Sophia Brooks

06/03/2025

IT FDN 110 A

Assignment #7

https://github.com/SEBrooks25/IntroToProg-Python-Mod07

# Classes & Objects

## Introduction

As programming projects grow in complexity, the importance of organizing code in a structured, readable, and reusable way becomes more critical. Object-oriented programming (OOP) offers a powerful approach for managing complexity by grouping related data and functionality into self-contained units called classes. Classes serve as blueprints for creating objects, which store their own data and provide methods for interacting with it. Concepts like abstraction, encapsulation, inheritance, and data validation all contribute to cleaner, more maintainable code. Distinguishing between different types of classes, such as data classes and processing classes, helps divide responsibilities and align with the Separation of Concerns (SoC) design pattern. Within data classes, tools like constructors, attributes, and properties enable detailed control over how data is stored, accessed, and protected. Using the self keyword allows each object to manage its own state, ensuring that objects remain independent even when built from the same class. Data validation practices, such as private attributes and property setters, help ensure program stability by catching errors early and enforcing data rules. Object-oriented code can also integrate smoothly with practical needs like file storage, using custom objects internally while converting them to dictionary format for JSON compatibility. In addition to writing effective code, developers benefit from using tools like Git, GitHub Desktop, and PyCharm to manage changes, collaborate with others, and track the history of their projects over time.

## Statements, Functions, & Classes

As programs grow in size and complexity, structuring code in a logical and consistent way becomes increasingly important. One of the core organizing tools in object-oriented programming is the class, which serves as a blueprint for creating objects. While a class defines the structure and behavior shared by all its objects (Figure 1), each object instance stores its own unique set of data. This separation supports isolation, ensuring that changes to one object don’t affect others. It also promotes reusability, since a single class can be used to generate many independent objects. By interacting with each object through the class’s public methods and attributes, programs benefit from abstraction, hiding the internal workings of each object behind a consistent and easy-to-use interface. These features make object-oriented programs easier to manage and extend as they develop.

When organizing code into classes, it's useful to think about their roles within the program. Some classes are designed specifically to store and manage data, while others focus on actions or processing. A data class holds information related to a single concept, such as a student's name and GPA. In contrast, a processing class might handle tasks like saving data to a file or converting it into a different format. Separating these responsibilities makes it easier to apply the Separation of Concerns design pattern, where each part of the program has a clear, focused role. Using different classes for data and processing also allows for better testing, flexibility, and reuse across projects. As programs expand, this separation becomes essential for maintaining clarity and reducing the chance of bugs.

Data classes typically include several key components: attributes, constructors, properties, and methods. Attributes are variables that store an object's data (Figure 1). Constructors are special methods, such as *__init__*, that run when a new object is created and set the initial values for its attributes (Figure 1). To protect and control access to data, data classes often use properties, which are special getter and setter methods. These properties make it possible to include validation or formatting whenever data is retrieved or updated. Inside class methods, the self keyword is used to refer to the specific object the method is working with. This allows each object to manage its own data, even when many objects are created from the same class. Using self ensures that changes made in one object don't interfere with others, preserving each object's independence. This pattern supports both encapsulation and abstraction, two fundamental principles of object-oriented design that help keep programs organized, scalable, and reliable.

```
# Define a class
class Book:  1 usage
    def __init__(self, title, author):
        self.title = title
        self.author = author

    def display_info(self):  1 usage
        print(f"Title: {self.title}, Author: {self.author}")

# Create an object from the class
my_book = Book( title: "Charlotte's Web",  author: "E. B. White")

# Use a method from the class
my_book.display_info()
```

Figure 1: Shows an example of class defining, attributes (*title, author*), constructor *__init__*,

## Adding Data Validation

As data becomes more central to a program's functionality, controlling how that data is accessed and modified is critical. One of the first steps in improving data security and consistency is marking attributes as private. This is typically done by adding an underscore before an attribute name. A single underscore (*_name*) signals that the attribute is intended for internal use, while a double underscore (*__name*) makes it harder to access the attribute

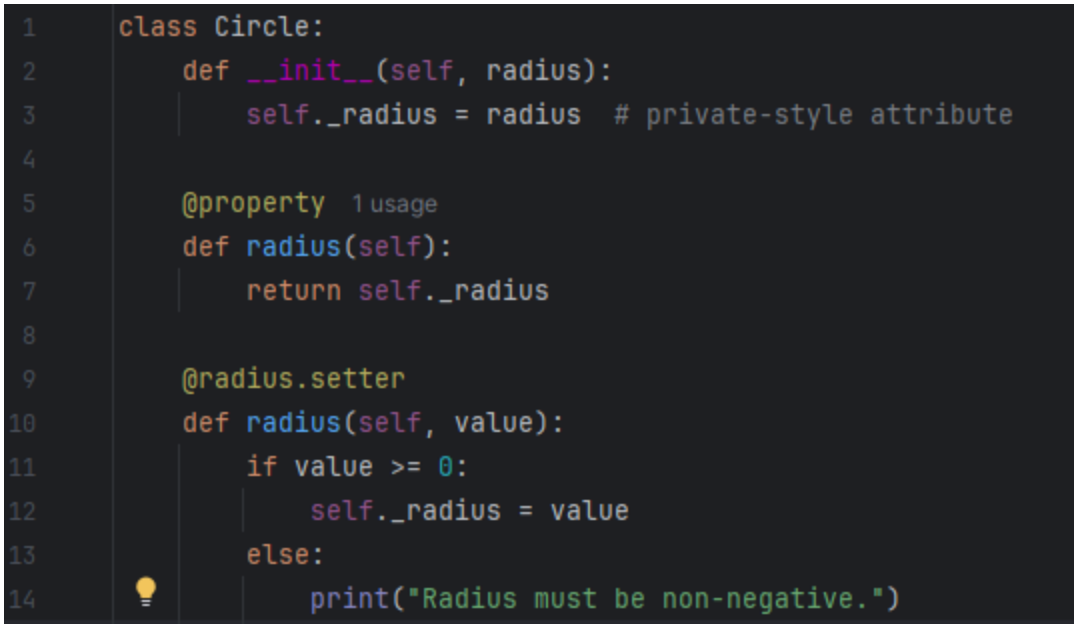
directly by enabling name mangling behind the scenes (Figure 2). Although Python does not strictly enforce these protections, using private attributes helps guide developers to interact with data in safe and predictable ways. This practice becomes especially important when attributes are tied to validation rules, since accessing or changing them directly could bypass necessary checks and lead to errors.

To manage access to private attributes and enforce validation rules, programs use special functions called properties. A property acts as a gatekeeper, wrapping data access in logic that can include formatting, validation, or error handling. Properties are defined using the @property decorator for getters and the @attribute_name.setter decorator for setters (Figure 3). This setup allows the code to appear as if it's accessing an attribute directly, while actually calling a method behind the scenes. For example, a property might allow a name to be retrieved in the title case or raise an error if a number is accidentally assigned to it. Using properties in this way adds a layer of protection and flexibility, making it easier to catch mistakes early and maintain clean, consistent data throughout the program.

While properties and private attributes control how data is accessed and changed, the broader concepts of abstraction and encapsulation explain why this structure is so effective. Abstraction focuses on simplifying complexity by exposing only the parts of a class that other code needs to interact with, hiding the inner workings behind a clear interface. Encapsulation, on the other hand, is about bundling the data (attributes) and the behavior (methods or properties) into a single, self-contained unit. These principles prevent accidental misuse of code and make it easier to update or extend functionality without breaking other parts of the program. When data is protected with private attributes, validated through properties, and organized with abstraction and encapsulation in mind, programs become more robust, less error-prone, and easier to maintain over time.

```
1  class Example:
2      def __init__(self):
3          self.name = "Public"            # No underscore: public
4          self._nickname = "Protected"    # One underscore: protected by convention
5          self.__secret = "Private"       # Two underscores: name mangled (private)
```

Figure 2: Shows the use of underscores for differing levels of security to attributes.



```python
class Circle:
    def __init__(self, radius):
        self._radius = radius  # private-style attribute

    @property  1 usage
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, value):
        if value >= 0:
            self._radius = value
        else:
            print("Radius must be non-negative.")
```

Figure 3: Shows the use of the "*@property*" and "*@attribute_name.setter*" decorators.

## Inherited Code

When working with classes in Python, certain built-in behaviors are automatically inherited from a special parent class called object. This inheritance includes several predefined magic methods, also known as dunder methods (short for "double underscore"), such as *__init__()* for constructors and *__str__()* for string representations (Figure 4). These methods are triggered automatically by Python during specific operations, such as creating an object or printing one. While these default versions are functional, they can be customized in user-defined classes to produce more meaningful behavior. Understanding how these methods work, and how to redefine them, is key to building clear and efficient object oriented programs.

In many cases, inherited methods are useful as is, but sometimes need to be adapted to suit the needs of a more specific class. This is done through a process called overriding, where a subclass redefines a method inherited from its parent class (Figure 5). The overridden method keeps the same name and parameters but replaces the inherited version with new behavior. For example, a parent class might have a *__str__()* method that returns just a name, while a child class overrides it to include additional information like a GPA or salary. Overriding methods allows each class to fine-tune its behavior while still benefiting from the reusable structure of the parent class.

These features highlight the power of inheritance, one of the core principles of object-oriented programming. Inheritance allows a child class to receive the attributes and methods of a parent class, reducing repetition and encouraging consistency across similar types of data. Instead of rewriting code for each new type of object, developers can build from existing classes and make modifications only where needed. This approach not only saves time but also makes programs easier to update and expand. Together with abstraction and encapsulation, inheritance helps form a complete object-oriented strategy, enabling developers to write modular, scalable, and maintainable code that adapts as projects grow in size and complexity.

```python
class Animal:  1 usage
    def __init__(self, species, sound):
        self.species = species
        self.sound = sound

    def __str__(self):
        return f"{self.species} says {self.sound}"

# Create object
a = Animal( species: "Dog",  sound: "Woof")

# Print object (calls __str__)
print(a)
```

Figure 4: Shows the use of the *__init__()* constructor and the *__str__()* for string representation.

```python
class Animal:  2 usages
    def speak(self):  1 usage
        return "Some generic sound"

class Dog(Animal):  1 usage
    def speak(self):  # Overriding the parent method  1 usage
        return "Woof!"

# Create objects
generic_animal = Animal()
dog = Dog()

# Call the method
print(generic_animal.speak())  # Output: Some generic sound
print(dog.speak())             # Output: Woof!

```

Figure 5: Shows the process of overriding.

## Working With Custom Objects

As programs shift from using dictionaries to custom classes like *Student*, working with structured objects becomes more organized and powerful. Each object contains its own attributes and behaviors, making it easier to manage data and apply validation through properties. However, when it comes time to save this data, especially to files like .json, objects must be converted back into a format that JSON can understand. Since the json module can only serialize basic data types like dictionaries, lists, and strings, objects need to be translated into dictionaries before writing them to a file. This extra step allows the program to take

advantage of object oriented design while still maintaining compatibility with external data storage.

To convert custom objects into JSON ready dictionaries, each object's attributes are manually extracted and used to build a dictionary with key-value pairs. For example, a Student object's *first_name*, *last_name*, and gpa attributes can be placed into a new dictionary and added to a list. This list of dictionaries can then be passed to *json.dump()* to save the data to a file (Figure 6). Converting data back to JSON is essential for preserving records, sharing data with other systems, or reloading it later. By combining custom object design with file conversion steps, programs become more flexible and ready for real-world data handling needs.

```python
import json

class Student:  1 usage
    def __init__(self, first_name, last_name, gpa):
        self.first_name = first_name
        self.last_name = last_name
        self.gpa = gpa

    def to_dict(self):  2 usages (1 dynamic)
        return {
            "first_name": self.first_name,
            "last_name": self.last_name,
            "gpa": self.gpa
        }

# Create a Student object
student1 = Student( first_name: "Alex",  last_name: "Rivera",  gpa: 3.8)

# Convert to dictionary and save to JSON
with open("Student.json", "w") as file:
    json.dump(student1.to_dict(), file, indent=2)
```

Figure 6: Shows the conversion of data back into a serialized format for a JSON file.

## Git vs GitHub

Git is a version control system that helps developers manage changes to their code over time. It tracks every update made to a file, allowing users to revisit earlier versions, identify what was changed, and by whom. One of Git's most powerful features is cloning, which allows developers to create a complete copy of a repository from a remote location, such as GitHub, to their local machine. From there, developers can create branches to work on new features or fixes without altering the main version of the code. These branches can be committed as checkpoints, each with a message that describes the changes made. When the changes are

ready, Git allows the branch to be merged back into the main branch, combining the updates into the original codebase. Git also supports remote repositories, which act as cloud-based backups that multiple people can access and contribute to. GitHub, a popular hosting service, integrates with Git to provide a visual interface, enabling collaboration, version history, and issue tracking from anywhere with internet access.

GitHub Desktop (Figure 7) is a graphical application that simplifies working with Git for those who prefer not to use command-line tools. It allows users to clone repositories, track changes, and push updates to GitHub using a straightforward point-and-click interface. This makes it easier to learn the basics of version control and manage project files without needing advanced Git knowledge. While more limited than terminal-based Git, GitHub Desktop is useful for smaller projects or for users who are just getting started.

For users working in PyCharm, Git integration provides a seamless way to manage version control directly from the development environment. To get started, a local Git repository can be created using the VCS > Create Git Repository menu. From there, changes to files are tracked automatically, and users can commit changes, view history, and revert edits without leaving the editor. To connect PyCharm to a GitHub account, users can go to File > Settings > GitHub, add an account, and authorize access. Once connected, projects can be shared to GitHub, and updates can be pushed and pulled directly from within PyCharm (Figure 8). This integration is especially helpful for streamlining the development process, as it keeps version control tools close to the code being written. It also improves team collaboration by syncing changes between local and remote repositories efficiently. Overall, using Git in PyCharm enhances workflow consistency, minimizes context switching, and encourages best practices in code management.

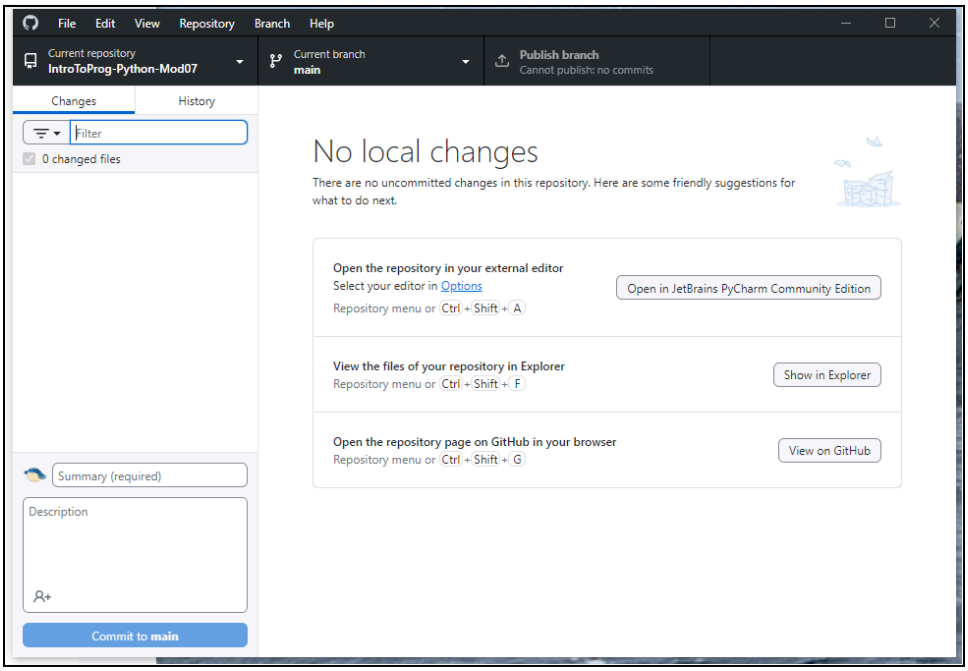


Figure 7: Shows GitHub Desktop window.

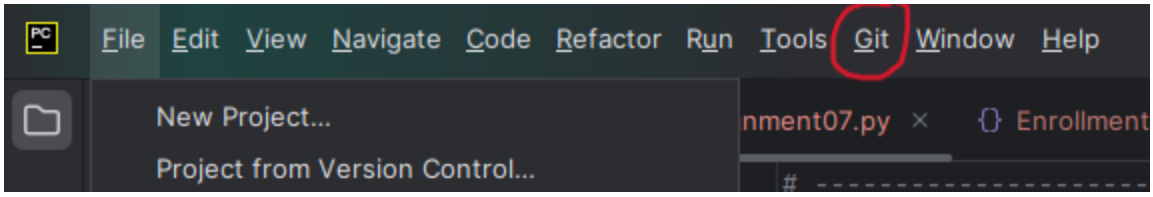


Figure 8: Shows the application of Git in PyCharm

## The Assignment

To extend the course registration program using the concepts introduced in Module 07, the script was restructured around class-based design and object-oriented programming. Four classes were created: *Person*, *Student*, *FileProcessor*, and *IO* (Figures 10, 11, 12, & 13 respectively). Each class was designed with a clear role to reinforce the Separation of Concerns pattern. The script begins by defining the two constants: MENU, which contains the interactive menu text, and FILE_NAME, which specifies the target JSON file for data persistence. These are declared at the top of the script to identify values that remain consistent throughout program execution (Figure 9). The students list was maintained as a global variable, but its structure was updated to store Student objects instead of dictionaries, offering better encapsulation and data integrity.

The *Student* class inherits structure from the *Person* class, and introduces a third attribute, *course_name*, which is managed using property methods. These properties include simple validation to ensure user input follows expected formats like alphabetic names. By incorporating class inheritance and property decorators, the code becomes more readable and better aligned with object-oriented best practices. The *to_dict()* method was added to the *Student* class to support converting objects into JSON-compatible dictionaries for file output. This method allowed the JSON file to retain a familiar format, with each record as a dictionary containing three key-value pairs: “first_name”, “last_name”, and “course_name”.

The *FileProcessor* and *IO* classes both use *@staticmethod* decorators and manage data processing and user interaction respectively. The *FileProcessor.read_data_from_file()* method uses *json.load()* to retrieve existing records from *Enrollments.json*, then converts each dictionary into a Student object that is appended to the students list. This ensures users can resume their work from previous sessions. The *write_data_to_file()* converts the list of *Student* objects back into dictionaries and saves them using *json.dump()*. Structured error handling was added to both methods to handle missing or corrupt files without terminating the program. The *IO* class supports all input and output operations, including menu display, data entry, and result presentation.

The script’s main loop (Figure 14) remains menu-driven, but each option now delegates functionality to methods in either the *FileProcessor* or *IO* class. Choosing option 1 invokes *input_student_data*, which prompts for input, validates the entries, creates a new Student object, and adds it to the global list. Option 2 calls *output_student_courses* to display all current registrations. Option 3 triggers saving to the JSON file, with feedback confirming the action. Option 4 exits the program. Wrapping the loop in an if *__name__ == "__main__":* block ensures the logic only runs when the script is executed directly. These improvements result in a more robust, modular application that is easier to test, debug, and expand in future assignments.

```python
import json

# Constants
MENU: str = '''---- Course Registration Program ----
  Select from the following menu:
    1. Register a Student for a Course
    2. Show current data
    3. Save data to a file
    4. Exit the program
----------------------------------------'''
FILE_NAME: str = "Enrollments.json"

# Variables
menu_choice: str = ""
students: list = []

```

Figure 9: Shows the defined constants, variables, and JSON file target.

```python
# Data Classes
class Person:  1 usage
    """Stores data about a person"""

    def __init__(self, first_name: str = "", last_name: str = ""):
        self.first_name = first_name
        self.last_name = last_name

    @property  2 usages
    def first_name(self) -> str:
        return self._first_name

    @first_name.setter  1 usage
    def first_name(self, value: str):
        if not value.isalpha():
            raise ValueError("First name must contain only letters.")
        self._first_name = value.strip().title()

    @property  2 usages
    def last_name(self) -> str:
        return self._last_name

    @last_name.setter  1 usage
    def last_name(self, value: str):
        if not value.isalpha():
            raise ValueError("Last name must contain only letters.")
        self._last_name = value.strip().title()

    def __str__(self) -> str:
        return f"{self.first_name},{self.last_name}"

```

Figure 10: Shows the *Person* class.

```python
class Student(Person):  2 usages
    """Stores data about a student (inherits Person)"""

    def __init__(self, first_name: str = "", last_name: str = "", course_name: str = ""):
        super().__init__(first_name, last_name)
        self.course_name = course_name

    @property  3 usages
    def course_name(self) -> str:
        return self._course_name

    @course_name.setter  1 usage
    def course_name(self, value: str):
        if not value:
            raise ValueError("Course name cannot be empty.")
        self._course_name = value.strip().title()

    def __str__(self) -> str:
        return f"{super().__str__()},{self.course_name}"

    def to_dict(self) -> dict:  1 usage (1 dynamic)
        return {"first_name": self.first_name, "last_name": self.last_name, "course_name": self.course_name}
```

Figure 11: Shows the *Student* class with inherited structure from the *Person* class.

```python
# Processing Classes
class FileProcessor:  2 usages
    """Processes data to and from a file"""

    @staticmethod  1 usage
    def read_data_from_file(file_name: str, student_data: list):
        """Reads JSON data from file and fill student_data with Student objects"""
        try:
            with open(file_name, "r") as file:
                data = json.load(file)
                student_data.clear()
                for item in data:
                    student = Student(item["first_name"], item["last_name"], item["course_name"])
                    student_data.append(student)
        except FileNotFoundError:
            IO.output_error_messages("File not found; starting with an empty list.")
        except Exception as e:
            IO.output_error_messages( message: "Error reading file.", e)

    @staticmethod  1 usage
    def write_data_to_file(file_name: str, student_data: list):
        """Writes student data to JSON file"""
        try:
            with open(file_name, "w") as file:
                json.dump([s.to_dict() for s in student_data], file)
        except Exception as e:
            IO.output_error_messages( message: "Error writing to file.", e)
```

Figure 12: Shows the *FileProcessor* class.

```python
# Presentation Classes
class IO:
    """Performs input and output"""

    @staticmethod  5 usages
    def output_error_messages(message: str, error: Exception = None):
        print("ERROR:", message)
        if error:
            print("DETAILS:", error)

    @staticmethod  1 usage
    def output_menu(menu: str):
        print(menu)

    @staticmethod  1 usage
    def input_menu_choice() -> str:
        return input("Please enter your choice: ").strip()

    @staticmethod  1 usage
    def input_student_data(student_data: list):
        try:
            first = input("Enter student's first name: ")
            last = input("Enter student's last name: ")
            course = input("Enter course name: ")
            student = Student(first, last, course)
            student_data.append(student)
        except Exception as e:
            IO.output_error_messages( message: "Invalid input for student registration.", e)

    @staticmethod  1 usage
    def output_student_courses(student_data: list):
        if not student_data:
            print("No student data available.")
        for student in student_data:
            print(student)
```

Figure 13: Shows *IO* class.

```python
# Menu Loop
while True:
    IO.output_menu(MENU)
    menu_choice = IO.input_menu_choice()

    if menu_choice == "1":
        IO.input_student_data(students)
    elif menu_choice == "2":
        IO.output_student_courses(students)
    elif menu_choice == "3":
        FileProcessor.write_data_to_file(FILE_NAME, students)
    elif menu_choice == "4":
        print("Exiting the program. Goodbye!")
        break
    else:
        IO.output_error_messages("Invalid menu option. Please choose 1-4.")

```

Figure 14: Shows Contents and use of the Menu Loop as a while loop.

## Summary

The skills and concepts introduced through object-oriented programming provide a strong foundation for writing scalable and reliable applications. Classes and objects make it possible to organize code in a way that supports reusability, modularity, and abstraction. Distinguishing between data and processing classes allows developers to maintain clear roles for different parts of the code, improving both flexibility and readability. Techniques like using private attributes, property methods, and constructors help enforce data integrity and provide built-in safeguards against user error. Concepts such as abstraction and encapsulation support a professional coding approach that hides complexity and focuses on clear interfaces. Inheritance and method overriding extend this power further, allowing one class to build on another while customizing its behavior as needed. Beyond the structure of the code itself, tools like Git and GitHub offer a safety net for experimentation and teamwork, while PyCharm's Git integration streamlines the development process from within the editor. These tools help ensure that work is saved, reviewed, and recoverable which is important for both solo and group development. Together, these practices form a comprehensive introduction to writing professional Python code. As these concepts are practiced and applied in real projects, they become essential tools in every developer's workflow.