Principles of Software Engineering and Data Bases

Davide Yi Xian Hu
Email: davideyi.hu@polimi.it
Date: 03 December 2024
Exercise Lecture: 06 - Introduction to Python

POLITECNICO
MILANO 1863

Exercise 1 - Contact Management System

Create a Python program that allows users to manage a contact list.

Implement a menu-driven system with options to add, view, search, and delete contacts using a dictionary to store contact details (name, phone, email).

Use functions for each operation, loops for menu navigation, and the `json` module for saving/loading data.

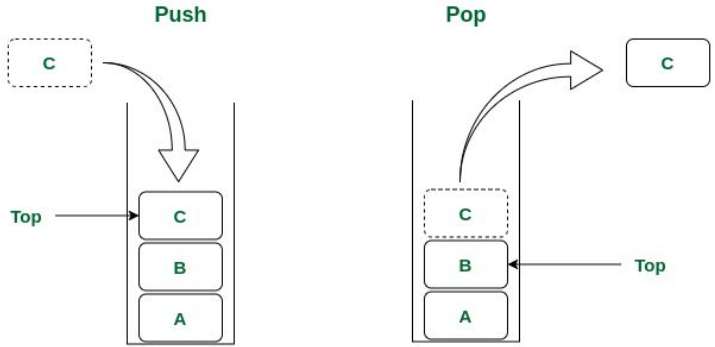
Incorporate input validation and use the `datetime` module to log the date when a contact is added.

Exercise 2 - Stack

Implement a program to simulate stack operations in Python.

The program should:

- Allow the user to add elements to the stack (**push** operation).
- Allow the user to remove the top element from the stack (**pop** operation).
- **Display** the current state of the stack after each operation.
- **Handle underflow** gracefully (when trying to pop from an empty stack).



Exercise 3 - Two Sum Problem

Given an array of integers `nums` and an integer `target`,
return the indices of the two numbers in the array such that they add up to target.

Additional: Use a dictionary to optimize the search process.

```
nums   = [2, 7, 11, 15]
target = 9

result = [0, 1]
```

Exercise 4 - Longest Substring

Given a string `s`, find the length of the longest substring without repeating characters.

```
s = "abcabcbb"
result = 3

s = "aaaaaa"
result = 1

s = "a!@#a!@"
result = 4
```

Exercise 5 - Attributes

```
class Dog:
    species = "Canis lupus familiaris"  # Class attribute (shared by all dogs)

    def __init__(self, name, age):
        self.name = name  # Instance attribute (unique to each dog)
        self.age = age    # Instance attribute (unique to each dog)

# Create two dog instances
dog1 = Dog("Buddy", 3)
dog2 = Dog("Lucy", 5)

# Access attributes
print(dog1.species)
print(dog2.name)

# Change the class attribute
Dog.species = "Canis familiaris"
print(dog1.species)
print(dog2.species)

# Change an instance attribute
dog1.name = "Max"
print(dog1.name)
print(dog2.name)
```

Exercise 5 - Attributes

```
class Dog:
    species = "Canis lupus familiaris"  # Class attribute (shared by all dogs)

    def __init__(self, name, age):
        self.name = name  # Instance attribute (unique to each dog)
        self.age = age    # Instance attribute (unique to each dog)

# Create two dog instances
dog1 = Dog("Buddy", 3)
dog2 = Dog("Lucy", 5)

# Access attributes
print(dog1.species)  # Output: Canis lupus familiaris (class attribute)
print(dog2.name)     # Output: Lucy (instance attribute)

# Change the class attribute
Dog.species = "Canis familiaris"
print(dog1.species)  # Output: Canis familiaris
print(dog2.species)  # Output: Canis familiaris

# Change an instance attribute
dog1.name = "Max"
print(dog1.name)     # Output: Max
print(dog2.name)     # Output: Lucy
```

Exercise 6 - Methods

```
class Employee:
    company = "TechCorp"  # Class attribute

    def __init__(self, name, salary):
        self.name = name  # Instance attribute
        self.salary = salary  # Instance attribute

    def get_info(self):  # Instance method
        return f"{self.name} works at {self.company} and earns {self.salary}."

    @classmethod
    def set_company(cls, new_company):  # Class method
        cls.company = new_company

    @staticmethod
    def is_high_salary(salary):  # Static method
        return salary > 100000

# Example usage
emp1 = Employee("Alice", 95000)
emp2 = Employee("Bob", 120000)

# Access instance method
print(emp1.get_info())

# Modify class attribute using class method
Employee.set_company("NewTech")
print(emp2.get_info())

# Use static method
print(Employee.is_high_salary(95000))
print(Employee.is_high_salary(120000))
```

Exercise 6 - Methods

```
class Employee:
    company = "TechCorp"  # Class attribute

    def __init__(self, name, salary):
        self.name = name  # Instance attribute
        self.salary = salary  # Instance attribute

    def get_info(self):  # Instance method
        return f"{self.name} works at {self.company} and earns {self.salary}."

    @classmethod
    def set_company(cls, new_company):  # Class method
        cls.company = new_company

    @staticmethod
    def is_high_salary(salary):  # Static method
        return salary > 100000

# Example usage
emp1 = Employee("Alice", 95000)
emp2 = Employee("Bob", 120000)

# Access instance method
print(emp1.get_info())  # Output: Alice works at TechCorp and earns 95000.

# Modify class attribute using class method
Employee.set_company("NewTech")
print(emp2.get_info())  # Output: Bob works at NewTech and earns 120000.

# Use static method
print(Employee.is_high_salary(95000))  # Output: False
print(Employee.is_high_salary(120000)) # Output: True
```

Exercise 7 - Properties

```
class Rectangle:
    def __init__(self, width, height):
        self._width = width  # Private attribute
        self._height = height  # Private attribute

    # Getter for width
    @property
    def width(self):
        return self._width

    # Setter for width with validation
    @width.setter
    def width(self, value):
        if value <= 0:
            raise ValueError("Width must be a positive value.")
        self._width = value

    # Getter for height
    @property
    def height(self):
        return self._height

    # Setter for height with validation
    @height.setter
    def height(self, value):
        if value <= 0:
            raise ValueError("Height must be a positive value.")
        self._height = value

    # Calculate area
    def area(self):
        return self._width * self._height

    # Calculate perimeter
    def perimeter(self):
        return 2 * (self._width + self._height)
```

```
if __name__ == "__main__":

    rect = Rectangle(10, 5)
    print(f"Width: {rect.width}, Height: {rect.height}")
    print(f"Area: {rect.area()}")
    print(f"Perimeter: {rect.perimeter()}")

    rect.width = 15
    rect.height = 8
    print(f"Updated Width: {rect.width},
            Updated Height: {rect.height}")
    print(f"Updated Area: {rect.area()}")
    print(f"Updated Perimeter: {rect.perimeter()}")

    try:
        rect.width = -5  # Invalid width
    except ValueError as e:
        print(f"Error: {e}")

    try:
        rect.height = 0  # Invalid height
    except ValueError as e:
        print(f"Error: {e}")

    rect._width = -10
    rect._height = -20
    print(f"Modified Width: {rect._width},
            Modified Height: {rect._height}")
    print(f"Area: {rect.area()}")
```

Exercise 8 - Bank Account

Create a BankAccount class to simulate basic banking operations.

The class should include attributes for account holder, account number, and balance, along with methods to deposit, withdraw, check balance, and transfer money between accounts.

Use a class attribute to represent the bank's name.

Demonstrate encapsulation by ensuring the balance can only be modified through methods and create multiple instances to show interaction between accounts.

Exercise 9 - Student

Create a Student class to manage student information and grades.

The class should include methods to add grades, calculate the average grade, retrieve grades for specific subjects, and generate a report card.

Use a dictionary to store grades by subject and implement a class method to initialize students from a data dictionary.

Exercise 10 - Task Management

Design and implement a Task Management System using Python classes.
The system should allow users to manage a list of tasks, each with attributes such as a title, description, due date, priority level, and completion status.

The system should support the following operations:
Add a new task: Create a task with all the necessary attributes.
Mark a task as complete: Change the status of a task to "completed."
View all tasks: Display all tasks, including their details and completion status.
View tasks by priority: Filter and display tasks based on their priority level.
Delete a task: Remove a task by its ID.

Exercise 11 - Product Management

Design a Product class that uses the `@property` decorator to manage private attributes.

Implement attributes with read-only (name), write-only (discount), and both getter and setter (price) access.

Include validation for setting values and demonstrate encapsulation and control over attribute access.