Python Lecture 7: Advanced Exception Handling and Introduction to Object-Oriented Programming

September 26, 2025

1 Summary of File Types

This section provides a summary table of text and binary file types, highlighting their differences and including simple Python code examples for usage.

2 Try-Except-Else-Finally and Usage with Files

The try-except-else-finally block is used for exception handling in Python. - **try**: Contains code that might raise an exception. - **except**: Catches and handles the exception. - **else**: Executes if no exception occurs in the try block. - **finally**: Always executes, regardless of whether an exception occurred or not. Useful for cleanup tasks like closing files.

When working with files, these blocks ensure graceful handling of errors such as file not found or I/O issues.

2.1 Example 1: Basic File Reading with Try-Except-Else-Finally

```
try:
    with open('example.txt', 'r') as file:
        content = file.read()
except FileNotFoundError:
    print("File not found!")
except IOError:
    print("An I/O error occurred!")
else:
    print("File content:", content)
finally:
    print("File operation complete.")
```

If the file exists, it prints the content and "File operation complete." If not, it handles the error and still runs finally.

2.2 Example 2: Writing to File with Error Handling

```
try:
    with open('output.txt', 'w') as file:
```

```
        file.write("Hello, World!")
except IOError:
    print("Error writing to file!")
else:
    print("Write successful.")
finally:
    print("Cleanup done.")
```

This ensures the write operation is attempted, handles errors, confirms success in else, and always runs finally.

3 Starting in Object-Oriented Programming in Python

3.1 Definition of OOP and Its Characteristics

Object-Oriented Programming (OOP) is a programming paradigm that uses "objects" to design applications and computer programs. Objects are instances of classes and can contain data (attributes) and code (methods). OOP focuses on modeling real-world entities as software objects that interact with each other.

Key characteristics of OOP: - **Encapsulation**: Bundling data and methods that operate on the data within a single unit (class), restricting direct access to some components. - **Abstraction**: Hiding complex implementation details and showing only essential features. - **Inheritance**: Allowing a new class to inherit properties and methods from an existing class, promoting code reuse. - **Polymorphism**: Enabling objects of different classes to be treated as objects of a common superclass, typically through method overriding.

3.2 Difference Between Class and Object

- **Class**: A blueprint or template that defines the structure and behavior (attributes and methods) for creating objects. It does not occupy memory until an object is created. - **Object**: An instance of a class. It is a real entity that occupies memory and has specific values for the attributes defined in the class.

How to create an object from a class:

```
class MyClass:
    pass  # Class definition

obj = MyClass()  # Creating an object
```

3.3 Difference Between Procedural Programming and Object-Oriented Programming

- **Procedural Programming**: Focuses on functions and procedures to perform tasks. It follows a top-down approach, where the program is divided into functions that operate on data. Data and functions are separate. Examples: C, Pascal. - **Object-Oriented Programming**: Focuses on objects that combine data and functions. It follows a bottom-up approach, emphasizing data hiding, reusability, and modeling real-world scenarios. Examples: Python, Java.

Procedural is suitable for simple, linear tasks, while OOP is better for complex, modular systems.

3.4 Content of the Class in Python

A class in Python typically contains: - **Attributes** (data members): Variables that store data. - **Methods**: Functions that define behaviors. - **Constructor** (`__init__`): A special method to initialize objects. - Optional: Class variables, static methods, etc.

Example:

```
class Example:
    class_var = "Shared"  # Class variable

    def __init__(self, value):
        self.instance_var = value  # Instance attribute

    def method(self):
        print(self.instance_var)
```

3.5 Data Members and Methods of the Class

- **Data Members (Attributes)**: Variables associated with the class or its instances. - Instance attributes: Unique to each object (defined in `__init__`). - Class attributes: Shared among all instances (defined directly in the class). - **Methods**: Functions defined inside the class that operate on data members. The first parameter is usually `self` (referring to the instance).

3.6 Difference Between Private, Public, Protected Data Members and Methods

In Python, access modifiers are conventions rather than strict enforcements: - **Public**: No underscore prefix. Accessible from anywhere. Example: `self.var`. - **Protected**: Single underscore prefix (`_var`). Intended for use within the class and subclasses. Accessible outside but by convention, treated as non-public. - **Private**: Double underscore prefix (`__var`). Name mangling makes it harder to access outside the class (becomes `_ClassName__var`). Not truly private but discourages external access.

Example:

```
class AccessExample:
    def __init__(self):
        self.public_var = "Public"
        self._protected_var = "Protected"
        self.__private_var = "Private"
```

3.7 Complete Example: Class Rectangle

Here is a complete example of a Rectangle class with constructor, accessors (get methods), modifiers (set methods), area calculation, perimeter calculation, and a print method.

```python
class Rectangle:
    def __init__(self, length, width):
        self._length = length  # Protected attribute
        self._width = width    # Protected attribute

    # Accessors (Getters)
    def get_length(self):
        return self._length

    def get_width(self):
        return self._width

    # Modifiers (Setters)
    def set_length(self, length):
        if length > 0:
            self._length = length
        else:
            print("Length must be positive.")

    def set_width(self, width):
        if width > 0:
            self._width = width
        else:
            print("Width must be positive.")

    # Area method
    def rectangle_area(self):
        return self._length * self._width

    # Perimeter method
    def rect_perimeter(self):
        return 2 * (self._length + self._width)

    # Print details
    def print_rectangle(self):
        print(f"Length: {self._length}, Width: {self._width}")
        print(f"Area: {self.rectangle_area()}")
        print(f"Perimeter: {self.rect_perimeter()}")

# Usage example
rect = Rectangle(5, 3)  # Create object
rect.print_rectangle()  # Print details

# Use setters
rect.set_length(10)
rect.set_width(4)

# Use getters and methods
print("New Length:", rect.get_length())
print("New Area:", rect.rectangle_area())
```

```
rect.print_rectangle()  # Print updated details
```

Output:

```
Length: 5, Width: 3
Area: 15
Perimeter: 16
New Length: 10
New Area: 40
Length: 10, Width: 4
Area: 40
Perimeter: 28
```

Aspect	**Text Files**	**Binary Files**	**Simple Python Code**
Data Representation	Human-readable characters (e.g., ASCII, UTF-8). Stored as strings.	Raw binary data (bytes). Not human-readable.	-
File Modes	'r', 'w', 'a', 'r+', etc.	'rb', 'wb', 'ab', 'rb+', etc.	-
Use Cases	Configuration files, logs, scripts, CSV/JSON data. When data needs to be edited manually.	Images (JPEG, PNG), videos (MP4), executables, serialized objects. When preserving exact byte structure is crucial.	-
Advantages	Easy to read/edit with text editors. Platform-independent for plain text.	Efficient for non-text data. No encoding issues.	-
Disadvantages	Inefficient for large non-text data. Encoding/decoding required.	Not readable without specialized tools. Platform-dependent sometimes (e.g., endianness).	-

Code Example	Writing and Reading Text File
	`with open('text.txt', 'w')as f: f.write('Hello World')Oopen('text.txt', 'r')as f: print(f.read())#`
	`Output: Hello World`
	Writing and Reading Binary File
	`with open('binary.bin', 'wb')as f: f.write(b'Hello World')Oopen('binary.bin', 'rb')as f: print(f.read())#`
	`Output: b'Hello World'`		

Table 1: Summary of Text and Binary Files with Differences and Code Examples