Phase 1: Python Fundamentals. I'll create comprehensive learning materials for each topic, and we'll move through them one by one.

Let's begin with the first topic:

Python Basic Syntax & Data Types

Variables in Python

Variables are containers for storing data values. Python has no command for declaring a variable - you create one by assigning a value.

```
# Variable assignment
name = "John"
age = 25
height = 5.9
is_student = True

print(name)    # Output: John
print(age)     # Output: 25
print(height)  # Output: 5.9
print(is_student)  # Output: True
```

Variable Naming Rules

- Must start with a letter or underscore
- Can contain letters, numbers, and underscores
- Case-sensitive
- Cannot use Python keywords

```
# Valid variable names
user_name = "Alice"
_private = "secret"
age2 = 30

# Invalid variable names (will cause errors)
# 2age = 30      # Cannot start with number
# user-name = "" # Cannot contain hyphens
# class = ""     # Cannot use Python keywords
```

Data Types

1. Numbers

```
# Integer
count = 10
negative_num = -5

# Float
price = 19.99
temperature = -15.5

# Complex (rarely used in automation)
complex_num = 3 + 4j

# Type checking
```

```
print(type(count))      # <class 'int'>
print(type(price))      # <class 'float'>
```

2. Strings

```
# String creation
single_quote = 'Hello World'
double_quote = "Hello World"
triple_quote = """This is a
multi-line string"""

# String operations
first_name = "John"
last_name = "Doe"
full_name = first_name + " " + last_name  # Concatenation
print(full_name)  # Output: John Doe

# String formatting
age = 25
message = f"My name is {first_name} and I am {age} years old"
print(message)  # Output: My name is John and I am 25 years old

# String methods (useful for automation)
email = "  USER@EXAMPLE.COM  "
print(email.lower())        # user@example.com
print(email.upper())        # USER@EXAMPLE.COM
print(email.strip())        # USER@EXAMPLE.COM (removes spaces)
print(email.replace("@", " AT "))  # USER AT EXAMPLE.COM
```

3. Lists

Lists are ordered collections that can store multiple items.

```
# Creating lists
fruits = ["apple", "banana", "orange"]
numbers = [1, 2, 3, 4, 5]
mixed_list = ["John", 25, True, 5.9]

# Accessing elements (0-indexed)
print(fruits[0])    # apple (first element)
print(fruits[-1])   # orange (last element)

# List operations
fruits.append("grape")           # Add to end
fruits.insert(1, "mango")       # Insert at position 1
fruits.remove("banana")         # Remove specific item
popped_item = fruits.pop()      # Remove and return last item

print(fruits)  # Current list after operations
print(f"Popped item: {popped_item}")

# List slicing
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(numbers[2:5])    # [3, 4, 5] (elements from index 2 to 4)
print(numbers[:3])     # [1, 2, 3] (first 3 elements)
print(numbers[5:])     # [6, 7, 8, 9, 10] (from index 5 to end)
```

4. Tuples

Tuples are ordered collections that cannot be changed (immutable).

```
# Creating tuples
coordinates = (10, 20)
colors = ("red", "green", "blue")
single_item_tuple = ("item",)  # Note the comma for single item

# Accessing elements
print(coordinates[0])  # 10
print(colors[1])       # green

# Tuple unpacking (very useful!)
x, y = coordinates
print(f"X: {x}, Y: {y}")  # X: 10, Y: 20

# Use cases in automation
browser_info = ("Chrome", "98.0", "Windows")
browser_name, version, os = browser_info
```

5. Sets

Sets are unordered collections of unique items.

```
# Creating sets
unique_numbers = {1, 2, 3, 4, 5}
duplicate_numbers = {1, 1, 2, 2, 3}  # Duplicates automatically removed
print(duplicate_numbers)  # {1, 2, 3}

# Set operations
browsers = {"Chrome", "Firefox", "Safari"}
browsers.add("Edge")
browsers.remove("Safari")
print(browsers)

# Set operations (useful for data comparison)
test_env_browsers = {"Chrome", "Firefox"}
prod_env_browsers = {"Chrome", "Firefox", "Safari", "Edge"}

common_browsers = test_env_browsers & prod_env_browsers  # Intersection
print(f"Common browsers: {common_browsers}")
```

6. Dictionaries

Dictionaries store key-value pairs.

```
# Creating dictionaries
user = {
    "name": "John Doe",
    "age": 30,
    "email": "john@example.com",
    "is_active": True
}

# Accessing values
print(user["name"])           # John Doe
print(user.get("age"))        # 30
print(user.get("phone", "N/A"))  # N/A (default if key doesn't exist)

# Adding/updating values
user["phone"] = "123-456-7890"
user["age"] = 31  # Update existing key

# Dictionary methods
print(user.keys())    # All keys
```

```
print(user.values())  # All values
print(user.items())   # All key-value pairs

# Use case in automation - test data
test_data = {
    "valid_user": {
        "username": "testuser",
        "password": "password123"
    },
    "invalid_user": {
        "username": "wronguser",
        "password": "wrongpass"
    }
}

print(test_data["valid_user"]["username"])  # testuser
```

Type Conversion

```
# String to number
age_str = "25"
age_int = int(age_str)
price_str = "19.99"
price_float = float(price_str)

# Number to string
count = 10
count_str = str(count)

# Boolean conversion
print(bool(1))      # True
print(bool(0))      # False
print(bool(""))     # False (empty string)
print(bool("text")) # True (non-empty string)
```

Practical Examples for Automation

```
# Example 1: Storing test user data
test_users = [
    {"username": "admin", "password": "admin123", "role": "administrator"},
    {"username": "user1", "password": "user123", "role": "regular"},
    {"username": "guest", "password": "guest123", "role": "guest"}
]

# Example 2: Browser configurations
browser_configs = {
    "chrome": {"driver_path": "/path/to/chromedriver", "headless": True},
    "firefox": {"driver_path": "/path/to/geckodriver", "headless": False}
}

# Example 3: Test URLs
test_urls = (
    "https://example.com/login",
    "https://example.com/dashboard",
    "https://example.com/profile"
)

# Example 4: Expected vs Actual results comparison
expected_titles = {"Login Page", "Dashboard", "User Profile"}
actual_titles = {"Login Page", "Dashboard", "Settings"}
missing_titles = expected_titles - actual_titles
```

```
print(f"Missing titles: {missing_titles}")  # {'User Profile'}
```

Practice Exercises

1. **Create a dictionary** containing information about 3 different browsers with their versions
2. **Create a list** of test URLs and practice accessing them by index
3. **Use string formatting** to create dynamic error messages
4. **Practice tuple unpacking** with coordinate data
5. **Use sets** to find common elements between two lists of supported features

Key Takeaways

- Variables don't need type declaration in Python
- Strings are immutable and have many useful methods
- Lists are mutable and ordered
- Tuples are immutable and ordered
- Sets contain unique items only
- Dictionaries store key-value pairs
- Use appropriate data types based on your needs in test automation

- Variables and naming conventions
- All major data types (numbers, strings, lists, tuples, sets, dictionaries)
- Type conversion
- Practical examples relevant to automation testing
- Practice exercises

Key points to remember:

- Python is dynamically typed (no need to declare variable types)
- Different data types serve different purposes in automation (lists for test data, dictionaries for configurations, etc.)
- String methods like `strip()`, `lower()`, `replace()` are very useful in test automation
- Lists and dictionaries will be your most-used data structures in Selenium automation

Practice suggestions:

1. Try creating different types of variables
2. Experiment with string formatting using f-strings
3. Create test data structures using lists and dictionaries
4. Practice accessing and modifying data in these structures

Python Control Flow

Control flow statements allow you to control the execution of your code based on conditions and repeat actions efficiently.

Conditional Statements (if/elif/else)

Basic if Statement

```
# Simple if statement
age = 18
if age >= 18:
    print("You are eligible to vote")

# if-else statement
password = "admin123"
if password == "admin123":
    print("Login successful")
else:
    print("Invalid password")
```

Multiple Conditions with elif

```
# Multiple conditions using elif
score = 85

if score >= 90:
    grade = "A"
elif score >= 80:
    grade = "B"
elif score >= 70:
    grade = "C"
elif score >= 60:
    grade = "D"
else:
    grade = "F"

print(f"Your grade is: {grade}")
```

Logical Operators

```
# AND operator
username = "admin"
password = "password123"

if username == "admin" and password == "password123":
    print("Access granted")
else:
    print("Access denied")

# OR operator
browser = "Chrome"
if browser == "Chrome" or browser == "Firefox":
    print("Supported browser")
else:
    print("Unsupported browser")

# NOT operator
is_logged_in = False
if not is_logged_in:
    print("Please log in first")
```

Nested Conditions

```
# Nested if statements
```

```
user_type = "admin"
is_active = True

if user_type == "admin":
    if is_active:
        print("Admin access granted")
    else:
        print("Admin account is inactive")
else:
    print("Regular user access")
```

Loops

For Loops

For loops are used to iterate over sequences (lists, tuples, strings, etc.)

```
# Basic for loop with list
browsers = ["Chrome", "Firefox", "Safari", "Edge"]
for browser in browsers:
    print(f"Testing on {browser}")

# Output:
# Testing on Chrome
# Testing on Firefox
# Testing on Safari
# Testing on Edge

# For loop with range
for i in range(5):
    print(f"Test iteration: {i}")
# Output: 0, 1, 2, 3, 4

# Range with start and end
for i in range(1, 6):
    print(f"Test case {i}")
# Output: 1, 2, 3, 4, 5

# Range with step
for i in range(0, 10, 2):
    print(f"Even number: {i}")
# Output: 0, 2, 4, 6, 8
```

While Loops

While loops continue executing as long as a condition is true.

```
# Basic while loop
count = 0
while count < 3:
    print(f"Attempt {count + 1}")
    count += 1

# While loop for retrying operations
max_retries = 3
current_retry = 0
success = False

while current_retry < max_retries and not success:
    print(f"Attempting login... (Attempt {current_retry + 1})")
    # Simulate login attempt
```

```
    login_result = True  # This would be your actual login logic

    if login_result:
        success = True
        print("Login successful!")
    else:
        current_retry += 1
        print("Login failed, retrying...")

if not success:
    print("Max retries reached. Login failed.")
```

Loop Control Statements

```
# break - exits the loop
test_data = ["valid_data", "invalid_data", "corrupted_data", "more_data"]

for data in test_data:
    if data == "corrupted_data":
        print("Corrupted data found, stopping tests")
        break
    print(f"Processing: {data}")

# continue - skips to next iteration
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

print("Odd numbers only:")
for num in numbers:
    if num % 2 == 0:  # Skip even numbers
        continue
    print(num)
```

Practical Automation Examples

Example 1: Validating Multiple Form Fields

```
# Form validation example
form_data = {
    "username": "testuser",
    "email": "test@example.com",
    "password": "password123",
    "confirm_password": "password123"
}

validation_errors = []

# Validate each field
if not form_data["username"]:
    validation_errors.append("Username is required")
elif len(form_data["username"]) < 3:
    validation_errors.append("Username must be at least 3 characters")

if "@" not in form_data["email"]:
    validation_errors.append("Invalid email format")

if len(form_data["password"]) < 8:
    validation_errors.append("Password must be at least 8 characters")

if form_data["password"] != form_data["confirm_password"]:
    validation_errors.append("Passwords do not match")
```

```
# Check validation results
if validation_errors:
    print("Validation failed:")
    for error in validation_errors:
        print(f"- {error}")
else:
    print("All validations passed!")
```

Example 2: Testing Multiple Browsers

```
# Multi-browser testing setup
browsers = ["Chrome", "Firefox", "Edge"]
test_urls = [
    "https://example.com/login",
    "https://example.com/dashboard",
    "https://example.com/profile"
]

test_results = []

for browser in browsers:
    print(f"\n--- Testing with {browser} ---")

    for url in test_urls:
        print(f"Testing URL: {url}")

        # Simulate test execution
        test_passed = True  # This would be your actual test logic

        if test_passed:
            result = f"{browser} - {url}: PASSED"
        else:
            result = f"{browser} - {url}: FAILED"

        test_results.append(result)
        print(result)

# Summary
print("\n--- Test Summary ---")
passed_tests = [result for result in test_results if "PASSED" in result]
failed_tests = [result for result in test_results if "FAILED" in result]

print(f"Total tests: {len(test_results)}")
print(f"Passed: {len(passed_tests)}")
print(f"Failed: {len(failed_tests)}")
```

Example 3: Retry Mechanism for Flaky Tests

```
# Retry mechanism for unreliable operations
def execute_test_with_retry(test_name, max_retries=3):
    for attempt in range(1, max_retries + 1):
        print(f"Executing {test_name} - Attempt {attempt}")

        # Simulate test execution (replace with actual test)
        import random
        test_result = random.choice([True, False])  # Random pass/fail

        if test_result:
            print(f"{test_name} PASSED on attempt {attempt}")
            return True
        else:
            print(f"{test_name} FAILED on attempt {attempt}")
```

```
            if attempt < max_retries:
                print("Retrying...")
            else:
                print(f"{test_name} FAILED after {max_retries} attempts")
                return False

# Execute tests with retry
test_cases = ["Login Test", "Navigation Test", "Form Submission Test"]

for test_case in test_cases:
    success = execute_test_with_retry(test_case)
    if not success:
        print(f"⚠  {test_case} requires manual investigation")
    print("-" * 40)
```

List Comprehensions (Bonus - Modern Python)

List comprehensions provide a concise way to create lists based on existing lists.

```
# Traditional way
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = []
for num in numbers:
    if num % 2 == 0:
        even_numbers.append(num)

# List comprehension way (more Pythonic)
even_numbers = [num for num in numbers if num % 2 == 0]
print(even_numbers)  # [2, 4, 6, 8, 10]

# Practical automation examples
test_urls = ["http://site1.com", "https://site2.com", "http://site3.com"]

# Get only HTTPS URLs
secure_urls = [url for url in test_urls if url.startswith("https")]

# Convert all URLs to uppercase
upper_urls = [url.upper() for url in test_urls]

# Extract domain names
domains = [url.split("//")[1] for url in test_urls]
print(domains)  # ['site1.com', 'site2.com', 'site3.com']
```

Common Patterns in Test Automation

Pattern 1: Data-Driven Testing

```
# Test data for different scenarios
test_scenarios = [
    {"name": "Valid Login", "username": "admin", "password": "admin123",
"expected": "success"},
    {"name": "Invalid Username", "username": "wrong", "password": "admin123",
"expected": "failure"},
    {"name": "Invalid Password", "username": "admin", "password": "wrong",
"expected": "failure"},
    {"name": "Empty Fields", "username": "", "password": "", "expected":
"failure"}
]

for scenario in test_scenarios:
```

```
    print(f"\nExecuting: {scenario['name']}")

    # Simulate login attempt
    if scenario['username'] == "admin" and scenario['password'] == "admin123":
        result = "success"
    else:
        result = "failure"

    # Verify expected result
    if result == scenario['expected']:
        print(f"✅ {scenario['name']}: PASSED")
    else:
        print(f"❌ {scenario['name']}: FAILED (Expected: {scenario['expected']},
Got: {result})")
```

Pattern 2: Environment-Based Configuration

```
# Different configurations for different environments
environments = {
    "dev": {"url": "https://dev.example.com", "timeout": 30},
    "staging": {"url": "https://staging.example.com", "timeout": 45},
    "prod": {"url": "https://example.com", "timeout": 60}
}

current_env = "dev"  # This could come from command line or config file

if current_env in environments:
    config = environments[current_env]
    print(f"Running tests on {current_env} environment")
    print(f"URL: {config['url']}")
    print(f"Timeout: {config['timeout']} seconds")
else:
    print(f"Unknown environment: {current_env}")
    print("Available environments:", list(environments.keys()))
```

Practice Exercises

1. **Create a grade calculator** that takes a list of scores and assigns grades using if/elif/else
2. **Write a loop** that processes a list of email addresses and identifies valid vs invalid ones
3. **Implement a retry mechanism** for a simulated flaky operation
4. **Create a data-driven test** using a list of dictionaries with test scenarios
5. **Use list comprehension** to filter test results based on status

Key Takeaways

- Use `if/elif/else` for conditional logic in your tests
- `for` loops are perfect for iterating over test data
- `while` loops are useful for retry mechanisms
- `break` and `continue` help control loop execution
- List comprehensions make data filtering more concise
- Control flow is essential for data-driven testing approaches

---

Functions - this is where you'll learn to organize your code into reusable blocks, which is crucial for creating maintainable automation scripts.

Python Functions

Functions are reusable blocks of code that perform specific tasks. They help organize your code, reduce repetition, and make your automation scripts more maintainable.

Basic Function Syntax

Defining and Calling Functions

```
# Basic function definition
def greet():
    print("Hello, World!")

# Calling the function
greet()  # Output: Hello, World!

# Function with parameters
def greet_user(name):
    print(f"Hello, {name}!")

greet_user("Alice")  # Output: Hello, Alice!
greet_user("Bob")    # Output: Hello, Bob!

# Function with return value
def add_numbers(a, b):
    result = a + b
    return result

sum_result = add_numbers(5, 3)
print(sum_result)  # Output: 8
```

Functions with Multiple Parameters

```
# Multiple parameters
def create_user_profile(name, age, email, is_active=True):
    profile = {
        "name": name,
        "age": age,
        "email": email,
        "is_active": is_active
    }
    return profile

# Function calls
user1 = create_user_profile("John", 25, "john@example.com")
user2 = create_user_profile("Jane", 30, "jane@example.com", False)

print(user1)
print(user2)
```

Default Parameters

```
# Function with default parameters
def wait_for_element(timeout=10, poll_frequency=0.5):
    print(f"Waiting for element with timeout: {timeout}s, polling every
{poll_frequency}s")
    return True

# Different ways to call
```

```
wait_for_element()                    # Uses all defaults
wait_for_element(20)                  # timeout=20, poll_frequency=0.5
wait_for_element(15, 1.0)            # timeout=15, poll_frequency=1.0
wait_for_element(poll_frequency=0.2)  # timeout=10, poll_frequency=0.2
```

Advanced Function Concepts

*args and **kwargs

```
# *args - variable number of positional arguments
def run_multiple_tests(*test_names):
    print(f"Running {len(test_names)} tests:")
    for test in test_names:
        print(f"- Executing: {test}")

run_multiple_tests("login_test", "dashboard_test", "logout_test")

# **kwargs - variable number of keyword arguments
def configure_browser(**options):
    print("Browser configuration:")
    for key, value in options.items():
        print(f"- {key}: {value}")

configure_browser(headless=True, window_size="1920x1080", timeout=30)

# Combining regular parameters, *args, and **kwargs
def execute_test(test_name, *test_data, **config):
    print(f"Executing test: {test_name}")
    print(f"Test data: {test_data}")
    print(f"Configuration: {config}")

execute_test("login_test", "user1", "password123", browser="chrome",
headless=True)
```

Lambda Functions (Anonymous Functions)

```
# Lambda function syntax
square = lambda x: x ** 2
print(square(5))  # Output: 25

# Lambda with multiple parameters
multiply = lambda x, y: x * y
print(multiply(3, 4))  # Output: 12

# Practical use with built-in functions
test_results = [
    {"name": "test1", "duration": 2.5, "status": "passed"},
    {"name": "test2", "duration": 1.8, "status": "failed"},
    {"name": "test3", "duration": 3.2, "status": "passed"}
]

# Sort by duration using lambda
sorted_by_duration = sorted(test_results, key=lambda x: x["duration"])
print("Tests sorted by duration:")
for test in sorted_by_duration:
    print(f"{test['name']}: {test['duration']}s")

# Filter passed tests using lambda
passed_tests = list(filter(lambda x: x["status"] == "passed", test_results))
print(f"\nPassed tests: {len(passed_tests)}")
```

Automation-Focused Function Examples

Example 1: Login Functions

```
def login_user(username, password, remember_me=False):
    """
    Simulate user login process
    Returns: dict with login result
    """
    print(f"Attempting login for user: {username}")

    # Simulate validation
    valid_credentials = {"admin": "admin123", "user": "user123"}

    if username in valid_credentials and valid_credentials[username] ==
password:
        result = {
            "success": True,
            "message": "Login successful",
            "user": username,
            "remember_me": remember_me
        }
    else:
        result = {
            "success": False,
            "message": "Invalid credentials",
            "user": username,
            "remember_me": False
        }

    print(result["message"])
    return result

# Test the login function
login_result = login_user("admin", "admin123", True)
if login_result["success"]:
    print(f"Welcome, {login_result['user']}!")
```

Example 2: Element Interaction Functions

```
def find_element_with_retry(locator, locator_type="id", max_retries=3,
wait_time=1):
    """
    Simulate finding an element with retry mechanism
    """
    import time

    for attempt in range(1, max_retries + 1):
        print(f"Attempt {attempt}: Looking for element by {locator_type} =
'{locator}'")

        # Simulate element search (replace with actual Selenium code later)
        element_found = attempt >= 2  # Simulate finding element on 2nd attempt

        if element_found:
            print(f"✅ Element found on attempt {attempt}")
            return {"found": True, "attempts": attempt}
        else:
            print(f"❌ Element not found, waiting {wait_time}s...")
            time.sleep(wait_time)

    print(f"❌ Element not found after {max_retries} attempts")
```

```
    return {"found": False, "attempts": max_retries}

def click_element(locator, locator_type="id"):
    """
    Click an element after finding it
    """
    element_result = find_element_with_retry(locator, locator_type)

    if element_result["found"]:
        print(f"Clicking element: {locator}")
        return True
    else:
        print(f"Cannot click element: {locator} - not found")
        return False

# Usage
click_element("submit-button", "id")
```

Example 3: Test Data Management Functions

```
def load_test_data(data_type):
    """
    Load different types of test data
    """
    test_data = {
        "users": [
            {"username": "admin", "password": "admin123", "role":
"administrator"},
            {"username": "user1", "password": "user123", "role": "regular"},
            {"username": "guest", "password": "guest123", "role": "guest"}
        ],
        "urls": {
            "login": "https://example.com/login",
            "dashboard": "https://example.com/dashboard",
            "profile": "https://example.com/profile"
        },
        "browsers": ["Chrome", "Firefox", "Edge"]
    }

    return test_data.get(data_type, [])

def get_user_by_role(role):
    """
    Get user data by role
    """
    users = load_test_data("users")
    for user in users:
        if user["role"] == role:
            return user
    return None

# Usage
admin_user = get_user_by_role("administrator")
print(f"Admin user: {admin_user}")

test_urls = load_test_data("urls")
print(f"Login URL: {test_urls['login']}")
```

Example 4: Test Result Management

```
def execute_test_case(test_name, test_function, *args, **kwargs):
    """
```

```
    Execute a test case and capture results
    """
    import time

    print(f"\n--- Executing: {test_name} ---")
    start_time = time.time()

    try:
        result = test_function(*args, **kwargs)
        end_time = time.time()
        duration = round(end_time - start_time, 2)

        test_result = {
            "name": test_name,
            "status": "PASSED" if result else "FAILED",
            "duration": duration,
            "timestamp": time.strftime("%Y-%m-%d %H:%M:%S")
        }

    except Exception as e:
        end_time = time.time()
        duration = round(end_time - start_time, 2)

        test_result = {
            "name": test_name,
            "status": "ERROR",
            "duration": duration,
            "error": str(e),
            "timestamp": time.strftime("%Y-%m-%d %H:%M:%S")
        }

    print(f"Result: {test_result['status']} ({test_result['duration']}s)")
    return test_result

def simulate_login_test():
    """Sample test function"""
    time.sleep(1)  # Simulate test execution time
    return True

def simulate_failing_test():
    """Sample failing test function"""
    time.sleep(0.5)
    return False

# Execute tests
results = []
results.append(execute_test_case("Login Test", simulate_login_test))
results.append(execute_test_case("Navigation Test", simulate_failing_test))

# Generate summary
def generate_test_summary(test_results):
    """
    Generate summary of test results
    """
    total_tests = len(test_results)
    passed_tests = len([r for r in test_results if r["status"] == "PASSED"])
    failed_tests = len([r for r in test_results if r["status"] == "FAILED"])
    error_tests = len([r for r in test_results if r["status"] == "ERROR"])
    total_duration = sum([r["duration"] for r in test_results])

    summary = {
        "total": total_tests,
        "passed": passed_tests,
        "failed": failed_tests,
```

```
        "errors": error_tests,
        "total_duration": round(total_duration, 2),
        "pass_rate": round((passed_tests / total_tests) * 100, 2) if total_tests
> 0 else 0
    }

    return summary

summary = generate_test_summary(results)
print(f"\n--- Test Summary ---")
print(f"Total Tests: {summary['total']}")
print(f"Passed: {summary['passed']}")
print(f"Failed: {summary['failed']}")
print(f"Errors: {summary['errors']}")
print(f"Pass Rate: {summary['pass_rate']}%")
print(f"Total Duration: {summary['total_duration']}s")
```

Function Documentation and Best Practices

Docstrings

```
def validate_email(email):
    """
    Validate email address format

    Args:
        email (str): Email address to validate

    Returns:
        bool: True if email is valid, False otherwise

    Example:
        >>> validate_email("user@example.com")
        True
        >>> validate_email("invalid-email")
        False
    """
    return "@" in email and "." in email.split("@")[1]

def setup_browser(browser_name, headless=False, window_size=None):
    """
    Setup browser for testing

    Args:
        browser_name (str): Name of browser ('chrome', 'firefox', 'edge')
        headless (bool, optional): Run in headless mode. Defaults to False.
        window_size (tuple, optional): Window size as (width, height). Defaults
to None.

    Returns:
        dict: Browser configuration dictionary

    Raises:
        ValueError: If browser_name is not supported
    """
    supported_browsers = ["chrome", "firefox", "edge"]

    if browser_name.lower() not in supported_browsers:
        raise ValueError(f"Unsupported browser: {browser_name}")

    config = {
        "browser": browser_name.lower(),
```

```
        "headless": headless,
        "window_size": window_size or (1920, 1080)
    }

    return config
```

Function Organization Patterns

```
# Group related functions together
class TestHelpers:
    """Collection of test helper functions"""

    @staticmethod
    def wait_and_click(element_id, timeout=10):
        """Wait for element and click it"""
        print(f"Waiting for and clicking element: {element_id}")
        return True

    @staticmethod
    def fill_form_field(field_id, value):
        """Fill a form field with value"""
        print(f"Filling field {field_id} with: {value}")
        return True

    @staticmethod
    def verify_text_present(expected_text):
        """Verify text is present on page"""
        print(f"Verifying text present: {expected_text}")
        return True

# Usage
TestHelpers.wait_and_click("submit-button")
TestHelpers.fill_form_field("username", "testuser")
```

Scope and Variable Lifetime

```
# Global vs Local scope
global_variable = "I'm global"

def demonstrate_scope():
    local_variable = "I'm local"
    print(f"Inside function: {global_variable}")
    print(f"Inside function: {local_variable}")

demonstrate_scope()
print(f"Outside function: {global_variable}")
# print(local_variable)  # This would cause an error

# Modifying global variables
test_count = 0

def run_test():
    global test_count
    test_count += 1
    print(f"Running test #{test_count}")

run_test()  # Running test #1
run_test()  # Running test #2
print(f"Total tests run: {test_count}")
```

Practice Exercises

1. **Create a function** that takes a list of URLs and returns only the HTTPS ones
2. **Write a function** that simulates form filling with default values
3. **Implement a retry function** that accepts any function as a parameter
4. **Create functions** for different types of element interactions (click, type, select)
5. **Build a test reporting function** that formats results nicely

Key Takeaways

- Functions help organize and reuse code
- Use parameters to make functions flexible
- Default parameters provide convenience
- *args and **kwargs handle variable arguments
- Lambda functions are useful for simple operations
- Always document your functions with docstrings
- Functions should have a single, clear purpose
- Use meaningful function and parameter names

Modules and Packages - this is where you'll learn to organize your code across multiple files and work with external libraries, which is essential for building scalable automation frameworks.

Python Modules and Packages

Modules and packages help you organize your code into separate files and folders, making your automation projects more maintainable and reusable.

What are Modules?

A module is simply a Python file containing functions, classes, and variables. Any `.py` file can be imported as a module.

Creating Your First Module

Let's create a module called `test_helpers.py`:

```
# test_helpers.py
"""
Test helper functions for automation
"""

def login_user(username, password):
    """Simulate user login"""
    print(f"Logging in user: {username}")
    if username == "admin" and password == "admin123":
        return {"success": True, "message": "Login successful"}
    return {"success": False, "message": "Invalid credentials"}

def wait_for_element(element_id, timeout=10):
    """Wait for element to be present"""
    print(f"Waiting for element: {element_id} (timeout: {timeout}s)")
    return True
```

```
def take_screenshot(filename="screenshot.png"):
    """Take a screenshot"""
    print(f"Taking screenshot: {filename}")
    return filename

# Module-level variables
DEFAULT_TIMEOUT = 10
SUPPORTED_BROWSERS = ["chrome", "firefox", "edge"]

# Module-level code (runs when imported)
print("Test helpers module loaded")
```

Importing Modules

```
# main.py - Different ways to import

# 1. Import entire module
import test_helpers

result = test_helpers.login_user("admin", "admin123")
print(result)
print(f"Default timeout: {test_helpers.DEFAULT_TIMEOUT}")

# 2. Import specific functions
from test_helpers import login_user, wait_for_element

result = login_user("user", "password")
wait_for_element("submit-button", 15)

# 3. Import with alias
import test_helpers as helpers

helpers.take_screenshot("test_screenshot.png")

# 4. Import specific items with alias
from test_helpers import login_user as login, SUPPORTED_BROWSERS as browsers

login("admin", "admin123")
print(f"Supported browsers: {browsers}")

# 5. Import all (use sparingly)
from test_helpers import *

# Now all functions are available directly
login_user("test", "test")
wait_for_element("button")
```

Built-in Modules

Python comes with many useful built-in modules:

```
# os module - Operating system interface
import os

print(f"Current directory: {os.getcwd()}")
print(f"Environment PATH: {os.environ.get('PATH', 'Not found')}")

# Create directory for test reports
reports_dir = "test_reports"
if not os.path.exists(reports_dir):
```

```
    os.makedirs(reports_dir)
    print(f"Created directory: {reports_dir}")

# datetime module - Date and time operations
from datetime import datetime, timedelta

current_time = datetime.now()
print(f"Test started at: {current_time.strftime('%Y-%m-%d %H:%M:%S')}")

# Calculate test deadline (1 hour from now)
deadline = current_time + timedelta(hours=1)
print(f"Test deadline: {deadline.strftime('%Y-%m-%d %H:%M:%S')}")

# json module - JSON data handling
import json

test_config = {
    "browser": "chrome",
    "headless": True,
    "timeout": 30,
    "test_urls": ["https://example.com", "https://test.com"]
}

# Save config to file
with open("config.json", "w") as f:
    json.dump(test_config, f, indent=2)

# Load config from file
with open("config.json", "r") as f:
    loaded_config = json.load(f)
    print(f"Loaded config: {loaded_config}")

# random module - Generate random data
import random

# Generate test data
random_emails = [
    f"user{random.randint(1000, 9999)}@example.com"
    for _ in range(3)
]
print(f"Random test emails: {random_emails}")

random_password = ''.join(random.choices("abcdefghijklmnopqrstuvwxyz0123456789",
k=8))
print(f"Random password: {random_password}")
```

Creating Packages

A package is a directory containing multiple modules. It must have an `__init__.py` file.

Package Structure Example

```
test_automation/
├── __init__.py
├── browser/
│   ├── __init__.py
│   ├── chrome_driver.py
│   └── firefox_driver.py
├── pages/
│   ├── __init__.py
│   ├── login_page.py
│   └── dashboard_page.py
```

```
├── utils/
│   ├── __init__.py
│   ├── helpers.py
│   └── data_loader.py
└── tests/
    ├── __init__.py
    ├── test_login.py
    └── test_dashboard.py
```

Package Files Content

```
# test_automation/__init__.py
"""
Test Automation Framework
Version: 1.0.0
"""

__version__ = "1.0.0"
__author__ = "Test Team"

# Make commonly used functions available at package level
from .utils.helpers import wait_for_element, take_screenshot
from .browser.chrome_driver import setup_chrome_driver

print(f"Test Automation Framework v{__version__} loaded")

# test_automation/utils/__init__.py
"""Utility modules for test automation"""

from .helpers import *
from .data_loader import load_test_data

# test_automation/utils/helpers.py
"""General helper functions"""

import time
from datetime import datetime

def wait_for_element(locator, timeout=10):
    """Wait for element with timeout"""
    print(f"Waiting for element: {locator}")
    time.sleep(1)  # Simulate wait
    return True

def take_screenshot(filename=None):
    """Take screenshot with timestamp"""
    if filename is None:
        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
        filename = f"screenshot_{timestamp}.png"

    print(f"Taking screenshot: {filename}")
    return filename

def generate_test_id():
    """Generate unique test ID"""
    return f"TEST_{datetime.now().strftime('%Y%m%d_%H%M%S')}"

# test_automation/utils/data_loader.py
"""Test data loading utilities"""

def load_test_data(data_type):
    """Load test data by type"""
    data = {
```

```
        "users": [
            {"username": "admin", "password": "admin123"},
            {"username": "user1", "password": "user123"}
        ],
        "urls": {
            "login": "https://example.com/login",
            "dashboard": "https://example.com/dashboard"
        }
    }
    return data.get(data_type, {})

def load_user_credentials(role="admin"):
    """Load user credentials by role"""
    users = load_test_data("users")
    return users[0] if role == "admin" else users[1]

# test_automation/browser/chrome_driver.py
"""Chrome driver setup"""

def setup_chrome_driver(headless=False):
    """Setup Chrome driver with options"""
    options = {
        "headless": headless,
        "window_size": (1920, 1080)
    }
    print(f"Setting up Chrome driver: {options}")
    return options
```

Using Your Package

```
# main.py - Using the package

# Import entire package
import test_automation

print(f"Framework version: {test_automation.__version__}")

# Use functions made available at package level
test_automation.wait_for_element("button", 15)
screenshot = test_automation.take_screenshot()

# Import specific modules
from test_automation.utils import helpers, data_loader
from test_automation.browser.chrome_driver import setup_chrome_driver

# Use module functions
test_id = helpers.generate_test_id()
print(f"Generated test ID: {test_id}")

# Load test data
users = data_loader.load_test_data("users")
admin_user = data_loader.load_user_credentials("admin")
print(f"Admin user: {admin_user}")

# Setup browser
browser_config = setup_chrome_driver(headless=True)
print(f"Browser config: {browser_config}")

# Import from nested packages
from test_automation.utils.helpers import wait_for_element as wait
from test_automation.utils.data_loader import load_test_data as load_data

wait("submit-button")
```

```
test_urls = load_data("urls")
```

Third-Party Packages with pip

Installing Packages

```
# Install single package
pip install requests

# Install specific version
pip install selenium==4.15.0

# Install from requirements file
pip install -r requirements.txt

# List installed packages
pip list

# Show package information
pip show selenium
```

Using Third-Party Packages

```
# requests - HTTP library (useful for API testing)
import requests

def test_api_endpoint(url, expected_status=200):
    """Test API endpoint"""
    try:
        response = requests.get(url, timeout=10)
        success = response.status_code == expected_status

        result = {
            "url": url,
            "status_code": response.status_code,
            "success": success,
            "response_time": response.elapsed.total_seconds()
        }

        print(f"API Test - {url}: {'PASSED' if success else 'FAILED'}")
        return result

    except requests.exceptions.RequestException as e:
        print(f"API Test Error - {url}: {e}")
        return {"url": url, "success": False, "error": str(e)}

# Test API endpoints
api_results = []
api_results.append(test_api_endpoint("https://httpbin.org/status/200"))
api_results.append(test_api_endpoint("https://httpbin.org/status/404", 404))

# BeautifulSoup - HTML parsing (useful for data extraction)
try:
    from bs4 import BeautifulSoup

    def parse_html_content(html_content):
        """Parse HTML and extract useful information"""
        soup = BeautifulSoup(html_content, 'html.parser')

        # Extract all links
        links = [link.get('href') for link in soup.find_all('a', href=True)]
```

```
        # Extract all form inputs
        inputs = [input_tag.get('name') for input_tag in soup.find_all('input')]

        return {
            "links": links,
            "form_inputs": inputs,
            "title": soup.title.string if soup.title else "No title"
        }

    # Example HTML
    sample_html = """
    <html>
        <head><title>Test Page</title></head>
        <body>
            <a href="/login">Login</a>
            <a href="/dashboard">Dashboard</a>
            <form>
                <input name="username" type="text">
                <input name="password" type="password">
                <input type="submit" value="Submit">
            </form>
        </body>
    </html>
    """

    parsed_content = parse_html_content(sample_html)
    print(f"Parsed content: {parsed_content}")

except ImportError:
    print("BeautifulSoup not installed. Run: pip install beautifulsoup4")
```

Module Search Path and PYTHONPATH

```
# sys module - System-specific parameters
import sys

print("Python module search paths:")
for path in sys.path:
    print(f"  {path}")

# Add custom path to module search
import os
custom_path = os.path.join(os.getcwd(), "custom_modules")
if custom_path not in sys.path:
    sys.path.append(custom_path)
    print(f"Added custom path: {custom_path}")
```

Creating a Requirements File

```
# create_requirements.py
"""Generate requirements.txt file"""

requirements = [
    "selenium>=4.15.0",
    "pytest>=7.0.0",
    "requests>=2.28.0",
    "beautifulsoup4>=4.11.0",
    "webdriver-manager>=3.8.0"
]
```

```
with open("requirements.txt", "w") as f:
    f.write("\n".join(requirements))

print("requirements.txt created with:")
for req in requirements:
    print(f"  {req}")
```

Practical Automation Framework Structure

```
# Example: Complete test framework structure

# config.py - Configuration management
"""Test configuration settings"""

import os
from datetime import datetime

class Config:
    # Browser settings
    DEFAULT_BROWSER = "chrome"
    HEADLESS = os.getenv("HEADLESS", "false").lower() == "true"
    IMPLICIT_WAIT = 10

    # URLs
    BASE_URL = os.getenv("BASE_URL", "https://example.com")

    # Test data
    TEST_DATA_DIR = "test_data"
    SCREENSHOTS_DIR = "screenshots"
    REPORTS_DIR = "reports"

    # Create directories
    @classmethod
    def setup_directories(cls):
        for directory in [cls.TEST_DATA_DIR, cls.SCREENSHOTS_DIR,
cls.REPORTS_DIR]:
            os.makedirs(directory, exist_ok=True)

# logger.py - Logging setup
"""Logging configuration"""

import logging
from datetime import datetime

def setup_logger(name="test_automation"):
    """Setup logger with file and console handlers"""

    # Create logger
    logger = logging.getLogger(name)
    logger.setLevel(logging.INFO)

    # Avoid duplicate handlers
    if logger.handlers:
        return logger

    # Create formatter
    formatter = logging.Formatter(
        '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
    )

    # Console handler
    console_handler = logging.StreamHandler()
```

```
    console_handler.setFormatter(formatter)
    logger.addHandler(console_handler)

    # File handler
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    log_file = f"test_log_{timestamp}.log"
    file_handler = logging.FileHandler(log_file)
    file_handler.setFormatter(formatter)
    logger.addHandler(file_handler)

    return logger

# main_test.py - Using the framework
"""Main test execution"""

from config import Config
from logger import setup_logger

def main():
    # Setup
    Config.setup_directories()
    logger = setup_logger()

    logger.info("Test execution started")
    logger.info(f"Base URL: {Config.BASE_URL}")
    logger.info(f"Browser: {Config.DEFAULT_BROWSER}")
    logger.info(f"Headless mode: {Config.HEADLESS}")

    # Your test execution logic here
    logger.info("All tests completed")

if __name__ == "__main__":
    main()
```

Best Practices

1. Module Organization

```
# Good: Clear, descriptive module names
# login_helpers.py
# browser_setup.py
# test_data_loader.py

# Bad: Vague names
# utils.py (too generic)
# stuff.py
# helpers.py (too generic)
```

2. Import Organization

```
# Good: Organized imports
# Standard library imports
import os
import sys
from datetime import datetime

# Third-party imports
import requests
from selenium import webdriver

# Local imports
```

```
from utils.helpers import wait_for_element
from pages.login_page import LoginPage
```

3. Package Documentation

```
# __init__.py
"""
Test Automation Framework

This package provides utilities for web automation testing including:
- Browser management
- Page object models
- Test data handling
- Reporting utilities

Usage:
    from test_automation import setup_browser, LoginPage

    driver = setup_browser("chrome")
    login_page = LoginPage(driver)
"""
```

Practice Exercises

1. **Create a module** with functions for different browser setups
2. **Build a package** for test utilities with multiple modules
3. **Create a configuration module** that loads settings from environment variables
4. **Write a data loader module** that can handle JSON, CSV files
5. **Set up a logging module** for your test framework

Key Takeaways

- Modules are individual Python files that can be imported
- Packages are directories with `__init__.py` files
- Use imports to access functionality from other modules
- Organize related functions into modules and packages
- Use third-party packages to extend functionality
- Structure your automation framework with clear module organization
- Document your modules and packages well

Key concepts covered:

- **Creating modules** - organizing functions in separate .py files
- **Import techniques** - different ways to import modules and functions
- **Built-in modules** - os, datetime, json, random for common tasks
- **Package structure** - organizing multiple modules in directories
- **Third-party packages** - using pip to install external libraries
- **Framework organization** - practical structure for automation projects

Important for automation because:

- **Organization**: Keep your code structured and maintainable
- **Reusability**: Share common functions across multiple test files

- **Collaboration**: Team members can work on different modules
- **Scalability**: Framework can grow without becoming unwieldy

Practical examples included:

- Test helper modules
- Complete package structure for automation framework
- Configuration and logging modules
- Using popular third-party libraries (requests, BeautifulSoup)
- Requirements file management

Best practices highlighted:

- Clear, descriptive module names
- Organized import statements
- Proper package documentation
- Directory structure for test frameworks

Practice what you've learned:

- Create modules for different automation tasks
- Build a simple package structure
- Practice different import techniques
- Set up a requirements.txt file for your project

Python Object-Oriented Programming (OOP)

What is OOP?

Object-Oriented Programming is a programming paradigm that organizes code into objects and classes. It helps create reusable, maintainable, and well-structured code.

1. Classes & Objects

What is a Class?

A class is a blueprint or template for creating objects. It defines attributes (data) and methods (functions) that objects will have.

What is an Object?

An object is an instance of a class. It's a real entity created from the class template.

Creating Your First Class

```
# Basic class definition
class Car:
    # Class attribute (shared by all instances)
    wheels = 4

    # Constructor method (initializer)
    def __init__(self, brand, model, year):
```

```
        # Instance attributes (unique to each object)
        self.brand = brand
        self.model = model
        self.year = year
        self.is_running = False

    # Instance method
    def start_engine(self):
        self.is_running = True
        print(f"{self.brand} {self.model} engine started!")

    def stop_engine(self):
        self.is_running = False
        print(f"{self.brand} {self.model} engine stopped!")

    def get_info(self):
        return f"{self.year} {self.brand} {self.model}"

# Creating objects (instances)
car1 = Car("Toyota", "Camry", 2023)
car2 = Car("Honda", "Civic", 2022)

# Using objects
print(car1.get_info())  # Output: 2023 Toyota Camry
print(car2.get_info())  # Output: 2022 Honda Civic

car1.start_engine()     # Output: Toyota Camry engine started!
print(car1.is_running)  # Output: True

# Accessing class attribute
print(car1.wheels)      # Output: 4
print(Car.wheels)       # Output: 4
```

Practice Exercise 1

```
# Create a Person class
class Person:
    def __init__(self, name, age, email):
        self.name = name
        self.age = age
        self.email = email

    def introduce(self):
        return f"Hi, I'm {self.name}, {self.age} years old."

    def have_birthday(self):
        self.age += 1
        print(f"Happy birthday! {self.name} is now {self.age} years old.")

# Test the class
person1 = Person("Alice", 25, "alice@email.com")
print(person1.introduce())
person1.have_birthday()
```

2. Inheritance

Inheritance allows a class to inherit attributes and methods from another class.

Basic Inheritance

```
# Parent class (Base class)
```

```
class Animal:
    def __init__(self, name, species):
        self.name = name
        self.species = species
        self.is_alive = True

    def eat(self):
        print(f"{self.name} is eating.")

    def sleep(self):
        print(f"{self.name} is sleeping.")

    def make_sound(self):
        print("Some generic animal sound")

# Child class (Derived class)
class Dog(Animal):
    def __init__(self, name, breed):
        # Call parent constructor
        super().__init__(name, "Canine")
        self.breed = breed

    # Method overriding
    def make_sound(self):
        print(f"{self.name} says Woof!")

    # Additional method specific to Dog
    def fetch(self):
        print(f"{self.name} is fetching the ball!")

class Cat(Animal):
    def __init__(self, name, color):
        super().__init__(name, "Feline")
        self.color = color

    def make_sound(self):
        print(f"{self.name} says Meow!")

    def climb_tree(self):
        print(f"{self.name} is climbing a tree!")

# Using inheritance
dog1 = Dog("Buddy", "Golden Retriever")
cat1 = Cat("Whiskers", "Orange")

# Inherited methods
dog1.eat()          # Output: Buddy is eating.
cat1.sleep()        # Output: Whiskers is sleeping.

# Overridden methods
dog1.make_sound()   # Output: Buddy says Woof!
cat1.make_sound()   # Output: Whiskers says Meow!

# Child-specific methods
dog1.fetch()        # Output: Buddy is fetching the ball!
cat1.climb_tree()   # Output: Whiskers is climbing a tree!

print(f"Dog breed: {dog1.breed}")     # Output: Dog breed: Golden Retriever
print(f"Cat species: {cat1.species}") # Output: Cat species: Feline
```

Multiple Inheritance

```
class Vehicle:
```

```
    def __init__(self, max_speed):
        self.max_speed = max_speed

    def start(self):
        print("Vehicle started")

class Electric:
    def __init__(self, battery_capacity):
        self.battery_capacity = battery_capacity

    def charge(self):
        print("Charging battery...")

# Multiple inheritance
class ElectricCar(Vehicle, Electric):
    def __init__(self, max_speed, battery_capacity, brand):
        Vehicle.__init__(self, max_speed)
        Electric.__init__(self, battery_capacity)
        self.brand = brand

    def drive_silent(self):
        print("Driving silently with electric motor")

tesla = ElectricCar(200, 100, "Tesla")
tesla.start()           # From Vehicle
tesla.charge()          # From Electric
tesla.drive_silent()    # Own method
```

3. Encapsulation

Encapsulation is about hiding internal details and controlling access to class members.

Private and Protected Members

```
class BankAccount:
    def __init__(self, account_number, initial_balance):
        self.account_number = account_number    # Public
        self._balance = initial_balance         # Protected (convention)
        self.__pin = 1234                      # Private (name mangling)

    # Public method
    def get_balance(self):
        return self._balance

    # Public method to deposit
    def deposit(self, amount):
        if amount > 0:
            self._balance += amount
            print(f"Deposited ${amount}. New balance: ${self._balance}")
        else:
            print("Invalid deposit amount")

    # Public method to withdraw
    def withdraw(self, amount, pin):
        if self.__validate_pin(pin):
            if amount <= self._balance:
                self._balance -= amount
                print(f"Withdrew ${amount}. New balance: ${self._balance}")
            else:
                print("Insufficient funds")
        else:
            print("Invalid PIN")
```

```
    # Private method
    def __validate_pin(self, pin):
        return pin == self.__pin

    # Method to change PIN
    def change_pin(self, old_pin, new_pin):
        if self.__validate_pin(old_pin):
            self.__pin = new_pin
            print("PIN changed successfully")
        else:
            print("Invalid old PIN")

# Using encapsulation
account = BankAccount("123456", 1000)

print(account.account_number)  # Accessible
print(account.get_balance())   # Accessible through method

account.deposit(500)
account.withdraw(200, 1234)

# This will raise an AttributeError
# print(account.__pin)  # Can't access private attribute directly

# But name mangling allows this (not recommended)
# print(account._BankAccount__pin)
```

Properties and Getters/Setters

```
class Temperature:
    def __init__(self, celsius=0):
        self._celsius = celsius

    @property
    def celsius(self):
        return self._celsius

    @celsius.setter
    def celsius(self, value):
        if value < -273.15:
            raise ValueError("Temperature cannot be below absolute zero")
        self._celsius = value

    @property
    def fahrenheit(self):
        return (self._celsius * 9/5) + 32

    @fahrenheit.setter
    def fahrenheit(self, value):
        self.celsius = (value - 32) * 5/9

# Using properties
temp = Temperature(25)
print(f"Celsius: {temp.celsius}")      # Output: Celsius: 25
print(f"Fahrenheit: {temp.fahrenheit}") # Output: Fahrenheit: 77.0

temp.fahrenheit = 86
print(f"Celsius: {temp.celsius}")      # Output: Celsius: 30.0

# This will raise ValueError
# temp.celsius = -300
```

4. Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common base class.

```
class Shape:
    def area(self):
        pass

    def perimeter(self):
        pass

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2 * (self.width + self.height)

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14159 * self.radius ** 2

    def perimeter(self):
        return 2 * 3.14159 * self.radius

class Triangle(Shape):
    def __init__(self, base, height, side1, side2, side3):
        self.base = base
        self.height = height
        self.side1 = side1
        self.side2 = side2
        self.side3 = side3

    def area(self):
        return 0.5 * self.base * self.height

    def perimeter(self):
        return self.side1 + self.side2 + self.side3

# Polymorphism in action
def print_shape_info(shape):
    print(f"Area: {shape.area()}")
    print(f"Perimeter: {shape.perimeter()}")
    print("-" * 20)

# Create different shapes
shapes = [
    Rectangle(5, 3),
    Circle(4),
    Triangle(6, 4, 5, 5, 6)
]

# Same function works with different object types
for shape in shapes:
    print(f"Shape: {type(shape).__name__}")
    print_shape_info(shape)
```

Practice Exercises

Exercise 1: Library Management System

```
class Book:
    def __init__(self, title, author, isbn):
        self.title = title
        self.author = author
        self.isbn = isbn
        self.is_available = True

    def borrow(self):
        if self.is_available:
            self.is_available = False
            return True
        return False

    def return_book(self):
        self.is_available = True

class Library:
    def __init__(self):
        self.books = []

    def add_book(self, book):
        self.books.append(book)

    def find_book(self, title):
        for book in self.books:
            if book.title.lower() == title.lower():
                return book
        return None

    def borrow_book(self, title):
        book = self.find_book(title)
        if book and book.borrow():
            print(f"'{title}' borrowed successfully")
        else:
            print(f"'{title}' is not available")

# Test the system
library = Library()
library.add_book(Book("Python Programming", "John Doe", "123456"))
library.add_book(Book("Web Development", "Jane Smith", "789012"))

library.borrow_book("Python Programming")
library.borrow_book("Python Programming")  # Should show not available
```

Exercise 2: Employee Hierarchy

```
class Employee:
    def __init__(self, name, employee_id, salary):
        self.name = name
        self.employee_id = employee_id
        self.salary = salary

    def work(self):
        print(f"{self.name} is working")

    def get_salary_info(self):
        return f"{self.name}: ${self.salary}"
```

```
class Developer(Employee):
    def __init__(self, name, employee_id, salary, programming_language):
        super().__init__(name, employee_id, salary)
        self.programming_language = programming_language

    def work(self):
        print(f"{self.name} is coding in {self.programming_language}")

    def code_review(self):
        print(f"{self.name} is reviewing code")

class Manager(Employee):
    def __init__(self, name, employee_id, salary, team_size):
        super().__init__(name, employee_id, salary)
        self.team_size = team_size

    def work(self):
        print(f"{self.name} is managing a team of {self.team_size}")

    def conduct_meeting(self):
        print(f"{self.name} is conducting a team meeting")

# Test polymorphism
employees = [
    Developer("Alice", "DEV001", 80000, "Python"),
    Manager("Bob", "MGR001", 100000, 5),
    Developer("Charlie", "DEV002", 75000, "JavaScript")
]

for employee in employees:
    print(employee.get_salary_info())
    employee.work()
    print()
```

Key Takeaways

1. **Classes** are templates, **Objects** are instances
2. **Inheritance** allows code reuse and creates "is-a" relationships
3. **Encapsulation** hides internal details and controls access
4. **Polymorphism** allows different objects to be used interchangeably
5. Use `super()` to call parent class methods
6. Use `@property` for getters and setters
7. Private attributes start with __, protected with _

What we covered:

1. **Classes & Objects** - The foundation of OOP
2. **Inheritance** - Code reuse and creating hierarchies
3. **Encapsulation** - Data hiding and access control
4. **Polymorphism** - Using different objects interchangeably

Key highlights:

- Practical examples with cars, animals, and bank accounts
- Real-world scenarios like Library Management and Employee systems

- Best practices for private/protected attributes
- Property decorators for getters/setters
- Multiple inheritance examples

Why this matters for Selenium:

OOP is crucial for Selenium automation because:

- **Page Object Model** uses classes to represent web pages
- **Test frameworks** organize tests using classes
- **Inheritance** helps create reusable test components
- **Encapsulation** protects sensitive test data

Python File Handling

File handling is crucial for test automation - you'll read test data, configuration files, and write test reports. Let's master working with different file types!

1. Basic File Operations

Opening and Closing Files

```
# Method 1: Manual file handling (not recommended)
file = open('example.txt', 'r')
content = file.read()
file.close()

# Method 2: Using 'with' statement (recommended)
with open('example.txt', 'r') as file:
    content = file.read()
    # File automatically closes when exiting the 'with' block
```

File Modes

```
# Common file modes:
# 'r'  - Read (default)
# 'w'  - Write (overwrites existing content)
# 'a'  - Append
# 'x'  - Create (fails if file exists)
# 'r+' - Read and write
# 'b'  - Binary mode (e.g., 'rb', 'wb')

# Examples
with open('data.txt', 'w') as file:    # Write mode
    file.write("Hello World")

with open('data.txt', 'a') as file:    # Append mode
    file.write("\nNew line added")

with open('data.txt', 'r') as file:    # Read mode
    content = file.read()
    print(content)
```

2. Reading Files

Different Ways to Read Files

```
# Create a sample file first
sample_content = """Line 1: Python is awesome
Line 2: File handling is important
Line 3: Test automation rocks
Line 4: Selenium with Python"""

with open('sample.txt', 'w') as file:
    file.write(sample_content)

# Method 1: Read entire file
with open('sample.txt', 'r') as file:
    content = file.read()
    print("Entire file:")
    print(content)
    print("-" * 30)

# Method 2: Read line by line
with open('sample.txt', 'r') as file:
    print("Reading line by line:")
    for line_number, line in enumerate(file, 1):
        print(f"Line {line_number}: {line.strip()}")
    print("-" * 30)

# Method 3: Read all lines into a list
with open('sample.txt', 'r') as file:
    lines = file.readlines()
    print("All lines as list:")
    for i, line in enumerate(lines):
        print(f"Index {i}: {repr(line)}")  # repr() shows \n characters
    print("-" * 30)

# Method 4: Read one line at a time
with open('sample.txt', 'r') as file:
    print("Reading first two lines only:")
    first_line = file.readline()
    second_line = file.readline()
    print(f"First: {first_line.strip()}")
    print(f"Second: {second_line.strip()}")
```

Practical File Reading Example

```
def read_config_file(filename):
    """Read configuration from a text file"""
    config = {}
    try:
        with open(filename, 'r') as file:
            for line in file:
                line = line.strip()
                if line and not line.startswith('#'):  # Skip empty lines and
comments
                    if '=' in line:
                        key, value = line.split('=', 1)
                        config[key.strip()] = value.strip()
        return config
    except FileNotFoundError:
        print(f"File {filename} not found!")
        return {}
```

```
# Create a config file
config_content = """# Application Configuration
url=https://example.com
username=testuser
password=testpass123
timeout=10
# Browser settings
browser=chrome"""

with open('config.txt', 'w') as file:
    file.write(config_content)

# Read the config
config = read_config_file('config.txt')
print("Configuration loaded:")
for key, value in config.items():
    print(f"{key}: {value}")
```

3. Writing Files

Basic Writing Operations

```
# Writing text to file
data_to_write = [
    "Test Case 1: Login functionality",
    "Test Case 2: Registration process",
    "Test Case 3: Password reset",
    "Test Case 4: Profile update"
]

# Write list to file
with open('test_cases.txt', 'w') as file:
    for test_case in data_to_write:
        file.write(test_case + '\n')

print("File written successfully!")

# Append additional data
with open('test_cases.txt', 'a') as file:
    file.write("Test Case 5: Logout functionality\n")
    file.write("Test Case 6: Search feature\n")

# Verify the content
with open('test_cases.txt', 'r') as file:
    print("File contents:")
    print(file.read())
```

Writing Test Results

```
import datetime

class TestReporter:
    def __init__(self, filename):
        self.filename = filename
        self.test_results = []

    def add_test_result(self, test_name, status, message=""):
        timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        result = {
            'test_name': test_name,
            'status': status,
```

```
            'message': message,
            'timestamp': timestamp
        }
        self.test_results.append(result)

    def write_report(self):
        with open(self.filename, 'w') as file:
            file.write("TEST EXECUTION REPORT\n")
            file.write("=" * 50 + "\n")
            file.write(f"Generated on: {datetime.datetime.now()}\n\n")

            passed = sum(1 for result in self.test_results if result['status']
== 'PASS')
            failed = sum(1 for result in self.test_results if result['status']
== 'FAIL')

            file.write(f"Summary: {passed} Passed, {failed} Failed\n")
            file.write("-" * 50 + "\n\n")

            for result in self.test_results:
                file.write(f"Test: {result['test_name']}\n")
                file.write(f"Status: {result['status']}\n")
                file.write(f"Time: {result['timestamp']}\n")
                if result['message']:
                    file.write(f"Message: {result['message']}\n")
                file.write("-" * 30 + "\n")

# Usage example
reporter = TestReporter('test_report.txt')
reporter.add_test_result("Login Test", "PASS")
reporter.add_test_result("Registration Test", "PASS")
reporter.add_test_result("Password Reset", "FAIL", "Email not sent")
reporter.add_test_result("Profile Update", "PASS")
reporter.write_report()

print("Test report generated!")
```

4. Working with JSON Files

JSON is perfect for configuration files and test data in automation.

Reading JSON Files

```
import json

# Create sample JSON data
test_data = {
    "users": [
        {
            "username": "john_doe",
            "password": "password123",
            "email": "john@example.com",
            "role": "admin"
        },
        {
            "username": "jane_smith",
            "password": "secret456",
            "email": "jane@example.com",
            "role": "user"
        }
    ],
    "test_config": {
```

```
        "base_url": "https://demo.site.com",
        "timeout": 30,
        "browser": "chrome",
        "headless": False
    }
}

# Write JSON to file
with open('test_data.json', 'w') as file:
    json.dump(test_data, file, indent=4)

print("JSON file created!")

# Read JSON from file
with open('test_data.json', 'r') as file:
    loaded_data = json.load(file)

print("Loaded JSON data:")
print(f"Base URL: {loaded_data['test_config']['base_url']}")
print(f"First user: {loaded_data['users'][0]['username']}")

# Pretty print JSON
print("\nPretty printed JSON:")
print(json.dumps(loaded_data, indent=2))
```

Practical JSON Example for Test Automation

```
class TestDataManager:
    def __init__(self, json_file):
        self.json_file = json_file
        self.data = self.load_data()

    def load_data(self):
        try:
            with open(self.json_file, 'r') as file:
                return json.load(file)
        except FileNotFoundError:
            print(f"JSON file {self.json_file} not found!")
            return {}
        except json.JSONDecodeError as e:
            print(f"Error parsing JSON: {e}")
            return {}

    def get_user_data(self, role=None):
        users = self.data.get('users', [])
        if role:
            return [user for user in users if user.get('role') == role]
        return users

    def get_config(self, key=None):
        config = self.data.get('test_config', {})
        if key:
            return config.get(key)
        return config

    def update_config(self, key, value):
        if 'test_config' not in self.data:
            self.data['test_config'] = {}
        self.data['test_config'][key] = value
        self.save_data()

    def save_data(self):
        with open(self.json_file, 'w') as file:
```

```
            json.dump(self.data, file, indent=4)

# Usage
data_manager = TestDataManager('test_data.json')

# Get admin users
admin_users = data_manager.get_user_data('admin')
print("Admin users:", admin_users)

# Get specific config
browser = data_manager.get_config('browser')
print(f"Browser: {browser}")

# Update config
data_manager.update_config('timeout', 45)
print("Config updated!")
```

5. Working with CSV Files

CSV files are excellent for storing test data in tabular format.

Basic CSV Operations

```
import csv

# Create sample CSV data
csv_data = [
    ['Test_ID', 'Test_Name', 'Expected_Result', 'Status'],
    ['TC001', 'Login with valid credentials', 'Login successful', 'Pass'],
    ['TC002', 'Login with invalid password', 'Error message displayed', 'Pass'],
    ['TC003', 'Login with empty username', 'Validation error', 'Fail'],
    ['TC004', 'Password reset functionality', 'Reset email sent', 'Pass']
]

# Write CSV file
with open('test_results.csv', 'w', newline='') as file:
    writer = csv.writer(file)
    writer.writerows(csv_data)

print("CSV file created!")

# Read CSV file
with open('test_results.csv', 'r') as file:
    reader = csv.reader(file)
    print("CSV Contents:")
    for row in reader:
        print(row)
```

Advanced CSV Operations

```
# Working with CSV as dictionaries
test_cases = [
    {'test_id': 'TC001', 'test_name': 'Login Test', 'priority': 'High',
'status': 'Pass'},
    {'test_id': 'TC002', 'test_name': 'Signup Test', 'priority': 'Medium',
'status': 'Pass'},
    {'test_id': 'TC003', 'test_name': 'Search Test', 'priority': 'Low',
'status': 'Fail'},
    {'test_id': 'TC004', 'test_name': 'Checkout Test', 'priority': 'High',
'status': 'Pass'}
]
```

```
# Write dictionary data to CSV
with open('test_cases.csv', 'w', newline='') as file:
    fieldnames = ['test_id', 'test_name', 'priority', 'status']
    writer = csv.DictWriter(file, fieldnames=fieldnames)

    writer.writeheader()  # Write column headers
    writer.writerows(test_cases)

print("Dictionary CSV created!")

# Read CSV as dictionaries
with open('test_cases.csv', 'r') as file:
    reader = csv.DictReader(file)
    print("Reading CSV as dictionaries:")
    for row in reader:
        print(f"ID: {row['test_id']}, Name: {row['test_name']}, Status:
{row['status']}")

# Filter and process CSV data
def get_failed_tests(csv_file):
    failed_tests = []
    with open(csv_file, 'r') as file:
        reader = csv.DictReader(file)
        for row in reader:
            if row['status'].lower() == 'fail':
                failed_tests.append(row)
    return failed_tests

failed = get_failed_tests('test_cases.csv')
print(f"\nFailed tests: {len(failed)}")
for test in failed:
    print(f"- {test['test_name']} (Priority: {test['priority']})")
```

CSV Test Data Provider

```
class CSVTestDataProvider:
    def __init__(self, csv_file):
        self.csv_file = csv_file

    def get_test_data(self, test_type=None):
        test_data = []
        try:
            with open(self.csv_file, 'r') as file:
                reader = csv.DictReader(file)
                for row in reader:
                    if test_type is None or row.get('test_type') == test_type:
                        test_data.append(row)
            return test_data
        except FileNotFoundError:
            print(f"CSV file {self.csv_file} not found!")
            return []

    def add_test_result(self, test_data):
        # Append new test result to CSV
        with open(self.csv_file, 'a', newline='') as file:
            if test_data:
                writer = csv.DictWriter(file, fieldnames=test_data.keys())
                writer.writerow(test_data)

# Create sample login test data
login_data = [
```

```
    {'test_type': 'login', 'username': 'valid_user', 'password': 'valid_pass',
'expected': 'success'},
    {'test_type': 'login', 'username': 'invalid_user', 'password': 'valid_pass',
'expected': 'error'},
    {'test_type': 'login', 'username': 'valid_user', 'password': 'wrong_pass',
'expected': 'error'},
    {'test_type': 'signup', 'username': 'new_user', 'email': 'new@test.com',
'expected': 'success'}
]

# Write sample data
with open('login_test_data.csv', 'w', newline='') as file:
    fieldnames = ['test_type', 'username', 'password', 'email', 'expected']
    writer = csv.DictWriter(file, fieldnames=fieldnames)
    writer.writeheader()
    for data in login_data:
        writer.writerow(data)

# Use the data provider
data_provider = CSVTestDataProvider('login_test_data.csv')
login_tests = data_provider.get_test_data('login')

print("Login test data:")
for test in login_tests:
    print(f"User: {test['username']}, Password: {test['password']}, Expected:
{test['expected']}")
```

6. Error Handling with Files

```
def safe_file_operation(filename, operation='read'):
    try:
        if operation == 'read':
            with open(filename, 'r') as file:
                return file.read()
        elif operation == 'write':
            with open(filename, 'w') as file:
                file.write("Sample content")
                return "Write successful"

    except FileNotFoundError:
        print(f"Error: File '{filename}' not found!")
        return None

    except PermissionError:
        print(f"Error: Permission denied for file '{filename}'!")
        return None

    except IOError as e:
        print(f"Error: IO operation failed - {e}")
        return None

    except Exception as e:
        print(f"Unexpected error: {e}")
        return None

# Test error handling
result = safe_file_operation('nonexistent.txt', 'read')
result = safe_file_operation('test.txt', 'write')
```

7. File Utilities for Test Automation

```
import os
import shutil
from pathlib import Path

class FileHelper:
    @staticmethod
    def create_directory(dir_path):
        """Create directory if it doesn't exist"""
        Path(dir_path).mkdir(parents=True, exist_ok=True)

    @staticmethod
    def file_exists(file_path):
        """Check if file exists"""
        return Path(file_path).exists()

    @staticmethod
    def delete_file(file_path):
        """Safely delete file"""
        try:
            if Path(file_path).exists():
                Path(file_path).unlink()
                return True
        except Exception as e:
            print(f"Error deleting file: {e}")
        return False

    @staticmethod
    def copy_file(source, destination):
        """Copy file from source to destination"""
        try:
            shutil.copy2(source, destination)
            return True
        except Exception as e:
            print(f"Error copying file: {e}")
            return False

    @staticmethod
    def get_file_size(file_path):
        """Get file size in bytes"""
        return Path(file_path).stat().st_size if Path(file_path).exists() else 0

    @staticmethod
    def clean_old_files(directory, days_old=7):
        """Remove files older than specified days"""
        import time
        current_time = time.time()
        days_in_seconds = days_old * 24 * 60 * 60

        for file_path in Path(directory).glob('*'):
            if file_path.is_file():
                file_age = current_time - file_path.stat().st_mtime
                if file_age > days_in_seconds:
                    file_path.unlink()
                    print(f"Deleted old file: {file_path}")

# Usage examples
FileHelper.create_directory('test_reports')
FileHelper.create_directory('test_data/screenshots')

if FileHelper.file_exists('test_data.json'):
    print("Test data file exists!")
```

```
# Copy important files
FileHelper.copy_file('test_data.json', 'test_reports/backup_data.json')
```

Practice Exercises

Exercise 1: Test Data Manager

```
# Create a comprehensive test data manager that can:
# 1. Read test data from JSON and CSV files
# 2. Write test results to files
# 3. Generate summary reports

class TestDataManager:
    def __init__(self):
        self.test_results = []

    def load_test_cases_from_csv(self, filename):
        # Implement CSV loading
        pass

    def load_config_from_json(self, filename):
        # Implement JSON config loading
        pass

    def record_test_result(self, test_name, status, duration, error_msg=None):
        # Record test execution result
        pass

    def generate_html_report(self, output_file):
        # Generate HTML test report
        pass

# Test the implementation
```

Exercise 2: Log File Analyzer

```
# Create a log file analyzer that can:
# 1. Parse application log files
# 2. Extract error messages
# 3. Generate summary statistics

def analyze_log_file(log_file):
    """Analyze log file and return statistics"""
    stats = {
        'total_lines': 0,
        'error_count': 0,
        'warning_count': 0,
        'info_count': 0,
        'errors': []
    }

    # Implement log analysis logic
    return stats

# Sample log content for testing
sample_log = """2023-12-01 10:00:01 INFO Application started
2023-12-01 10:00:02 INFO User logged in: john_doe
2023-12-01 10:00:03 WARNING Database connection slow
2023-12-01 10:00:04 ERROR Failed to save user data
2023-12-01 10:00:05 INFO User logged out: john_doe"""
```

Key Takeaways

1. **Always use `with` statement** for file operations
2. **Handle exceptions** when working with files
3. **JSON** is perfect for configuration and structured data
4. **CSV** is ideal for tabular test data
5. **Path operations** should use `pathlib.Path` for better cross-platform support
6. **File utilities** make test automation more robust

Why This Matters for Selenium

File handling is essential because you'll:

- Read test data from JSON/CSV files
- Store configuration settings
- Generate test reports
- Save screenshots and logs
- Manage test artifacts

What we covered:

1. **Basic File Operations** - Opening, reading, writing files
2. **Reading Files** - Different methods and practical examples
3. **Writing Files** - Including test report generation
4. **JSON Files** - Perfect for configuration and test data
5. **CSV Files** - Ideal for tabular test data
6. **Error Handling** - Safe file operations
7. **File Utilities** - Helper functions for automation

Key highlights:

- **Test Report Generation** - Automated reporting functionality
- **Test Data Management** - JSON and CSV data providers
- **Configuration Management** - Reading settings from files
- **Error Handling** - Robust file operations
- **Real-world Examples** - Test case management, log analysis

Why this is crucial for Selenium:

- **Test Data**: Store login credentials, test scenarios in JSON/CSV
- **Configuration**: Browser settings, URLs, timeouts
- **Reports**: Generate test execution reports
- **Screenshots**: Save test evidence
- **Logs**: Capture and analyze test logs

Python Exception Handling

Exception handling is crucial for robust test automation. When tests run, many things can go wrong - elements not found, network issues, timeouts. Proper exception handling makes your tests stable and maintainable.

1. Understanding Exceptions

What are Exceptions?

Exceptions are errors that occur during program execution. Instead of crashing, Python allows you to "catch" and handle these errors gracefully.

Common Built-in Exceptions

```
# Common exceptions you'll encounter:

# 1. ValueError - Invalid value for operation
try:
    number = int("not_a_number")
except ValueError as e:
    print(f"ValueError: {e}")

# 2. TypeError - Wrong data type
try:
    result = "hello" + 5
except TypeError as e:
    print(f"TypeError: {e}")

# 3. KeyError - Dictionary key doesn't exist
try:
    data = {"name": "John"}
    age = data["age"]
except KeyError as e:
    print(f"KeyError: Missing key {e}")

# 4. IndexError - List index out of range
try:
    numbers = [1, 2, 3]
    value = numbers[5]
except IndexError as e:
    print(f"IndexError: {e}")

# 5. FileNotFoundError - File doesn't exist
try:
    with open("nonexistent.txt", "r") as file:
        content = file.read()
except FileNotFoundError as e:
    print(f"FileNotFoundError: {e}")

# 6. ZeroDivisionError - Division by zero
try:
    result = 10 / 0
except ZeroDivisionError as e:
    print(f"ZeroDivisionError: {e}")
```

2. Basic Try/Except Structure

Simple Exception Handling

```
def safe_division(a, b):
    """Safely divide two numbers"""
    try:
        result = a / b
        return result
    except ZeroDivisionError:
        print("Error: Cannot divide by zero!")
        return None

# Test the function
print(safe_division(10, 2))    # Output: 5.0
print(safe_division(10, 0))    # Output: Error message, then None

def safe_conversion(value):
    """Safely convert string to integer"""
    try:
        return int(value)
    except ValueError:
        print(f"'{value}' is not a valid number")
        return 0

# Test conversion
print(safe_conversion("123"))     # Output: 123
print(safe_conversion("abc"))     # Output: Error message, then 0
```

Catching Multiple Exceptions

```
def process_user_input(user_input):
    """Process user input with multiple exception handling"""
    try:
        # Try to convert to number and perform calculation
        number = float(user_input)
        result = 100 / number
        return f"Result: {result}"

    except ValueError:
        return "Error: Please enter a valid number"

    except ZeroDivisionError:
        return "Error: Cannot divide by zero"

    except Exception as e:
        return f"Unexpected error: {e}"

# Test with different inputs
print(process_user_input("10"))      # Output: Result: 10.0
print(process_user_input("0"))       # Output: Error: Cannot divide by zero
print(process_user_input("abc"))     # Output: Error: Please enter a valid
number
```

Multiple Exceptions in One Block

```
def read_and_process_file(filename):
    """Read file and process data with multiple exception types"""
    try:
        with open(filename, 'r') as file:
            data = file.read()
```

```
            lines = data.split('\n')
            numbers = [int(line) for line in lines if line.strip()]
            return sum(numbers) / len(numbers)  # Calculate average

    except (FileNotFoundError, PermissionError) as e:
        print(f"File access error: {e}")
        return None

    except ValueError as e:
        print(f"Data conversion error: {e}")
        return None

    except ZeroDivisionError:
        print("Error: File is empty or contains no valid numbers")
        return None

# Create test file
with open('numbers.txt', 'w') as f:
    f.write("10\n20\n30\n40")

# Test the function
average = read_and_process_file('numbers.txt')
print(f"Average: {average}")
```

3. Using Else and Finally

The Complete Try/Except Structure

```
def complete_file_operation(filename, data):
    """Demonstrate complete try/except/else/finally structure"""
    file_handle = None

    try:
        print("Attempting to open file...")
        file_handle = open(filename, 'w')
        file_handle.write(data)
        print("Data written successfully!")

    except PermissionError:
        print("Error: Permission denied to write file")
        return False

    except IOError as e:
        print(f"Error: IO operation failed - {e}")
        return False

    else:
        # Executes only if no exception occurred
        print("File operation completed without errors")
        return True

    finally:
        # Always executes, regardless of exceptions
        if file_handle:
            file_handle.close()
            print("File handle closed")
        print("Cleanup completed")

# Test the function
result = complete_file_operation('test_output.txt', 'Hello, World!')
print(f"Operation successful: {result}")
```

Practical Example with Database-like Operations

```
class DatabaseConnection:
    def __init__(self, connection_string):
        self.connection_string = connection_string
        self.connected = False

    def connect(self):
        print(f"Connecting to {self.connection_string}")
        # Simulate connection
        if "invalid" in self.connection_string:
            raise ConnectionError("Invalid connection string")
        self.connected = True
        print("Connected successfully!")

    def disconnect(self):
        if self.connected:
            self.connected = False
            print("Disconnected from database")

def perform_database_operation(connection_string, query):
    """Perform database operation with proper exception handling"""
    db = DatabaseConnection(connection_string)

    try:
        db.connect()
        print(f"Executing query: {query}")

        if "DROP" in query.upper():
            raise ValueError("Dangerous operation not allowed")

        # Simulate query execution
        result = f"Query '{query}' executed successfully"
        return result

    except ConnectionError as e:
        print(f"Connection failed: {e}")
        return None

    except ValueError as e:
        print(f"Query error: {e}")
        return None

    except Exception as e:
        print(f"Unexpected error: {e}")
        return None

    else:
        print("Database operation completed successfully")

    finally:
        # Always cleanup, even if exceptions occurred
        db.disconnect()
        print("Database cleanup completed")

# Test database operations
result1 = perform_database_operation("valid_db_connection", "SELECT * FROM
users")
print(f"Result 1: {result1}\n")

result2 = perform_database_operation("invalid_connection", "SELECT * FROM
users")
print(f"Result 2: {result2}\n")
```

```
result3 = perform_database_operation("valid_db_connection", "DROP TABLE users")
print(f"Result 3: {result3}")
```

4. Custom Exceptions

Creating Custom Exception Classes

```
# Custom exception classes for test automation
class TestAutomationError(Exception):
    """Base exception for test automation"""
    pass

class ElementNotFoundError(TestAutomationError):
    """Raised when web element is not found"""
    def __init__(self, element_locator, timeout=10):
        self.element_locator = element_locator
        self.timeout = timeout
        super().__init__(f"Element '{element_locator}' not found within
{timeout} seconds")

class TestDataError(TestAutomationError):
    """Raised when test data is invalid or missing"""
    def __init__(self, data_type, message="Invalid test data"):
        self.data_type = data_type
        super().__init__(f"{data_type}: {message}")

class BrowserError(TestAutomationError):
    """Raised when browser operations fail"""
    pass

# Using custom exceptions
class WebElementSimulator:
    def __init__(self):
        self.elements = {
            "login_button": True,
            "username_field": True,
            "submit_button": False  # Simulate missing element
        }

    def find_element(self, locator, timeout=10):
        if locator not in self.elements:
            raise ElementNotFoundError(locator, timeout)

        if not self.elements[locator]:
            raise ElementNotFoundError(locator, timeout)

        return f"Element '{locator}' found"

# Test custom exceptions
def test_login_flow():
    """Test login flow with custom exception handling"""
    simulator = WebElementSimulator()

    try:
        # Test finding existing element
        username = simulator.find_element("username_field")
        print(username)

        # Test finding missing element
        submit = simulator.find_element("submit_button")
        print(submit)
```

```
    except ElementNotFoundError as e:
        print(f"Test failed: {e}")
        print(f"Element locator: {e.element_locator}")
        print(f"Timeout used: {e.timeout}")
        return False

    except TestAutomationError as e:
        print(f"General automation error: {e}")
        return False

    return True

# Run the test
success = test_login_flow()
print(f"Test passed: {success}")
```

Practical Test Data Validation

```
class TestDataValidator:
    @staticmethod
    def validate_user_data(user_data):
        """Validate user test data with custom exceptions"""
        required_fields = ['username', 'password', 'email']

        if not isinstance(user_data, dict):
            raise TestDataError("User Data", "Data must be a dictionary")

        # Check required fields
        for field in required_fields:
            if field not in user_data:
                raise TestDataError("User Data", f"Missing required field:
{field}")

            if not user_data[field] or not user_data[field].strip():
                raise TestDataError("User Data", f"Field '{field}' cannot be
empty")

        # Validate email format (simple check)
        email = user_data['email']
        if '@' not in email or '.' not in email:
            raise TestDataError("Email", "Invalid email format")

        # Validate password strength
        password = user_data['password']
        if len(password) < 6:
            raise TestDataError("Password", "Password must be at least 6
characters")

        return True

def test_user_registration(test_users):
    """Test user registration with validation"""
    successful_tests = 0

    for i, user in enumerate(test_users, 1):
        try:
            print(f"\nTesting user {i}: {user.get('username', 'Unknown')}")
            TestDataValidator.validate_user_data(user)
            print("✓ Validation passed - User data is valid")
            successful_tests += 1

        except TestDataError as e:
            print(f"✗ Validation failed: {e}")
```

```
        except Exception as e:
            print(f"✗ Unexpected error: {e}")

    print(f"\nResults: {successful_tests}/{len(test_users)} users passed
validation")

# Test data with various scenarios
test_users = [
    {
        "username": "john_doe",
        "password": "password123",
        "email": "john@example.com"
    },
    {
        "username": "jane_smith",
        "password": "12345",  # Too short
        "email": "jane@example.com"
    },
    {
        "username": "",  # Empty username
        "password": "password123",
        "email": "empty@example.com"
    },
    {
        "username": "bob_wilson",
        "password": "securepass",
        "email": "invalid-email"  # Invalid email
    },
    {
        "username": "alice_brown",
        "password": "strongpassword",
        "email": "alice@example.com"
    }
]

# Run validation tests
test_user_registration(test_users)
```

5. Exception Handling Best Practices

1. Be Specific with Exceptions

```
# ✗ BAD: Too generic
def bad_example():
    try:
        # Some risky operation
        result = risky_operation()
    except Exception:
        print("Something went wrong")

# ✓ GOOD: Specific exception handling
def good_example():
    try:
        result = risky_operation()
    except ValueError as e:
        print(f"Invalid value provided: {e}")
    except ConnectionError as e:
        print(f"Network connection failed: {e}")
    except Exception as e:
        print(f"Unexpected error: {e}")
```

```
def risky_operation():
    # Simulate various possible errors
    import random
    error_type = random.choice(['value', 'connection', 'other'])

    if error_type == 'value':
        raise ValueError("Invalid input value")
    elif error_type == 'connection':
        raise ConnectionError("Network unavailable")
    else:
        raise RuntimeError("Unknown error occurred")
```

2. Log Exceptions Properly

```
import logging
import traceback

# Configure logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler('test_errors.log'),
        logging.StreamHandler()
    ]
)

class TestLogger:
    @staticmethod
    def log_exception(test_name, exception, include_traceback=True):
        """Log exception details for debugging"""
        error_msg = f"Test '{test_name}' failed: {type(exception).__name__}:
{exception}"
        logging.error(error_msg)

        if include_traceback:
            logging.error(f"Traceback:\n{traceback.format_exc()}")

def test_with_logging(test_name, test_function):
    """Run test with proper exception logging"""
    try:
        logging.info(f"Starting test: {test_name}")
        result = test_function()
        logging.info(f"Test '{test_name}' passed")
        return result

    except Exception as e:
        TestLogger.log_exception(test_name, e)
        return None

    finally:
        logging.info(f"Test '{test_name}' completed")

# Example test functions
def sample_test_1():
    return "Test 1 passed"

def sample_test_2():
    raise ValueError("Sample error for testing")

def sample_test_3():
    result = 10 / 0  # This will raise ZeroDivisionError
    return result
```

```
# Run tests with logging
test_with_logging("Sample Test 1", sample_test_1)
test_with_logging("Sample Test 2", sample_test_2)
test_with_logging("Sample Test 3", sample_test_3)
```

3. Retry Mechanism

```
import time
import random

def retry_on_exception(max_attempts=3, delay=1, exceptions=(Exception,)):
    """Decorator to retry function on specific exceptions"""
    def decorator(func):
        def wrapper(*args, **kwargs):
            for attempt in range(max_attempts):
                try:
                    return func(*args, **kwargs)
                except exceptions as e:
                    if attempt == max_attempts - 1:
                        raise e
                    print(f"Attempt {attempt + 1} failed: {e}")
                    print(f"Retrying in {delay} seconds...")
                    time.sleep(delay)
            return None
        return wrapper
    return decorator

# Example usage with network operations
@retry_on_exception(max_attempts=3, delay=2, exceptions=(ConnectionError,
TimeoutError))
def unreliable_network_call():
    """Simulate unreliable network call"""
    print("Making network call...")

    # Simulate random failure
    if random.random() < 0.7:  # 70% chance of failure
        raise ConnectionError("Network connection failed")

    return "Network call successful!"

# Test retry mechanism
try:
    result = unreliable_network_call()
    print(f"Success: {result}")
except Exception as e:
    print(f"Final failure: {e}")
```

6. Exception Handling for Test Automation

Test Framework with Exception Handling

```
import datetime
import traceback

class TestFramework:
    def __init__(self):
        self.test_results = []
        self.current_test = None

    def run_test(self, test_name, test_function, *args, **kwargs):
```

```
        """Run a single test with comprehensive exception handling"""
        self.current_test = test_name
        start_time = datetime.datetime.now()

        test_result = {
            'test_name': test_name,
            'start_time': start_time,
            'status': 'RUNNING',
            'error_message': None,
            'duration': None
        }

        try:
            print(f"\n▫ Running test: {test_name}")
            result = test_function(*args, **kwargs)

            test_result['status'] = 'PASSED'
            test_result['result'] = result
            print(f"✓ Test '{test_name}' PASSED")

        except AssertionError as e:
            test_result['status'] = 'FAILED'
            test_result['error_message'] = str(e)
            print(f"✗ Test '{test_name}' FAILED: {e}")

        except ElementNotFoundError as e:
            test_result['status'] = 'FAILED'
            test_result['error_message'] = f"Element not found: {e}"
            print(f"✗ Test '{test_name}' FAILED: {e}")

        except TestDataError as e:
            test_result['status'] = 'SKIPPED'
            test_result['error_message'] = f"Invalid test data: {e}"
            print(f"⚠ Test '{test_name}' SKIPPED: {e}")

        except Exception as e:
            test_result['status'] = 'ERROR'
            test_result['error_message'] = f"Unexpected error: {e}"
            test_result['traceback'] = traceback.format_exc()
            print(f"✹ Test '{test_name}' ERROR: {e}")

        finally:
            end_time = datetime.datetime.now()
            test_result['end_time'] = end_time
            test_result['duration'] = (end_time - start_time).total_seconds()
            self.test_results.append(test_result)
            self.current_test = None

    def run_test_suite(self, tests):
        """Run multiple tests"""
        print("🚀 Starting test suite execution...")

        for test_name, test_function, args, kwargs in tests:
            self.run_test(test_name, test_function, *args, **kwargs)

        self.print_summary()

    def print_summary(self):
        """Print test execution summary"""
        total = len(self.test_results)
        passed = sum(1 for t in self.test_results if t['status'] == 'PASSED')
        failed = sum(1 for t in self.test_results if t['status'] == 'FAILED')
        errors = sum(1 for t in self.test_results if t['status'] == 'ERROR')
        skipped = sum(1 for t in self.test_results if t['status'] == 'SKIPPED')
```

```
        print(f"\n📊 TEST SUMMARY")
        print("=" * 50)
        print(f"Total Tests: {total}")
        print(f"✅ Passed: {passed}")
        print(f"❌ Failed: {failed}")
        print(f"💥 Errors: {errors}")
        print(f"⚠️ Skipped: {skipped}")
        print(f"Success Rate: {(passed/total)*100:.1f}%" if total > 0 else "No
tests run")

# Sample test functions
def test_login_success():
    # Simulate successful login
    username = "valid_user"
    password = "valid_pass"

    if username == "valid_user" and password == "valid_pass":
        return "Login successful"
    else:
        raise AssertionError("Login failed")

def test_login_invalid_credentials():
    # Simulate login with invalid credentials
    username = "invalid_user"
    password = "wrong_pass"

    # This should fail
    if username != "valid_user":
        raise AssertionError("Invalid username provided")

def test_element_not_found():
    # Simulate element not found scenario
    element_id = "nonexistent_button"
    available_elements = ["login_button", "username_field", "password_field"]

    if element_id not in available_elements:
        raise ElementNotFoundError(element_id, timeout=10)

def test_invalid_data():
    # Simulate invalid test data
    user_data = {"username": "", "password": "test123"}

    if not user_data["username"]:
        raise TestDataError("User Data", "Username cannot be empty")

def test_unexpected_error():
    # Simulate unexpected error
    result = 10 / 0  # This will cause ZeroDivisionError

# Create and run test suite
framework = TestFramework()

test_suite = [
    ("Login Success Test", test_login_success, [], {}),
    ("Login Invalid Credentials", test_login_invalid_credentials, [], {}),
    ("Element Not Found Test", test_element_not_found, [], {}),
    ("Invalid Data Test", test_invalid_data, [], {}),
    ("Unexpected Error Test", test_unexpected_error, [], {})
]

framework.run_test_suite(test_suite)
```

7. Context Managers for Resource Management

```
class WebDriverManager:
    """Context manager for web driver with exception handling"""

    def __init__(self, browser_type="chrome"):
        self.browser_type = browser_type
        self.driver = None

    def __enter__(self):
        try:
            print(f"Starting {self.browser_type} browser...")
            # Simulate browser startup
            if self.browser_type == "invalid":
                raise BrowserError("Invalid browser type")

            self.driver = f"{self.browser_type}_driver_instance"
            print("Browser started successfully")
            return self.driver

        except Exception as e:
            print(f"Failed to start browser: {e}")
            raise

    def __exit__(self, exc_type, exc_value, traceback):
        try:
            if self.driver:
                print("Closing browser...")
                self.driver = None
                print("Browser closed successfully")
        except Exception as e:
            print(f"Error closing browser: {e}")

        # Handle exceptions that occurred in the with block
        if exc_type:
            print(f"Exception in test: {exc_type.__name__}: {exc_value}")
            return False  # Don't suppress the exception

# Using context manager
def test_with_browser():
    try:
        with WebDriverManager("chrome") as driver:
            print(f"Using driver: {driver}")
            print("Performing test actions...")

            # Simulate test that might fail
            if True:  # Change to False to test exception handling
                print("Test completed successfully")
            else:
                raise Exception("Test failed!")

    except BrowserError as e:
        print(f"Browser setup failed: {e}")
    except Exception as e:
        print(f"Test execution failed: {e}")

# Run test
test_with_browser()
```

Key Takeaways

1. **Always be specific** - Catch specific exceptions rather than generic `Exception`

2. **Use proper cleanup** - Use `finally` or context managers for resource cleanup
3. **Log exceptions** - Include proper logging for debugging
4. **Create custom exceptions** - For domain-specific errors in test automation
5. **Implement retry logic** - For handling flaky network/UI operations
6. **Don't ignore exceptions** - Always handle or re-raise appropriately
7. **Use context managers** - For automatic resource management

Why This Matters for Selenium

Exception handling is critical in Selenium because:

- **Elements may not be found** - Network delays, page loading issues
- **Timeouts occur** - Pages load slowly, elements appear late
- **Browser crashes** - Unexpected browser behavior
- **Network issues** - API calls, resource loading failures
- **Test data problems** - Invalid inputs, missing data

Proper exception handling makes your Selenium tests:

- More reliable and stable
- Easier to debug when failures occur
- Better at handling real-world scenarios
- More maintainable over time

What we covered:

1. **Understanding Exceptions** - Common built-in exceptions
2. **Try/Except Structure** - Basic and multiple exception handling
3. **Else and Finally** - Complete exception handling flow
4. **Custom Exceptions** - Creating domain-specific errors
5. **Best Practices** - Specific exceptions, logging, retry mechanisms
6. **Test Framework Integration** - Real-world test automation scenarios
7. **Context Managers** - Resource management with exception safety

Key highlights:

- **Custom Exception Classes** for test automation (ElementNotFoundError, TestDataError)
- **Test Framework Example** with comprehensive exception handling
- **Retry Mechanisms** for handling flaky operations
- **Proper Logging** for debugging failed tests
- **Context Managers** for browser resource management

Why this is crucial for Selenium:

Exception handling is absolutely essential because Selenium tests deal with:

- **Web elements that may not exist**
- **Network timeouts and delays**
- **Browser crashes and instabilities**

- **Dynamic content loading**
- **Test data validation issues**

Without proper exception handling, your Selenium tests will be fragile and hard to maintain.