Python Fundamentals & PyTest Framework – Step-by-Step Automation Guide

A comprehensive instructor-led training programme designed for QA engineers, automation beginners, and manual testers transitioning to automation roles.

INTRODUCTION

Why Python for Automation?

Simple Syntax

Python's readable, English-like syntax reduces the learning curve for automation beginners. Write tests faster with less code complexity.

Rich Ecosystem

Extensive libraries for testing including PyTest, Selenium, Playwright, Requests, and more. Everything you need is readily available.

Industry Standard

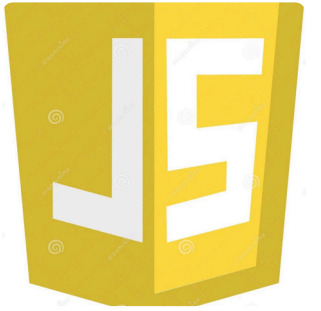
Used by major organisations worldwide for test automation. High demand for Python automation skills in QA roles.

Where Python is Used in QA/QE

Common Applications

- Web UI automation with Selenium and Playwright
- API testing with Requests and PyTest
- Mobile automation with Appium
- Performance testing script development
- Test data generation and manipulation
- CI/CD pipeline integration

Python vs Other Automation Languages



Python

Easy syntax, vast libraries, quick prototyping, ideal for beginners

Java

More verbose, strongly typed, enterprise adoption, steeper learning curve

JavaScript

Front-end focused, asynchronous nature, Node.js ecosystem, full-stack capability

Python strikes an optimal balance between simplicity and power, making it the preferred choice for automation beginners whilst remaining robust enough for enterprise solutions.

Real-World Example: Test Automation Script

Scenario

A QA engineer needs to verify login functionality across multiple user accounts. Python enables rapid test creation with minimal code.

Key Benefits

- Readable test logic
- Easy maintenance
- Reusable components
- Fast execution

```
# Simple login test example
def test_login_valid_user():
 username = "test@example.com"
 password = "SecurePass123"

 result = login(username, password)

 assert result == "success"
 assert dashboard_visible() == True
```

Real-World Example: Data Processing Script

Python excels at automating repetitive QA tasks such as test data preparation, log analysis, and report generation.

```
# Process test results from CSV file
import csv

def analyze_test_results(filename):
    passed = 0
    failed = 0

    with open(filename, 'r') as file:
        reader = csv.DictReader(file)
        for row in reader:
            if row['status'] == 'PASS':
                passed += 1
            else:
                failed += 1

    print(f"Passed: {passed}, Failed: {failed}")
    return passed, failed
```

This automation reduces manual effort from hours to seconds whilst eliminating human error in data analysis.

Knowledge Check: Introduction to Python

Question 1

Which of the following is NOT an advantage of Python for test automation?

A) Easy-to-read syntax
B) Extensive testing libraries
C) Requires compilation before execution
D) Strong community support

Answer: C – Python is interpreted, not compiled

Question 2

What type of testing can Python be used for?

A) Web UI automation only
B) API testing only
C) Multiple types including UI, API, and mobile
D) Database testing only

Answer: C – Python supports diverse testing needs

Interview Keywords: Python Fundamentals

Interpreted Language

Python code is executed line-by-line without requiring compilation, enabling rapid development and testing cycles.

Scripting Language

Designed for writing scripts to automate tasks, making it ideal for test automation and DevOps workflows.

Readability

Emphasises clean, readable code through significant whitespace and clear syntax conventions.

SECTION 2

Python Syntax Basics

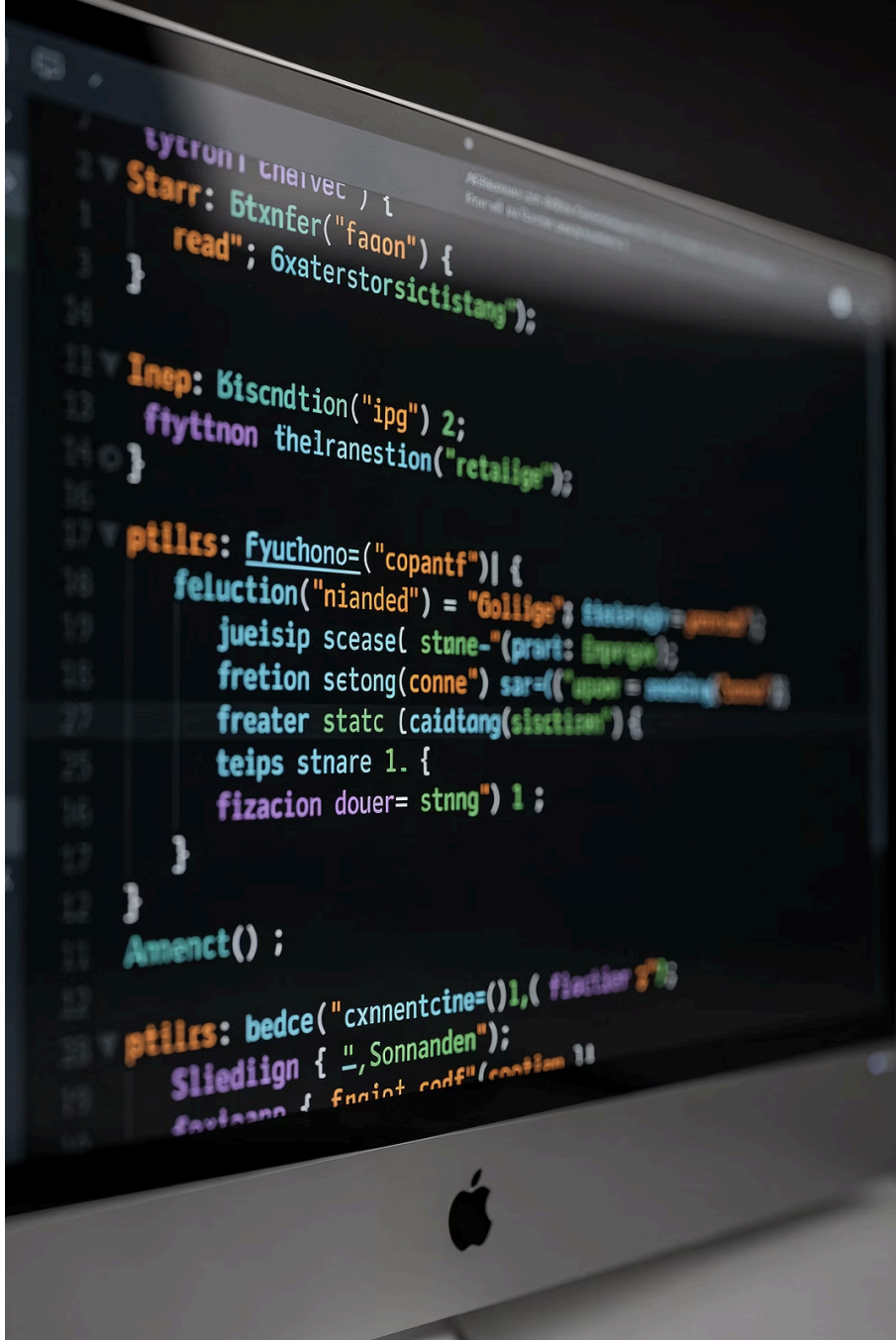
Python's syntax is designed for readability and simplicity. Unlike many languages, Python uses indentation to define code blocks rather than brackets or keywords.

Key Characteristics

- Indentation matters (4 spaces standard)
- No semicolons required
- Comments use # symbol
- Case-sensitive language

```
# This is a comment
def greet(name):
 message = "Hello, " + name
 print(message)
 return message

greet("Tester")
```



Variables and Data Types

Variables in Python are dynamically typed, meaning you don't need to declare their type explicitly. Python infers the type based on the assigned value.

01

Variable Declaration

Simply assign a value using the = operator

02

Type Inference

Python automatically determines the data type

03

Dynamic Reassignment

Variables can change type during execution

```
# Variable examples
test_name = "Login Test" # String
test_count = 42 # Integer
pass_rate = 95.5 # Float
is_automated = True # Boolean
```

Core Data Types for Automation

Integer (int)

Whole numbers for counters, test IDs, status codes

```
test_id = 1001
```

Float

Decimal numbers for response times, percentages

```
response_time = 2.5
```

String

Text data for usernames, URLs, test data

```
username = "tester"
```

Boolean

True/False for test results, conditions

```
test_passed = True
```

Collection Data Types

List

Ordered, mutable collection – ideal for test data sets

```
test_users = [
    "user1@test.com",
    "user2@test.com",
    "user3@test.com"
]
```

Dictionary

Key-value pairs – perfect for test data and configurations

```
user_data = {
    "username": "testuser",
    "password": "Test@123",
    "role": "admin",
    "active": True
}
```

Tuple

Ordered, immutable collection – for fixed test configurations

```
credentials = (
    "admin",
    "password123"
)
```

Trainer Note: Emphasise that dictionaries are extensively used in API testing for request/response handling and test data management.

Input and Output Operations

Python provides simple methods for user interaction and displaying results, essential for creating interactive test scripts and logging test outcomes.

Output with print()

```
# Basic output
print("Test execution started")

# Formatted output
test_name = "Login Test"
status = "PASSED"
print(f"{test_name}: {status}")

# Multiple values
print("Total:", 10, "Passed:", 8)
```

Input with input()

```
# Get user input
environment = input("Enter env: ")

# Convert input type
num_tests = int(input("Tests: "))

# Use in test script
username = input("Username: ")
password = input("Password: ")
```

Python Operators

Arithmetic

```
+ - * /
% // **
```

Mathematical operations for calculations

Comparison

```
== != > <
>= <=
```

Compare values in assertions

Logical

```
and or not
```

Combine conditions in tests

```
# Operator examples in testing context
total_tests = 100
passed_tests = 85
failed_tests = total_tests - passed_tests

pass_rate = (passed_tests / total_tests) * 100

if pass_rate >= 90 and failed_tests < 15:
 print("Quality threshold met")
```

Practical Example: User Input Validation

Real-world scenario demonstrating data types, operators, and basic logic for validating test data inputs.

```
# Validate user registration data
username = input("Enter username: ")
age = int(input("Enter age: "))
email = input("Enter email: ")

# Validation checks
is_valid_username = len(username) >= 5 and len(username) <= 20
is_valid_age = age >= 18 and age <= 100
is_valid_email = "@" in email and "." in email

# Overall validation
if is_valid_username and is_valid_age and is_valid_email:
 print("PASS: All validation checks passed")
else:
 print("FAIL: Validation failed")
 if not is_valid_username:
 print(" - Username must be 5-20 characters")
 if not is_valid_age:
 print(" - Age must be 18-100")
 if not is_valid_email:
 print(" - Email format invalid")
```

Practical Example: Data Comparison

Comparing expected versus actual results – a fundamental skill in test automation using comparison operators and data types.

```
# Test case: Verify API response data
expected_status_code = 200
actual_status_code = 200

expected_response_time = 2.0
actual_response_time = 1.8

expected_user_count = 50
actual_user_count = 50

# Perform comparisons
status_match = expected_status_code == actual_status_code
performance_acceptable = actual_response_time <= expected_response_time
count_match = expected_user_count == actual_user_count

# Test result
test_passed = status_match and performance_acceptable and count_match

print(f"Test Result: {'PASS' if test_passed else 'FAIL'}")
print(f"Status Code Check: {status_match}")
print(f"Performance Check: {performance_acceptable}")
print(f"User Count Check: {count_match}")
```

Knowledge Check: Python Basics

Question 1

Which data type would you use to store a list of test case IDs that should not be modified?

A) List
B) Dictionary
C) Tuple
D) Set

Answer: C – Tuples are immutable, making them ideal for fixed data

Question 2

What will be the result of: 10 // 3 in Python?

A) 3.33
B) 3
C) 4
D) Error

Answer: B – Floor division (//) returns the integer quotient

SECTION 3

Control Statements: Conditional Logic

Conditional statements allow your test scripts to make decisions based on conditions, enabling dynamic test execution and intelligent error handling.

Basic Structure

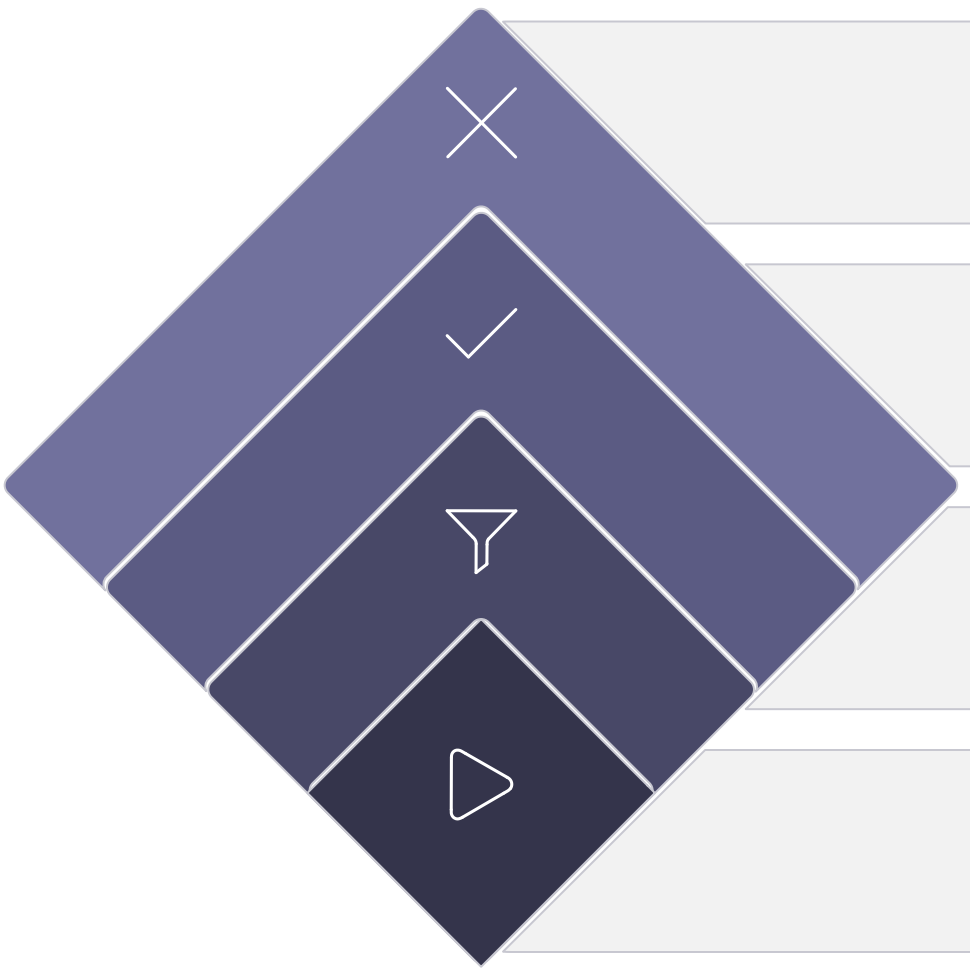
```
if condition:
    # Execute if true
elif another_condition:
    # Execute if first false
else:
    # Execute if all false
```

Testing Example

```
response_code = 404

if response_code == 200:
 print("PASS: Success")
elif response_code == 404:
 print("FAIL: Not found")
else:
 print("FAIL: Error")
```

Decision Making Flow



False Path

Execute alternative block

True Path

Execute matching block

Check Condition

Evaluate conditional expression

Start

Begin test sequence

Conditional statements create decision points in your test execution. The programme evaluates conditions sequentially and executes the first matching block, then continues to the next statement after the entire if-elif-else structure.

Looping Statements: For Loops

For loops iterate over sequences, allowing you to execute the same test logic across multiple data points efficiently.

Common Use Cases

- Test multiple user accounts
- Validate list of URLs
- Process test data sets
- Iterate through test results

Trainer Note: Emphasise data-driven testing concepts

```
# Loop through test users
users = [
 "user1@test.com",
 "user2@test.com",
 "user3@test.com"
]

for user in users:
 result = test_login(user)
 print(f"{user}: {result}")

# Loop with range
for i in range(5):
 print(f"Test iteration {i+1}")
```

Looping Statements: While Loops

While loops continue executing as long as a condition remains true, useful for scenarios requiring dynamic repetition based on runtime conditions.

```
# Retry logic example
max_retries = 3
attempt = 0
success = False

while attempt < max_retries and not success:
    attempt += 1
    print(f"Attempt {attempt}")
    success = perform_test()

if success:
    print("Test passed")
else:
    print("Test failed after retries")
```

When to Use While Loops

- Retry mechanisms in tests
- Polling for element visibility
- Waiting for API responses
- Unknown iteration count scenarios

Important: Always ensure while loops have a clear exit condition to avoid infinite loops

Loop Control: Break and Continue

break Statement

Immediately exits the loop entirely, regardless of the loop condition

```
for test in test_suite:
result = execute(test)
if result == "CRITICAL_FAIL":
print("Critical failure")
break  # Stop all tests
```

continue Statement

Skips the current iteration and moves to the next one

```
for test in test_suite:
 if test.skip:
 continue # Skip this test
 result = execute(test)
 log_result(result)
```

Case Study: Test Execution Data Loop

Real-world scenario demonstrating how loops enable data-driven testing by executing the same test logic across multiple input datasets.

```
# Test data set
login_test_data = [
 {"username": "admin@test.com", "password": "Admin@123", "expected": "success"},
 {"username": "user@test.com", "password": "User@123", "expected": "success"},
 {"username": "invalid@test.com", "password": "wrong", "expected": "failure"},
 {"username": "", "password": "", "expected": "failure"},
 {"username": "test@test.com", "password": "short", "expected": "failure"}
]

# Execute test for each data set
passed = 0
failed = 0

for test_case in login_test_data:
 actual_result = login(test_case["username"], test_case["password"])
 expected = test_case["expected"]

 if actual_result == expected:
 passed += 1
 print(f"PASS: {test_case['username']}")
 else:
 failed += 1
 print(f"FAIL: {test_case['username']} - Expected {expected}, got {actual_result}")

print(f"\nSummary: {passed} passed, {failed} failed")
```

Knowledge Check: Control Statements

Question 1

What is the primary difference between 'break' and 'continue' statements?

A) break exits the loop, continue skips to next iteration
B) continue exits the loop, break skips to next iteration
C) They perform the same function
D) break is for while loops, continue for for loops

Answer: A – break terminates the loop; continue skips current iteration

Question 2

Which loop type is best suited for implementing retry logic in test automation?

A) for loop with range()
B) while loop with condition
C) Nested for loops
D) List comprehension

Answer: B – while loops allow condition-based repetition ideal for retries

SECTION 4

Functions in Python

Functions are reusable blocks of code that perform specific tasks. In test automation, functions enable you to write test logic once and reuse it across multiple test cases, improving maintainability and reducing redundancy.

Reusability

Write once, use many times across different tests

Maintainability

Update logic in one place rather than throughout codebase

Readability

Break complex tests into named, understandable units

Function Syntax and Structure

Basic Anatomy

```
def function_name(parameters):
    """Docstring describes function"""
    # Function body
    # Perform operations
    return result
```

Key Components

- **def keyword:** Declares a function
- **Parameters:** Input values (optional)
- **Docstring:** Function documentation
- **Return:** Output value (optional)

Simple Example

```
def verify_status_code(actual, expected):
 """Compare actual vs expected code"""
 if actual == expected:
 return "PASS"
 else:
 return "FAIL"

# Usage
result = verify_status_code(200, 200)
print(result) # Output: PASS
```

Parameters and Return Values

Functions can accept input parameters and return output values, creating flexible, reusable test components.

1

Input Parameters

Data passed into the function

2

Processing

Function logic executes

3

Return Value

Result sent back to caller

```
def calculate_pass_rate(total_tests, passed_tests):
 """Calculate test pass rate percentage"""
 if total_tests == 0:
 return 0
 pass_rate = (passed_tests / total_tests) * 100
 return round(pass_rate, 2)

# Function call with arguments
rate = calculate_pass_rate(100, 87)
print(f"Pass rate: {rate}%") # Output: Pass rate: 87.0%
```

Real-World Example: Reusable Login Function

A practical demonstration of function reusability in test automation. This login function can be called from multiple test cases, ensuring consistent login logic across your entire test suite.

```
def login_to_application(username, password, expected_result="success"):
 """
 Perform login operation and verify result

 Parameters:
 username: User's email or username
 password: User's password
 expected_result: Expected outcome (success/failure)

 Returns:
 Boolean indicating if test passed
 """
 print(f"Attempting login: {username}")

 # Perform login operation (simplified)
 if username and password and len(password) >= 8:
 actual_result = "success"
 else:
 actual_result = "failure"

 # Verify result
 test_passed = (actual_result == expected_result)

 if test_passed:
 print(f"✓ PASS: Login {expected_result} as expected")
 else:
 print(f"  FAIL: Expected {expected_result}, got {actual_result}")

 return test_passed

# Reuse across multiple test cases
login_to_application("user@test.com", "ValidPass123", "success")
login_to_application("", "password", "failure")
login_to_application("user@test.com", "short", "failure")
```

Knowledge Check: Functions

Question 1

What is the primary benefit of using functions in test automation?

A) Functions make code run faster
B) Functions enable code reusability and maintainability
C) Functions are required by PyTest
D) Functions automatically fix bugs

Answer: B – Reusability and maintainability are core benefits

Question 2

Which keyword is used to send a value back from a function?

A) send
B) output
C) return
D) yield

Answer: C – return sends values back to the function caller

Interview Keywords: Functions

Reusability

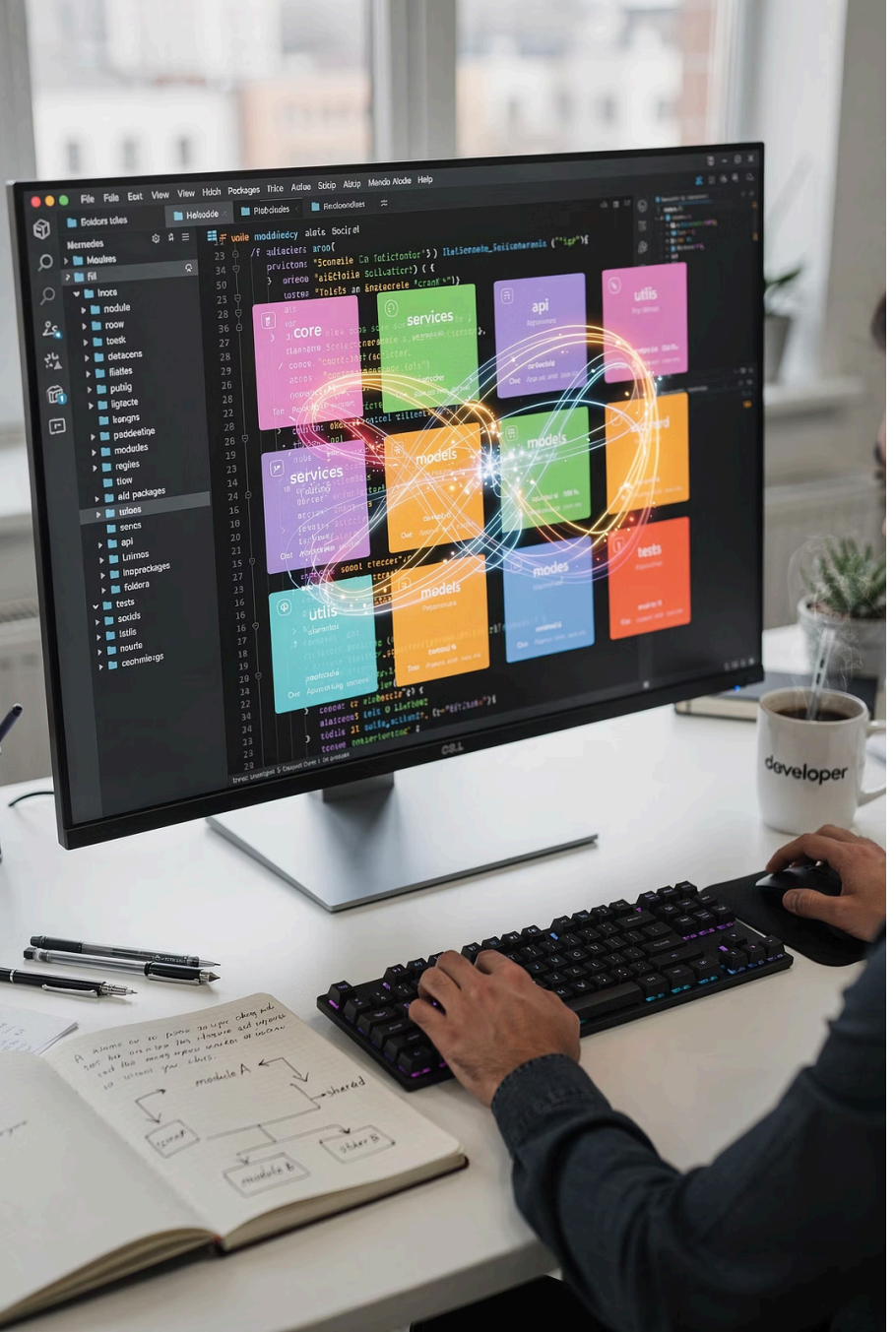
Write test logic once and invoke it multiple times across different test scenarios, reducing code duplication and maintenance effort.

Modularity

Breaking complex test suites into smaller, independent functions that can be developed, tested, and maintained separately.

Abstraction

Hiding implementation details behind simple function interfaces, making test code more readable and easier to understand.



SECTION 5

Python Modules & Packages

Modules and packages help organise code into logical units, essential for maintaining large test automation projects. They enable code reuse across multiple test files and create clear project structure.

01

Module

A single Python file containing related functions, classes, and variables

02

Package

A directory containing multiple modules and an __init__.py file

03

Library

Collection of packages providing extensive functionality

Understanding Modules

A module is simply a Python file (.py) containing definitions and statements. Modules allow you to logically organise your test automation code.

Why Modules Matter

- Separate concerns logically
- Reuse code across test files
- Improve code organisation
- Enable team collaboration
- Simplify maintenance

Example Module Structure

```
# File: test_helpers.py
def verify_response(actual, expected):
 return actual == expected

def log_result(test_name, result):
 print(f"{test_name}: {result}")

# File: test_login.py
import test_helpers

result = test_helpers.verify_response(
 200, 200
)
test_helpers.log_result(
 "Status Check", result
)
```

The Import Statement

Python's import statement loads modules into your code, making their functions and variables available for use.

Import Entire Module

```
import math

result = math.sqrt(16)
```

Import Specific Items

```
from math import sqrt, pi

result = sqrt(16)
```

Import with Alias

```
import datetime as dt

now = dt.datetime.now()
```

Import All (Avoid)

```
from math import *

# Not recommended
```

Best Practice: Use specific imports or module imports with aliases for clarity and avoiding namespace conflicts

Built-in Modules for Testing

datetime Module

```
import datetime

# Timestamp for logs
now = datetime.datetime.now()
print(f"Test run: {now}")

# Date calculations
start = datetime.date(2024, 1, 1)
end = datetime.date(2024, 12, 31)
duration = end - start
print(f"Days: {duration.days}")
```

random Module

```
import random

# Generate test data
user_id = random.randint(1000, 9999)
print(f"User ID: {user_id}")

# Select random test case
test_cases = ["TC001", "TC002",
 "TC003"]
selected = random.choice(test_cases)
print(f"Running: {selected}")
```

Creating Custom Modules

Organise your test automation code by creating custom modules that group related functionality together.

```
# File: api_helpers.py
"""Helper functions for API testing"""

def build_headers(auth_token):
 """Construct API request headers"""
 return {
 "Authorization": f"Bearer {auth_token}",
 "Content-Type": "application/json"
 }

def verify_status_code(response, expected=200):
 """Verify HTTP status code"""
 return response.status_code == expected

def extract_json_value(response, key):
 """Extract value from JSON response"""
 return response.json().get(key)

# File: test_api.py
"""API test suite"""
import api_helpers

# Use imported functions
headers = api_helpers.build_headers("abc123")
status_ok = api_helpers.verify_status_code(response, 200)
user_id = api_helpers.extract_json_value(response, "user_id")
```

Case Study: Modular Test Project Structure

Professional test automation projects use modular structure to separate concerns and improve maintainability. This case study demonstrates a typical organisation pattern.

Project Structure

```
test_project/
├── helpers/
│   ├── __init__.py
│   ├── api_helper.py
│   ├── db_helper.py
│   └── ui_helper.py
├── test_data/
│   ├── __init__.py
│   └── users.py
├── tests/
│   ├── test_login.py
│   ├── test_api.py
│   └── test_checkout.py
└── config.py
```

Benefits of This Structure

- Clear separation of test logic from helpers
- Reusable helper functions across tests
- Centralised test data management
- Easy to locate and modify code
- Scalable as project grows
- Team members understand organisation

Trainer Note: Demonstrate creating and importing custom modules in live coding session

Knowledge Check: Modules & Packages

Question 1

What is the difference between a module and a package in Python?

A) No difference, they're the same thing
B) Module is a single file, package is a directory with modules
C) Module is for tests, package is for production code
D) Package is deprecated, use modules instead

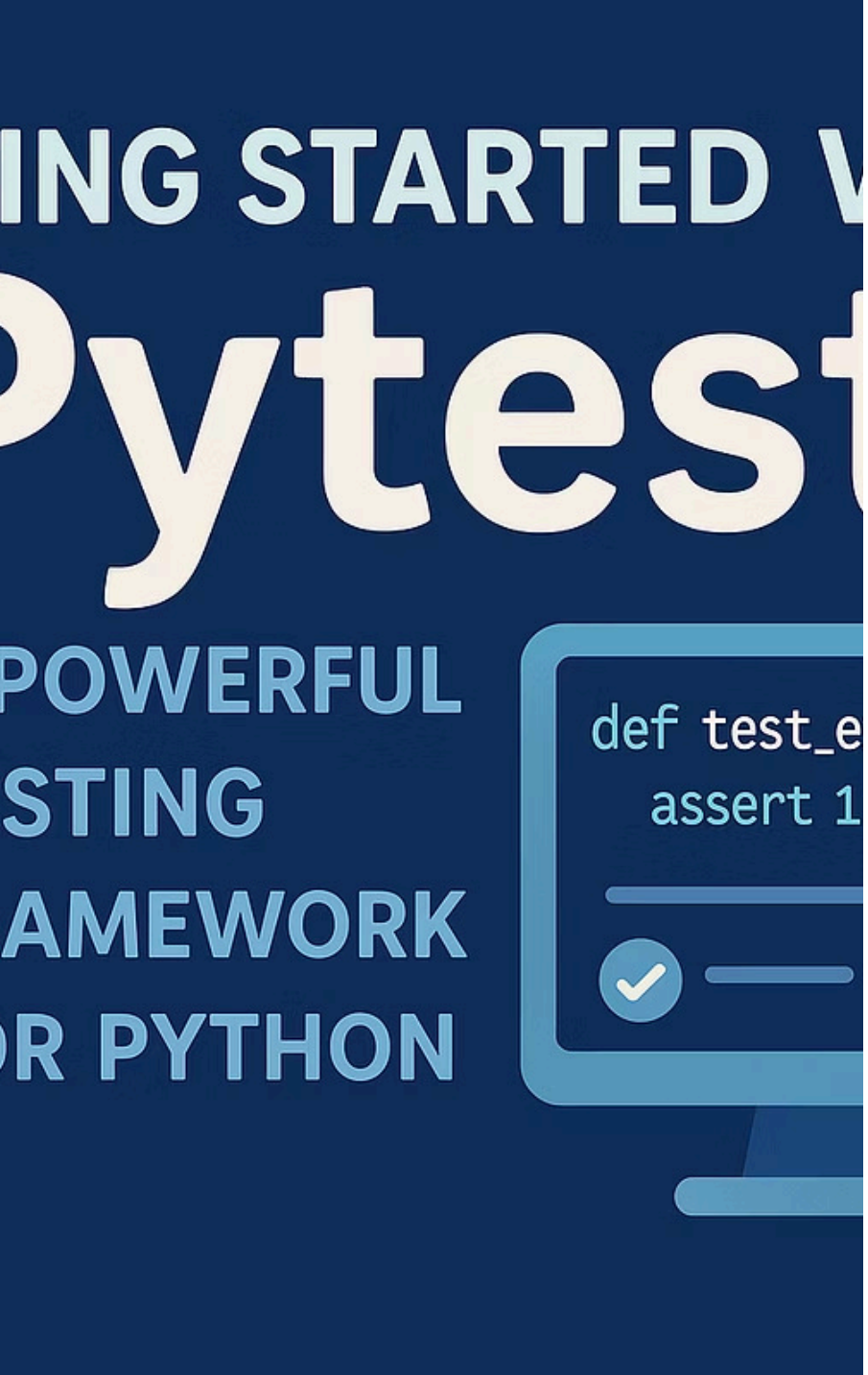
Answer: B – Module = single .py file; Package = directory with __init__.py

Question 2

Which import statement is considered best practice?

A) from module import *
B) import module or from module import specific_item
C) Imports should be avoided
D) Only built-in modules can be imported

Answer: B – Specific imports or module imports provide clarity



SECTION 6

Introduction to PyTest Framework

PyTest is a mature, feature-rich testing framework that makes it easy to write small, readable tests whilst scaling to support complex functional testing for applications and libraries.

Simple & Powerful

Minimal boilerplate code with powerful features when you need them

Auto-Discovery

Automatically finds and runs your tests based on naming conventions

Rich Assertions

Detailed failure messages showing exactly what went wrong

Why PyTest is Popular for Automation

Key Advantages

- **Minimal syntax:** Less boilerplate than unittest
- **Powerful fixtures:** Flexible test setup and teardown
- **Parametrisation:** Run same test with multiple data sets
- **Plugin ecosystem:** Extensive plugins for various needs
- **Parallel execution:** Run tests concurrently
- **Detailed reports:** Clear, actionable test results
- **Industry adoption:** Used by major organisations globally

PyTest Architecture Overview

PyTest follows a systematic approach to test execution. It discovers test files and functions based on naming patterns, collects them into a test session, executes each test whilst managing fixtures and setup/teardown, and finally generates comprehensive reports showing results and failures.

Trainer Note: Emphasise that understanding this flow helps debug test execution issues