**PRACTICAL 1**

**AIM: Design test cases using Boundary value analysis**

**THEORY:** Boundary Value Analysis (BVA) is a testing technique used to identify errors at the

boundaries rather than within the ranges of input values. It involves testing at the edges of

input ranges, including maximum, minimum, just inside/outside boundaries, typical values,

and error values.

**CODE:**

def is\_valid\_age(age):

if 18 <= age <= 60:

return "Valid"

else:

return "Invalid"

test\_cases = [

(18, "Valid"),

(17, "Invalid"),

(19, "Valid"),

(59, "Valid"),

(60, "Valid"),

(61, "Invalid")

]

for i, (age, expected) in enumerate(test\_cases):

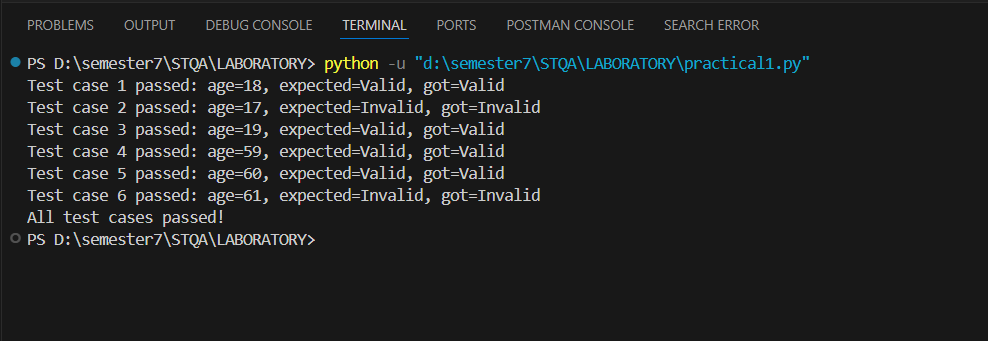
result = is\_valid\_age(age)

assert result == expected, f"Test case {i+1} failed: age={age}, expected={expected}, got={result}"

print(f"Test case {i+1} passed: age={age}, expected={expected}, got={result}")

print("All test cases passed!")

**OUTPUT:**

****

**PRACTICAL 2**

**AIM:** **Testcase Design Using Equivalence class Partitioning**

**THEORY:** Equivalence Class Partitioning (ECP) is a testing technique that divides input data into partitions of equivalent data from which test cases can be derived. The idea is to reduce the number of test cases to a manageable level while still ensuring adequate coverage. Each partition represents a set of values that are expected to be treated similarly by the software.

**CODE:**

def is\_valid\_age(age):

if 18 <= age <= 60:

return "Valid"

else:

return "Invalid"

test\_cases = [

(30, "Valid"),

(17, "Invalid"),

(10, "Invalid"),

(61, "Invalid"),

(70, "Invalid"),

]

for i, (age, expected) in enumerate(test\_cases):

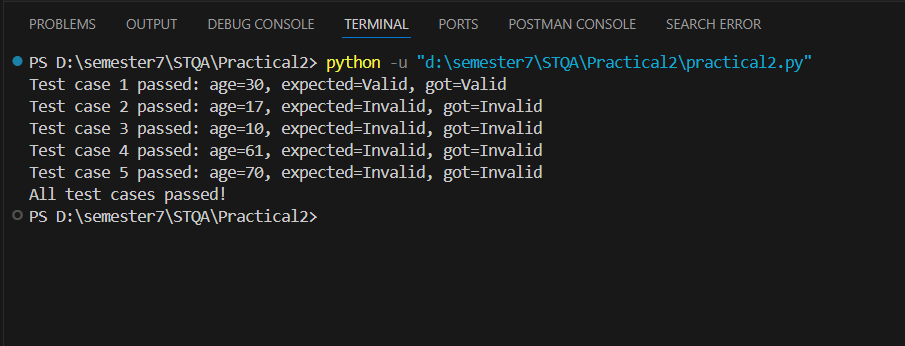
result = is\_valid\_age(age)

assert result == expected, f"Test case {i+1} failed: age={age}, expected={expected}, got={result}"

print(f"Test case {i+1} passed: age={age}, expected={expected}, got={result}")

print("All test cases passed!")

**OUTPUT:**

****

**PRACTICAL 3**

**AIM**: Design independent paths by calculating cyclomatic complexity using date problem

**THEORY:** Cyclomatic complexity is a software metric used to measure the complexity of a program. It is calculated by using the control flow graph of the program, where nodes represent the

commands/statements of the program, and edges represent the control flow between the nodes.

Here how to calculate cyclomatic complexity:

1. Identify the control flow graph of the program.

2. Count the number of edges (E) in the graph.

3. Count the number of nodes (N) in the graph.

4. Count the number of connected components (P) (usually P is 1 if the program is a single connected graph).

The cyclomatic complexity (V(G)) is then calculated using the formula: V(G)=E−N+2PV(G)

= E - N + 2PV(G)=E−N+2P

**CODE:**

def is\_leap\_year(year):

if (year % 4 == 0 and year % 100 != 0) or year % 400 == 0:

return True

else:

return False

def is\_valid\_date(day, month, year):

if year < 1:

return False

if month < 1 or month > 12:

return False

if day < 1:

return False

if month in {1, 3, 5, 7, 8, 10, 12}:

if day > 31:

return False

elif month in {4, 6, 9, 11}:

if day > 30:

return False

elif month == 2:

if is\_leap\_year(year):

if day > 29:

return False

else:

if day > 28:

return False

return True

print(is\_valid\_date(26, 6, 2024)) # True, 2024 is a leap year

print(is\_valid\_date(29, 2, 2021)) # False, 2021 is not a leap year

print(is\_valid\_date(31, 4, 2021)) # False, April has 30 days

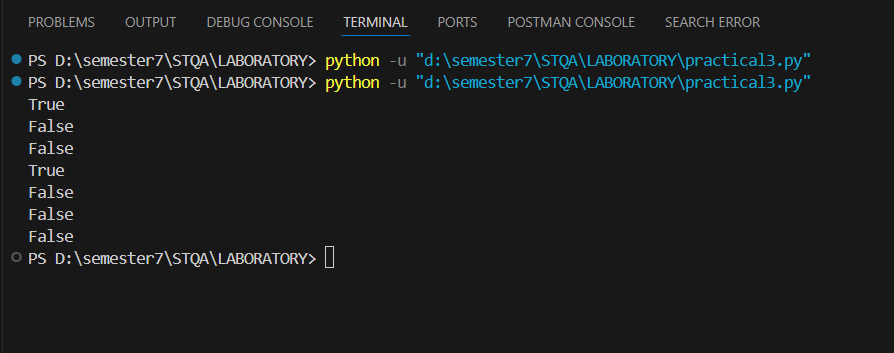
print(is\_valid\_date(31, 12, 2021)) # True, December has 31 days

print(is\_valid\_date(0, 1, 2021)) # False, day cannot be less than 1

print(is\_valid\_date(15, 13, 2021)) # False, month cannot be greater than 12

print(is\_valid\_date(15, 10, -1)) # False, year cannot be less than 1

**OUTPUT:**

****

**PRACTICAL 4**

Aim : Design test cases using Decision table

Program :

decision\_table = [

{"credit\_score": "Good", "income\_level": "High", "collateral": "Yes", "expected\_result": "Approve loan"},

{"credit\_score": "Good", "income\_level": "High", "collateral": "No", "expected\_result": "Approve loan"},

{"credit\_score": "Good", "income\_level": "Low", "collateral": "Yes", "expected\_result": "Approve loan"},

{"credit\_score": "Good", "income\_level": "Low", "collateral": "No", "expected\_result": "Deny loan"},

{"credit\_score": "Bad", "income\_level": "High", "collateral": "Yes", "expected\_result": "Approve loan"},

{"credit\_score": "Bad", "income\_level": "High", "collateral": "No", "expected\_result": "Deny loan"},

{"credit\_score": "Bad", "income\_level": "Low", "collateral": "Yes", "expected\_result": "Deny loan"},

{"credit\_score": "Bad", "income\_level": "Low", "collateral": "No", "expected\_result": "Deny loan"},

]

# Step 2: Define a function that simulates the loan approval process

def loan\_approval\_system(credit\_score, income\_level, collateral):

if credit\_score == "Good" and (income\_level == "High" or collateral == "Yes"):

return "Approve loan"

elif credit\_score == "Bad" and income\_level == "High" and collateral == "Yes":

return "Approve loan"

else:

return "Deny loan"

# Step 3: Generate and execute test cases

def run\_test\_cases(decision\_table):

for i, test\_case in enumerate(decision\_table):

credit\_score = test\_case["credit\_score"]

income\_level = test\_case["income\_level"]

collateral = test\_case["collateral"]

expected\_result = test\_case["expected\_result"]

# Get the actual result from the loan approval system

actual\_result = loan\_approval\_system(credit\_score, income\_level, collateral)

# Check if the actual result matches the expected result

if actual\_result == expected\_result:

print(f"Test Case {i+1} PASSED")

else:

print(f"Test Case {i+1} FAILED: expected {expected\_result} but got {actual\_result}")

# Run the test cases

run\_test\_cases(decision\_table)

**Practical 6**

**AIM :** Understand The Automation Testing Approach (Theory Concept)

**Introduction**

Automation testing is a critical aspect of modern software development, aimed at enhancing the efficiency, accuracy, and coverage of software testing processes. This report delves into the theoretical concepts of automation testing, its significance, various approaches, tools, and best practices.

**Objectives**

The primary objectives of this report are:

* To understand the concept of automation testing.
* To explore different approaches and methodologies in automation testing.
* To identify popular tools used in automation testing.
* To discuss best practices and challenges associated with automation testing.

**Understanding Automation Testing**

**Definition**

Automation testing involves using automated tools and scripts to perform tests on software applications. Unlike manual testing, where testers manually execute test cases, automation testing employs software to control the execution of tests and compare actual outcomes with expected results.

**Significance**

* **Efficiency**: Automation testing significantly reduces the time required to execute repetitive test cases.
* **Accuracy**: It minimizes human error, ensuring more reliable and consistent test results.
* **Coverage**: Automated tests can cover more test scenarios, including edge cases, which might be impractical to test manually.
* **Reusability**: Test scripts can be reused across different versions of the software, reducing the effort needed for regression testing.

**Approaches to Automation Testing**

**Linear Scripting**

This is the simplest form of test automation, where test scripts are written in a linear fashion without any modularity. It's suitable for small projects but becomes difficult to maintain as the project grows.

* **Pros**: Quick to implement, no need for advanced programming skills.
* **Cons**: Difficult to maintain and scale, high redundancy.

**Modular Testing**

In this approach, the application is divided into smaller, manageable modules. Test scripts are developed for each module and then integrated to form a comprehensive test suite.

* **Pros**: Easier to maintain, reusable scripts, better scalability.
* **Cons**: Requires good planning and understanding of the application structure.

**Data-Driven Testing**

Data-driven testing involves separating test logic from test data. Test data is stored in external files (e.g., Excel, CSV), and the same test scripts are executed with different sets of data.

* **Pros**: High reusability, easy to manage large data sets, better test coverage.
* **Cons**: Requires more initial setup, complexity in managing data files.

**Keyword-Driven Testing**

This approach extends data-driven testing by using keywords to represent actions to be performed on the application. Test scripts use these keywords to drive the testing process.

* **Pros**: High reusability, easy for non-technical users to understand and modify, scalable.
* **Cons**: Requires a well-defined keyword library, initial setup can be complex.

**Hybrid Testing**

Hybrid testing combines multiple automation frameworks to leverage the benefits of each. For example, a hybrid approach may use data-driven and keyword-driven testing together.

* **Pros**: Flexibility, better maintainability, can handle complex testing scenarios.
* **Cons**: Requires thorough understanding and planning, can be complex to implement.

**Popular Automation Testing Tools**

1. **Selenium**: An open-source tool primarily used for web application testing. It supports multiple programming languages (Java, C#, Python) and browsers.
2. **JUnit**: A widely-used testing framework for Java applications, ideal for unit testing.
3. **TestNG**: Similar to JUnit but with more powerful features, such as annotations, parallel execution, and data-driven testing.
4. **Appium**: An open-source tool for mobile application testing, supporting both Android and iOS pluuatforms.
5. **Cucumber**: A tool that supports Behaviour-Driven Development (BDD), allowing tests to be written in plain language.

**Best Practices in Automation Testing**

* **Start Small**: Begin with automating high-value and high-risk test cases.
* **Maintainability**: Write clear, modular, and reusable test scripts.
* **Consistency**: Ensure consistent test environments to avoid discrepancies in test results.
* **Regular Updates**: Keep test scripts updated with application changes.
* **Reporting and Logging**: Implement comprehensive reporting and logging to facilitate debugging and analysis.
* **Continuous Integration**: Integrate automation testing with CI/CD pipelines to ensure regular and automatic execution of tests.

**Challenges in Automation Testing**

* **Initial Cost and Setup**: High initial investment in tools and setup.
* **Skill Requirements**: Requires skilled personnel with programming and testing knowledge.
* **Maintenance**: Continuous maintenance of test scripts is necessary to keep up with application changes.
* **Complex Scenarios**: Automating complex test scenarios can be challenging and time-consuming.

**Conclusion**

Automation testing is an indispensable part of modern software development, offering significant advantages in terms of efficiency, accuracy, and coverage. Understanding various automation testing approaches and employing best practices can help organizations overcome challenges and reap the full benefits of automation. As software complexity continues to grow, investing in robust automation frameworks and tools will be crucial for maintaining high-quality software products.

This report provides a comprehensive understanding of the automation testing approach, covering theoretical concepts, methodologies, tools, best practices, and challenges. By leveraging this knowledge, teams can implement effective automation testing strategies to enhance their software development processes.