keywords**: Written in Robot Framework syntax, they are mainly used as wrappers around sets of other keywords or to implement actions not readily available in existing library keywords. User Keywords improve readability, understandability, maintainability, and structure. These keywords always call other keywords or commands within their body, which is why they are also called **higher-level keywords**. In other literature, such keywords are also referred to as **Business Keywords** or **Composite Keywords**.

   Example:

```
*** Keywords ***
Login User
    [Arguments]    ${login}    ${password}
    Set Login Name    ${login}
    Set Password    ${password}
    Execute Login
```

2. **Library Keywords**: Typically written in Python, but they may also be implemented using other technologies. These keywords typically interact with the system under test (SUT) or the system to be controlled by RPA, or execute specific actions such as calculations or conversions. From the viewpoint of Robot Framework, such keywords are not composed of other keywords and do form the lowest level of keywords. Therefore, they are also referred to as **low-level keywords**. In other literature, such keywords are also called **Technical Keywords** or **Atomic Keywords**.

A **User Keyword** consists of a **name**, optional **arguments**, and a **body** of keyword calls that may invoke other user keywords, library keywords, or other statements such as variable definitions or flow control.

During execution, each keyword call is logged, providing fine-grained detail in the execution logs. This includes all levels of keywords—from those called directly by a test or task to those nested within user keywords, all the way down to the execution of library keywords. This granular logging and detailed execution documentation is one of the key advantages of Robot Framework compared to other automation tools.

1.3.4 Resource Files & Libraries

LEARNING OBJECTIVES

K1 LO-1.3.4

Recall the difference between Resource Files and Libraries and their artifacts

While tests and tasks are organized into suites, **keywords** are organized into **Resource Files** and **Keyword Libraries**.

- **Resource Files**: Contain **User Keywords** and are also used to organize the importing of libraries and the definition of variables. These are considered to be part of the test|task data in the *Definition Layer*.
- **Keyword Libraries**: Contain **Library Keywords**, which are typically implemented in Python or other technologies and, except for the standard libraries, are not part of Robot Framework itself. They can be either custom-made or third-party libraries implemented by the Robot Framework community. These are considered to be part of the *Adaptation Layer*.

Central resource files and libraries allow the separation of concerns, making the automation more modular and reusable across multiple suites, tests or tasks.

The concepts of organizing are fundamental to working with Robot Framework and contribute to its flexibility and scalability in both test automation and RPA.

1.4 Specification Styles

LEARNING OBJECTIVES

LO-1.4

Recall the three specification styles of Robot Framework

Specification styles define how tests or tasks are structured, focusing on how actions and verifications are described. While **Keyword-Driven Testing (KDT)** and **Behavior-Driven Development (BDD)** are commonly associated with testing, the principles behind these styles are adaptable to other forms of automation, such as RPA.

Both styles can be mixed, even within the same test or task, but it is strongly recommended to have separate styles for separate purposes and not mix them within the same body. One practical solution would be to define acceptance test cases that cover users' expectations in a declarative *Behavior-Driven Style*, while using keywords that are implemented in an imperative *Keyword-Driven style*. Further system level test cases, that are not covering acceptance criteria could be written in a *Keyword-Driven style*.

The approach of both styles is different in that way, that the *Behavior-Driven Style* is a **declarative** specification, where the script describe/declare what the system should do or how it should behave, while the *Keyword-Driven Style* is an **imperative** specification, where the script specifies what the automation should do to control the system.

Beside these two different specification approaches how to write/formulate your automation script and their step sequences, there is also a third specification method, **Data-Driven Specification** that can be combined with the other two styles, to define the data that is used in the automation.

1.4.1 Keyword-Driven Specification

LEARNING OBJECTIVES

LO-1.4.1

Understand the basic concepts of Keyword-Driven Specification

In **Keyword-Driven Specification**, automation steps are expressed through a sequence of mostly **imperative commands**. Keywords define the specific actions that must be executed in a particular order, similar to procedural programming. The emphasis is on the **actions performed by the automation/tester**.

For example, in Robot Framework, a Keyword-Driven test might look like:

```
*** Test Cases ***
Verify Foundation Link
    Open Page       http://robotframework.org
    Click Button    FOUNDATION
    Verify Title    Foundation | Robot Framework
    Verify Url      https://robotframework.org/foundation
```

Verifications or assertions can be imperative, though they are often phrased as assertions, such as `Title Should Be    Foundation | Robot Framework`, adding flexibility to how outcomes are checked.

The advantage of this style lies in its **clarity** and **structure**. It provides a straightforward representation of the task flow, making it easy to understand what actions will be executed.

Separation of the executed step/keyword and its arguments/data with spaces improves the readability of tests or tasks. Flow and data can be parsed separately by the consumer.

1.4.2 Behavior-Driven Specification

LEARNING OBJECTIVES

LO-1.4.2

Understand the basic concepts of Behavior-Driven Specification

Behavior-Driven Specification originates from **Behavior-Driven Development (BDD)** and its **Gherkin-Style**, where steps are written to describe the system's behavior from the user's perspective. This style often incorporates **embedded arguments** into the steps and uses natural language constructs like **Given, When, Then, And & But**.

In Robot Framework, behavior-driven tests may look like:

```
*** Test Cases ***
Opening Foundation Page
    Given "robotframework.org" is open
    When the user clicks the "FOUNDATION" button
    Then the page title should be "Foundation | Robot Framework"
    And the url should be "https://robotframework.org/foundation"
```

The prefixes `Given`, `When`, `Then`, `And` and `But` are basically ignored by Robot Framework if a keyword is found matching the rest of the name. A key difference between Robot Framework's behavior-driven style and BDD frameworks like **Cucumber** or most others is the ability in Robot Framework to use **multiple keyword layers**. In other BDD frameworks the code that implements a sentence like `Given "robotframework.org" is open.` is referred to as a step definition. Step definitions are written in a programming language (typically Java, JavaScript, Ruby, or Python) and map natural language steps from a Gherkin feature file to code. Therefore there are no multiple layers of keywords that can be logged into execution protocols. Robot Framework allows you to create **user keywords** that can further call other user or library keywords, providing greater flexibility, modularity and much more detailed logging.

1.4.3 Comparing Keyword-Driven and Behavior-Driven Specification

LEARNING OBJECTIVES

LO-1.4.3

Recall the differences between Keyword-Driven and Behavior-Driven Specification

The core difference between **Keyword-Driven** and **Behavior-Driven** styles lies in their focus:

- **Keyword-Driven Style** emphasizes **what actions** need to be performed in a specific order, making it action-centric. It is an **imperative** style, comparable to procedural programming. It is structured, clear, and well-suited for scenarios where the steps are more technical or detailed and involve a larger number of keyword calls within a test or task. Additionally, this style is better suited for complex tasks or handling complex data, as it enables a clear separation between keyword names and their argument values.

- **Behavior-Driven Style** emphasizes **how the system behaves** from the user's point of view, using more natural language and focusing on expected outcomes. It is a **declarative** style that can be compared to writing user stories or acceptance criteria. It is optimized for **business-oriented** descriptions of functionality and is often more suitable for communicating with non-technical stakeholders. This style can get less understandable when the amount of steps increases or the amount of defined data in the steps increases.

Both styles can be applied within Robot Framework, offering flexibility depending on the context of the automation task.

1.4.4 Data-Driven Specification

LEARNING OBJECTIVES

K1 LO-1.4.4

Recall the purpose of Data-Driven Specification

Data-Driven Specification originates from **Data-Driven Testing** and is a method where the test data and expected results are separated from the test script that controls the flow.

While in **Robotic Process Automation (RPA)**, the data used in an automation workflow is typically acquired dynamically from an external source, in testing, the data is specifically chosen to cover different scenarios or cases. Therefore, this method of defining data combinations statically in the suite files is normally not applicable to RPA.

The purpose of **Data-Driven Testing** is to automate the same sequence of actions or scenario with different sets of input and/or expected output data.

In this style, a single user keyword, which contains the whole test logic or sequence of actions, is executed with multiple data variations, making it highly effective for repetitive tests, where the logic stays the same but the data changes, without duplicating the test logic for each case.

Robot Framework offers a convenient feature for this approach through **Test Templates**.

Benefits of Data-Driven Specification:

- **Efficiency**: Reduces the need to write redundant test cases by reusing the same workflow with different data inputs.
- **Clarity**: Keeps the test logic separate from the data, making it easier to manage large data sets.
- **Scalability**: Suitable for scenarios where the same functionality needs to be tested under various conditions, such as verifying form inputs or performing calculations with different values.

See 3.4 Using Data-Driven Specification for more details and examples on Data-Driven Specification.

1.5 Organization and Licensing

1.5.1 Open Source License

LEARNING OBJECTIVES

LO-1.5.1

Recall the type of open-source license under which Robot Framework is distributed

Robot Framework is licensed under the **Apache License 2.0**, a permissive open-source license. The key characteristics of this license include:

- **Permissive**: The license allows users to freely use, modify, and distribute the software, including for commercial purposes, without significant restrictions.
- **No Warranty**: The software is provided "as-is," without any warranties or guarantees of performance.
- **Attribution**: Users must keep the original authorship and any changes made to the code visible, ensuring transparency regarding contributions and modifications.

This licensing structure encourages broad usage and contribution while maintaining a legal framework that protects both users and developers.

1.5.2 About the Robot Framework Foundation

LEARNING OBJECTIVES

LO-1.5.2

List and recall the key objectives and organizational form of the Robot Framework Foundation

The **Robot Framework Foundation** (officially known as **Robot Framework ry**) is a non-profit association based in Helsinki, Finland, dedicated to promoting the use, development, and maintenance of the open-source Robot Framework. The foundation ensures that Robot Framework remains freely available and viable for both test automation and robotic process automation (RPA) in the future.

Key objectives of the foundation include:

- **Support for Core Development**: The foundation funds and enables the core development, maintenance, and evolution of the Robot Framework, ensuring it is freely available to everyone. It also supports ecosystem and user-contributed projects that further enhance the framework's capabilities.
- **Democratic Governance**: The foundation operates under democratic principles, with a **Board of Directors** elected annually by its members. The board oversees the foundation's operations, and membership primarily consists of companies that contribute financially to support the framework's ongoing development through membership fees.
- **Platform Maintenance**: The foundation is responsible for maintaining key infrastructure, such as the official website, GitHub repositories, and community platforms. These resources are crucial to sustaining a healthy ecosystem and fostering collaboration among users and contributors.

- **Community Support and Events**: The foundation plays a central role in organizing **RoboCon**, the annual Robot Framework User Conference, which brings together users, developers, and contributors to share knowledge and insights. Additionally, it helps to disseminate knowledge about test automation and RPA through community events and documentation efforts.
- **Funding of Ecosystem Projects**: Whenever possible, the foundation finances open-source projects that are proposed by community members, aiming to support broader ecosystem development and innovation.

As a non-profit association, all funds are directed towards the development and promotion of the Robot Framework, ensuring that it remains accessible to all users without commercial restrictions.

More information, including a list of foundation members, is available at **robotframework.org/foundation**.

This structure and mission ensure that Robot Framework continues to grow and serve the needs of its community while maintaining an open and democratic approach to its development and governance.

1.5.3 Robot Framework Webpages

LEARNING OBJECTIVES

LO-1.5.3

Recall the official webpages for Robot Framework and its resources

The official pages for Robot Framework and its related resources are maintained by the foundation. These include:

- **robotframework.org**: The main page providing an overview, documentation, and access to resources.
- **github.com/robotframework**: The official repository for the framework's source code and other components.

2 Getting Started with Robot Framework

This chapter introduces participants to the foundational concepts of Robot Framework. It covers the basics of how automation specifications are structured, how suites are organized, and the execution process. Participants will learn how Robot Framework is run and explore the generated reports and logs that document test results.

The chapter also provides an overview of suite structures, the role of libraries and resource files, and how to import them. Additionally, it delves into the core syntax of Robot Framework, focusing on how keywords are defined and used, the types of keyword arguments, and how keyword documentation is interpreted to ensure clarity and maintainability.

2.1 Suite File & Tree Structure

LEARNING OBJECTIVES

LO-2.1

Understand which files and directories are considered suites and how they are structured in a suite tree.

When executing Robot Framework, it either parses directory trees or files, depending on which paths are given.

The given path to Robot Framework where it starts parsing is considered the **Root Suite**.

If the path to a single file is given as **Root Suite** directly to Robot Framework, only this file is parsed.

If a directory path is given, starting at this location, Robot Framework will parse all `*.robot` files and directories within this path. Robot Framework analyzes all containing files and determines if they contain test cases or tasks. If they do, they are considered **Suite Files** or **Low-Level Suites**. All directories that either directly or indirectly contain a Suite File are considered **Suites Directories** or **Higher-Level Suites**.

The ordering of suites during execution is, by default, defined by their name and hierarchy. All files and directories, which are suites in one directory, are considered on the same level and are executed in case-insensitive alphabetical order.

It is possible to define the order without influencing the name of the suite by adding a prefix followed by two underscores `__` to the name of the directory or file. This prefix is NOT considered part of the name. So `01__First_Suite.robot` sets the suite name to `First Suite`, while `2_Second_Suite.robot` sets the suite name to `2 Second Suite`.

One or more underscores in file or directory names are replaced by space characters as suite names.

Legend:

```
▷ Directory (No Suite)
▶ Suite Directory
□ File (No Suite)
■ Suite File
```

Example:

```
----- Tree Structure / Order --------- | ---- Suite Name ---------
 ▶ Root_Suite                          | Root Suite
   ■ A_Suite.robot                     | A Suite
   ▶ Earlier_Suite_Directory           | Earlier Suite Directory
     ■ B_Suite.robot                   | B Suite
     ■ C_Suite.robot                   | C Suite
   ▷ Keywords (No Suite)               |
     □ technical_keywords.resource     |
   ▶ Later_Suite_Directory             | Later Suite Directory
     ■ 01__First_Suite.robot           | First Suite
     ■ 02__Second_Suite.robot          | Second Suite
     ▶ 03__Third_Suite                 | Third Suite
       ■ 01__First_Sub_Suite.robot     | First Sub Suite
       ■ 02__Second_Sub_Suite.robot    | Second Sub Suite
     ■ 04__Fourth_Suite.robot          | Fourth Suite
```

2.1.1 Suite Files

LEARNING OBJECTIVES

K1 LO-2.1.1

Recall the conditions and requirements for a file to be considered a Suite file

Robot Framework parses files with the extension `.robot` and searches for test cases or tasks within these files.

A parsed file that contains at least one test case or task is called a **Suite File**.

A Suite File **either** contains `*** Test Cases ***` (in Test Suites) **or** `*** Tasks ***` (in Task Suites), but it CANNOT contain both types simultaneously.

2.1.2 Sections and Their Artifacts

LEARNING OBJECTIVES

K1 LO-2.1.2

Recall the available sections in a suite file and their purpose.

Robot Framework data files are defined in different sections. These sections are recognized by their header row. The format is `*** <Section Name> ***` with three asterisks before and after the section name and section names in **Title Case** separated by a space.

The following sections are recognized by Robot Framework and are recommended to be used in the order they are listed:

- `*** Settings ***`
- `*** Variables ***`
- `*** Test Cases ***` or `*** Tasks ***` (mandatory in Suite Files)
- `*** Keywords ***`
- `*** Comments ***`

The sections `*** Settings ***`, `*** Variables ***`, `*** Keywords ***`, and `*** Comments ***` are optional in suite files and can be omitted if not needed.

2.1.2.1 Introduction to `*** Settings ***` Section

LEARNING OBJECTIVES

K1 LO-2.1.2.1-1

Recall the available settings in a suite file.

K2 LO-2.1.2.1-2

Understand the concepts of suite settings and how to define them.

The `*** Settings ***` section is used to configure various aspects of the test|task suite. It allows you to import keywords from external libraries (`Library`) or resource files (`Resource`), and import variables (`Variables`) from variable files (not part of this syllabus) that are needed for execution in the containing tests|tasks.

In this section, the suite name, that is normally derived from the file name, can be redefined with the `Name` setting and its documentation can be defined with the `Documentation` setting.

Additional metadata can be defined by multiple `Metadata` entries, which can contain key-value pairs that can be used to store additional information about the suite, like the author, the version, or related requirements of the suite.

This section can also define keywords called for execution flow control, such as `Suite Setup` and `Suite Teardown`, which are executed before and after the suite's tests run. See 4.1 Setups (Suite, Test|Task, Keyword) and 4.2 Teardowns (Suite, Test|Task, Keyword) for more information.

Additionally, some settings can define defaults for all tests|tasks in the suite, which can be extended or overwritten in the individual tests|tasks. Those settings are prefixed with either `Test` or `Task`, according to the type of suite and the following section type (`*** Test Cases ***` or `*** Tasks ***`), like `Test Timeout`, while the local setting is in square brackets and without the prefix like: `[Timeout]`.

- `Test Setup`/`Task Setup` (locally: `[Setup]`) and `Test Teardown`/`Task Teardown` (locally `[Teardown]`) define which keywords are executed before and after each individual test|task. The local setting overrides the suite's default. See 4.1 Setups (Suite, Test|Task, Keyword) and 4.2 Teardowns (Suite, Test|Task, Keyword) for more information.
- `Test Timeout`/`Task Timeout` (locally `[Timeout]`) defines the maximum time a test|task is allowed to run before it is marked as failed. The local setting overrides the suite's default.
- `Test Tags`/`Task Tags` (locally `[Tags]`) define tags that are assigned to tests|tasks in the suite and can be used to filter tests|tasks for execution or for attributing information to the tests|tasks. The local setting appends or removes tags defined by the suite's default. See 4.4 Test|Task Tags and Filtering Execution for more information.
- `Test Template`/`Task Template` (locally `[Template]`) defines a template keyword that defines the test|task body and is typically used for Data-Driven Testing where each test has the same keywords but different argument data. The local setting overrides the suite's default.

Similar to test|task tags, also keyword tags can be defined in the `*** Settings ***` section with the `Keyword Tags` (locally `[Tags]`) setting, which can be used to set keyword tags to the keywords. The local setting appends or removes tags defined by the suite's default.

2.1.2.2 Introduction to `*** Variables ***` Section

LEARNING OBJECTIVES

LO-2.1.2.2

Recall the purpose of the `*** Variables ***` section.

This section is used to define suite variables that are used in the suite or its tests|tasks or inside their keywords.

The most common use case is to define these variables as constants that contain a static value during execution. This can either be a default value, that may be overwritten by globally defined variables via the Command Line Interface (CLI) or a constant value that is used in multiple places in the suite.

In some cases, these variables are also dynamically reassigned during the execution of the suite, but this is not recommended and should be avoided if possible, because this may lead to test|task runtime dependancies and errors caused by these side-effects that are hard to debug and find.

See 3.2.2 `*** Variables ***` Section for more information about the `*** Variables ***` section.

2.1.2.3 Introduction to `*** Test Cases ***` or `*** Tasks ***` Section

LEARNING OBJECTIVES

K2 LO-2.1.2.3

Understand the purpose of the `*** Test Cases ***` or `*** Tasks ***` section.

This section defines the executable elements of a suite. Test cases and tasks are technically synonyms for each other. However, users have to choose one of the two modes of suite execution that Robot Framework offers.

Each test case or task is structured using an indentation-based format. The first un-indented line specifies the name of the test|task, followed by indented lines containing **keyword calls** and their possible **arguments** and test|task-specific settings. These optional settings like `[Setup]`, `[Teardown]`, and `[Timeout]` can be applied to individual test cases or tasks to control their behavior or provide additional details.

A suite file must contain either a `*** Test Cases ***` or a `*** Tasks ***` section, but not both. Resource files cannot contain either of them.

See 2.6 Writing Test|Task and Calling Keywords for more information about the `*** Test Cases ***` or `*** Tasks ***` section.

2.1.2.4 Introduction to `*** Keywords ***` Section

LEARNING OBJECTIVES

K2 LO-2.1.2.4

Understand the purpose and limitations of the `*** Keywords ***` section.

This section allows you to define **locally scoped user keywords** that can only be used within this suite where they are defined, while keywords defined in resource files can be used in any suite that imports these resource files. Keywords defined in a suite are therefore not reusable outside the suite, but they are often used to organize and structure tests|tasks for improved readability and maintainability. This section is particularly useful for defining suite-specific actions, such as **Suite Setup** keywords or similar kinds, which are relevant only to the suite they belong to.

While these keywords are not globally accessible, they serve a crucial role in making the suite more modular and understandable by breaking down complex sequences into smaller, manageable parts. Defining keywords locally in this section enhances the maintainability of the tests|tasks within the suite, ensuring that even large and intricate suites remain well-structured and easy to manage.

See 3.3.1 `*** Keywords ***` Section for more information about the `*** Keywords ***` section.

2.1.2.5 Introduction to `*** Comments ***` Section

This section is used to add comments to the suite file or resource file. All content in this section is ignored by Robot Framework and is not executed or parsed.

2.2 Basic Suite File Syntax

LEARNING OBJECTIVES

K2 LO-2.2

Understand the basic syntax of test cases and tasks.

Suite files and resource files share the same syntax, however they differ in their capabilities. Resource files are explained in more detail in 2.4.2 Resource Files 3.1 Resource File Structure.

2.2.1 Separation and Indentation

LEARNING OBJECTIVES

K3 LO-2.2.1

Understand and apply the mechanics of indentation and separation in Robot Framework.

As mentioned before, Robot Framework uses an indentation-based and space-separated syntax to structure keywords, test cases, and tasks.

Two or more spaces are used to separate or indent statements in Robot Framework files, while a single space is a valid character in tokens (e.g. keyword names, argument values, variables, etc.). The clear recommendation for separators is to use **four spaces** or more to unambiguously make it visible to a potential reader where elements are separated or indented.

A statement in Robot Framework is a logical line that contains specific data tokens, which are separated by multiple spaces (separator tokens) and typically end with a line break (end-of-line token). To create a statement spanning multiple lines, literal lines can be continued by adding `...` (three dots) and a separator token at the beginning of the next line, maintaining the same indentation level as the line being continued.

Example 1: A keyword call is a statement that consists of a keyword name and its arguments, which are separated by two or more spaces from the keyword name and from each other. An optional assignment of the return value can be possible as well. The line comments starting with a hash `#` show the tokens in the statement.

Plain example for better readability:

```
*** Test Cases ***
Test Case Name
      Keyword Call     argument one     argument two
      Keyword Call
      ...              argument one
      ...              argument two
      ${variable_assignment}     Keyword Getter Call
```

Example with tokens in comments:

```
*** Test Cases ***
# TESTCASE HEADER |
Test Case Name
# TESTCASE    | EOL
```

```
      Keyword Call      argument one      argument two
# SEP |   KEYWORD | SEP | ARGUMENT | SEP | ARGUMENT | EOL
      Keyword Call
# SEP |   KEYWORD | EOL
      ...             argument one
# SEP | CONTINUATION | ARGUMENT | EOL
      ...             argument two
# SEP | CONTINUATION | ARGUMENT | EOL
      ${variable_assignment}      Keyword Getter Call
# SEP |      ASSIGNMENT      | SEP |      KEYWORD      | EOL
```

In the example above, the test case `Test Case Name` contains three keyword calls. The first keyword call `Keyword Call` has two arguments, `argument one` and `argument two`. The second keyword call even though it is split over two lines is considered one logical line and identical to the first keyword call. The third keyword call is a keyword call that assigns the return value of the keyword `Keyword Getter Call` to the variable `${variable_assignment}`.

Example 2: In the `*** Settings ***` section, the settings are separated from their values by four or more spaces.

```
*** Settings ***
# SETTINGS HDR |
Documentation     This is the first line of documentation.
#  SETTING   | SEP |                 VALUE                | EOL
...    # just CONTINUATION and End Of Line
...             This is the second line of documentation.
# CONTINUATION |                VALUE                  | EOL
Resource     keywords.resource
# SET   | SEP |      VALUE      | EOL
```

All elements themselves in their section are written without any indentation. So, settings in the `*** Settings ***` section, test cases in the `*** Test Cases ***` section, and keywords in the `*** Keywords ***` section are written without any indentation. However, when defining tests|tasks and keywords, indentation is used to define their body, while their name is still un-indented, e.g., after a test case name, all subsequent lines that are part of the test case body are indented by two or more spaces.

That means that a body statement always starts with a separator token, followed by a data token, like e.g. variable or keyword as seen in the examples above.

The body ends when either a new un-indented element (e.g. test case or keyword) is defined or another section like `*** Keywords ***` starts or the end of the file is reached.

Within the body of tests|tasks and keywords, control structures like loops or conditions can be used. Their content should be indented by additional four spaces to make it clear that they are part of the control structure. However, this is not mandatory and only a recommendation to make the file more readable.

While single tabulators (`\t`) as well as two or more spaces are valid separators, it is recommended to use multiple spaces for indentation and separation and avoid tabulators. This can prevent issues where different editors align text to a grid (e.g., 4 spaces) when using tabs, making it difficult for users to distinguish between tabs and spaces. It could cause a single tabulator to look the same as a single space in the editor, which would lead to misinterpretation of the file structure by a human reader.

2.2.2 Line Breaks, Continuation and Empty Lines

LEARNING OBJECTIVES

K3 LO-2.2.2

Be able to use line breaks and continuation in a statement.

Empty lines are allowed and encouraged to structure data files and make them more readable. In the next example, the sections are visibly separated by two empty lines, and the tests are separated by one empty line. Empty lines are technically not relevant and are ignored while parsing the file.

By default, each statement is terminated by a line break, allowing only one statement per literal line. However, for better readability, or to add line breaks in documentation, statements can span multiple lines by using `...` (three dots) and a separator at the start of the next line with the same indentation level as the line being continued.

A line continuation can only be performed where a separator is expected, like between a keyword name and its arguments or between two arguments or between a setting and its value(s). In the following example the two keyword calls are logically identical, even though the second one is split over three literal lines.

In documentation settings, line breaks with continuation are interpreted as a line break character. In Robot Framework documentation syntax, a single line break is treated as a space after interpretation, whereas two consecutive line breaks are considered a paragraph break. This allows you to structure documentation in a more readable and organized manner.

Example:

```
*** Settings ***
Documentation     This is the first paragraph of suite documentation.
...
...               This is the second paragraph of suite documentation.
Resource     keywords.resource


*** Test Cases ***
Test Case Name
      [Documentation]    This is the first paragraph of test documentation.
      ...
      ...                This is the second paragraph of test documentation.
      Keyword Call     argument one     argument two
      Keyword Call
      ...              argument one
      ...              argument two
      ${variable_assignment}     Keyword Getter Call
```

2.2.3 In-line Comments

LEARNING OBJECTIVES

LO-2.2.3

Be able to add in-line comments to suites.

In Robot Framework comments can be added to lines after the content by starting the comment with a separator (multiple spaces) and a hash `#`. The hash `#` is used to indicate that the rest of the line is a comment and is ignored by Robot Framework. The same works at the very start of a line, which makes the whole line a comment.

Hashes in the middle of a value are considered normal characters and do not need to be escaped.

If an argument value or any other token shall start with a hash (`#`) and is preceded by a separator (multiple spaces), the hash must be escaped by a backslash `\` like `Click Element By Css    \#element_id`.

Block comments are not supported in Robot Framework, so each line that shall be a comment must be prefixed with a hash `#`. Alternatively, the `*** Comments ***` section can be used to add multi-line comments to files.

2.2.4 Escaping of Control Characters

LEARNING OBJECTIVES

LO-2.2.4

Understand how to escape control characters in Robot Framework.

In Robot Framework strings are not quoted which leads to situations where users need to be able to define, if a specific character shall be interpreted as part of the value or as a control character.

Some examples are:

- the `#` hash character that is used to start a comment as described above.
- variables that are started by e.g. `${` (See 3.2 Variables)
- multiple spaces that are considered as separators
- equal sign `=` that is used to assign named arguments to keywords

All those characters or character sequences that are interpreted as control characters can be escaped by a backslash `\`. This means that the character following the backslash is interpreted as a normal character and not as a control character.

This leads to the fact that a backslash itself must be escaped by another backslash to be interpreted as a normal backslash character. Therefore it is strongly recommended to use forward slashes `/` as path separators in paths also on windows environments and avoid backslashes `\` whenever possible.

Leading and trailing spaces in values are normally considered being part of the separator surrounding the values. If values shall contain leading or trailing spaces they must be either enclosed in backslashes `\` or replaced by the special variable `${SPACE}` that contains a single space character.

Example:

```
*** Test Cases ***
Test of Escaping
    Log    \# leading hash.                    # This logs "# leading hash."
    Log    \ lead & trail \                    # This logs " lead & trail "
    Log    ${SPACE}and now 5 More: \ \ \ \ \    # This logs " and now 5 More:     "
    Log    Not a \${variable}                  # This logs "Not a ${variable}"
    Log    C:\\better\\use\\forward\\slashes    # This logs "C:\better\use\forward\slashes"
```

2.2.5 Example Suite File

LEARNING OBJECTIVES

LO-2.2.5

Understand the structure of a basic suite file.

In the following example, two test cases are defined in a suite file.

- `Login User With Password`

- `Denied Login With Wrong Password`

Both test the login functionality of a system by calling four keywords in their bodies.

In the `*** Settings ***` section, the suite is documented, and the keywords for connecting to the server, logging in, and verifying the login are imported from a resource file. The settings of this section are not indented, but their values are separated by four or more spaces.

In the `*** Test Cases ***` section, there are two test cases defined. The first test case, `Login User With Password`, connects to the server, logs in with the username `ironman` and the password `1234567890`, and verifies that the login was successful with the user's name `Tony Stark`. In this test, the first called keyword is `Connect To Server` without any arguments, while the second called keyword is `Login User`, and it has two argument values: `ironman` and `1234567890`.

The second test case, `Denied Login With Wrong Password`, connects to the server, tries to log in with the username `ironman` and the password `123`, and expects an error to be raised and the login to be denied.

Clearly visible due to the indentation by four spaces, the body of the test cases contains the keywords that are called to execute the test case. In the test case body, some keyword calls have arguments that are separated by two or more spaces from the keyword name.

The following tests will be executed in the order they are defined in the suite file. First, the `Login User With Password` test case will be executed, followed by the `Denied Login With Wrong Password` test case.

Example Suite File Content:

robot_files/TestSuite.robot

```
*** Settings ***
Documentation     A suite for valid and invalid login tests.
...
...               Keywords are imported from the resource file.
Resource          keywords.resource


*** Test Cases ***
Login User With Password
    Connect To Server
    Login User            ironman    1234567890    # Login with valid credentials
    Verify Valid Login    Tony Stark    # Verify that the login was successful by checking the user name
    Close Server Connection

Denied Login With Wrong Password
    Connect To Server
    Run Keyword And Expect Error    # this keyword calls another keyword and expects an error
    ...        *Invalid Password*   # it expects an error containing `Invalid Password`
    ...        Login User           # this keyword is called with two arguments
    ...        ironman
    ...        123#wrong            # A hash in the middle of a string is not a comment
    Verify Unauthorized Access
    Close Server Connection
```

2.3 Executing Robot

LEARNING OBJECTIVES

K1 LO-2.3

Recall the three components of the Robot Framework CLI.

Robot Framework comes with three executables when being installed which are designed to be used via the command-line interface (CLI).

- `robot` is the main executable that is used to execute suites.
- `rebot` (not part of this syllabus) is used to post-process execution results and generate reports.
- `libdoc` (not part of this syllabus) is used to generate keyword documentation for libraries and resource files. See 2.5 Keyword Interface and Documentation

2.3.1 `robot` command & help

LEARNING OBJECTIVES

K2 LO-2.3.1

Understand how to run the `robot` command and its basic usage.

The `robot` command is used to run a Robot Framework execution, which will execute suites and their containing tests|tasks.

At a basic level, you can run `robot` by providing the path to a suite file or suite directory containing suite files as last argument.

```
robot <path_to_root_suite>
```

In case of the 2.2.5 Example Suite File where a single suite file named `TestSuite.robot` is stored in a directory `robot_files`, to execute the example test suite the following command is used, if the current working directory of the terminal is the directory containing the `robot_files` directory:

```
> robot robot_files
```

This command starts the Robot Framework execution by first parsing all files in the given directory tree that have the extension `.robot`, then creates an execution model and finally, executes all suites with their test cases from that model. During execution, the results of each test case are printed to the console and at the end a summary is printed and reports are generated.

Example Console Output:

Console Output

```
> robot robot_files
==============================================================================
Robot Files
==============================================================================
Robot Files.TestSuite :: A test suite for valid login.
```

```
==============================================================================
Login User With Password                                              | PASS |
------------------------------------------------------------------------------
Denied Login With Wrong Password                                      | PASS |
------------------------------------------------------------------------------
Robot Files.TestSuite :: A test suite for valid login.                | PASS |
2 tests, 2 passed, 0 failed
==============================================================================
Robot Files                                                           | PASS |
2 tests, 2 passed, 0 failed
==============================================================================
Output:  /path/to/output.xml
Log:     /path/to/log.html
Report:  /path/to/report.html
```

The `robot` command can optionally be configured with additional options to control the execution behavior, such as setting output formats, specifying specific tests to run, or controlling logging levels and many more. These options are named arguments that are passed to the `robot` command BEFORE the path to the suite file or directory. To learn more about these options, you can use the help of the `robot` command like: `robot --help`.

2.3.2 Execution Artifacts

LEARNING OBJECTIVES

LO-2.3.2

Explain the execution artifacts generated by Robot Framework.

After executing a suite, Robot Framework, by default, generates three output files in the output directory. These artifacts provide detailed execution results:

- **`output.xml`**: A machine-readable file containing **ALL** logged execution details, limited by the given log-level.
- **`log.html`**: A detailed log file that provides an HTML view of the execution, including keyword-level details.
- **`report.html`**: A summary report that gives an overview of the execution results, including statistics of tests|tasks executed, passed, and failed.

`log.html` and `report.html` are generated based on the information stored in `output.xml`.

A unique feature of Robot Framework is, that it logs each keyword call and its arguments with its log outputs and timestamps, so that it is possible to have a very detailed view of the execution flow and the data that was used during the execution. In case of a failure it is possible to see the exact keyword call that failed and the arguments that were used, which can be very helpful for debugging or reporting. Furthermore you also get all passed keywords and even the non-executed keywords to protocol the whole execution flow.

2.3.3 Status

LEARNING OBJECTIVES

LO-2.3.3

Recall the four different status labels used by Robot Framework.

Robot Framework uses different status labels to indicate the result of an execution:

On Suite, Test Case, Task and Keyword Level:

- `PASS`: Indicates that the item was successfully executed without unexpected errors.
- `FAIL`: Shows that the item encountered an error and did not pass.
- `SKIP`: Indicates that the item was intentionally skipped, either by tagging or during execution, typically because some condition was not met.

Additional Keyword Status:

- `NOT RUN`: Refers to keywords that were not executed during execution, e.g. due to previous failure or conditions.

`SKIP` is explained in more detail in later chapters.

Atomic elements like Library Keywords or Robot Framework language statements do define their own status.

Composite elements like suites (composed of tests|tasks), tests|tasks (composed of keywords) and User Keywords (composed of Library Keywords and Robot Framework statements) do define their status based on the status of their child elements.

2.3.3.1 PASS

LEARNING OBJECTIVES

LO-2.3.3.1

Understand when an element is marked as `PASS`.

This status is used if an element was executed successfully without any errors or exceptions.

Atomic elements are `PASS` if they were executed successfully without reporting an error by raising an exception.

Composite elements are `PASS` if all their executed body elements are pass. E.g. in case of User Keywords this means that if all keywords or Robot Framework language statements that were directly called by that User Keyword were `PASS` the User Keyword itself is considered `PASS`.

Library Keywords like `Run Keyword And Expect Error`, from BuiltIn Library, do `PASS` if the keyword they are internally calling does raise an error with the expected message or type.

That means that a composite element like suite, test|task or User Keyword may be `PASS` even if some of its deeper child elements are `FAIL`.

2.3.3.2 FAIL

LEARNING OBJECTIVES

LO-2.3.3.2

Understand when an element is marked as `FAIL`.

This status is used if an element was executed but encountered an unexpected error or exception.

A failure typically causes the subsequent keywords not to be executed and to receive the status `NOT RUN`. Exceptions are teardowns, as explained in Chapter 4: Advanced Structuring and Execution.

Atomic elements are `FAIL` if they were tried to be executed but raised an exception.

Composite elements are `FAIL` if at least one of their executed direct body elements are `FAIL`. Therefore a failure typically distributes upwards through the hierarchy of elements until it reaches the root suite.

A User Keyword is `FAIL` if one of its called Library Keywords is `FAIL`. A test|task is `FAIL` if one of its directly called Keywords is `FAIL`. A suite (file) is `FAIL` if one of its test|task is `FAIL` and a suite (directory) is `FAIL` if one of its suites (file) is `FAIL`.

2.3.4 Logging possibilities (Log vs Console)

LEARNING OBJECTIVES

K2 LO-2.3.4

Understand the difference between log messages and console output.

There are basically two kinds of logging information in Robot Framework.

- **Console Output**: The console output is the output that is printed to the terminal where the `robot` command was executed. It is typically not persistent but can be already seen during execution.
- **Log Messages**: Log messages are written to the `output.xml` and therefore also `log.html` file and are persistent. They are typically created by the Library Keywords that are executed and can be used to log information about the execution. Also Robot Framework itself does log information to the `output.xml` like assigned values of arguments or the return values of keywords.

Log messages can be written with different levels of severity (i.e. `INFO`, `DEBUG`, `TRACE`, `WARN` or `ERROR`). Which levels are written to the log can be controlled by the log level of an execution. Further information in later chapters.

2.4 Keyword Imports

Robot Framework has a modular design that allows users to import keywords from external sources. Without importing external keywords into a suite, only the keywords from Robot Framework's BuiltIn library are available for use, due to them being imported automatically. Also the Robot Framework language statements themselves are available for use without importing them.

External keywords can be imported from either libraries or resource files. Both types of sources are using different syntax to import their keywords.

2.4.1 Libraries

LEARNING OBJECTIVES

K1 LO-2.4.1-1

Recall the purpose of keyword libraries and how to import them.

K1 LO-2.4.1-2

Recall the three types of libraries in Robot Framework.

From a user perspective there are three different kinds of libraries:

- **Robot Framework Standard Libraries**: These are libraries that are shipped with Robot Framework and are available without any additional installation. See documentation of ext: Robot Framework Standard Libraries for more information.
- **3rd Party Libraries** / **External Libraries**: These are libraries have been developed and maintained by community members and have to be installed/downloaded separately.
- **Custom Libraries**: These libraries are developed by the users themselves to solve specific problems or to encapsulate more complex functionality.

Further more detailed information about the different types of libraries and are described in later chapters.

To import a library into a suite or resource file the `Library` setting is used in the `*** Settings ***` section followed by the name of the library as long as they are located in the Python module search path, which automatically happens if they are installed via `pip`. The name of the library is case-sensitive and should be taken from the library's keyword documentation. By default, libraries in Robot Framework are implemented in Python and the name of the library is the name of the Python module that contains the library implementation.

Alternatively, if a library is not in Python module search path, a library can be imported using the path to the Python module. See 2.4.3 Import Paths.

Be aware that the library `BuiltIn` is always imported invisibly and does not need to be imported explicitly.

Example:

```
*** Settings ***
Library    OperatingSystem
Library    Browser
Library    DatabaseLibrary
```

Once a library is imported, all keywords from that library are available for use in that suite or resource file. Which keywords are available can be seen in the keyword documentation of the library or may be visible in the IDE by code completion, depending on the IDE extension being used.

2.4.2 Resource Files

LEARNING OBJECTIVES

K1 LO-2.4.2-1

Recall the purpose of resource files.

K3 LO-2.4.2-2

Use resource files to import new keywords.

As mentioned before resource files are used to organize and store keywords and variables that are used in multiple suites.

They share a similar structure and the same syntax as suite files, but they do not contain test cases or tasks. See 2.2 Basic Suite File Syntax for more information about the structure of suite files and 3.1 Resource File Structure for more details of their structure.

They can contain other keyword imports, which cause the keywords from the imported libraries or resource files to be available in the suites where the resource file is imported. The same applies for variables that are defined and imported from other resource files. Therefore keywords from a library that have been imported in a resource file are also available in the suite that imports that resource file.

To import a resource file into a suite or resource file the `Resource` setting is used in the `*** Settings ***` section followed by the path to the resource file. See 2.4.3 Import Paths for more information about the path to the resource file.

Resource files shall have the extension `.resource` to make it clear what they are. `.resource` and `.robot` extensions are also recognized by IDE extensions, supporting Robot Framework.

Example:

```
*** Settings ***
Resource    local_keywords.resource
Resource    D:/keywords/central_keywords.resource
```

See more about the structure of resource files in 3.1 Resource File Structure and how keywords and variables are created in the sections following that.

2.4.3 Import Paths

LEARNING OBJECTIVES

K2 LO-2.4.3

Understand the different types of paths that can be used to import libraries and resource files.

When importing libraries or resource files via a path, the path can be either an absolute path or a relative path. If a relative path is given, the path is resolved relative to the data file that is importing the library or resource file.

If an **absolute path** is given, the resource file or library is searched for at the given path.

If a **relative path** is given, the resource file or library is searched for relative to the data file that is importing it and then relative to the Python *module search path*. This *module search path* is defined by the Python interpreter that executes Robot Framework and can be influenced by the environment variable `PYTHONPATH` or by using the CLI-Argument `--pythonpath` when executing `robot`.

For **path separators**, it is strongly recommended to always use forward slashes (`/`), and even on Windows, not to use backslashes (`\`). This is because backslashes are used as escape characters in Robot Framework, which can cause issues when used in paths. Forward slashes are supported on all operating systems when used in Robot Framework.

When choosing the location of resource files or libraries, it should be taken into that consideration that absolute paths are typically not portable and therefore should be avoided. Relative paths are portable as long as they are related to the data file that is importing using them, as long as that relative path is part of the project structure.

However the most stable and recommended way is to use the **Python Path/module search path** to import them. That path needs to be defined when executing Robot Framework but can lead to more uniform and stable imports, because each suite or resource file can use the same path to import the same resource file or library, independent of the location of the importing suite or resource file.

2.5 Keyword Interface and Documentation

LEARNING OBJECTIVES

K2 LO-2.5

Understand the structure of keyword interfaces and how to interpret keyword documentation.

Library Keywords and User Keywords that are defined in a resource file should have a documentation text that describes what the keyword does and how it should be used.

Robot Framework is capable of generating **Keyword Documentation** files that contains a library- or resource-documentation, all keywords, their argument interfaces, and their documentation texts. This documentation file can be generated with the `libdoc` command and can be used to provide a reference for users who want to use the keywords.

Basically all standard and external 3rd party libraries offer these Keyword Documentations as online available HTML pages.

Robot Framework offers the Keyword Documentation of its Standard Libraries at https://robotframework.org/robotframework .

2.5.1 Documented Keyword Information

LEARNING OBJECTIVES

K1 LO-2.5.1

Recall the information that can be found in a keyword documentation.

The Keyword Documentation is structured so, that it contains first the library or resource documentation, followed by a list of all keywords that are available in that library or resource file.

Each library or resource documentation can contain the following information sections for keywords:

- **Name**: The name of the keyword as it is called.
- **Arguments** (opt.): The argument interface that the keyword expects/offers its types and default values.
- **Return Type** (opt.): The type of the return value of the keyword.
- (*) **Tags** (opt.): The tags that are assigned to the keyword to categorize keywords.
- **Documentation** (opt.): The documentation text that describes what the keyword does and how it should be used.

(*) Understanding keyword tags is not part of the syllabus.

The following keywords are part of the Standard Libraries of Robot Framework. Their documentation has been generated by the Robot Framework tool `libdoc` which is included in Robot Framework.

2.5.1.1 Example Keyword `Should Be Equal`

Documentation of `Should Be Equal` from `BuiltIn` library

`Should Be Equal` is part of the BuiltIn library and is documented as follows:

Should Be Equal

Arguments

```
first
second
msg                 = None
values              = True
ignore_case         = False
formatter           = str
strip_spaces        = False
collapse_spaces     = False
```

Documentation

Fails if the given objects are unequal.

Optional `msg`, `values` and `formatter` arguments specify how to construct the error message if this keyword fails:

Examples:

Should Be Equal	${x}	expected		
Should Be Equal	${x}	expected	Custom error message	
Should Be Equal	${x}	expected	Custom message	values=False
Should Be Equal	${x}	expected	ignore_case=True	formatter=repr

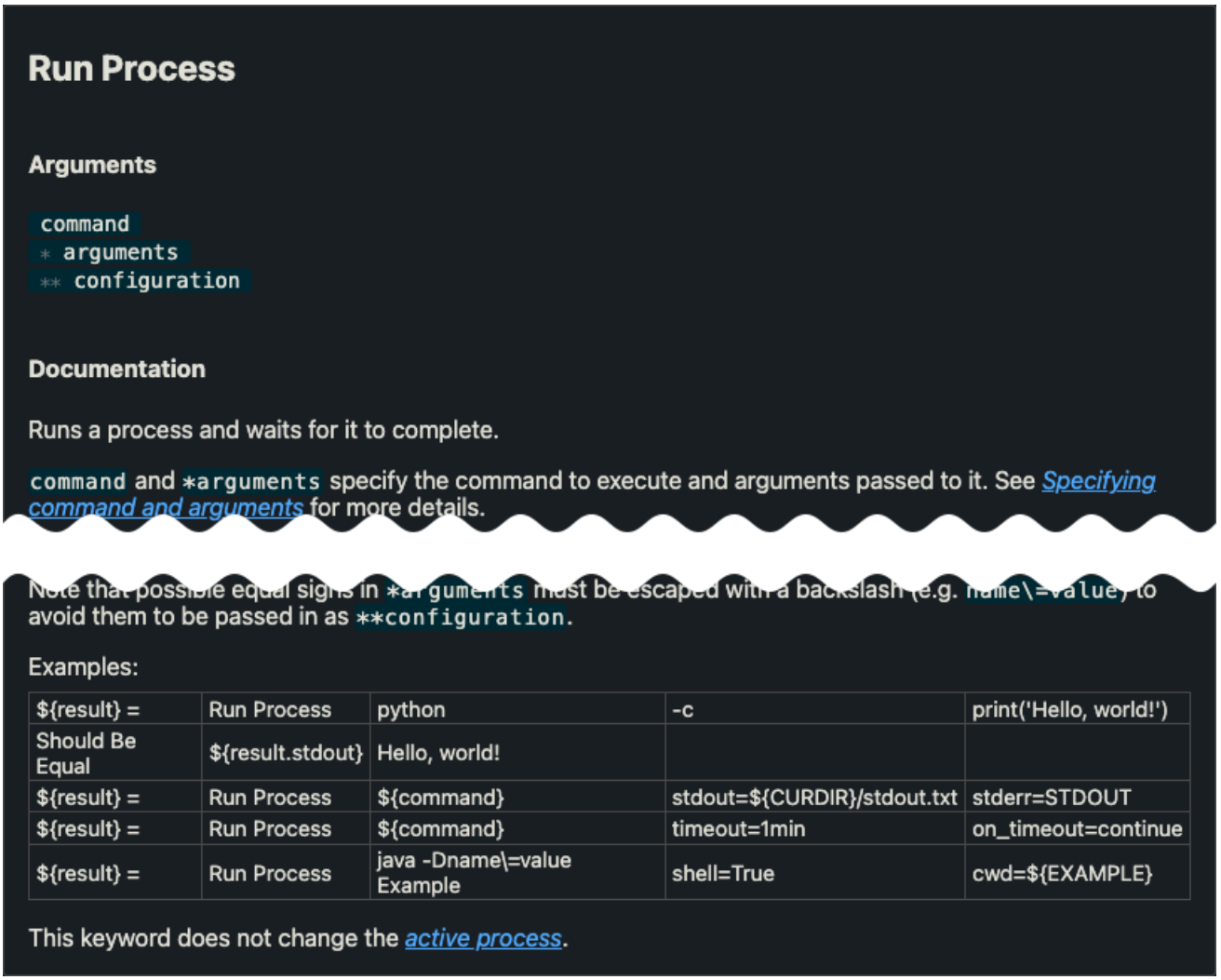
`strip_spaces` is new in Robot Framework 4.0 and `collapse_spaces` is new in Robot Framework 4.1.

This keyword has 2 "Mandatory Arguments" and 6 "Optional Arguments". All of them can be called positionally or by name.

2.5.1.2 Example Keyword `Run Process`

Documentation of `Run Process` from `Process` library

`Run Process` is part of the Process library and is documented as follows:



Run Process

Arguments

```
command
* arguments
** configuration
```

Documentation

Runs a process and waits for it to complete.

`command` and `*arguments` specify the command to execute and arguments passed to it. See *Specifying command and arguments* for more details.

Note that possible equal signs in `*arguments` must be escaped with a backslash (e.g. `name\=value`) to avoid them to be passed in as `**configuration`.

Examples:

${result} =	Run Process	python	-c	print('Hello, world!')
Should Be Equal	${result.stdout}	Hello, world!		
${result} =	Run Process	${command}	stdout=${CURDIR}/stdout.txt	stderr=STDOUT
${result} =	Run Process	${command}	timeout=1min	on_timeout=continue
${result} =	Run Process	java -Dname\=value Example	shell=True	cwd=${EXAMPLE}

This keyword does not change the *active process*.

This keyword has one **Mandatory Arguments** `command` which can be called positionally or by name. The latter two arguments are optional.

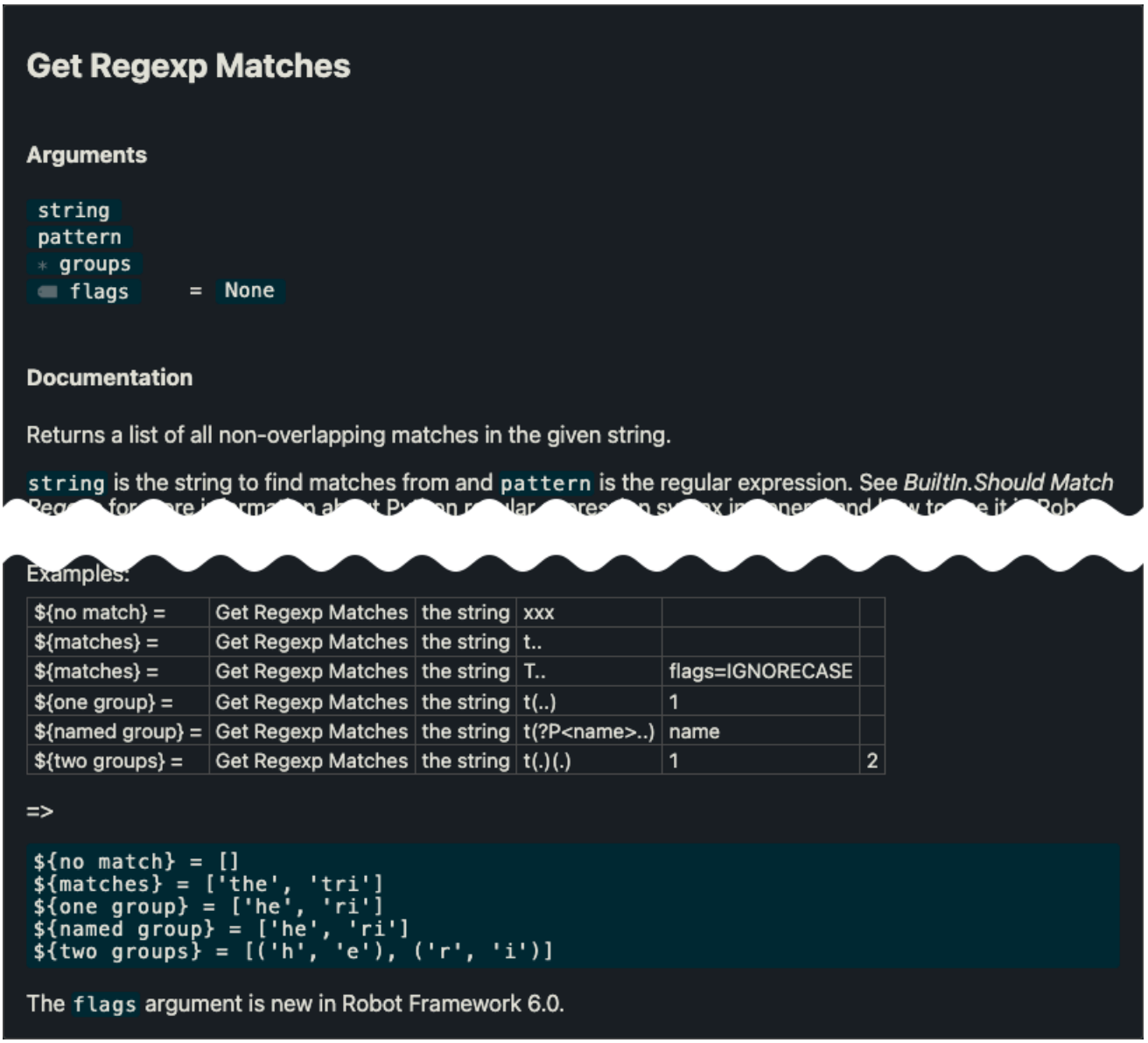
The argument `arguments` is a "Variable Number of Positional Arguments" and can only be set by position. Therefore, if it shall be set, all preceding arguments must be set by position as well. See 2.5.2.5 Variable Number of Positional Arguments for more information about this kind of argument.

The argument `configuration` is a "Free Named Argument" and can only be set by names. See 2.5.2.7 Free Named Arguments for more information about this kind of argument.

2.5.1.3 Example Keyword `Get Regexp Matches`

Documentation of `Get Regexp Matches` from `String` library

`Get Regexp Matches` is part of the String library and is documented as follows:



This keyword has 2 "Mandatory Arguments" that can be called positionally or by name. The last two arguments are optional.

The argument `groups` is a "Variable Number of Positional Arguments" and can only be set by position. Therefore, if it shall be set, all preceding arguments must be set by position as well. See 2.5.2.5 Variable Number of Positional Arguments for more information about this kind of argument.

The argument `flags` is a "Named-Only Argument" and can only be set by name. See 2.5.2.6 Named-Only Arguments for more information about this kind of argument.

2.5.2 Keyword Arguments

LEARNING OBJECTIVES

K2 LO-2.5.2

Understand the difference between argument kinds.

Most library keywords can be parameterized with arguments that are passed to the keyword when it is called to customize its behavior. The more business oriented keywords are the less arguments they typically have.

Keyword arguments can be grouped into different argument kinds. On the one hand you can group them by their definition attributes and on the other hand by their usage kind.

The relevant distinction of usage kinds is between using **Positional Arguments**, **Named Arguments**, or **Embedded Arguments**. How to use them is described in 2.6 Writing Test|Task and Calling Keywords.

Another important information is if an argument is mandatory or optional. See the next two sections for more information about these two kinds of arguments.

Most arguments can either be set by their position or by their name. But there are some kinds of arguments that can only be set positionally, like **Variable Number of Positional Arguments**, or only be set named, like **Named-Only Arguments** or **Free Named Arguments**.

The order is as follows:

1. **Positional or Named Arguments** (can be mandatory or optional)
2. **Variable Number of Positional Arguments** (optional)
3. **Named-Only Arguments** (can be mandatory or optional)
4. **Free Named Arguments** (optional)

2.5.2.1 Mandatory Arguments

LEARNING OBJECTIVES

K2 LO-2.5.2.1

Understand the concept of mandatory arguments and how they are documented.

Arguments that do not have a default value, must be set when the keyword is called. These arguments have to be before arguments with default values in the argument interface of the keywords.

See the argument named `first` and `second` in the `Should Be Equal` keyword documentation in the beginning of this section.

If too few arguments are provided, the keyword call will fail with an error message.

Example:

```
*** Test Cases ***
Tests Will Pass
    Should Be Equal    One    One

Test Will Fail
    Should Be Equal    One    Two

Test Will Fail Due to Missing Args
    Should Be Equal    One
```

The first Test will pass, because both argument values are equal. The second Test will fail, because the argument values are not equal. The third Test will fail before the keyword `Should Be Equal` is actually being executed, because the keyword expects at least two arguments. The Error Message would be:
`Keyword 'BuiltIn.Should Be Equal' expected 2 to 8 arguments, got 1.`

Two arguments are mandatory and the additional six arguments are optional in the `Should Be Equal` keyword.

2.5.2.2 Optional Arguments

LEARNING OBJECTIVES

LO-2.5.2.2

Understand the concept of optional arguments and how they are documented.

Arguments that have a default value can be omitted when the keyword is called, causing these arguments to be set to their default value. These arguments are listed after the mandatory arguments in the argument interface. Default values are defined and represented in the docs by the equal sign (`=`) after the argument name and a value after that.

Also "Variable Number of Positional Arguments", represented with a single star (`*`) prefix, and "Free Named Arguments", represented with a double star (`**`) prefix are optional arguments.

E.g. the argument `msg` in the `Should Be Equal` keyword documentation has the default value `None` and `ignore_case` has the default value `False`.

In that particular keyword these optional arguments can be used to activate some special features like ignoring the case of the compared strings or to provide a custom error message.

Omitting some optional arguments but still using others is possible independent of their order by setting these arguments by their name. See 2.6 Writing Test|Task and Calling Keywords.

2.5.2.3 Embedded Arguments

LEARNING OBJECTIVES

LO-2.5.2.3

Recall the concept of keywords with embedded arguments used in Behavior-Driven Specification and how they are documented.

Keywords can include arguments embedded directly into their names, a feature primarily used for Behavior-Driven Development (BDD). Embedded arguments are mandatory and must be provided in the exact position defined within the keyword name.

Keyword names include arguments defined using the scalar variable syntax with dollar and curly braces (`${var_name}`). This syntax explicitly defines these as arguments, distinguishing them from the rest of the keyword name.

Example keyword names are:

- `"${url}" is open`
- `the user clicks the "${button}" button`
- `the page title should be ${exp_title}`
- `the url should be ${exp_url}`

Example Test Case:

```
*** Test Cases ***
Foundation Page should be Accessible
    Given "robotframework.org" is open
    When the user clicks the "FOUNDATION" button
    Then the page title should be Foundation | Robot Framework
    And the url should be https://robotframework.org/foundation
```

The optional prefixes `Given`, `When`, `Then`, `And` and `But` are basically ignored by Robot Framework if a keyword is found matching the rest of the name including the embedded arguments. In the example test case some keywords are designed so that the arguments are surrounded by double quotes (`"`) for better visibility.

A mix of embedded arguments and "normal" arguments is possible to fully support BDD. In the keyword documentation the embedded arguments are written in variable syntax with dollar-curly-braces (`${var_name}`) to indicate that they are not part of the keyword name but are arguments. They can also be defined using regular expressions to allow for more complex argument structures, which is not part of this syllabus.

2.5.2.4 Positional or Named Arguments

LEARNING OBJECTIVES

K1 LO-2.5.2.4

Recall how "Positional or Named Arguments" are marked in the documentation and their use case.

Except for "Positional-Only Arguments", which are not part of this syllabus, all arguments that are positioned before "Variable Number of Positional Arguments", "Named-Only Arguments", or "Free Named Arguments" in the argument interface of a keyword are "Positional or Named Arguments".

As their name states, they can be set either by their position or by their name, but not by both at the same time for one argument. If an argument shall be set by its position, all preceding arguments must be set by their position as well.

These arguments can either be mandatory or optional with a default value.

They are not specially marked in the keyword documentation with any prefix, because they are the default kind of arguments in Robot Framework.

2.5.2.5 Variable Number of Positional Arguments

LEARNING OBJECTIVES

K1 LO-2.5.2.5

Recall how "Variable Number of Positional Arguments" are marked in the documentation and their use case.

A special case of optional arguments that can only be set by their position are "Variable Number of Positional Arguments". These are also referred to as `*args` or `*varargs` in Python. Some keywords need to collect a variable amount of values into one argument, because it is not possible to define the amount of values in advance.

One example for this kind of keyword is 2.5.1.2 Example Keyword `Run Process` from the Process library. This keyword executes a `command` with variable amount of `arguments` and waits for the process to finish. Depending on the command to be executed different amount of arguments are needed for that command.

This variable argument is marked with a single asterisk `*` before the argument name in the keyword documentation.

When calling this keyword, the first positional argument is assigned to `command`, while all subsequent positional arguments are collected into `arguments`. Because of this behavior, no additional positional arguments can be used after these "Variable Number of Positional Arguments". As a result, any arguments following these "Variable Number of Positional Arguments" must be named arguments, regardless of whether they are mandatory or optional arguments with a default value.

Also see 2.5.1.3 Example Keyword `Get Regexp Matches`.

2.5.2.6 Named-Only Arguments

LEARNING OBJECTIVES

LO-2.5.2.6

Recall what properties "Named-Only Arguments" have and how they are documented.

All arguments that are defined after a "Variable Number of Positional Arguments" (`*varargs`) are "Named-Only Arguments". However it is also possible to create "Named-Only Arguments without a preceding "Variable Number of Positional Arguments".

"Named-Only Arguments" are marked with a "LABEL" sign before the argument name in the keyword documentation.

Those arguments can not be set positionally. All positional values would be consumed by the "Variable Number of Positional Arguments". So they must be called by their name followed by an equal sign `=` and the value of the argument.

"Named-Only Arguments" can be mandatory or optional with a default value.

2.5.2.7 Free Named Arguments

LEARNING OBJECTIVES

LO-2.5.2.7

Recall how free named arguments are marked in documentation.

Another special case of "Named-Only Arguments" are "Free Named Arguments." These arguments are similar to the "Variable Number of Positional Arguments" in that they can collect multiple values. However, instead of collecting positional values, they gather all named values that are not explicitly defined as argument names. In this case all values given to the keyword as arguments, that do contain an unescaped equal sign (`=`) are considered as named arguments.

Free named arguments are marked with two asterisks `**` before the argument name in the keyword documentation.

The example of the `Run Process` keyword also has a free named argument `** configuration`.

When calling this keyword all named arguments that are not explicitly defined as argument names are collected into the `configuration` argument and will be available as a dictionary in the keyword implementation.

They are optional and can be omitted.

With this configuration it is possible to e.g. redirect the output of the process to a file or to set the working directory of the process.

Example redirecting stdout and stderr to a file:

```
*** Test Cases ***
Send 5 IPv4 Pings On Windows
    Run Process    ping    -n    5    -4    localhost    stdout=ping_output.txt    stderr=ping_error.tx
```

2.5.2.8 Argument Types

LEARNING OBJECTIVES

LO-2.5.2.8

Understand the concept of argument types and automatic type conversion.

Keywords may define the expected types of their argument values. The Robot Framework specification is predominantly a string-based language, therefore most statically defined argument values are strings. However, the actual implementation of the keyword may expect a different type of argument, like an integer.

If an argument type is defined and Robot Framework has a matching converter function available, that can convert the given type to the expected type, the conversion is tried automatically. If the conversion fails, the keyword call will fail with an error message before the actual keyword code is executed. Robot Framework brings some built-in converters for common types like integer, float, boolean, list, dictionary, etc. Library developers can also register their own converters for none-supported types.

Defining types for arguments is nowadays the recommended way to let Robot Framework convert the given arguments to the expected type, however it is optional.

Lets imagine a keyword that clicks on a specific coordinate on the screen, e.g. `Click On Coordinates`. This keyword would expect two integer arguments, one for the `x`-coordinate and one for the `y`-coordinate.

That keyword can now claim that it expects two integer arguments by defining type hints for these arguments. Type hints are shown in the keyword documentation at the arguments after the optional default value.

Robot Framework in that case tries to convert the given string arguments to the integer type.

Example:

```
*** Test Cases ***
Test Conversion
    Click On Coordinates    10    20    # This will work
    Click On Coordinates    10    Not_A_Number  # This will fail
```

In the first call the keyword will be called with the integer values `10` and `20` and will work as expected. The second keyword call will fail, because the second argument is not a number and cannot be converted to an integer. The error message would be: `ValueError: Argument 'y' got value 'Not_A_Number' that cannot be converted to integer.`

The advantage of using type hints is that the user get more information about what kind of values are expected and the keyword implementation can be simpler, because it can rely on the arguments being of the expected type.

2.5.2.9 Return Types

LEARNING OBJECTIVES

K2 LO-2.5.2.9

Understand the concept of return type hints.

Keywords may gather information and return these to the caller of that keyword to be stored in a variable and used in further keyword calls. So, a keyword can `RETURN` values to the caller as functions do in programming languages.

If the keyword implementation offers a type hint for the return value, this is documented in the keyword documentation. Similar to the argument types, return types are optional and a more recent feature of Robot Framework and therefore not widely used, yet.

It is important to know that keywords without a return type hint are often still returning values! This is typically documented in the *Documentation* part of the keyword documentation.

2.5.3 Keyword Documentation & Examples

LEARNING OBJECTIVES

LO-2.5.3

Understand how to read keyword documentation and how to interpret the examples.

Keyword documentation is an important part of the keyword implementation. Good keyword names that clearly communicate what a keyword does are even more important, but this should not give the impression that descriptive documentation is unnecessary.

Documentation is sometimes lean and sometimes extensive, depending on the complexity of the keyword. The documentation should describe what the keyword does, how it should be used, and what the expected arguments and possible returned values are. Depending on the complexity it may also be useful to provide examples of how the keyword can be used.

User Keywords typically have less extensive documentation because they are used in a narrower context and cannot be configured with arguments as much as library keywords from generic external libraries.

Examples in the documentation are commonly either written in table format or as code blocks.

Table Example of `Should Be Equal`:

Should Be Equal	`${x}`	expected		
Should Be Equal	`${x}`	expected	Custom error message	
Should Be Equal	`${x}`	expected	Custom message	values=False
Should Be Equal	`${x}`	expected	ignore_case=True	formatter=repr

Code block example:

```
Should Be Equal    ${x}    expected
Should Be Equal    ${x}    expected    Custom error message
Should Be Equal    ${x}    expected    Custom message    values=False
Should Be Equal    ${x}    expected    ignore_case=True    formatter=repr
```

2.6 Writing Test|Task and Calling Keywords

LEARNING OBJECTIVES

K2 LO-2.6

Understand how to call imported keywords and how to structure keyword calls.

A typical test case or task is a sequence of keyword calls that are executed in a specific order. As learned before these keywords need to be imported into the suite or resource file before they can be used. When using keywords in a test|task or User Keyword, it is important to indent the keyword calls correctly. With the exception of returning values, which are described in Chapter 3, the name of the keyword is the first element of the keyword call followed by the arguments that are separated by two or more spaces.

The following example shows different ways to call imported keywords in a test case based on the `Should Be Equal` keyword from the BuiltIn library.

The keyword name should be written as defined in the keyword documentation and may have single spaces or other special characters in it. After the keyword name the arguments are set. All arguments are separated by multiple spaces from the keyword name and from each other and can also include single spaces. Argument values are stripped from leading and trailing spaces, but spaces within the argument value are preserved.

If an argument shall contain more than one consecutive spaces or start or end with spaces, the spaces must be escaped by a backslash `\` to prevent them from being interpreted as a part of a "multi-space-separator".

Example:

```
*** Test Cases ***
Mandatory Positional Arguments
    [Documentation]    Only mandatory arguments are use positionally
    Should Be Equal    1    1

Mixed Positional Arguments
    [Documentation]    Mandatory and optional arguments are used positionally.
    ...
    ...    It is hard to figure out what the values are doing and which arguments are filled,
    ...    without looking into the keyword documentation.
    ...    Even though the argument `values` is kept at its default value `True` it must be set if late
    Should Be Equal    hello    HELLO    Values are case-insensitive NOT equal    True    True

All Named Arguments
    [Documentation]    Arguments are used named.
    ...
    ...    It is clear what the values are doing and which arguments are filled and order is not releva
    ...    The argument `values` can be omitted and the order can be mixed
    Should Be Equal    first=hello    second=HELLO    ignore_case=True    msg=Values are case-insensiti

Mixed Named and Positional Arguments
    [Documentation]    Arguments are used named and positional.
    ...
    ...    The positional arguments must be in order, but the subsequent named arguments may be in an a
    ...    The first arg has the string value `" hello  spaces "` and the second arg has the string valu
    Should Be Equal    \ hello \ spaces \    HELLO \ SPACE    ignore_case=True    strip_spaces=True    m
```

2.6.1 Positional Arguments

LEARNING OBJECTIVES

K2 LO-2.6.1

Understand the concept of how to set argument values positionally.

When calling keywords, arguments can often be set positionally in the order they are defined in the keyword documentation. An exception to this are "Named-Only Arguments" and "Free Named Arguments" that can only be set by their name.

However, only using positional values can lead to poor readability as you can see in the previous example: `Mixed Positional Arguments` Some keywords do not have an obvious order of arguments. In these cases, calling keywords with named arguments can lead to better readability and understanding of the keyword call.

Using arguments positionally is very handy for arguments that are obvious and easy to understand. In the early login example the following keyword calls exists:

```
*** Test Cases ***
Login User With Password
    Login User    ironman    1234567890
```

In that case it should be obvious that the first argument is the username and the second argument is the password. Also the following keyword call should be easy to understand but could still be more explicit by using named arguments.

```
*** Test Cases ***
Click on x and y
    Click On Coordinates    82    70
    Click On Coordinates    x=82    y=70
```

Calling keywords that have a "Variable Number of Positional Arguments" does require to set all preceding arguments by their position if the "Variable Number of Positional Arguments" shall be set.

Example:

```
*** Test Cases ***
Run Process Without Arguments
    ${dir}  Run Process    command=dir
    Log    ${dir.stdout}

Run Process With Arguments
    ${ping}    Run Process    ping    -c    2    127.0.0.1
    Log    ${ping.stdout}
```

In the second test `Run Process With Arguments` the first given value `ping` is assigned to the argument `command` and all following values are collected into the `arguments` argument of the keyword `Run Process` as a list of values.

2.6.2 Named Arguments

LEARNING OBJECTIVES

K2 LO-2.6.2

Understand the concept of named arguments and how to set argument values by their name.

Keyword Calls with non-obvious arguments should use named argument calls if possible. Also setting one optional argument but leaving the others at their default value is an indication to use named arguments.

Named arguments are set by their name followed by an equal sign `=` and the value of the argument. All named arguments must be set after the positional arguments are set but can be set in any order.

Equal signs are valid argument values and could therefore be misinterpreted as named arguments, if the text before the equal sign is an existing argument name or if "Free Named Arguments" are available at the called keyword. To prevent that, an equal sign in argument values can be escaped by a backslash `\`.

Example of escaping conflicting equal signs:

```
*** Test Cases ***
Test Escaping Equal Sign
    Should Be Equal    second\=2    Second\=2    ignore_case=True
```

The argument `first` does get the value `second=2` and the argument `second` does get the value `Second=2`.

2.6.3 Embedded Arguments / Using Behavior-Driven Specification

LEARNING OBJECTIVES

LO-2.6.3

Recall how to use embedded arguments.

Embedded Arguments are mostly used in Behavior-Driven Development (BDD) using Robot Frameworks Behavior-Driven Specification style.

Embedded Arguments are part of the keyword name as described in 2.5.2.3 Embedded Arguments.

When calling keywords with embedded arguments, all characters that are at the position where the embedded argument is expected are used as the argument value.

See the example in section 2.5.2.3 Embedded Arguments.

See also 2.5.2.3 Embedded Arguments for more information about how to use embedded arguments.

3 Keyword Design, Variables, and Resource Files

This chapter introduces the essential components of Robot Framework: **Keywords**, **Variables**, and **Resource Files**. These building blocks allow users to create reusable, structured, and maintainable automation solutions. Understanding these concepts is critical for developing efficient automation in both testing and RPA contexts.

3.1 Resource File Structure

Resource Files in Robot Framework are used to store reusable keywords, variables, and organize imports of other resource files and libraries. See 2.4.2 Resource Files for an introduction to Resource Files.

Resource Files are typically used in many suites to share common keywords and variables across different tests|tasks. Therefore, they should be designed to be modular, reusable, and maintainable. Keywords and variables defined in one resource file should therefore be related to each other to store similar functionality or data. This relation can be based on a common purpose, a common abstraction layer, or a common context.

For example all user keywords and variables that do control or test a specific part or dialog of an application could be stored together in one resource file.

Resource files are imported using the `Resource` setting in the `*** Settings ***` section so that the path to the resource file is given as an argument to the setting. The extension for resource files shall be `.resource`.

Unless the resource file is given as an absolute path, it is first searched relatively to the directory where the importing file is located. If the file is not found there, it is then searched from the directories in Python's module search path. See 2.4.3 Import Paths for more details.

3.1.1 Sections in Resource Files

See 2.1.2 Sections and Their Artifacts for an introduction to sections in suites.

In difference to suite files, resource files do **not** allow the `*** Test Cases ***` or `*** Tasks ***` sections.

The allowed sections in recommended order are:

- `*** Settings ***` to import libraries and other resource files.

  This section has common but also different settings available than in suites.

  Common settings are:

  - `Library` to import libraries.
  - `Resource` to import other resource files.
  - `Variables` to import variable files. (Not part of this syllabus)
  - `Documentation` to provide documentation for the resource file.

  Additional settings are:

  - `Keyword Tags` to set tags for all keywords in the resource file. defining and using Keyword tags is not part of this syllabus.

  Other settings available in suites are not available in resource files.

- `*** Variables ***` to define variables.

  See 3.2.2 `*** Variables ***` Section for more details about defining variables in resource files. Other than in suites these variables can be used outside this resource file, if it is imported in another file.

- `*** Keywords ***` to define user keywords.

  See 3.3.1 `*** Keywords ***` Section for more details about defining keywords in resource files. Other than in suites these keywords can be used outside this resource file, if it is imported in another file.

- `*** Comments ***` is used to store comments and is ignored and not parsed by Robot Framework. (same as in suites)

3.2 Variables

LEARNING OBJECTIVES

K2 LO-3.2-1

Understand how variables in Robot Framework are used to store and manage data

K1 LO-3.2-2

Recall the relevant five different ways to create and assign variables

Variables in Robot Framework are used to store values that can be referenced and reused throughout suites, test cases, tasks, and keywords. They help manage dynamic data or centrally maintained data, reducing hardcoding in multiple locations and making automation flexible.

Variables can be created and assigned in various ways, such as:

- Definition in the `*** Variables ***` section in suites or resource files. (see 3.2.2 `*** Variables ***` Section)
- Capturing return values from keywords. (see 3.2.3 Return values from Keywords)
- Inline assignment using the `VAR` statement. (see 3.2.4 `VAR` Statement)
- As arguments passed to keywords. (see 3.3.5 User Keyword Arguments)
- By the command line interface of Robot Framework. (See 5.1.3 Global Variables via Command Line)
- (*) By internal implementation of library keywords.
- (*) By importing variables from variable files.

(*) These methods are not part of this syllabus.

Beside variables created by the user, Robot Framework also supports **Built-in Variables** that are explained in the 5.1.6 Built-In Variables chapter.

3.2.1 Variable Syntax and Access Types

LEARNING OBJECTIVES

K1 LO-3.2.1-1

Recall the four syntactical access types to variables with their prefixes

K1 LO-3.2.1-2

Recall the basic syntax of variables

Variables in Robot Framework are defined by three attributes:

- **Prefix**: `$`, `@`, or `&` to define the access type to the variable. (`%` for environment variables)
- **Delimiter**: `{}` to enclose the variable name.
- **Variable Name**: The string that addresses the variable. i.e. just the `variable_name` or more advanced access ways.

Variable names are case-insensitive and as keywords, containing single spaces and underscores are ignored when matching variable names. Robot Framework supports Unicode and allows the use of special characters and even Emojis in variable names.

In case these prefixes followed by a curly brace opening (`${`) should be used as characters in a normal string and not as a variable, they must be escaped by a backslash like `\${` to be treated as text rather than a variable start.

Robot Framework, implemented in Python, can work with any object stored in variables, and syntactically distinguishes four types of accessing variables:

- **Scalar Variables**: Store values as a single entity and are represented by the dollar-syntax `${variable_name}`.
- **List Variables**: Store multiple values in a list structure. They are created using the at-syntax `@{list_variable_name}`.
- **Dictionary Variables**: Store key-value pairs in a dictionary structure. They are created using the ampersand-syntax `&{dictionary_variable_name}`.
- **Environment Variables** (read-only): Read access to environment variables of the operating system using the percent-syntax `%{ENV_VAR_NAME}`.

These different syntactical handling methods allow the users to also create and handle lists and dictionaries natively in Robot Framework. However, these prefixes just define the access type to the variable, and the actual data stored in the variable can be of any type, including strings, numbers, lists, dictionaries, or even objects.

When creating variables, different syntax is used to define the type of the variable as described in the next sections, but when accessing the variable, the scalar variable syntax with a dollar sign `$` as the prefix is used in most cases. More details about list-like and dictionary-like variables, and when to use `@` or `&` when accessing these variables, can be found in the 5.1 Advanced Variables chapter.

3.2.2 `*** Variables ***` Section

LEARNING OBJECTIVES

LO-3.2.2-1

Create variables in the Variables section

LO-3.2.2-2

Use the correct variable prefixes for assigning and accessing variables

Variables can be defined in the `*** Variables ***` section within both suite files and resource files.

- Variables defined in a **suite file** are accessible throughout that specific suite, enabling consistent use across all test|tasks, and keywords executed within that suite.
- Variables defined in a **resource file**, however, are accessible in all files that import the resource file directly or indirectly by imports of other resource files. This allows for the sharing of variables across multiple suites or files while maintaining modularity and reusability.

This section is evaluated before any other section in a resource or suite file, and therefore variables defined here can be used in any other section of the file.

This section is typically used to define constants or to initialize variables that may be re-assigned during execution and more globally used.

Variables created in this section:

- are not indented,
- must be created either as `scalar ($)`, `list-like (@)`, or `dictionary-like (&)` variables,
- can be followed by an optional single space and equal sign (`=`) to improve readability,
- are separated from their following value(s) by multiple spaces,
- can be defined in multiple lines using the `...` syntax.
- have a **suite scope** in the suite created or imported to.

Because two or more spaces are used to separate elements in a row, all values are stripped of leading and trailing spaces, identical to arguments of keyword calls (see 2.2.4 Escaping of Control Characters to be able to define these spaces.

Variable values in Robot Framework can include other variables, and their values will be concatenated at runtime when the line is executed. This means that when a variable is used within another variable's value, the final value is resolved by replacing the variables with their actual content during execution.

Variables defined in the `*** Variables ***` section are recommended to be named in uppercase to distinguish them from local variables defined in test cases or keywords.

3.2.2.1 Scalar Variable Definition

LEARNING OBJECTIVES

K3 LO-3.2.2.1-1

Create and assign scalar variables

K2 LO-3.2.2.1-2

Understand how multiple lines can be used to define scalar variables

Example of creating scalar variables:

```
*** Variables ***
${NAME}       Robot Framework
${VERSION}    8.0
${TOOL}       ${NAME}, version: ${VERSION}
```

The variable `${TOOL}` will be resolved to `Robot Framework, version: 8.0` at runtime.

If the value of a scalar variable is long, you can split it into multiple lines for better readability using the `...` syntax. By default, multiple values are concatenated with a space.

You can also define a custom separator by specifying the last value as a lowercase `separator=` followed by the desired separator value (e.g., newline: `separator=\n`). Alternatively, you can use no separator at all by specifying `separator=` to join the values into a single string.

In the rare case that `separator=` should be taken literally as part of the variable value, it must be escaped with a backslash, like `\separator=`, to be treated as text rather than as a separator definition.

Example:

```
*** Variables ***
${EXAMPLE}        This value is joined
...               together with a space.
```

```
${MULTILINE}      First line.
...               Second line.
...               separator=\n
${SEARCH_URL}     https://example.com/search
...               ?query=robot+framework
...               &page=1
...               &filter=recent
...               &lang=en
...               &category=test-automation
...               separator=
```

`${SEARCH_URL}` will contain:

```
https://example.com/search?query=robot+framework&page=1&filter=recent&lang=en&category=test-automation
```

3.2.2.2 Primitive Data Types

LEARNING OBJECTIVES

LO-3.2.2.2

Understand how to access primitive data types

Robot Framework does support primitive data types as part of the syntax.

These are:

- **Strings**: a sequence of unicode characters.
- **Integers**: whole numbers (negative/positive) are written in variable syntax like: `${42}` or `${0}`.
- **Floats**: numbers with a decimal point (negative/positive) are written in variable syntax like: `${3.14}` or `${1.0}`.
- **Booleans**: `${True}` or `${False}`.
- **None**: a special value representing the absence of a value written as `${None}`.

Except for Strings, which are defined without any quotation or enclosure, the other primitive data types are defined by using the scalar variable syntax `${variable_value}`.

These values are case-insensitive and can be used in any context where a variable is accepted.

Example:

```
*** Variables ***
${STRING}            This is a string
${STILL_STRING}      8270    # These are the four characters 8, 2, 7, and 0
${INTEGER}           ${42}
${FLOAT}             ${3.14}   # Dot is used as decimal separator
${BOOLEAN}           ${True}   # Case-insensitive
${NOTHING}           ${NONE}
${EMPTY_STRING}
${ANSWER}            The answer is ${INTEGER}    # This will be 'The answer is 42'
```

IMPORTANT

When using other types than strings and concatenating them with a string, the other value will be converted to a string before concatenation.

3.2.2.3 List Variable Definition

LEARNING OBJECTIVES

K2 LO-3.2.2.3

Understand how to set and access data in list variables

List variables store multiple values and are defined using the at-syntax `@{variable_name}`. You can define as many values as needed, with each additional value separated by multiple spaces or line continuation using the `...` syntax.

Example:

```
*** Variables ***
@{NAMES}        Matti       Teppo
@{EMPTY_LIST}
@{NUMBERS}      one         two       three
...             four        five      six
```

Single values of list-like variables can be accessed by the dollar-syntax (`$`) followed by their index in square brackets (`[]`), starting with 0, like `${NAMES}[0]` for `Matti` and `${NAMES}[1]` for `Teppo`.

Example:

```
*** Test Cases ***
List Example
    Log    First Name: ${NAMES}[0]    # Logs 'First Name: Matti'
    Log    Second Name: ${NAMES}[1]   # Logs 'Second Name: Teppo'
```

3.2.2.4 Dictionary Variable Definition

LEARNING OBJECTIVES

K2 LO-3.2.2.4

Understand how to set and access data in dict variables

Dictionary variables store key-value pairs and use the ampersand-syntax `&{variable_name}`. Key-value pairs are assigned using the `key=value` format.

Example:

```
*** Variables ***
&{USER1}        name=Matti     address=xxx         phone=123
&{USER2}        name=Teppo     address=yyy         phone=456
&{COMBINED}     first=1        second=${2}         third=third
&{EMPTY_DICT}
```

You can escape equal signs in keys with a backslash (`\=`) to prevent misinterpretation.

Values of all dictionary-like variables can be accessed by the dollar-syntax (`$`) followed by the key in square brackets (`[]`), like `${USER1}[name]` for `Matti` and `${USER1}[address]` for `xxx`. No quotes are needed around the key name.

If dictionaries are created in Robot Framework by using the `&{}` syntax, they are **ordered**, which means they persist assignment order of the key-value pairs and can be iterated, and **support attribute access**, allowing to reference dictionary keys using syntax like `${USER1.name}`. Dictionaries or dictionary-like values can also be created by keywords and might have a different data type and therefore can not be accessed by attribute access.

Variables can also be used to set the accessed key dynamically by using the variable in the square brackets. Assuming `${key}` contains the value `phone`, `${USER1}[${key}]` would resolve to `123`.

3.2.3 Return values from Keywords

LEARNING OBJECTIVES

LO-3.2.3

Be able to assign return values from keywords to variables

In Robot Framework, values returned by keywords can be assigned to variables, enabling data to be passed between different keywords.

These variables have a **local scope** in the block where they are created, i.e., in the test|task or keyword where the assignment is made. If a variable has already been defined in the `*** Variables ***` section and therefore has a **suite scope**, it will just be locally overwritten/masked by the new variable with the same name. Once the block is left, the original variable with its original value is accessible again. See 5.1.2 Variable Scopes for more information.

An assignment is always constructed by the variable or variables that shall be assigned to, followed by an optional equal sign (`=`) and the keyword call that shall be executed and will return the value(s) to be assigned.

3.2.3.1 Assigning to Scalar Variables

In the simplest case, a keyword returns exactly one value, which can be assigned to a scalar variable using the dollar-syntax `${variable_name}`.

```
*** Settings ***
Library    OperatingSystem


*** Test Cases ***
Returning Example
    ${server_log} =    Get File    server.log
    Should Contain    ${server_log}    Successfully started
```

In this example, the content of the file `server.log`, which is returned by the `Get File` keyword, is stored in the `${server_log}` variable and later verified by the `Should Contain` keyword. Although the `=` sign is optional, its usage makes the assignment visually more explicit.

If keywords return multiple values, still the scalar variable syntax with `${var}` is used. All values are assigned to the variable as a list of values and can be accessed as described in the 3.2.2.3 List Variable Definition section.

```
*** Settings ***
Library    OperatingSystem


*** Test Cases ***
Returning a List Example
```

```
    ${files} =    List Files In Directory    server/logs
    Log    First File: ${files}[0]
    Log    Last File: ${files}[-1]
```

In cases where a keyword returns a defined number of values, they can be assigned to multiple scalar variables in one assignment. In the following example, the keyword `Split Path` returns two values, the path and the file name.

```
*** Settings ***
Library    OperatingSystem


*** Test Cases ***
Multiple Return Example
    ${path}    ${file} =    Split Path    server/logs/server.log
    Should Be Equal    ${path}    server/logs
    Should Be Equal    ${file}    server.log
```

3.2.4 `VAR` Statement

LEARNING OBJECTIVES

K2 LO-3.2.4

Understand how to create variables using the VAR statement

The `VAR` statement in Robot Framework is a way to create and assign values to variables directly within a test|task or keyword during execution. While the `*** Variables ***` section allows defining variables for a whole suite, the `VAR` statement is used within the body of a test|task or keyword, allowing more control over when and where the variable is created.

The `VAR` statement is case-sensitive and is followed by the variable name and an optional equal sign (`=`) and the value(s) to be assigned. The syntax is very similar to the `*** Variables ***` section. Scalar variables, lists, and dictionaries are created the same way and multiple values can also be assigned in multiple lines using the `...` syntax. Strings can be concatenated with the `separator=` syntax as well.

Example:

```
*** Test Cases ***
Test with VAR
    VAR    ${filename}    test.log
    ${file} =    Get File    ${filename}
    ${time} =    Get Time
    ${length} =    Get Length    ${file}
    VAR    &{file_info}
    ...    name=${filename}
    ...    content=${file}
    ...    time=${time}
    ...    length=${length}
    IF    $login == "matti"
        VAR    &{USER}    name=Matti    address=xxx    phone=123
    ELSE
        VAR    &{USER}    name=Teppo    address=yyy    phone=456
    END
```

Example use cases for the `VAR` statement:

- **Combining values during test|task execution**: Variables that shall have content based on information gathered during test|task execution.
- **Conditional assignments**: In some scenarios, it may be necessary to assign different values to a variable based on conditions that occur during test|task execution.
- **Initialization of variables**: In a FOR-loop (see 5.2.4 FOR Loops), it may be necessary to collect information and add it to a list. This list can be initialized with the `VAR` statement as an empty list before the loop starts and then filled with values during the loop.

By default, variables created with the `VAR` statement have a **local scope** in the test|task, or keyword where they are defined. This means that they cannot be accessed outside that specific test|task or keyword, ensuring that variables do not interfere with other parts of the test|task suite.

However, the `VAR` statement can also be used to create variables with a broader scope, using `scope=`, such as suite-wide or global variables, when needed. These variables can then be accessed outside of the test|task or keyword where they were originally created.

For more details on this topic, refer to the section on 5.1.2 Variable Scopes.

3.2.5 Variable Scope Introduction

LEARNING OBJECTIVES

K2 LO-3.2.5

Understand how `local` and `suite` scope variables are created

In Robot Framework, variables have different scopes, which define where they can be accessed and used. Understanding the scope of variables is crucial for managing data within tests and keywords.

- **`LOCAL` Scope**: Variables created within a test|task or keyword, by **assignment of return values**, as keyword arguments or **`VAR`** statement, are by default `LOCAL` to that specific test|task or keyword body.

  They cannot be accessed outside of that block and are destroyed once the block is completed. This means that a local variable created in one test|task can neither be accessed inside the body of a called keyword nor in a subsequent test|task or other keywords.

- **`SUITE` Scope**: Variables defined at the suite level, for example in the `*** Variables ***` section or through importing resource files, are available to all tests|tasks and keywords called within the suite.

  That means that they can be accessed inside a keyword, called from a test|task of that suite even, if this variable is not created as part of the argument interface of that keyword.

Examples and more details on variable scope, such as `TEST` and `GLOBAL` scope can be found in the 5.1.2 Variable Scopes section.

3.3 User Keyword Definition & Arguments

User Keywords in Robot Framework allow users to create their own keywords by combining existing keywords into reusable higher-level actions. They help improve readability, maintainability, and modularity in automation by abstracting complex sequences into named actions. User Keywords are defined syntactically very similarly to tests|tasks and are defined in the `*** Keywords ***` section of a suite file or resource file.

3.3.1 `*** Keywords ***` Section

The `*** Keywords ***` section of suite and resource files is indentation-based similar to the `*** Test Cases ***` section. The user keywords defined are unindented, while their body implementation is indented by multiple spaces.

See these sections for more details about 2.2 Basic Suite File Syntax and 2.6 Writing Test|Task and Calling Keywords.

This section can be part of suites or resource files. While keywords defined in suites can solely be used in the suite they are defined in, keywords defined in resource files can be used in any suite that imports these resource files.

Example definition of a user keyword:

```
*** Keywords ***
Verify Valid Login
    [Arguments]    ${exp_full_name}
    ${version}=    Get Server Version
    Should Not Be Empty    ${version}
    ${name}=    Get User Name
    Should Be Equal    ${name}    ${exp_full_name}
```

As a reference for how defined keywords are documented, see 2.5 Keyword Interface and Documentation.

3.3.2 User Keyword Names

LEARNING OBJECTIVES

LO-3.3.2

Recall the rules how keyword names are matched.

The names of User Keywords should be descriptive and clear, reflecting the purpose of the keyword. Well-named keywords make tests more readable and easier to understand. Robot Framework supports Unicode and allows the use of special characters and even Emojis in keyword names.

Keyword names are case-insensitive and can include single spaces. Also spaces and underscores will be ignored when matching keyword names. So the keywords `Login To System`, and `log_into_system` are considered identical.

To identify keywords that shall be executed, Robot Framework uses a matching algorithm that is case-insensitive and ignores spaces and underscores.

- If then a full match is found, that keyword is used.

- If no full match is found, the prefixes `Given`, `When`, `Then`, `And`, and `But` (case-insensitive), which are used in Behavior-Driven Specification style, are removed from the called keyword name to find a match.
- If still no match is found, Robot Framework tries to match the name with keywords that have embedded arguments.

By default, if not explicitly defined by the library developers, all Library Keywords are named in **Title Case** with capital letters at the beginning of each word, and spaces between words.

Project may choose a different naming convention for User Keywords, but it is recommended to be consistent across the project for User Keyword names.

They are defined without indentation, and the subsequent lines until the next unindented line are considered the body of the keyword. The following topics explain how to structure the body of a keyword.

3.3.3 User Keyword Settings

LEARNING OBJECTIVES

K1 LO-3.3.3

Recall all available settings and their purpose for User Keywords

User keywords can have similar settings as test cases, and they have the same square bracket syntax separating them from keyword calls. All available settings are listed below and explained in this section or in sections linked below.

- `[Documentation]` Used for setting user keyword documentation. (see 3.3.4 User Keyword Documentation)
- `[Arguments]` Specifies user keyword arguments to hand over values to the keyword. (see 3.3.5 User Keyword Arguments)
- `[Setup]`, `[Teardown]` Specify user keyword setup and teardown. (see 4.2 Teardowns (Suite, Test|Task, Keyword))
- `[Tags]` (*) Sets tags for the keyword, which can be used for filtering in documentation and attribution for post-processing results.
- `[Timeout]` (*) Sets the possible user keyword timeout.
- `[Return]` (*) Deprecated.

(*) The application of these settings are not part of this syllabus.

3.3.4 User Keyword Documentation

LEARNING OBJECTIVES

K1 LO-3.3.4

Recall the significance of the first logical line and in keyword documentation for the log file.

Each keyword can have a `[Documentation]` setting to provide a description of the keyword's purpose and usage.

The first logical line, until the first empty row, is used as the *short documentation* of the keyword in the `log.html` test protocol.

Proper documentation helps maintain clarity, especially in larger projects. It is a good practice to document what the keyword does, any important notes regarding its usage, and additional information about the arguments it accepts if not self-explanatory.

User keywords can be documented in the Robot Framework documentation format.

IMPORTANT

> The syntax of this format has similarities to Markdown, but is more limited and not compatible with Markdown!

This format includes:

- `*bold*` = **bold**
- `_italic_` = *italic*
- `_*bold italic*_` = ***bold italic***
- `` `code` `` = `code`
- Tables
- Lists
- Links
- Images
- Heading levels

3.3.5 User Keyword Arguments

LEARNING OBJECTIVES

K2 LO-3.3.5

Understand the purpose and syntax of the [Arguments] setting in User Keywords.

User Keywords can accept arguments, which make them more dynamic and reusable in various contexts. The `[Arguments]` setting is used to define the arguments a user keyword expects.

See also Chapter 2 2.5.2 Keyword Arguments for an introduction to argument kinds.

Arguments are defined by `[Arguments]` followed by the argument names separated by multiple spaces in the syntax of scalar variables.

Since Robot Framework 7.3 User Keywords can define argument types like `string`, `number`, etc., as described in the 2.5.2.8 Argument Types section.

3.3.5.1 Defining Mandatory Arguments

LEARNING OBJECTIVES

K1 LO-3.3.5.1-1

Recall what makes an argument mandatory in a user keyword.

K3 LO-3.3.5.1-2

Define User Keywords with mandatory arguments.

Arguments defined as scalar variable (`${arg}`) without a default value are mandatory and must be provided when calling the keyword.

Example that defines a keyword with two arguments:

```
*** Keywords ***
Verify File Contains
```

```
    [Documentation]    Verifies that a file contains a specific text.
    ...
    ...    The keyword opens the file specified by the file path and checks if it contains the expected
    [Arguments]    ${file_path}    ${expected_content}
    ${server_log} =    Get File    ${file_path}
    Should Contain    ${server_log}    ${expected_content}
```

All variables defined in the `[Arguments]` are local to the keyword body and do not exist outside of the keyword.

This keyword may be called in a test case like this:

```
*** Test Cases ***
Check Server Log
    Verify File Contains    server.log    Successfully started
```

In that case, the argument `${file_path}` is assigned the value `server.log`, and `${expected_content}` is assigned the value `Successfully started`.

3.3.5.2 Defining Optional Arguments

LEARNING OBJECTIVES

K1 LO-3.3.5.2-1

Recall how to define optional arguments in a user keyword.

K3 LO-3.3.5.2-2

Define User Keywords with optional arguments.

Optional arguments are defined by assigning default values to them in the `[Arguments]` setting. All optional arguments must be defined after all mandatory arguments.

Default values are assigned using an equal sign (`=`), followed by the default value without any spaces, such as `${ignore_case}=True`, which would set the string `True` as default.

The assigned default values can also include previously defined variables, such as `${ignore_case}=${True}`, where `${True}` represents the boolean value `True`.

Example:

```
*** Keywords ***
Verify File Contains
    [Documentation]    Verifies that a file contains a specific text.
    ...
    ...    The keyword opens the file specified by the ``file_path``
    ...    and checks if it contains the ``expected_content``.
    ...
    ...    By default, the verification is case-insensitive
    ...    but can be changed with the optional argument ``ignore_case``.
    [Arguments]    ${file_path}    ${expected_content}    ${encoding}=utf-8    ${ignore_case}=${True}
    ${server_log} =    Get File    ${file_path}    ${encoding}
    Should Contain    ${server_log}    ${expected_content}    ignore_case=${ignore_case}
```

3.3.5.3 Defining Embedded Arguments

LEARNING OBJECTIVES

K2 LO-3.3.5.3-1

Describe how embedded arguments are replaced by actual values during keyword execution.

K2 LO-3.3.5.3-2

Understand the role of embedded arguments in Behavior-Driven Development (BDD) style.

In Robot Framework, **embedded arguments** allow the inclusion of arguments directly within the keyword name itself. This approach is particularly useful for creating **Behavior-Driven Development (BDD)**-style test cases or for making keyword names more readable and meaningful.

With embedded arguments, placeholders are used within the keyword name, which are replaced by actual values when the keyword is executed. These arguments are written as scalar variables with dollar signs and curly braces, as shown in the following example:

```
*** Keywords ***
The file '${file_name}' should contain '${expected_content}'
    ${file_content} =    Get File    ${file_name}
    Should Contain    ${file_content}    ${expected_content}
```

When this keyword is called, the placeholders `${file_name}` and `${expected_content}` are replaced by the actual values provided in the keyword call. For instance, in the following example, `${file_name}` is replaced with `server.log` and `${expected_content}` with `Successfully started`:

```
*** Test Cases ***
Test File Content
    Given the server log level is 'INFO'
    When the server is started successfully
    Then the file 'server.log' should contain 'Successfully started'
```

Quotes around the embedded arguments are treated as regular characters within the keyword name but can improve readability and help distinguish embedded arguments from the rest of the keyword name.

Embedded arguments can become problematic when the keyword name becomes overly long or complicated. To address this, a mix of embedded arguments and regular arguments can be used. This approach can help manage more complex data structures and enhance readability.

Example of mixed embedded and regular arguments:

```
*** Test Cases ***
Embedded and normal arguments
    Given the user is on the pet selection page
    When the user adds    2    cat fish
    And the user sets    3    dogs
    And the user removes    1    dogs
    Then the number of cat fish should be    2
    And the number of dogs should be    count=2

*** Keywords ***
the user is on the pet selection page
```

```
    Open Pet Selection Page

the number of ${animals} should be
    [Arguments]    ${count}
    ${current_count}    Get Animal Count    ${animals}
    Should Be Equal As Numbers    ${current_count}    ${count}

the user ${action}
    [Arguments]    ${amount}    ${animal}
    IF    '${action}' == 'adds'
        Add Items To List    animal_list    ${animal}    ${amount}
    ELSE IF    '${action}' == 'removes'
        Remove Items From List    animal_list    ${animal}    ${amount}
    ELSE IF    '${action}' == 'sets'
        Set Amount To List    animal_list    ${animal}    ${amount}
    ELSE
        Skip    Test skipped due to invalid action
    END
```

3.3.5.4 Other Argument Kinds

Other argument kinds like **Named-Only Arguments**, **Free Named Arguments**, or **Variable Number of Positional Arguments** should be known, but their definition and usage are not part of this syllabus.

3.3.6 RETURN Statement

LEARNING OBJECTIVES

K2 LO-3.3.6-1

Understand how the `RETURN` statement passes data between different keywords.

K3 LO-3.3.6-2

Use the `RETURN` statement to return values from a user keyword and assign it to a variable.

The `RETURN` statement (case-sensitive) in Robot Framework is used to return values from a User Keyword to be used in further test steps or stored in variables. This allows test execution to pass data between different keywords.

It can return one or more values. If more than one value is returned, they can either be assigned to multiple variables or stored as a list in a single variable.

Example:

```
*** Keywords ***
Get File Name From Path
    [Arguments]    ${file_path}
    ${path}    ${file} =    Split Path    ${file_path}
    RETURN    ${file}
```

The `RETURN` statement is normally used at the end of a keyword definition, because it will end the keyword execution at that point and return to the caller. However, this behavior can be used to conditionally end a keyword execution early together with an `IF` or `TRY-EXCEPT` statement.

IMPORTANT

The `RETURN` statement of a keyword cannot return the returned value from a called keyword directly like in other programming languages. The return value must be stored in a variable first and then be returned by the `RETURN` statement. So the first keyword is **invalid** while the second **valid** is!

invalid

```
*** Keywords ***
Get ISO Time
    RETURN    Evaluate    datetime.datetime.now().isoformat()
```

valid

```
*** Keywords ***
Get ISO Time
    ${time}    Evaluate    datetime.datetime.now().isoformat()
    RETURN    ${time}
```

3.3.7 Keyword Conventions

LEARNING OBJECTIVES

K1 LO-3.3.7

Recall the naming conventions for user keywords.

When defining User Keywords, it is recommended to follow conventions to ensure consistency and readability across the project. These may be taken from community best practices or defined within the project team.

Keyword Conventions should contain agreements on:

- **Naming Case**: Which case shall be used? (e.g. `Title Case`, `camelCase`, `snake_case`, `kebab-case`, or `Sentence case`, etc.) (from a readability perspective, `Title Case` or `Sentence case` are recommended)
- **Grammatical Form/Mood**: Which form shall be used for actions and verifications/assertions? (e.g. `Imperative` for both like `Click Button`, `Verify Text`. Or e.g. `Declarative`/`Indicative` for assertions like `Text Should Be`, `Element Should Be Visible`)
- **Word/Character Count**: How many words or characters shall be used in a keyword name? (e.g. less than 7 words)
- **Argument Count**: How many arguments shall a keyword have? (e.g. less than 5)
- **Documentation**: How shall the documentation be structured and which information shall be included or is it required at all?

3.4 Using Data-Driven Specification

LEARNING OBJECTIVES

K2 LO-3.4

Understand the basic concept and syntax of Data-Driven Specification

The **Data-Driven Specification** style in Robot Framework separates test|task logic from data, enabling tests|tasks to be executed with multiple data sets efficiently. This approach involves using a single higher-level keyword to represent the entire workflow, while the test data is defined as rows of input and expected output values.

3.4.1 Test|Task Templates

LEARNING OBJECTIVES

K2 LO-3.4.1-1

Understand how to define and use test|task templates

K1 LO-3.4.1-2

Recall the differences between the two different approaches to define Data-Driven Specification

For each test|task, a template keyword can be defined that contains the workflow logic.

At the suite level, the `Test Template` or `Task Template` setting can be used to specify that keyword. All tests|tasks in the suite will reuse this keyword for execution with different data sets.

Alternatively, the `[Template]` setting can be used at the test|task level. The tests|tasks would not have any other keyword calls but would instead define the data rows to be passed to the template keyword.

`Test Setup`|`Test Teardown` and `Task Setup`|`Task Teardown` can be used together with templates.

3.4.1.1 Multiple Named Test|Task With One Template

LEARNING OBJECTIVES

K1 LO-3.4.1.1

Recall the syntax and properties of multiple named test|task with one template

The following example has six different test|task, each with different name and different data sets, all using the `Login With Invalid Credentials Should Fail` keyword template.

```
*** Settings ***
Test Setup       Open Login Page
Test Template    Login With Invalid Credentials Should Fail
Test Teardown    Close Page

*** Test Cases ***                 USERNAME         PASSWORD
Invalid User Name                  invalid          ${VALID PASSWORD}
```

```
Invalid Password                    ${VALID USER}     invalid
Invalid User Name and Password      invalid           invalid
Empty User Name and Password        ${EMPTY}          ${EMPTY}
    [Tags]    Empty
Empty User Name                     ${EMPTY}          ${VALID PASSWORD}
Empty Password                      ${VALID USER}     ${EMPTY}
```

The advantage of this approach is that each test|task is executed separately with its own name and data set. Each test|task appears in the statistics and reports. Single tests|tasks can be filtered and re-executed or tagged, like the test case `Empty User Name and Password`.

It is possible to add header names to the data columns in the line of `*** Test Cases ***` or `*** Tasks ***` to describe the data columns to improve readability.

3.4.1.2 Named Test|Task With Multiple Data Rows:

LEARNING OBJECTIVES

LO-3.4.1.2

Recall the syntax and properties of named test|task with multiple data rows

A slightly different approach is to define multiple data rows for a single test|task.

This is still possible with a single template defined in the `*** Settings ***` section, but in this case it would also make sense to define the template locally for each test|task with the `[Template]` setting. With this approach, it is possible to define different scenarios in the same suite file, which can be useful for testing different aspects of the same functionality.

```
*** Test Cases ***
Invalid Logins
    [Template]    Login With Invalid Credentials Should Fail
    invalid          ${VALID PASSWORD}
    ${VALID USER}    invalid
    invalid          whatever
    ${EMPTY}         ${VALID PASSWORD}
    ${VALID USER}    ${EMPTY}
    ${EMPTY}         ${EMPTY}

Valid Logins
    [Template]    Login With Valid Credentials Should Pass
    ${VALID USER}             ${VALID PASSWORD}
    ${VALID LONG USER}        ${VALID LONG PASSWORD}
    ${VALID COMPLEX USER}     ${VALID COMPLEX PASSWORD}
```

If one data row fails, this template execution is marked FAIL and the test|task is marked FAIL, but **the other data rows are still executed**.

This approach creates only a single tests|tasks for multiple data rows in the logs and reports, which can be beneficial statistically.

However, this approach has also its drawbacks:

- Test|task setup and teardown are executed only once for all data rows of one test|task. If there is a setup and teardown needed for each data row, a keyword setup or teardown is needed.
- The test|task name is not unique for each data row, which can make it harder to understand the failing data row in the logs.
- Filtering and re-execution of some or single data rows is not possible.

3.5 Advanced Importing of Keywords and Naming Conflicts

LEARNING OBJECTIVES

K1 LO-3.5

Recall that naming conflicts can arise from the import of multiple resource files.

As stated before, it is possible to organize imports and available keywords in Robot Framework by using Resource Files. By default, all keywords or variables created or imported in a resource file are available to those suites and files that are importing that higher-level resource file.

This can lead to complex import hierarchies or the importing of libraries multiple times, which should be avoided.

Due to this mechanism, the number of keywords available to a suite can be quite large, and naming conflicts, especially with keywords from third-party keyword libraries, can occur. These conflicts need to be resolved.

Some keyword libraries have the option to be configured to change their behavior, which may also change the available keywords they offer.

3.5.1 Importing Hierarchies

LEARNING OBJECTIVES

K2 LO-3.5.1

Understand how transitive imports of resource files and libraries work.

Let's assume the following libraries and resource files shall be used:

- **Library** `A`
- **Library** `B`
- **Library** `Operating System`
- **Resource** `tech_keywordsA.resource`
- **Resource** `tech_keywordsB.resource`
- **Resource** `variables.resource`
- **Resource** `functional_keywords.resource`

The respective files could look like this:

tech_keywordsA.resource:

```
*** Settings ***
Library    A
Library    Operating System
```

tech_keywordsB.resource:

```
*** Settings ***
Library    B
Resource    variables.resource
```

functional_keywords.resource:

```
*** Settings ***
Resource    tech_keywordsA.resource
Resource    tech_keywordsB.resource
```

suite.robot:

```
*** Settings ***
Resource    functional_keywords.resource
```

In this case, the suite `suite.robot` has access to all keywords from all keyword libraries, as well as all variables and user keywords from all resource files. With this transitive importing it is possible to organize user keywords and imports of libraries in a hierarchical way.

It shall be avoided to create circular imports, where `A.resource` imports `B.resource` and `B.resource` imports `A.resource`.

It should be avoided to import the same library in different places multiple times. If the exact same library with the same configuration (see the next section) is imported again, it will be ignored because Robot Framework already has it in its catalog. However, if the library is imported with different configurations, it may be imported multiple times, but depending on the library's internal behavior, the new configuration may have no effect on the existing keywords, or other side effects may occur.

Therefore, the recommendation is to import libraries only in one resource file with one configuration and use that import file in all places where the library is needed to make its keywords available.

3.5.2 Library Configuration

LEARNING OBJECTIVES

LO-3.5.2

Be able to configure a library import using arguments.

Some libraries offer or need additional configuration to change their behavior or make them work. This is typically global behavior like internal timeouts, connection settings to systems, or plugins that should be used.

If this is possible, the library documentation will have an `Importing` section directly before the list of keywords.

Library importing arguments are used in the same way as keyword calls with arguments. If possible, it is recommended to set the arguments as named arguments to make usage more readable and future-proof. These arguments follow the Library path or name, separated by multiple spaces.

Example with the Telnet library:

```
*** Settings ***
Library    Telnet    newline=LF    encoding=ISO-8859-1    # set newline and encoding using named argumen
```

Another example that cannot be used without configuration is the Remote library. Remote libraries are libraries that are connected remotely via a network connection. So the actual library is running as a server, and the library `Remote` is connecting as a client and connects the keywords of the server to Robot Framework. Therefore, it needs the server's address and port to connect to. Because there may be more than one Remote Library, we need to define the used library name as well.

```
*** Settings ***
Library     Remote     uri=http://127.0.0.1:8270        AS     EmbeddedAPI
Library     Remote     uri=http://remote.devices.local:8270        AS     DeviceAPI
```

In this example, two remote libraries are imported. The upper-case `AS` statement is used to define the name of the library that shall be used in the suite.

They are now available as `EmbeddedAPI` and `DeviceAPI` in the suite.

3.5.3 Naming Conflicts

LEARNING OBJECTIVES

K2 LO-3.5.3

Explain how naming conflicts can happen and how to mitigate them.

Naming conflicts can occur when two or more keywords have the same name. If a proper IDE is used, that can be detected, and users can be warned after they have created a duplicate user keyword name.

Project teams may not have this influence over imported third-party libraries that have the same keyword names. Due to the fact that keywords from library and resource files are imported in the scope of the importing suite, it may be unavoidable to have naming conflicts.

One example of these kinds of conflicts is the two libraries `Telnet` and `SSHLibrary`, which at the current time both have multiple keywords with the same name. This is because they both work with network connections and have similar functionality. Keywords like `Open Connection`, `Login`, `Read`, `Close Connection`, and many more are common.

These conflicts cannot be resolved by Robot Framework if they are coming from the same kind of source, like two libraries. The error message will be like this:

```
Multiple keywords with name 'Open Connection' found. Give the full name of the keyword you want to use:
    SSHLibrary.Open Connection
    Telnet.Open Connection
```

As proposed by Robot Framework, to resolve naming conflicts, the easiest way to mitigate this is to use the full names of the keywords, including the library name, when calling them.

Example:

```
*** Test Cases ***
Using Telnet and SSHLibrary
    Telnet.Open Connection
    Telnet.Login    ${username}    ${password}
    ${telnet_init} =    Telnet.Read Until Prompt
    Telnet.Close Connection

    SSHLibrary.Open Connection    ${host}    ${port}
```

```
    SSHLibrary.Login    ${username}    ${password}
    ${ssh_init} =    SSHLibrary.Read Until Prompt
    SSHLibrary.Close Connection
```

When using full names for libraries that were imported with the `AS` statement, the name of the library is used as a prefix to the keyword name.

```
*** Test Cases ***
Using Remote Libraries
    EmbeddedAPI.Close Contact   15
    DeviceAPI.Verify Contact    15    1
```

4 Advanced Structuring and Execution

As a Robot Framework automation project expands, the increasing number of tests|tasks adds complexity to the project. This chapter explores advanced structuring and execution techniques to effectively manage this complexity and control the execution flow.

We will cover methods for error handling and cleaning up after failed tests|tasks using **Teardowns**, as well as preparing individual or multiple suites and tests|tasks for execution with **Setups**. Additionally, filtering subsets of tests|tasks based on tags will be discussed, which is essential for managing test|task execution efficiently.

4.1 Setups (Suite, Test|Task, Keyword)

LEARNING OBJECTIVES

K1 LO-4.1-1

Recall the purpose and benefits of Setups in Robot Framework

K1 LO-4.1-2

Recall the different levels where a Setup can be defined

Setups in Robot Framework are used to prepare the environment or system for execution or to verify that the requirements/preconditions needed for execution are met. They can be defined at the suite, test|task, or keyword level and are executed before the respective scope begins execution.

A **Setup** is a single keyword with potential argument values that is called before all other keywords; or before tests|tasks in Suite Setup.

Examples of typical use cases for Setups are:

- Establishing connections to databases or services.
- Initializing test data or configurations.
- Setting the system under test to a known state.
- Logging into applications or systems.
- Navigating to the feature under test.

4.1.1 Suite Setup

LEARNING OBJECTIVES

K1 LO-4.1.1-1

Recall key characteristics, benefits, and syntax of Suite Setup

K2 LO-4.1.1-2

Understand when Suite Setup is executed and used

A **Suite Setup** is executed before any tests|tasks or child suites within the suite are run. It is used to prepare the environment or perform actions that need to occur before the entire suite runs. Since it is only executed once before all tests|tasks or child suites, it can save time, rather than executing the action for each test|task individually.

Key characteristics of Suite Setup:

- Suite Setup is a single keyword call with potential argument values.
- Executed before any tests|tasks and child suites in the suite.
- If the Suite Setup fails, all tests|tasks in the suite and its child suites are marked as failed, and they are not executed.
- Logged in the execution log as a separate section, indicating the setup status.

Typical use cases:

- Ideal for checking **preconditions** that must be met before running the tests|tasks.
- Ensuring that the environment is ready for execution.
- Starting services or applications required for the suite.
- Preparing a system under automation to meet the suite's requirements.
- Loading configurations or resources shared across multiple tests|tasks.

Example of defining a Suite Setup:

```
*** Settings ***
Suite Setup    Initialize Environment    dataset=Config_C3
```

4.1.2 Test|Task Setup

LEARNING OBJECTIVES

K1 LO-4.1.2-1

Recall key characteristics, benefits, and syntax of Test Setup

K2 LO-4.1.2-2

Understand when Test|Task Setup is executed and used

A **Test|Task Setup** is executed before a single test|task runs. It is used to prepare the specific conditions required for that test|task.

You can define a default Test|Task Setup in the `*** Settings ***` section of the suite using the `Test Setup`|`Task Setup` setting. This setup will be applied to all tests|tasks within the suite unless overridden and executed before each test|task.

Individual tests|tasks can override the default setup by specifying their own `[Setup]` setting within the test|task. To disable the setup for a specific test|task, you can set `[Setup]  NONE`, which means that no setup will be executed for that test|task.

Key characteristics of Test|Task Setup:

- Test|Task Setup is a single keyword call with potential argument values.
- Executed before the test|task starts.
- If the Test|Task Setup fails, the test|task is marked as failed, and its body, including its main keywords, is not executed.
- Can be set globally for all tests|tasks in a suite and overridden locally.
- Logged in the execution log as a separate section, indicating the setup status.

Typical use cases:

- Setting up data unique to the test|task.
- Executing preparation steps to navigate to the automated task or feature under test.
- Distinguishing phases of a test|task in *setup* (aka *preparation* or *precondition checking*), *steps*, and *teardown* (aka *clean up* or *postconditions*).

Example of defining a default Test|Task Setup in the suite settings and overriding it on a test case:

```
*** Settings ***
Test Setup    Login As Standard User


*** Test Cases ***
User Action Test With Default Setup    # Default Test Setup is applied
    Perform User Actions    0815

Another User Action With Default Setup    # Default Test Setup is applied
    Perform another User Action    4711

Admin Access Test With Local Setup
    [Setup]    Login As Admin    # Override the default setup
    Perform Admin Actions    007

No Setup Test
    [Setup]    NONE    # Override and disable the setup by case-sensitive NONE
    Perform Actions Without Login    000
```

4.1.3 Keyword Setup

LEARNING OBJECTIVES

K1 LO-4.1.3

Recall key characteristics and syntax of Keyword Setup

A **Keyword Setup** is executed before the body of a user keyword is executed. It allows for preparation steps specific to that keyword or ensures that the keyword's requirements are met before execution.

Key characteristics of Keyword Setup:

- Keyword Setup is a single keyword call with potential argument values.
- Executed before the keyword's body.
- If the Keyword Setup fails, the keyword's body is not executed.
- Logged in the execution log as a separate section, indicating the setup status.

Typical use cases:

- Opening connections or files needed by the keyword.
- Initializing variables or data structures.
- Ensuring preconditions specific to the keyword are met.

Example of defining a Keyword Setup:

```
*** Keywords ***
Process Data
    [Setup]    Open Data Connection
    Process the Data
```

4.2 Teardowns (Suite, Test|Task, Keyword)

LEARNING OBJECTIVES

K2 LO-4.2-1

Understand the different levels where and how Teardowns can be defined and when they are executed

K1 LO-4.2-2

Recall the typical use cases for using Teardowns

In automation, tests|tasks are typically executed in a linear sequence. This linear execution can lead to issues when a preceding test|task fails, potentially affecting subsequent tests|tasks due to an unclean state of the system under test or the automated environment. To prevent such issues, Robot Framework provides the **Teardown** functionality, which can be defined at the suite, test|task, or keyword level.

As mentioned before, a failure resulting in a keyword with the status `FAIL` will cause Robot Framework not to execute all subsequent keywords of the current test|task. These not-executed keywords will receive the status `NOT RUN`.

A **Teardown** is a single keyword call with potential argument values that is executed after the child suites, test|tasks, and keywords have completed execution, regardless of the outcome, even if previously executed elements have failed. It ensures that necessary cleanup actions are performed, maintaining the integrity of the environment for subsequent executions.

Typical use cases for Teardowns include:

- Cleaning up the system under test after a test|task has been executed.
- Closing connections to databases, files, or other resources.
- Resetting the system under test to a known state.
- Closing user sessions or logging out users.

By utilizing teardowns effectively, you can ensure that each test|task starts with a clean state, reducing dependencies between tests|tasks and improving the reliability of your automation project.

4.2.1 Suite Teardown

LEARNING OBJECTIVES

K1 LO-4.2.1-1

Recall key characteristics, benefits, and syntax of Suite Teardown

K2 LO-4.2.1-2

Understand when Suite Teardown is executed and used

A **Suite Teardown** is executed after all tests|tasks and all child suites in a suite have been executed.

The Suite Teardown is executed regardless of the outcome of the tests|tasks within the suite, even if the suite setup fails.

Key characteristics of Suite Teardown:

- Suite Teardown is a single keyword call with potential argument values.
- Executed after all tests|tasks and child suites have completed.
- Runs even if the Suite Setup fails or any test|task within the suite fails.
- If the Suite Teardown fails, all tests|tasks in the suite are marked as failed in reports and logs.
- All keywords within the Suite Teardown are executed, even if one of them fails, ensuring all cleanup actions are attempted.

Typical use cases:

- Cleaning up the environment after all test|task executions.
- Performing actions that need to occur after the entire suite has finished running.

Example of defining a Suite Teardown:

```
*** Settings ***
Suite Teardown    Close All Resources    force=True
```

4.2.2 Test|Task Teardown

LEARNING OBJECTIVES

K1 LO-4.2.2-1

Recall key characteristics, benefits, and syntax of Test|Task Teardown

K2 LO-4.2.2-2

Understand when Test|Task Teardown is executed and used

A **Test|Task Teardown** is executed after a single test|task body has been executed. It is used for cleaning up actions specific to that test|task. The Test|Task Teardown is executed regardless of the test|task's outcome, even if the test|task's setup fails.

In Robot Framework, you can define a default Test|Task Teardown in the `*** Settings ***` section of the suite using the `Test Teardown`|`Task Teardown` setting. This default teardown will be applied to all tests|tasks within the suite unless overridden.

Individual tests|tasks can override the default teardown by specifying their own `[Teardown]` setting within the test|task. If you want to disable the teardown for a specific test|task, you can set `[Teardown]  NONE`, which effectively means that no teardown will be executed for that test|task.

It is recommended to define the local `[Teardown]` setting as the last line of the test|task.

Key characteristics of Test|Task Teardown:

- Test|Task Teardown is a single keyword call with potential argument values.
- Executed after the test|task has been executed, regardless of its status.
- Runs even if the Test|Task Setup fails.
- If the Test|Task Teardown fails, the test|task is marked as failed in reports and logs.
- All keywords within the Test|Task Teardown are executed, even if one of them fails.
- Can be set globally for all tests|tasks in a suite and overridden locally.

Typical use cases:

- Logging out of an application after a test|task completes.
- Deleting test data created during the test|task.
- Restoring configurations altered during the test|task.
- Distinguishing phases of a test|task in *setup* (aka *preparation* or *precondition checking*), *steps*, and *teardown* (aka *clean up* or *postconditions*).

Example of defining a default Test|Task Teardown in the suite settings:

```
*** Settings ***
Test Teardown    Logout User    # Default Teardown for all tests


*** Test Cases ***
Test with Default Teardown    # Default Teardown is applied
    Login User
    Do Some Testing

Another Test with Default Teardown    # Default Teardown is applied
    Login User
    Do Some other Testing

Custom Teardown Test
    Perform Test Steps
    [Teardown]    Cleanup Specific Data    # Override the default teardown

No Teardown Test
    Perform Other Steps
    [Teardown]    NONE    # Override and disable the teardown by case-sensitive NONE
```

4.2.3 Keyword Teardown

LEARNING OBJECTIVES

K1 LO-4.2.3

Recall key characteristics, benefits, and syntax of Keyword Teardown

A **Keyword Teardown** is executed after a user keyword body has been executed. It allows for cleanup actions specific to that keyword, ensuring that any resources used within the keyword are properly released independently of failed child keyword calls.

For better readability, it should be written as the last line of a keyword.

Key characteristics of Keyword Teardown:

- Keyword Teardown is a single keyword call with potential argument values.
- Executed after the keyword body has been executed, regardless of its status.
- Runs even if the keyword's setup fails.
- All keywords within the Keyword Teardown are executed, even if one of them fails.

Typical use cases:

- Closing temporary files or connections opened within the keyword.
- Resetting variables or states altered during keyword execution.
- Logging additional information after keyword execution.

Example of defining a Keyword Teardown:

```
*** Keywords ***
Process Data
    Open Data Connection
    Process the Data
    [Teardown]    Close Data Connection
```

4.3 Initialization Files

LEARNING OBJECTIVES

LO-4.3

Recall how to define Initialization Files and its purpose

As Robot Framework automation projects grow, organizing tests|tasks into directories becomes essential for managing complexity and maintaining a clear structure. When suites are created from directories, these directories can contain multiple suites and tests|tasks, forming a hierarchical suite structure. However, directories alone cannot hold suite-level settings or information. To address this, Robot Framework uses **initialization files**, which allow you to define suite-level settings for directories.

An **initialization file** is a file named `__init__.robot` placed inside a directory that acts as a suite. This file can contain suite-level settings that apply to the directory suite.

4.3.1 Purpose of Initialization Files

Initialization files enable you to:

- Define `Suite Setup` and `Suite Teardown` keywords for the directory suite.
- Set the name of the suite with the `Name` setting if it should be different from the directory name.
- Specify suite-level settings such as `Documentation` and `Metadata`.
- Set default `Test Setup`, `Test Teardown`, `Test Tags`, and `Test Timeout` for all tests|tasks within the directory (these can be overridden/extended in lower-level suites or tests|tasks).

4.3.2 Suite Setup and Suite Teardown of Initialization Files

LEARNING OBJECTIVES

LO-4.3.2

Understand the execution order of Suite Setup and Suite Teardown in Initialization Files and their sub-suites and tests|tasks

As previously explained, **Suite Setup** and **Suite Teardown** are used to prepare and clean up the environment before and after a suite's execution. Initialization files provide a centralized place to define these setups and teardowns for all sub-suites and their tests|tasks within a directory structure. Thus, it is possible to define one Suite Setup that is executed at the very start of the execution before any other Suite Setup, Test|Task Setup, and Test|Task is executed. The Suite Teardown of an initialization file is executed after all sub-suites in the directory and their tests|tasks have been completed.

4.3.3 Allowed Sections in Initialization Files

LEARNING OBJECTIVES

> LO-4.3.3
>
> Recall the allowed sections and their content in Initialization Files

Initialization files have the same structure and syntax as regular suite files but with some limitations. The following sections are allowed in initialization files:

- `*** Settings ***` **Section (required)**:
  - `Name`: Set a custom name for the suite directory.
  - `Documentation`: Provide documentation for the suite.
  - `Metadata`: Add metadata to the suite.
  - `Suite Setup`: Define a keyword to be executed before any tests|tasks or child suites.
  - `Suite Teardown`: Define a keyword to be executed after all tests|tasks and child suites have completed.
  - `Test Setup`|`Task Setup`: Set a default setup keyword for all tests|tasks in the suite (can be overridden in lower-level suites or tests|tasks).
  - `Test Teardown`|`Task Teardown`: Set a default teardown keyword for all tests|tasks in the suite (can be overridden in lower-level suites or tests|tasks).
  - `Test Timeout`|`Task Timeout`: Define a default timeout for all tests|tasks in the suite (can be overridden in lower-level suites or tests|tasks).
  - `Test Tags`|`Task Tags`: Assign tags to all tests|tasks in the suite (applied recursively to all lower-level suites and tests|tasks and can be extended or reduced there).
  - `Library`, `Resource`, `Variables`: Import necessary libraries, resource files, or variable files.
  - `Keyword Tags`: Assign tags to all keywords in the local `*** Keywords ***` section.
- `*** Variables ***` **Section (optional)**:

  Define variables that are available to the initialization file.
- `*** Keywords ***` **Section (optional)**:

  Define keywords that are available to the initialization file for Suite Setup, Suite Teardown, Test Setup, or Test Teardown.
- `*** Comments ***` **Section (optional)**:

  Add comments to the initialization file.

Important Note: Variables and keywords defined or imported in the initialization file are **not** available to lower-level suites or tests|tasks. They are local to the initialization file itself. To share variables or keywords across multiple suites or tests|tasks, use resource files and import them where needed.

4.3.4 Example of an Initialization File

```
*** Settings ***
Documentation     Initialization file for the Sample Suite
Suite Setup       Initialize Environment
Suite Teardown    Cleanup Environment


*** Variables ***
${BASE_URL}       http://example.com
```

```
*** Keywords ***
Initialize Environment
    Start Server
    Set Base URL            ${BASE_URL}
    Import Dataset          ${BASE_URL}/imports    dataset=Config_C3
    Verify Server Status    ${BASE_URL}   status=OK

Cleanup Environment
    Reset Database
    Stop Server
```

4.4 Test|Task Tags and Filtering Execution

LEARNING OBJECTIVES

K1 LO-4.4

Recall the purpose of Test|Task Tags in Robot Framework

In Robot Framework, **tags** offer a simple yet powerful mechanism for classifying and controlling the execution of tests|tasks. Tags are free-form text labels that can be assigned to tests|tasks to provide metadata, enable flexible test selection, and organize test results.

Tags are also used to create a statistical summary of the test|task results in the execution protocols.

Important Note: Tags are case-insensitive in Robot Framework, but the first appearance of a tag in a test|task is used as the tag name in reports and logs in its current case.

4.4.1 Assigning Tags to Tests|Tasks

LEARNING OBJECTIVES

K1 LO-4.4.1

Recall the syntax and different ways to assign tags to tests|tasks

Tags can be assigned to tests|tasks in several ways:

1. **At the Suite Level** using the `Test Tags` setting in the `*** Settings ***` section or in an initialization file (`__init__.robot`). This assigns tags to all tests|tasks within the suite:

   ```
   *** Settings ***
   Test Tags    smoke    regression
   ```

   This will assign the tags `smoke` and `regression` to all tests|tasks in the suite.

2. **At the Test|Task Level** using the `[Tags]` setting within individual tests|tasks. These tags are added in addition to any suite-level tags:

   ```
   *** Test Cases ***
   Valid Login Test|Task
       [Tags]    login    critical    -smoke
       Perform Login Steps
   ```

   This test|task will have the tags `login`, `critical`, and any tags assigned at the suite level, except `smoke`. Adding a minus sign (`-`) before a tag removes it from the test|task's tags.

3. **Using Variables** in tags to dynamically assign tag values:

   ```
   *** Variables ***
   ${ENV}    production
   ```

```
*** Test Cases ***
Data Processing Test|Task
    [Tags]    environment:${ENV}
    Process Data
```

This test|task will have a tag `environment:production`.

4. **By Keyword** `Set Tags` **or** `Remove Tags` to dynamically assign or remove tags during test|task execution:

   See BuiltIn library documentation for more information.

4.4.2 Using Tags to Filter Execution

LEARNING OBJECTIVES

K2 LO-4.4.2

Understand how to filter tests|tasks using the command-line interface of Robot Framework

Tags can be used to select which tests|tasks are executed or skipped when running a suite. This is accomplished using command-line options when executing Robot Framework.

While tags are case-insensitive you should always use the lowercase version of the tag when filtering for tests|tasks with logical operators because logical operators are case-sensitive and uppercase. `AND`, `OR`, and `NOT` are the logical operators that can be used to combine tags in the filtering, but **they are not part of this syllabus!**

4.4.2.1 Including Tests|Tasks by Tags

To include only tests|tasks that have a specific tag, use the `--include` (or `-i`) option followed by the tag name:

```
robot --include smoke path/to/tests
```

This command will execute only the tests|tasks that have the `smoke` tag.

4.4.2.2 Excluding Tests|Tasks by Tags

To exclude tests|tasks that have a specific tag, use the `--exclude` (or `-e`) option followed by the tag name:

```
robot --exclude slow path/to/tests
```

This command will execute all tests|tasks except those that have the `slow` tag. The excluded tests|tasks will not be executed or logged at all. Use `--skip` to not execute tests|tasks but include them in the logs as skipped. See 4.5.1 Skipping By Tags Selection (CLI) for more information.

4.4.2.3 Combining Include and Exclude Options

You can combine `--include` and `--exclude` options to fine-tune which tests|tasks are executed:

```
robot --include regression --exclude unstable path/to/tests
```

This command will execute tests|tasks that have the `regression` tag but exclude any that also have the `unstable` tag.

4.4.2.4 Using Tag Patterns

Tags can include patterns using wildcards `*` and `?` to match multiple tags:

- `*` matches any number of characters.
- `?` matches any single character.

Examples:

- Include tests|tasks with tags starting with `feature-`:

  ```
  robot --include feature-* path/to/tests
  ```

- Exclude tests|tasks with tags ending with `-deprecated`:

  ```
  robot --exclude *-deprecated path/to/tests
  ```

4.4.3 Reserved Tags

Tags starting with `robot:` are reserved for internal use by Robot Framework and should not be used in user-defined tags. Using own tags with this prefix may lead to unexpected behavior in test execution and reporting.

- `robot:exclude`: Marks tests|tasks that should be excluded from execution similar to `--exclude`.
- `robot:skip`: Marks tests|tasks that should be skipped during execution similar to `--skip`.

4.5 SKIP Test|Task Status

LEARNING OBJECTIVES

K1 LO-4.5-1

Recall the use case and purpose of skipping tests|tasks in Robot Framework

K1 LO-4.5-2

Recall the different ways to skip tests|tasks in Robot Framework

In addition to `PASS` and `FAIL`, Robot Framework introduces the `SKIP` status to indicate that a test|task was explicitly skipped **during** execution. The `SKIP` status is useful when certain tests|tasks should not be executed, for example, due to unfulfilled preconditions, incomplete test logic, or unsupported environments. Skipped tests|tasks appear in logs and reports, clearly marked as skipped.

Reasons to Use SKIP

- **Temporary Exclusion of Tests|Tasks**: To prevent tests|tasks with known issues from running until the issue is resolved.
- **Conditional Execution**: To skip tests|tasks dynamically based on runtime conditions, e.g., if a Suite Setup detects an issue.
- **Unsupported Scenarios**: To mark tests|tasks as skipped in environments where they cannot run, while still ensuring they are logged as such.

4.5.1 Skipping By Tags Selection (CLI)

LEARNING OBJECTIVES

K1 LO-4.5.1

Recall the differences between skip and exclude

Tests|tasks can be skipped with `--skip` by tags when executing Robot Framework, similar to `--exclude`. The difference between `--skip` and `--exclude` is that `--skip` will mark the tests|tasks as skipped in the report and log, while `--exclude` will not execute them at all. Therefore skip is better for documenting that a specific test|task was not executed for a specific reason.

Example: If there is a defect in the System under Test (SUT) and a test|task has been written to reproduce the defect and tests its resolution, but the defect is not yet resolved, the test|task can be tagged with the defect-number and skipped until the defect has been resolved.

Example: Assuming there are different test environments and some tests can only be executed in specific environments, the tests can be tagged with the environment name and skipped in all other environments.

- **Command Line Option**: Use the `--skip` option to skip tests|tasks based on tags or tag patterns:

```
robot --skip BUG-42 --skip mobile path/to/tests
```

- **Reserved Tag `robot:skip`**: Add the `robot:skip` tag to tests|tasks to mark them as skipped: This ensures the test|task appears in reports as skipped but is not executed.

4.5.2 Skipping Dynamically During Execution

Tests|tasks can be skipped dynamically within their execution with the `Skip` keyword based on runtime conditions.

The `Skip` keyword does stop the execution of a test|task and mark it as skipped with a custom message. If a Test|Task Teardown exists, it will be executed.

4.5.3 Automatically Skipping Failed Tests

Tests|tasks can be automatically marked as skipped if they fail:

- **Command Line Option**: Use `--skiponfailure` with tags or tag patterns:

  ```
  robot --skiponfailure flaky path/to/tests
  ```

- **Reserved Tag** `robot:skip-on-failure`: Tag tests|tasks to skip automatically on failure.

5 Exploring Advanced Constructs

This chapter introduces more advanced constructs of Robot Framework. These topics are often not needed for simple automation cases but can be very useful in more complex situations. Although it is not expected that Robot Framework Certified Professionals will be able to use them, it is important to be aware of the possibilities and to understand the basic concepts.

5.1 Advanced Variables

Variables in Robot Framework, and in programming languages in general, can be more complex and can store various types of data. Robot Framework also offers multiple ways to create different kinds of values and types. However, the built-in language support is limited to the basic 3.2.2.2 Primitive Data Types, 3.2.2.3 List Variable Definition, and 3.2.2.4 Dictionary Variable Definition.

This chapter provides more advanced knowledge about the different variable scopes, lists, dictionaries, their syntax, and some background on the most important Built-In Variables.

Understanding the **priority** and **scope** of variables in Robot Framework is crucial for effective test automation. Variables can be defined in multiple places and ways, and their availability and precedence depend on where and how they are created.

5.1.1 Variable Priorities

LEARNING OBJECTIVES

K2 LO-5.1.1

Understand the difference between statically defined and dynamically created variables in Robot Framework

Variables can originate from various sources, and when variables with the same name exist, Robot Framework resolves them based on their priority.

Several factors influence variable priority in Robot Framework: the type of variable, the time of (re-)definition, and the variable's scope.

In general, there are two types of variables regarding how they are created:

- Statically defined or imported variables (e.g., in the `*** Variables ***` section, command-line options, imported resource files)
- Dynamically created variables during Robot Framework execution (e.g., using the `VAR` syntax, assignment of return values from keywords or keyword arguments)

Built-in variables cannot generally be sorted into one of these categories, as some are predefined globally while others are created during execution with a `SUITE` or `TEST` scope.

Examples:

- `${TEST_NAME}` is dynamically set during execution to the name of the currently running test case.
- `${OUTPUT_DIR}` is statically defined before the execution and contains the directory where `output.xml`, `log.html` and `report.html` are written.
- `${LOG_LEVEL}` is by default set statically via command line options or `INFO` as default, but can be changed, with the keyword `Set Log Level` during exection.

5.1.1.1 Statically Defined or Imported Variables

LEARNING OBJECTIVES

K1 LO-5.1.1.1

Recall the priority of statically defined or imported variables in Robot Framework

The rule of thumb here is: **"First come, first served!"**

The time of definition has the greatest impact on the priority of these variables.

In descending order, the priority is as follows:

1. **Global Command-Line Variables**: Variables defined via command-line options like `--variable` or `--variablefile` have the highest priority. See 5.1.3 Global Variables via Command Line for more details.
2. `*** Variables ***` **Section**: Variables defined in the `*** Variables ***` section of a suite are set before any resource file from the `*** Settings ***` section is imported. See 3.2.2 `*** Variables ***` Section for more details.
3. **Resource Files**: Variables from resource files are imported in the order they are specified in the `*** Settings ***` section. See 2.4.2 Resource Files for more details.

   Within a resource file, the same order applies: variables defined in the `*** Variables ***` section of a resource file have higher priority than variables imported from other resource files.

However, variables defined during Robot Framework execution can overwrite or shadow these variables.

5.1.1.2 Dynamically Created Variables

LEARNING OBJECTIVES

LO-5.1.1.2

Recall the priority of dynamically created variables in Robot Framework

Variables created or modified during execution have a higher priority than statically defined or imported variables.

The rule of thumb here is: **"Last one wins!"**

The scope of a variable defines its lifetime and availability. As long as a variable is in scope, the last definition takes precedence over the previous ones.

For example, a local variable defined as a 3.3.5 User Keyword Arguments has a higher priority than a suite variable defined in the `*** Variables ***` section of the suite file. However, once the keyword body scope is exited, the suite variable is back in scope with higher priority and the local variable is no longer existent.

5.1.2 Variable Scopes

LEARNING OBJECTIVES

LO-5.1.2

Recall the different variable scopes in Robot Framework

Variables in Robot Framework have different scopes, determining where they can be accessed and how long they are available.

5.1.2.1 . Global Scope

LEARNING OBJECTIVES

LO-5.1.2.1

Recall how to define global variables and where they can be accessed

- **Definition**: Variables accessible everywhere during the test execution.
- **Creation**:
  - Set from the command line using `--variable` or `--variablefile` options. (static)
  - Created during execution using the `VAR` syntax with the `scope=GLOBAL` argument. (dynamic)
- **Usage**: Ideal for configuration parameters that need to be consistent across the entire test run.

Because global variables set via the command line have the highest priority, they can override other variables defined in the suite or resource files. The most common use case for global variables is to define environment-specific or execution configurations, such as URLs, credentials, browser types, API keys, or similar data.

See 5.1.3 Global Variables via Command Line for more details.

Recommendation: Global variables should always be defined using uppercase letters, like `${GLOBAL_VARIABLE}`, to distinguish them from local variables. Every global variable should have a corresponding default value defined either in a `*** Variables ***` section or imported from variable files, so that editors and IDEs can provide auto-completion and static code analysis.

5.1.2.2 · Suite Scope

LEARNING OBJECTIVES

LO-5.1.2.2

Recall how to define suite variables and where they can be accessed

- **Definition**: Variables accessible within the test suite where they are defined, including all its tests|tasks and keywords.
- **Creation**:
  - Defined in the `*** Variables ***` section of the suite file. (static)
  - Imported from resource or variable files. (static)
  - Set during the execution of a suite using the `VAR` syntax with the `scope=SUITE` argument. (dynamic)
- **Usage**: Useful for sharing data among tests/tasks within the same suite or configuring suite-specific settings or setting default values for global variables.

Suite scope is not recursive; variables in a higher-level suite, e.g. defined in 4.3 Initialization Files, are not available in lower-level suites. Use resource files to share variables across suites.

Variables with a suite scope are generally statically defined or imported variables, but they can also be created dynamically during the execution of a suite. In this latter case, they have a higher priority than statically defined variables and can shadow or overwrite them.

If a variable is defined in the `*** Variables ***` section of a suite file and is dynamically defined using the `VAR` syntax at the suite level, the variable value is overwritten with the new value.

If a global variable is defined using the command line, and a suite-level variable with the same name is dynamically defined, the suite variable now shadows the global variable and has higher priority as long as the suite is in scope. Once the suite is finished or a sub-suite is executed, the global variable returns to scope with higher priority.

Recommendation: Suite variables should be defined using uppercase letters, like `${SUITE_VARIABLE}`, to distinguish them from local variables. These variables should be defined in the `*** Variables ***` section of the suite file, even if they are dynamically overwritten during execution, so they are visible in the editor or IDE and can be used for auto-completion and static code analysis.

5.1.2.3 . Test|Task Scope

LEARNING OBJECTIVES

K1 LO-5.1.2.3

Recall how to define test|task variables and where they can be accessed

- **Definition**: Variables accessible within a single test|task and within all keywords it calls.
- **Creation**:
  - Created during test execution using the `VAR` syntax with the `scope=TEST` or `scope=TASK` argument. (dynamic)
- **Usage**: Appropriate for data that is specific to a single test|task.

Test|Task variables cannot be created in suite setup or teardown, nor can they be imported. Test|Task scope variables are not available in other tests|tasks, even within the same suite. They can only be created dynamically, so they have higher priority than suite or global variables while in scope. Once a test|task is finished, the variables are no longer available. If they have shadowed a suite or global variable, that variable returns to scope.

Recommendation: Test|Task variables should be used only when there is a clear need to share data across multiple keywords within a single test|task and when this is known by all team members. Otherwise, it is better to use local variables. Editor and IDE support for these variables is limited, so they should be used with caution.

5.1.2.4 . Local Scope

LEARNING OBJECTIVES

K1 LO-5.1.2.4

Recall how to define local variables and where they can be accessed

- **Definition**: Variables accessible only within the keyword or test|task where they are defined.
- **Creation**:
  - Variables assigned by keyword return values.
  - Variables defined using the `VAR` syntax (optional: with `scope=LOCAL`) within a keyword or test|task.
  - Keyword arguments.
- **Usage**: Commonly used to temporarily store data and pass it to other keywords.

Local variables are the most commonly used variables in Robot Framework and have the fewest side effects. They should be preferred over other variable scopes unless there is an explicit need to share data across scope boundaries.

Recommendation: Local variables should always be defined using lowercase letters, like `${local_variable}`, to distinguish them from other variables.

Example of local variables:

```
*** Test Cases ***
Test People In Room
    ${trainer_count}    Get Trainers In Room    # returns the integer 2
```

```
    ${trainee_count}    Get Trainees In Room    # returns the integer 12
    ${total_people}    Calculate Sum    ${trainer_count}    ${trainee_count}
    Should Be Equal As Numbers    ${total_people}    14

*** Keywords ***
Calculate Sum
    [Arguments]    ${num1}    ${num2}
    ${result}    Evaluate    ${num1} + ${num2}
    RETURN    ${result}
```

In this example, the variable `${trainer_count}` is only available in the test case itself and not in the keyword `Calculate Sum`. Therefore, its value has to be passed as an argument to `Calculate Sum`, which assigns the value stored in `${trainer_count}` to the local variable `${num1}` within `Calculate Sum`. Additionally, `${result}` is only available within `Calculate Sum`, and only its value is returned to the test case, where it is assigned to `${total_people}`.

5.1.3 Global Variables via Command Line

As described earlier, global variables can be statically defined via command-line options.

The command line option `--variable` or `-v` can be used to define global variables. This option can be used multiple times to define multiple variables. The syntax is `--variable name:value` where `name` is the variable name without `${}` and `value` is the assigned value.

Only scalar string values are supported.

Examples:

- Simple String: `${name}` == `Robot` (str)

  ```
  robot --variable name:Robot .
  ```

- String with Spaces: `${hello}` == `Hello world` (str)

  ```
  robot -v "hello:Hello world" .
  ```

- Multiple Variables: `${name}` == `Robot` (str), `${version}` == `4.0` (str), `${patch}` == `${EMPTY}`

  ```
  robot -v "name:Robot Framework" -v version:4.0 -v patch: .
  ```

5.1.4 List-Variables (Advanced)

As explained in the `*** Variables ***` section under 3.2.2.3 List Variable Definition, Robot Framework natively supports creating lists. However, the at-syntax `@{var}` has different meanings when assigning values versus accessing values.

5.1.4.1 Assigning List Variables

LEARNING OBJECTIVES

K1 LO-5.1.4.1

> Recall that assignments to `@{list}` variables convert values to lists automatically

Using the at-syntax (`@{}`) is required to define a list variable with `VAR` syntax or in the `*** Variables ***` section, but it is optional when assigning return values, which are list-like, from keywords to a variable.

Example:

```
*** Test Cases ***
Test List Variables
    @{participants}    Get Participants                # returns a list of names
    ${trainers}        Get Trainers                    # returns a list of trainers
```

Both assignments will contain a list if the keyword returns a list of values.

However, if a keyword returns something other than a list but still list-like, it will be assigned without changes to the scalar variable `${trainers}` and will be converted to a list when using the at-syntax, as in `@{participants}`. List-like values can include Tuples, Sets, Dictionary Keys, or generator functions. As long as a value is iterable, it can be assigned to a list variable using the at-syntax to ensure it is a list after assignment.

Note: Strings are iterable in Python; however, they are explicitly **NOT** converted to a list when assigned to a list variable to prevent mistakes.

5.1.4.2 Accessing List Variables

> LEARNING OBJECTIVES
>
> > LO-5.1.4.2
> >
> > Recall that `@{list}` unpacks the values of a list variable when accessed

Variables containing a list are generally accessed with the normal dollar-syntax `${var}`. You can also access single values within a list using `${var}[0]` or `${var}[-1]`, and Robot Framework supports slicing, similar to Python, with `${var}[1:3]` or `${var}[1:]`.

However, in some cases, it is necessary to unpack the values of a list variable to use them as a sequence of multiple individual values. This is done using the at-syntax `@{var}` when accessing the variable. Unpacking works for iterable values, but is NOT possible with strings!

Example:

```
*** Variables ***
@{participants}    Alice    Bob    Charlie


*** Test Cases ***
Test List Variables
    Log Many    Alice    Bob    Charlie        # Logs three entries:    "Alice", "Bob", and "Charlie"
    Log Many    @{participants}               # Logs three entries:    "Alice", "Bob", and "Charlie"
    Log Many    ${participants}               # Logs only one entry:   "['Alice', 'Bob', 'Charlie']"
```

In the first two cases, the keyword `Log Many` is called with three arguments; in the last case, it is called with only one argument, which is a list of three values.

This is particularly needed when using FOR-Loops. See 5.2.4 FOR Loops for more details.

5.1.5 Dict-Like

As explained in the `*** Variables ***` section under 3.2.2.4 Dictionary Variable Definition, Robot Framework natively supports creating dictionaries. However, the ampersand-syntax `&{var}` has different meanings when assigning values and when accessing values.

5.1.5.1 Assigning Dictionary Variables

LEARNING OBJECTIVES

LO-5.1.5.1

Recall that assignments to `&{dict}` variables automatically convert values to Robot Framework Dictionaries and enable dot-access

Using the ampersand-syntax (`&{}`) is required to define a dictionary variable with `VAR` syntax or in the `*** Variables ***` section, but it is optional when assigning return values from keywords to a variable that returns dictionaries.

Example:

```
*** Test Cases ***
Test Dictionary Variables
    &{participant}    Get Participant    number=4    # returns a dictionary with keys "name" and "age"
    ${trainer}        Get Trainer        number=1    # returns a dictionary with keys "name" and "age"
```

In the following example, the first assignment to `&{participant}` causes an automatic conversion to a Robot Framework Dictionary, also known as DotDict. These special dictionary types can be accessed using dot-access like `${participant.name}` or `${participant.age}`, instead of the usual dictionary access like `${trainer}[name]` or `${trainer}[age]`.

5.1.5.2 Accessing Dictionary Variables

LEARNING OBJECTIVES

LO-5.1.5.2

Recall that `&{dict}` unpacks to multiple key=value pairs when accessed

Variables containing dictionaries are typically accessed using the normal dollar-syntax `${var}`. You can also access individual values by their keys using `${var}[key]` or `${var.key}` for Robot Framework Dictionaries.

However, in some cases, it is useful to unpack the key-value pairs of a dictionary variable to use them as a sequence of multiple key-value pairs. This is done using the ampersand-syntax `&{var}` when accessing the variable.

Example:

```
*** Variables ***
&{participant_one}    name=Alice    age=23
&{participant_two}    name=Bob      age=42

*** Test Cases ***
Test Dictionary Variables
    Log Participant    John    33
    Log Participant    name=Pekka    age=44
    Log Participant    &{participant_one}
```

```
    Log Participant    &{participant_two}

*** Keywords ***
Log Participant
    [Arguments]    ${name}    ${age}
    Log    ${name} is ${age} years old
```

Instead of calling the keyword `Log Participant` with two arguments, it is possible to use the unpacked dictionary variables `&{participant_one}` and `&{participant_two}` to call the keyword with two named arguments. The dictionary keys act as the argument names and the values as the argument values.

5.1.6 Built-In Variables

LEARNING OBJECTIVES

LO-5.1.6

Recall that Robot Framework provides access to execution information via Built-In variables

Robot Framework has a set of built-in variables that can be used in test cases, keywords, and other places. Some examples are:

Variable	Description
`${EMPTY}`	An empty string.
`${SPACE}`	A single space character.
`${CURDIR}`	An absolute path to the directory where the current suite or resource file is located. This variable is case-sensitive.
`${EXECDIR}`	An absolute path to the directory where test execution was started from.
`${OUTPUT_DIR}`	An absolute path to the directory where output files, like `output.xml`, `log.html`, and `report.html`, are written.
`${TEMPDIR}`	An absolute path to the system temporary directory. In UNIX-like systems, this is typically `/tmp`, and in Windows, it is `c:\Documents and Settings\<user>\Local Settings\Temp`.

Additionally, suite-related or test|task-related variables are available. These variables can have different values during test execution, and some are not available at all times. Altering the value of these variables does not affect the original values.

Variable	Description
`${SUITE_NAME}`	The name of the current suite.
`${SUITE_SOURCE}`	The path to the file where the current suite is defined.
`${SUITE_DOCUMENTATION}`	The documentation of the current suite.
`${TEST_NAME}`	The name of the current test.

Variable	Description
`${TEST_DOCUMENTATION}`	The documentation of the current test.
`${PREV_TEST_STATUS}`	The status of the previous test.

These variables can be used in test cases, keywords, and other places to access information about the current test execution.

5.2 Control Structures

Robot Framework is a Turing-complete language and supports all common control structures, including IF-Statements, FOR-Loops, WHILE-Loops and more. While it is not expected that RFCPs can write complex control structures, they should understand their purpose.

In some cases, it is necessary to use control structures to handle different cases, iterate over a list of values, or execute an action until a condition is met.

5.2.1 IF Statements

LEARNING OBJECTIVES

LO-5.2.1

Understand the purpose and basic concept of IF-Statements

The `IF` / `ELSE IF` / `ELSE` syntax in Robot Framework is used to control the flow of test|task execution by allowing certain keywords to run only when specific conditions are met. This is achieved by evaluating conditions written as Python expressions, enabling dynamic decision-making within your tests|tasks.

The `IF` statement begins with the `IF` token and ends with an `END`, enclosing the keywords executed when the condition is true. An optional `ELSE` or `ELSE IF` can specify alternative actions when the initial condition is false. This structure enhances the flexibility and responsiveness of your tests|tasks, allowing them to adapt based on variables and outcomes encountered during execution.

5.2.1.1 Basic IF Syntax

When certain keywords should be executed only if a condition is met, the IF statement can be used.

Structure

```
IF    <condition>
    <keywords>
    <keywords>
END
```

Example

```
*** Test Cases ***
Check Status
    IF    $status == 'SUCCESS'
        Log    Operation was successful.
    END
```

It executes the `Log` keyword if `${status}` is the string `SUCCESS`.

5.2.2 IF/ELSE Structure

To execute different alternative actions based on various conditions, use the IF/ELSE structure.

Structure

```
IF    <condition1>
    <keywords if condition1 is true>
ELSE IF    <condition2>
    <keywords if condition2 is true>
ELSE
    <keywords if all conditions are false>
END
```

Example

```
*** Test Cases ***
Evaluate Score
    IF    $score >= 90
        Log    Grade A
    ELSE IF    $score >= 80
        Log    Grade B
    ELSE
        Log    Grade C or below
    END
```

5.2.3 Inline IF Statement

For single conditional keywords, the simplified inline IF statement can be used.

Structure

```
IF    <condition>    <keyword>    [arguments]
```

Example

```
*** Test Cases ***
Quick Check
    IF    $user == 'Admin'    Log    Admin access granted.
```

Executes the `Log` keyword if `${user}` equals to the string `'Admin'`.

No `END` is needed for inline IF.

5.2.4 FOR Loops

LEARNING OBJECTIVES

LO-5.2.4

Understand the purpose and basic concept of FOR Loops

The `FOR` loop in Robot Framework repeats a set of keywords multiple times, iterating over a sequence of values. This allows you to perform the same actions for different items without duplicating code, enhancing the efficiency and readability of your keyword logic.

Robot Framework has four types of FOR loops; this chapter focuses on the basic `FOR-IN` loop.

- `FOR-IN` is used to iterate over a list of values.

The other types are `FOR-IN-RANGE`, `FOR-IN-ENUMERATE`, and `FOR-IN-ZIP`, which are more advanced and less commonly required.

- `FOR-IN-RANGE` iterates over a range of numbers.
- `FOR-IN-ENUMERATE` iterates over a list of values and their indexes.
- `FOR-IN-ZIP` iterates over multiple lists simultaneously.

The `FOR` loop begins with the `FOR` token, followed by a loop variable, the `IN` token, and the iterable variable or list of values. The loop variable takes on each value in the sequence one at a time, executing the enclosed keywords for each value.

5.2.4.1 Basic FOR Loop Syntax

When you need to execute the same keywords for each item in a list or sequence, you can use the FOR-IN loop.

Structure

```
FOR    ${loop_variable}    IN    <value1>    <value2>    ...    <valueN>
    <keywords>
    <keywords>
END
```

Since `<value1>    <value2>    ...    <valueN>` can be the same as an unpacked list like `@{values}`, this is the most common way to use the FOR loop.

Structure

```
FOR    ${loop_variable}    IN    @{iterable_values}
    <keywords>
    <keywords>
END
```

Examples:

Example

```
*** Test Cases ***
Process Fruit List
    FOR    ${fruit}    IN    apple    banana    cherry
        Log    Processing ${fruit}
    END
```

This would essentially be the same as this:

Example

```
*** Variables ***
@{fruits} =    apple    banana    cherry

*** Test Cases ***
Process Fruit List
    FOR    ${fruit}    IN    @{fruits}
        Log    Processing ${fruit}
    END
```

Or this:

Example

```
*** Test Cases ***
Process Fruits separately
    Log    Processing apple
    Log    Processing banana
    Log    Processing cherry
```

5.2.5 WHILE Loops

LEARNING OBJECTIVES

LO-5.2.5

Understand the purpose and basic concept of WHILE Loops

While the `FOR` loop iterates over a known amount of values, `WHILE` loops repeat their body as long as a condition is met. This is typically used in cases where the number of iterations is not known in advance or depends on a dynamic condition.

One example use case would be scrolling down a page until a certain element is visible. In this case, you would use a `WHILE` loop to keep scrolling until the element is found or a maximum iteration limit is reached.

The `WHILE` loop begins with the `WHILE` token, followed by a condition that evaluates to true or false. If the condition is true, the loop body is executed, and the condition is re-evaluated. If the condition is false, the loop is exited, and execution continues with the next keyword after the `END`. The condition is similar to an IF statement, a Python expression that evaluates to a boolean value.

Structure

```
WHILE    <condition>
    <keywords>
    <keywords>
END
```

Example

```
*** Test Cases ***
Scroll Down Until Element Visible
    ${element_visible}    Get Element Visibility    <locator>
    WHILE    not $element_visible
        Scroll Down
```

```
        ${element_visible}    Get Element Visibility    <locator>
    END
```

`WHILE` loops have a configurable iteration limit in Robot Framework. When the maximum number of iterations is reached, the loop exits with a failure, causing the test|task or keyword to fail. This prevents infinite loops and ensures that tests|tasks do not hang indefinitely.

5.2.6 BREAK and CONTINUE

LEARNING OBJECTIVES

LO-5.2.6

Understand the purpose and basic concept of the BREAK and CONTINUE statements

In some cases, it is helpful to stop a loop or skip the remaining part of a loop and continue with the next iteration. This can be achieved with the `BREAK` and `CONTINUE` statements.

- `BREAK` stops the current loop and exits it immediately.
- `CONTINUE` skips the remaining part of the current iteration and continues with the next iteration.

These can, of course, be combined with `IF` statements to control the loop flow.

Example 1 `BREAK`:

Suppose we want to search for an element on a page and scroll down until it is visible. This time, we do not know the number of pages we can scroll, so we use the `WHILE` loop. However, we want the loop to iterate and `BREAK` once we have found the element.

Example with BREAK

```
*** Test Cases ***
Scroll Down Until Element Visible
    WHILE    True    # This would loop to the max iteration limit
        ${element_visible}    Get Element Visibility    <locator>
        IF    ${element_visible}    BREAK
        Scroll Down
    END
```

Here we used `BREAK` to exit the loop before scrolling down if the element is visible.

`CONTINUE` is useful when you want to skip the remaining part of the current iteration and continue with the next iteration if a condition is met. In that case, combine `IF` and `CONTINUE` to control the loop flow.

Example 2 `CONTINUE`:

Example with CONTINUE

```
*** Settings ***
Library     Collections


*** Variables ***
&{PARTICIPANT_1}    name=Alice      age=23
```

```
&{PARTICIPANT_2}    name=Bob        age=42
&{PARTICIPANT_3}    name=Charlie    age=33
&{PARTICIPANT_4}    name=Pekka      age=44
@{PARTICIPANTS}     ${PARTICIPANT_1}    ${PARTICIPANT_2}    ${PARTICIPANT_3}    ${PARTICIPANT_4}


*** Test Cases ***
Find Older Participants
    ${older_participants}    Get Older Participants    ${PARTICIPANTS}    40
    Should Be Equal    ${older_participants}[0][name]    Bob
    Should Be Equal    ${older_participants}[1][name]    Pekka


*** Keywords ***
Get Older Participants
    [Arguments]    ${participants}    ${minimum_age}
    VAR    @{older_participants}                                    # Creates an empty list
    FOR    ${participant}    IN    @{participants}                  # Iterates over all participants
        IF    ${participant.age} < ${minimum_age}    CONTINUE       # Skips the remaining part of the
        Log    Participant ${participant.name} is older than 40     # Logs participant name if age is
        Append To List    ${older_participants}    ${participant}    # BuiltIn keyword to append a valu
    END
    RETURN    ${older_participants}
```

Glossary

Term	Definition
Behavior-Driven	A testing methodology that encourages collaboration between developers, QA, and non-technical stakeholders to define test cases.
Data-Driven Specification	A testing approach where test cases are executed with multiple sets of data to validate functionality.
Generic Test Automation Architecture (gTAA)	A framework that provides a structured approach to test automation, promoting reusability and maintainability.
Keywords	Reusable functions or actions defined in the test automation framework.
Keyword-Driven	A testing approach where test cases are defined using keywords that represent actions or operations.
Library	A collection of keywords and functions that can be used in test automation.
Libdoc	A tool used to generate keyword documentation for libraries and resource files.
Rebot	The main executable used to execute suites and post-process execution results to generate reports.
Resource Files	Files that contain shared keywords and variables that can be imported into test suites.
Root Suite	The top-level suite that contains all other suites and test cases.
Suite Directory	A directory that contains multiple suite files, which can include test cases and tasks organized hierarchically.
Suite File	A *.robot file that contains at least one test case or task.
Task	A unit of work that can be executed, similar to a test case but typically focused on automation tasks.
Task Suite	Suite files that have at least one task and do not contain any test cases.
Test Automation	The use of software tools to execute tests automatically, reducing manual effort.
Test Cases Section	This section defines the executable elements of a suite, specifically test cases.
Test Suite	Suite files that have at least one test case and do not contain any tasks.
Tasks Section	This section defines the executable elements of a suite, specifically tasks.

Term	Definition
Comments Section	This section is used to add comments to the suite file or resource file. All content in this section is ignored by Robot Framework.
Keyword Section	This section allows you to define locally scoped user keywords that can only be used within the same suite where they are defined.
Robot Framework Sections	Different parts of a Robot Framework suite file that organize the content.
Settings Section	This section is used to configure various aspects of the test/task suite.
Variables Section	This section is used to define suite variables that are used in the suite or its tests/tasks or inside their keywords.

Learning Objectives

ID	K-Level	Content
LO-1.1	K1	Recall the two main use cases of Robot Framework
LO-1.1.1	K1	Recall the test levels Robot Framework is mostly used for
LO-1.2.1	K1	Recall the layers of the Generic Test Automation Architecture (gTAA) and their corresponding components in Robot Framework
LO-1.2.2	K1	Recall what is part of Robot Framework and what is not
LO-1.2.3	K1	Recall the technology Robot Framework is built on and the prerequisites for running it
LO-1.3	K1	Recall the key attributes of the syntax that makes Robot Framework simple and human-readable
LO-1.3.3	K2	Explain the difference between User Keywords and Library Keywords
LO-1.3.4	K1	Recall the difference between Resource Files and Libraries and their artifacts
LO-1.4	K1	Recall the three specification styles of Robot Framework
LO-1.4.1	K2	Understand the basic concepts of Keyword-Driven Specification
LO-1.4.2	K2	Understand the basic concepts of Behavior-Driven Specification
LO-1.4.3	K1	Recall the differences between Keyword-Driven and Behavior-Driven Specification
LO-1.4.4	K1	Recall the purpose of Data-Driven Specification
LO-1.5.1	K1	Recall the type of open-source license under which Robot Framework is distributed
LO-1.5.2	K1	List and recall the key objectives and organizational form of the Robot Framework Foundation
LO-1.5.3	K1	Recall the official webpages for Robot Framework and its resources
LO-2.1	K2	Understand which files and directories are considered suites and how they are structured in a suite tree.
LO-2.1.1	K1	Recall the conditions and requirements for a file to be considered a Suite file
LO-2.1.2	K1	Recall the available sections in a suite file and their purpose.
LO-2.1.2.1-1	K1	Recall the available settings in a suite file.

ID	K-Level	Content
`LO-2.1.2.1-2`	K2	Understand the concepts of suite settings and how to define them.
`LO-2.1.2.2`	K1	Recall the purpose of the `*** Variables ***` section.
`LO-2.1.2.3`	K2	Understand the purpose of the `*** Test Cases ***` or `*** Tasks ***` section.
`LO-2.1.2.4`	K2	Understand the purpose and limitations of the `*** Keywords ***` section.
`LO-2.2`	K2	Understand the basic syntax of test cases and tasks.
`LO-2.2.1`	K3	Understand and apply the mechanics of indentation and separation in Robot Framework.
`LO-2.2.2`	K3	Be able to use line breaks and continuation in a statement.
`LO-2.2.3`	K3	Be able to add in-line comments to suites.
`LO-2.2.4`	K2	Understand how to escape control characters in Robot Framework.
`LO-2.2.5`	K2	Understand the structure of a basic suite file.
`LO-2.3`	K1	Recall the three components of the Robot Framework CLI.
`LO-2.3.1`	K2	Understand how to run the `robot` command and its basic usage.
`LO-2.3.2`	K2	Explain the execution artifacts generated by Robot Framework.
`LO-2.3.3`	K1	Recall the four different status labels used by Robot Framework.
`LO-2.3.3.1`	K2	Understand when an element is marked as `PASS`.
`LO-2.3.3.2`	K2	Understand when an element is marked as `FAIL`.
`LO-2.3.4`	K2	Understand the difference between log messages and console output.
`LO-2.4.1-1`	K1	Recall the purpose of keyword libraries and how to import them.
`LO-2.4.1-2`	K1	Recall the three types of libraries in Robot Framework.
`LO-2.4.2-1`	K1	Recall the purpose of resource files.
`LO-2.4.2-2`	K3	Use resource files to import new keywords.
`LO-2.4.3`	K2	Understand the different types of paths that can be used to import libraries and resource files.
`LO-2.5`	K2	Understand the structure of keyword interfaces and how to interpret keyword documentation.
`LO-2.5.1`	K1	Recall the information that can be found in a keyword documentation.

ID	K-Level	Content
LO-2.5.2	K2	Understand the difference between argument kinds.
LO-2.5.2.1	K2	Understand the concept of mandatory arguments and how they are documented.
LO-2.5.2.2	K2	Understand the concept of optional arguments and how they are documented.
LO-2.5.2.3	K1	Recall the concept of keywords with embedded arguments used in Behavior-Driven Specification and how they are documented.
LO-2.5.2.4	K1	Recall how "Positional or Named Arguments" are marked in the documentation and their use case.
LO-2.5.2.5	K1	Recall how "Variable Number of Positional Arguments" are marked in the documentation and their use case.
LO-2.5.2.6	K1	Recall what properties "Named-Only Arguments" have and how they are documented.
LO-2.5.2.7	K1	Recall how free named arguments are marked in documentation.
LO-2.5.2.8	K2	Understand the concept of argument types and automatic type conversion.
LO-2.5.2.9	K2	Understand the concept of return type hints.
LO-2.5.3	K2	Understand how to read keyword documentation and how to interpret the examples.
LO-2.6	K2	Understand how to call imported keywords and how to structure keyword calls.
LO-2.6.1	K2	Understand the concept of how to set argument values positionally.
LO-2.6.2	K2	Understand the concept of named arguments and how to set argument values by their name.
LO-2.6.3	K1	Recall how to use embedded arguments.
LO-3.2-1	K2	Understand how variables in Robot Framework are used to store and manage data
LO-3.2-2	K1	Recall the relevant five different ways to create and assign variables
LO-3.2.1-1	K1	Recall the four syntactical access types to variables with their prefixes
LO-3.2.1-2	K1	Recall the basic syntax of variables
LO-3.2.2-1	K3	Create variables in the Variables section
LO-3.2.2-2	K3	Use the correct variable prefixes for assigning and accessing variables
LO-3.2.2.1-1	K3	Create and assign scalar variables
LO-3.2.2.1-2	K2	Understand how multiple lines can be used to define scalar variables

ID	K-Level	Content
`LO-3.2.2.2`	K2	Understand how to access primitive data types
`LO-3.2.2.3`	K2	Understand how to set and access data in list variables
`LO-3.2.2.4`	K2	Understand how to set and access data in dict variables
`LO-3.2.3`	K3	Be able to assign return values from keywords to variables
`LO-3.2.4`	K2	Understand how to create variables using the VAR statement
`LO-3.2.5`	K2	Understand how `local` and `suite` scope variables are created
`LO-3.3.2`	K1	Recall the rules how keyword names are matched.
`LO-3.3.3`	K1	Recall all available settings and their purpose for User Keywords
`LO-3.3.4`	K1	Recall the significance of the first logical line and in keyword documentation for the log file.
`LO-3.3.5`	K2	Understand the purpose and syntax of the [Arguments] setting in User Keywords.
`LO-3.3.5.1-1`	K1	Recall what makes an argument mandatory in a user keyword.
`LO-3.3.5.1-2`	K3	Define User Keywords with mandatory arguments.
`LO-3.3.5.2-1`	K1	Recall how to define optional arguments in a user keyword.
`LO-3.3.5.2-2`	K3	Define User Keywords with optional arguments.
`LO-3.3.5.3-1`	K2	Describe how embedded arguments are replaced by actual values during keyword execution.
`LO-3.3.5.3-2`	K2	Understand the role of embedded arguments in Behavior-Driven Development (BDD) style.
`LO-3.3.6-1`	K2	Understand how the `RETURN` statement passes data between different keywords.
`LO-3.3.6-2`	K3	Use the `RETURN` statement to return values from a user keyword and assign it to a variable.
`LO-3.3.7`	K1	Recall the naming conventions for user keywords.
`LO-3.4`	K2	Understand the basic concept and syntax of Data-Driven Specification
`LO-3.4.1-1`	K2	Understand how to define and use test|task templates
`LO-3.4.1-2`	K1	Recall the differences between the two different approaches to define Data-Driven Specification
`LO-3.4.1.1`	K1	Recall the syntax and properties of multiple named test|task with one template
`LO-3.4.1.2`	K1	Recall the syntax and properties of named test|task with multiple data rows

ID	K-Level	Content
LO-3.5	K1	Recall that naming conflicts can arise from the import of multiple resource files.
LO-3.5.1	K2	Understand how transitive imports of resource files and libraries work.
LO-3.5.2	K3	Be able to configure a library import using arguments.
LO-3.5.3	K2	Explain how naming conflicts can happen and how to mitigate them.
LO-4.1-1	K1	Recall the purpose and benefits of Setups in Robot Framework
LO-4.1-2	K1	Recall the different levels where a Setup can be defined
LO-4.1.1-1	K1	Recall key characteristics, benefits, and syntax of Suite Setup
LO-4.1.1-2	K2	Understand when Suite Setup is executed and used
LO-4.1.2-1	K1	Recall key characteristics, benefits, and syntax of Test Setup
LO-4.1.2-2	K2	Understand when Test|Task Setup is executed and used
LO-4.1.3	K1	Recall key characteristics and syntax of Keyword Setup
LO-4.2-1	K2	Understand the different levels where and how Teardowns can be defined and when they are executed
LO-4.2-2	K1	Recall the typical use cases for using Teardowns
LO-4.2.1-1	K1	Recall key characteristics, benefits, and syntax of Suite Teardown
LO-4.2.1-2	K2	Understand when Suite Teardown is executed and used
LO-4.2.2-1	K1	Recall key characteristics, benefits, and syntax of Test|Task Teardown
LO-4.2.2-2	K2	Understand when Test|Task Teardown is executed and used
LO-4.2.3	K1	Recall key characteristics, benefits, and syntax of Keyword Teardown
LO-4.3	K1	Recall how to define an Initialization Files and its purpose
LO-4.3.2	K2	Understand the execution order of Suite Setup and Suite Teardown in Initialization Files and their sub-suites and tests|tasks
LO-4.3.3	K1	Recall the allowed sections and their content in Initialization Files
LO-4.4	K1	Recall the purpose of Test|Task Tags in Robot Framework
LO-4.4.1	K1	Recall the syntax and different ways to assign tags to tests|tasks

ID	K-Level	Content
`LO-4.4.2`	K2	Understand how to filter tests|tasks using the command-line interface of Robot Framework
`LO-4.5-1`	K1	Recall the use case and purpose of skipping tests|tasks in Robot Framework
`LO-4.5-2`	K1	Recall the different ways to skip tests|tasks in Robot Framework
`LO-4.5.1`	K1	Recall the differences between skip and exclude
`LO-5.1.1`	K2	Understand the difference between statically defined and dynamically created variables in Robot Framework
`LO-5.1.1.1`	K1	Recall the priority of statically defined or imported variables in Robot Framework
`LO-5.1.1.2`	K1	Recall the priority of dynamically created variables in Robot Framework
`LO-5.1.2`	K1	Recall the different variable scopes in Robot Framework
`LO-5.1.2.1`	K1	Recall how to define global variables and where they can be accessed
`LO-5.1.2.2`	K1	Recall how to define suite variables and where they can be accessed
`LO-5.1.2.3`	K1	Recall how to define test|task variables and where they can be accessed
`LO-5.1.2.4`	K1	Recall how to define local variables and where they can be accessed
`LO-5.1.4.1`	K1	Recall that assignments to `@{list}` variables convert values to lists automatically
`LO-5.1.4.2`	K1	Recall that `@{list}` unpacks the values of a list variable when accessed
`LO-5.1.5.1`	K1	Recall that assignments to `&{dict}` variables automatically convert values to Robot Framework Dictionaries and enable dot-access
`LO-5.1.5.2`	K1	Recall that `&{dict}` unpacks to multiple key=value pairs when accessed
`LO-5.1.6`	K1	Recall that Robot Framework provides access to execution information via Built-In variables
`LO-5.2.1`	K2	Understand the purpose and basic concept of IF-Statements
`LO-5.2.4`	K2	Understand the purpose and basic concept of FOR Loops
`LO-5.2.5`	K2	Understand the purpose and basic concept of WHILE Loops
`LO-5.2.6`	K2	Understand the purpose and basic concept of the BREAK and CONTINUE statements