W7: Object Oriented Programming 1

and Exception Handling

Coding for Creative Robotics

Sonia Litwin & Rohit Ramesh Thampy
15/11/2024

Agenda

Today's Session Overview

Announcements:

- Creative Briefs Check-in
- Muse Headband Data Usage
- Additional Readings & Podcast

Lecture Topics:

- Introduction to Object-Oriented Programming
- Core Concepts of OOP
- Exception Handling in Python
- Q&A and Discussion

Projects Submission

Important Update

Announcement

Submission via Moodle is compulsory.

Please ensure your projects are uploaded before the 6th December deadline.

If you have any questions about this, please reach out after the lecture.

Creative Briefs

Project Progress Check-in

Discussion Points:

- How is everyone progressing with your projects?
- Any challenges you'd like to discuss?
- Success stories to share?

Remember:

These sessions are a chance to collaborate and problem-solve together.

The path to innovation often involves overcoming obstacles.

Muse Headband Data

Project Requirements Update

Important Notice:

• Muse Headband is NOT required for testing your projects

• In projects you will use the data that will be recorded next week

• Data is saved in CSV format

Available Resources:

• Rich insights for analysis

• Pattern exploration

• Emotional sensing data

Additional Resources

Supplementary Materials

Reading Materials:

- Emotion Sensing from EEG

Revisit last week's concepts

- Muse Documentation

Available on GitHub

Podcast Episode #2

An in-depth discussion on emotion sensing using EEG data

Please review these materials before our next class

Object-Oriented Programming

Introduction to OOP Concepts

Starting Point:

Let's examine this simple program to understand why OOP is powerful for modeling complex systems like robots.

This example will help us see the limitations of procedural programming.

```
name = input("Enter your name: ")
color = input("Enter your favorite color: ")
print(f"{name} likes {color}")
```

Revisiting Previous Code

Understanding Current Limitations

Simple Program Structure:

- Code runs from top to bottom
- Efficient for basic tasks
- Becomes unwieldy as complexity grows

Consider Adding:

- Location data
- Unique identifier
- Multiple attributes
- Complex relationships

Using Data Structures

a) Tuples

Understanding Tuples:

- Groups related values together
- Immutable collections
- Index-based access
- Best for short, simple data combinations

```
user = (name, color)
print(f"{user[0]} likes {user[1]}")

# Tuples are immutable
# user[0] = "New Name"  # Error!
```

Index positions can become unclear in larger programs

Using Data Structures

b) Lists

List Properties:

- Mutable collection
- Can modify contents
- Similar index-based limitations
- More flexible than tuples

```
user = [name, color]
print(f"{user[0]} likes {user[1]}")

# Lists are mutable
user.append("new_data")
user[0] = "New Name"  # Works!
```

Adding data makes positional access increasingly complex

Using Data Structures

c) Dictionaries

Better Organization:

- Uses descriptive keys
- Clear data access
- More intuitive structure
- Closer to real-world modeling

```
user = {
    "name": name,
    "color": color
}
print(f"{user['name']} likes {user['color']}")

# Adding new data is clearer
user["location"] = "London"
```

Still can't define functions tied to the user

Procedural vs. Object-Oriented

Different Approaches to Problem Solving

Procedural Programming:

- Code runs sequentially
- Functions act on data structures
- Data and behavior are separate
- Limited for complex systems

OOP Solution:

- Objects contain data and behavior
- Models real-world entities
- Natural organization
- Better for complex systems

Introducing Classes

The Heart of Custom Data

Real-world Objects:

Imagine a group of robots where each robot has:

- A name
- A color
- A main sensing ability

```
# Define our own data type
class Robot:
    def __init__(self, name, color, sensor):
        self.name = name
        self.color = color
        self.sensor = sensor

# Create a robot instance
robot1 = Robot("Bot1", "blue", "camera")
robot2 = Robot("Bot2", "red", "lidar")
```

This class acts as a blueprint for creating robots

Defining a Class

Creating Custom Data Types

Class Components:

- Initialization method
- Instance attributes
- Class methods

self.name = name links the instance's name attribute to the provided value

```
class RobotProfile:
    def __init__(self, name, color, sensor):
        self.name = name
        self.color = color
        self.sensor = sensor

    def describe(self):
        return f"{self.name} is {self.color} with
{self.sensor}"

# Each instance maintains independent data
bot1 = RobotProfile("Bot1", "blue", "camera")
bot2 = RobotProfile("Bot2", "red", "lidar")
```

Why Use Classes?

Benefits of Object-Oriented Design

Encapsulation:

Bundle data and methods within a single unit

Modularity:

Code is organized into logical units

Reusability:

Classes can be reused across programs

Real-World Modeling:

Easier to model complex entities

```
class Robot:
    def __init__(self, name):
        self.name = name
        self.sensors = []

    def add_sensor(self, sensor):
        self.sensors.append(sensor)

    def get_status(self):
        return f"{self.name} has {len(self.sensors)} sensors"

# Reusable across programs
robot1 = Robot("Explorer")
```

Concepts Introduction

Functional Paradigms

Traditional Approach:

• Abstraction: Think generally about the problem

• Decomposition: Break down into smaller functions

• Organization: Arrange functions to solve the problem

Limitation: Separates data from methods, which isn't how we naturally think about solving problems

```
# Traditional functional approach
def move_robot(position, distance):
    return position + distance

def turn_robot(direction, angle):
    return direction + angle

# Functions separate from data
pos = 0
dir = 90
pos = move_robot(pos, 10)
dir = turn_robot(dir, 45)
```

Concepts Introduction

Object-Oriented World

Thinking in Objects:

- Abstraction Level:
  Think in terms of objects interacting
- Decomposition Level:
  Define kinds of objects
- Organization Level:
  Create instances and define interactions

```
class Robot:
    def __init__(self):
        self.position = 0
        self.direction = 90

    def move(self, distance):
        self.position += distance

    def turn(self, angle):
        self.direction += angle

# Object combines data and behavior
robot = Robot()
robot.move(10)
robot.turn(45)
```

Object-Oriented Programming

Core Definition and Principles

Key Principles:

- Encapsulation
- Inheritance
- Polymorphism

A programming paradigm centered around objects, which contain data (attributes) and behavior (methods)

```
class Robot:
    def __init__(self, name):
        self._name = name  # Encapsulation

    def get_name(self):
        return self._name

class ArtBot(Robot):  # Inheritance
    def __init__(self, name, tool):
        super().__init__(name)
        self.tool = tool

    def draw(self):
        return f"Drawing with {self.tool}"
```

The Core Principles of OOP

Building Blocks of Object-Oriented Design

Key Principles:

- Abstraction: Simplifying complex systems
- Encapsulation: Bundling data and methods
- Inheritance: Creating hierarchical relationships
- Polymorphism: Different forms, same interface

Today we will focus on the first three principles

```
# Example combining multiple principles
class Robot:
    def __init__(self, name):
        self._name = name    # Encapsulation

    def move(self):         # Abstraction
        pass

class DrawingRobot(Robot): # Inheritance
    def __init__(self, name, tool):
        super().__init__(name)
        self._tool = tool

    def move(self):        # Polymorphism
        print(f"Moving while drawing with {self._tool}")
```

Abstraction

Using Abstract Base Classes (ABC)

Key Concepts:

- Focus on essential qualities
- Hide unnecessary details
- Enforce implementation rules
- Create consistent interfaces

Abstract classes ensure all subclasses implement required methods

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

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

    def area(self):
        return 3.14 * self.radius ** 2
```

Encapsulation

Data and Behavior Protection

Protection Mechanisms:

- Private attributes (underscore convention)
- Getter and setter methods
- Control over data access
- Implementation hiding

Encapsulation enables control over how data is accessed and modified

```
class ArtBot:
    def __init__(self, name, color):
        self._name = name    # Protected
        self._color = color  # Protected

    def get_color(self):
        return self._color

    def set_color(self, color):
        if isinstance(color, str):
            self._color = color
        else:
            raise ValueError(
                "Color must be a string"
            )
```

Inheritance

Extending Class Capabilities

Inheritance Features:

- Inherit properties and behaviors
- Extend functionality
- Override methods
- Use super() for parent methods

Create specialized classes while reusing common functionality

```
class Robot:
    def __init__(self, name):
        self.name = name

    def move(self):
        print(f"{self.name} is moving")

class ArtBot(Robot):
    def __init__(self, name, tool):
        super().__init__(name)
        self.tool = tool

    def draw(self):
        print(f"{self.name} draws with {self.tool}")
```

Agents & the Environment

Understanding Robot Interactions

Key Components:

- Agents: Individual robots with behaviors
- Environment: Space for interactions
- Interactions:
  - Detecting obstacles
  - Communicating with agents
  - Adjusting to environment

```
class Environment:
    def __init__(self):
        self.agents = []
        self.obstacles = []

    def add_agent(self, agent):
        self.agents.append(agent)

    def update(self):
        for agent in self.agents:
            agent.sense(self)
            agent.act(self)

class Agent:
    def sense(self, env):
        # Detect nearby objects
        pass

    def act(self, env):
        # Adjust behavior
        pass
```

Class Instances, Objects, and Methods

Working with Class Components

Key Concepts:

- Class Instances: Objects created from class
- Objects: Instances with data and behavior
- Methods: Functions defined within class

When we create an instance of Robot, we're making a specific object with all the capabilities described in Robot

```
class Robot:
    def __init__(self, name, type="generic"):
        self.name = name
        self.type = type
        self.status = "idle"

    def start(self):
        self.status = "active"
        return f"{self.name} is now {self.status}"

# Creating instances
helper = Robot("Helper", "assistant")
cleaner = Robot("Cleaner", "maintenance")

# Each instance has its own state
helper.start()  # "Helper is now active"
print(cleaner.status)  # still "idle"
```

Interface

Defining Consistent Interactions

Interface Concepts:

- Set of methods a class must implement
- Defines interaction patterns
- Creates consistent behaviors
- Enables polymorphism

Any class that inherits from Drawable must have a draw() method

```
from abc import ABC, abstractmethod

class Drawable(ABC):
    @abstractmethod
    def draw(self):
        pass

class CircleBot(Drawable):
    def draw(self):
        return "Drawing a circle"

class SquareBot(Drawable):
    def draw(self):
        return "Drawing a square"

# Both classes implement the same interface
bots = [CircleBot(), SquareBot()]
for bot in bots:
    print(bot.draw())
```

Exception Handling

Managing Runtime Errors

Key Components:

- Try blocks for risky code
- Except blocks for error handling
- Else for success case
- Finally for cleanup

Gracefully handle errors to prevent program crashes

```
try:
    bot = ArtBot("PainterBot", "Blue")
    bot.draw()
except ValueError as e:
    print(f"Error: {e}")
else:
    print("Operation successful")
finally:
    print("Cleanup complete")
```

Exception Types

Common Python Exceptions

Frequently Used:

- ValueError: Invalid values
- TypeError: Wrong data type
- FileNotFoundError: Missing files
- ZeroDivisionError: Division by zero

Understanding common exceptions helps write more robust code

```
def process_robot_data(data):
    try:
        # Type error if not a number
        speed = float(data['speed'])

        # Value error if negative
        if speed < 0:
            raise ValueError(
                "Speed must be positive"
            )

        # Zero division error possible
        acceleration = distance / time

    except (TypeError, ValueError,
            ZeroDivisionError) as e:
        print(f"Error: {e}")
```

Tutorial Time!

Sonia Litwin & Rohit Ramesh Thampy

15/11/2024