Programming Paradigms & Exception handling

Novice

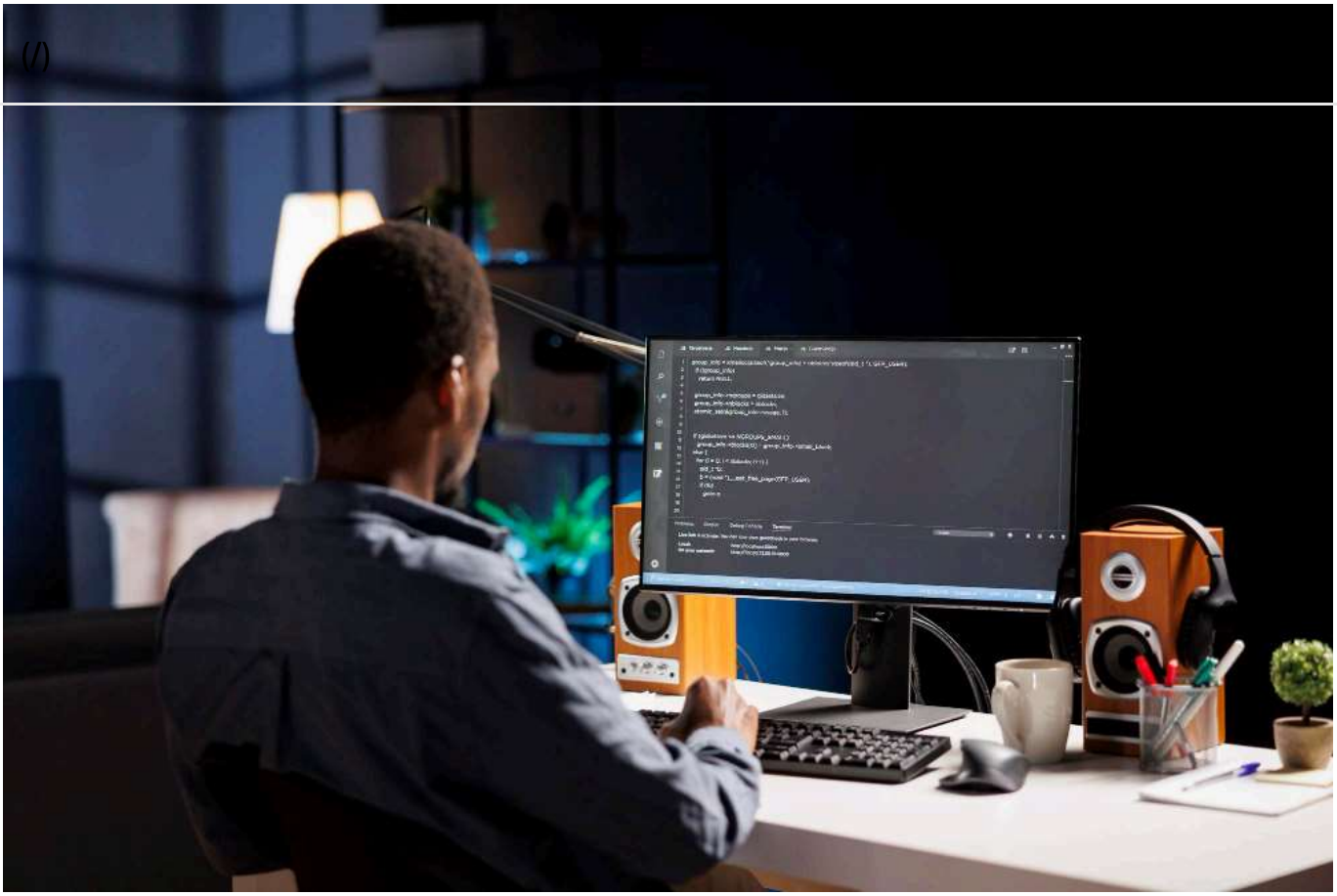
Weight: 1

Ongoing second chance project - started Sep 30, 2024 1:00 AM, must end by Oct 12, 2024 1:00 AM

An auto review will be launched at the deadline

In a nutshell…

- **Auto QA review:** 0.0/33 mandatory
- **Altogether: 0.0%**
  - Mandatory: 0.0%
  - Optional: no optional tasks



This project introduces you to the fundamental concepts of Object-Oriented Programming (OOP) in Python and Exception Handling. You'll learn about classes, objects, the benefits of OOP, explore how to handle errors gracefully, and be introduced to the basics of testing.

Project Objectives:

By the end of this project, you should be able to:

- Explain the core concepts of OOP: classes, objects, encapsulation, and abstraction.
- Discuss the significance of OOP in software development and its advantages over other programming paradigms.
- Define classes and create objects in Python.
- Understand the difference between class attributes, instance methods, and the role of the `self` keyword within classes.
- Differentiate between syntax errors and exceptions in Python.
- Identify common Python exceptions and understand their causes.
- Utilize `try`, `except`, `else`, and `finally` blocks to handle exceptions effectively.
- Raise exceptions using the `raise` keyword and create custom exceptions for specific errors in your code.
- Explain the importance of testing in software development.
- Describe different types of testing, with a focus on unit testing.
- Write basic unit tests using Python's `unittest` module to verify the functionality of your code.
- Structure test cases effectively and understand how test runners work.

This project equips you with the foundational knowledge of OOP and exception handling in Python. These skills are essential for building well-structured, maintainable, and robust Python applications.

(/)

Quiz questions

Great! You've completed the quiz successfully! Keep going! (Show quiz)

Tasks

0. Create a Simple Bank Account Class

mandatory

Score: 0.0% (*Checks completed: 0.0%*)

Objective: Understand the fundamentals of OOP in Python by implementing a `BankAccount` class that encapsulates banking operations. Use command line arguments to interact with instances of this class.

Task Description:

You will create two Python scripts: `bank_account.py`, which contains the `BankAccount` class, and `main-0.py`, which interfaces with the class through command line arguments to perform banking operations.

`bank_account.py`:

1. **Class Definition:**
   - Define a class named `BankAccount`.
   - Use the `__init__` method to initialize an `account_balance` attribute. Optionally, accept an initial balance parameter, defaulting to zero.
2. **Encapsulation and Behaviors:**
   - Implement `deposit(amount)`, `withdraw(amount)`, and `display_balance()` methods.
   - `deposit` should add the specified amount to `account_balance`.
   - `withdraw` should deduct the amount from `account_balance` if funds are sufficient, returning `True`; otherwise, return `False` and do not alter the balance.
   - `display_balance` should print the current balance in a user-friendly format.

`main-0.py` for Command Line Interaction:

This script utilizes `BankAccount` through command line arguments for banking operations.

(7)

```
import sys
from bank_account import BankAccount

def main():
    account = BankAccount(100)  # Example starting balance
    if len(sys.argv) < 2:
        print("Usage: python main.py <command>:<amount>")
        print("Commands: deposit, withdraw, display")
        sys.exit(1)

    command, *params = sys.argv[1].split(':')
    amount = float(params[0]) if params else None

    if command == "deposit" and amount is not None:
        account.deposit(amount)
        print(f"Deposited: ${amount}")
    elif command == "withdraw" and amount is not None:
        if account.withdraw(amount):
            print(f"Withdrew: ${amount}")
        else:
            print("Insufficient funds.")
    elif command == "display":
        account.display_balance()
    else:
        print("Invalid command.")

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

Sample Command Line Usage and Expected Outputs:

1. **Deposit:**

```
python main-0.py deposit:50
```

Expected Output: `Deposited: $50`

1. **Withdraw with Sufficient Funds:**

```
python main-0.py withdraw:20
```

Expected Output: `Withdrew: $20`

1. **Withdraw with Insufficient Funds:**

```
python main-0.py withdraw:150
```

Expected Output: `Insufficient funds.`

1. **Display Balance:**

```
(/) python main-0.py display
```

Expected Output: `Current Balance: $[amount]`

Implementation Notes for you:

- Ensure your `BankAccount` class in `bank_account.py` correctly implements the specified functionalities and adheres to the principles of encapsulation.
- Use `main.py` to test your `BankAccount` class by performing various operations. Adjust the initial balance as needed for testing different scenarios.
- This task combines learning OOP concepts with practical command line interaction, enhancing your understanding of Python programming.

Repo:

- GitHub repository: `alx_be_python`
- Directory: `programming_paradigm`
- File: `bank_account.py`

☐ Done? | Check your code | QA Review

1. Robust Division Calculator with Command Line Arguments

mandatory

Score: 0.0% (*Checks completed: 0.0%*)

Objective: Implement a division calculator that robustly handles errors like division by zero and non-numeric inputs using command line arguments.

Task Description:

Create two Python scripts: `robust_division_calculator.py`, which contains the division logic including error handling, and `main.py`, which interfaces with the user through the command line.

`robust_division_calculator.py`:

Define a function `safe_divide(numerator, denominator)` that performs division, handling potential errors:

- **Division by Zero:** Use a try-except block to catch `ZeroDivisionError`.
- **Non-numeric Input:** Attempt to convert arguments to floats. Use a try-except block to catch `ValueError` for non-numeric inputs.
- Return appropriate messages for errors or the result for successful division.

`main.py` for Command Line Interaction:

This script will import `safe_divide` from `robust_division_calculator.py` and use it to divide numbers provided as command line arguments.

(7)

```
import sys
from robust_division_calculator import safe_divide

def main():
    if len(sys.argv) != 3:
        print("Usage: python main.py <numerator> <denominator>")
        sys.exit(1)

    numerator = sys.argv[1]
    denominator = sys.argv[2]

    result = safe_divide(numerator, denominator)
    print(result)

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

Expected Behavior:

The script is executed from the command line with two additional arguments representing the numerator and denominator. Here are sample commands and the expected outputs:

- **Normal Division:**

```
python main.py 10 5
```

Expected Output: `The result of the division is 2.0`

- **Division by Zero:**

```
python main.py 10 0
```

Expected Output: `Error: Cannot divide by zero.`

- **Invalid Input (Non-numeric):**

```
python main.py ten 5
```

Expected Output: `Error: Please enter numeric values only.`

Implementation Notes for you:

- Focus on error handling within `safe_divide` in `robust_division_calculator.py`. Ensure you cover the scenarios detailed above.
- Test your function using `main.py` by passing different types of inputs via command line arguments. This method allows you to quickly assess how well your error handling works in various situations.
- This task helps you practice writing error-resistant code, a crucial skill in software development.

Repo:

- GitHub repository: `alx_be_python`
- Directory: `programming_paradigm`
- File: `robust_division_calculator.py`

☐ Done? | Check your code | QA Review

2. Writing Unit Tests for a Simple Calculator Class

mandatory

Score: 0.0% (*Checks completed: 0.0%*)

Objective: Learn the basics of unit testing in Python by writing tests for a provided `SimpleCalculator` class that supports addition, subtraction, multiplication, and division operations.

Provided: `simple_calculator.py`

You're given a `SimpleCalculator` class with basic arithmetic operations. Your task is to write unit tests to verify its correctness.

```
# simple_calculator.py

class SimpleCalculator:
    """A simple calculator class that supports basic arithmetic operations."""

    def add(self, a, b):
        """Return the addition of a and b."""
        return a + b

    def subtract(self, a, b):
        """Return the subtraction of b from a."""
        return a - b

    def multiply(self, a, b):
        """Return the multiplication of a and b."""
        return a * b

    def divide(self, a, b):
        """Return the division of a by b. Returns None if b is zero."""
        if b == 0:
            return None
        return a / b
```

Task: Write Unit Tests in `test_simple_calculator.py`

Create a `test_simple_calculator.py` script to define and run unit tests for each method in the `SimpleCalculator` class. Your tests should cover various scenarios to ensure the class functions correctly.

Guidelines for Writing Tests:

1. **Import the Necessary Modules:**
    - Import the `unittest` module and the `SimpleCalculator` class from `simple_calculator.py`.

2. **Define a Test Class:**
(/)
   - Create a test class that inherits from `unittest.TestCase`.
3. **Write Test Methods:**
   - Write at least one test method for each operation (`add`, `subtract`, `multiply`, `divide`) provided by the `SimpleCalculator`.
   - Include tests for edge cases, such as dividing by zero.
4. **Use Assertions to Verify Results:**
   - Utilize `self.assertEqual()` to check for expected outcomes.
   - For the `divide` method, ensure you test both normal operation and division by zero.
5. **Running Your Tests:**
   - Run your tests using the command line: `python -m unittest test_simple_calculator.py`.

Example Test Method Structure:

```
import unittest
from simple_calculator import SimpleCalculator

class TestSimpleCalculator(unittest.TestCase):

    def setUp(self):
        """Set up the SimpleCalculator instance before each test."""
        self.calc = SimpleCalculator()

    def test_addition(self):
        """Test the addition method."""
        self.assertEqual(self.calc.add(2, 3), 5)
        self.assertEqual(self.calc.add(-1, 1), 0)
        # Add more assertions to thoroughly test the add method.

# Remember to write additional test methods for subtract, multiply, and divide.
```

Note for you:

- Your goal is to think like a tester and identify as many relevant test cases as possible for each method.
- Pay special attention to potential edge cases, such as division by zero, which could lead to unexpected behaviors if not properly handled.
- Writing comprehensive tests not only helps ensure your code is working correctly but also improves your understanding of how the code operates under different conditions.

Repo:

- GitHub repository: `alx_be_python`
- Directory: `programming_paradigm`
- File: `test_simple_calculator.py`

(/) Done? | Check your code | QA Review

3. Implementing Basic OOP for a Library Management System

mandatory

Score: 0.0% (*Checks completed: 0.0%*)

Objective: Solidify understanding of basic OOP concepts in Python by implementing a system that tracks books in a library, focusing on classes, object instantiation, and method invocation.

Your Task: `library_management.py`

- **Implement a `Book` class** with public attributes `title` and `author`, and a private attribute `_is_checked_out` to track its availability.
- **Implement a `Library` class** with a private list `_books` to store instances of `Book`. Include methods to `add_book`, `check_out_book(title)`, `return_book(title)`, and `list_available_books`.

Provided for Testing: `main.py`

This script demonstrates how to interact with your `Book` and `Library` classes.

```
from library_management import Book, Library

def main():
    # Setup a small library
    library = Library()
    library.add_book(Book("Brave New World", "Aldous Huxley"))
    library.add_book(Book("1984", "George Orwell"))

    # Initial list of available books
    print("Available books after setup:")
    library.list_available_books()

    # Simulate checking out a book
    library.check_out_book("1984")
    print("\nAvailable books after checking out '1984':")
    library.list_available_books()

    # Simulate returning a book
    library.return_book("1984")
    print("\nAvailable books after returning '1984':")
    library.list_available_books()

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

Expected Outputs for Each Step in `main.py`:

1. **After Initial Setup:**

```
Available books after setup:
Brave New World by Aldous Huxley
1984 by George Orwell
```

1. **After Checking Out '1984':**

```
Available books after checking out '1984':
Brave New World by Aldous Huxley
```

1. **After Returning '1984':**

```
Available books after returning '1984':
Brave New World by Aldous Huxley
1984 by George Orwell
```

Note for you:

- Your `Book` class should provide methods to check a book out and return it, affecting its availability.
- Your `Library` class needs to manage a collection of books, including adding new books to the collection, checking a book out (which marks it as unavailable), returning it (making it available again), and listing all available books.
- Implementing these functionalities requires careful thought about how objects interact with each other in terms of state and behavior.
- Use the provided `main.py` for testing your implementation. The expected outputs give you a clear indication of how your program should behave if implemented correctly.

Repo:

- GitHub repository: `alx_be_python`
- Directory: `programming_paradigm`
- File: `library_management.py`

☐ Done? | Check your code | >_ Get a sandbox | QA Review

« Back (/concepts/103450?project_id=100760)