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

September 30, 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 Examples of Classes

3.7.1 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
```

3.7.2 Class Square

A Square class, similar to Rectangle but with a single side length.

```
class Square:
    def __init__(self, side):
        self._side = side  # Protected attribute

    # Accessors (Getters)
    def get_side(self):
        return self._side

    # Modifiers (Setters)
    def set_side(self, side):
        if side > 0:
            self._side = side
        else:
            print("Side must be positive.")

    # Area method
    def square_area(self):
        return self._side ** 2

    # Perimeter method
    def square_perimeter(self):
        return 4 * self._side

    # Print details
    def print_square(self):
        print(f"Side: {self._side}")
        print(f"Area: {self.square_area()}")
        print(f"Perimeter: {self.square_perimeter()}")

# Usage example
sq = Square(4)  # Create object
sq.print_square()  # Print details

# Use setter
sq.set_side(6)
```

```
36
37 # Use getter and methods
38 print("New Side:", sq.get_side())
39 print("New Area:", sq.square_area())
40 sq.print_square()  # Print updated details
```

Output:

```
Side: 4
Area: 16
Perimeter: 16
New Side: 6
New Area: 36
Side: 6
Area: 36
Perimeter: 24
```

3.7.3 Class Cylinder

A Cylinder class with radius and height, calculating volume and surface area.

```
1  import math
2
3  class Cylinder:
4      def __init__(self, radius, height):
5          self._radius = radius  # Protected attribute
6          self._height = height  # Protected attribute
7
8      # Accessors (Getters)
9      def get_radius(self):
10         return self._radius
11
12     def get_height(self):
13         return self._height
14
15     # Modifiers (Setters)
16     def set_radius(self, radius):
17         if radius > 0:
18             self._radius = radius
19         else:
20             print("Radius must be positive.")
21
22     def set_height(self, height):
23         if height > 0:
24             self._height = height
25         else:
26             print("Height must be positive.")
27
28     # Volume method
29     def cylinder_volume(self):
30         return math.pi * (self._radius ** 2) * self._height
31
```

```
    # Surface area method
    def cylinder_surface_area(self):
        return 2 * math.pi * self._radius * (self._radius +
            self._height)

    # Print details
    def print_cylinder(self):
        print(f"Radius: {self._radius}, Height: {self._height}")
        print(f"Volume: {self.cylinder_volume():.2f}")
        print(f"Surface Area:
            {self.cylinder_surface_area():.2f}")

# Usage example
cyl = Cylinder(3, 5)  # Create object
cyl.print_cylinder()  # Print details

# Use setters
cyl.set_radius(4)
cyl.set_height(6)

# Use getters and methods
print("New Radius:", cyl.get_radius())
print("New Volume:", f"{cyl.cylinder_volume():.2f}")
cyl.print_cylinder()  # Print updated details
```

Output:

```
Radius: 3, Height: 5
Volume: 141.37
Surface Area: 150.80
New Radius: 4
New Volume: 301.59
Radius: 4, Height: 6
Volume: 301.59
Surface Area: 251.33
```

3.7.4 Class Complex

A Complex number class with real and imaginary parts, supporting basic operations.

```
class Complex:
    def __init__(self, real, imag):
        self._real = real  # Protected attribute
        self._imag = imag  # Protected attribute

    # Accessors (Getters)
    def get_real(self):
        return self._real

    def get_imag(self):
        return self._imag

```

```
    # Modifiers (Setters)
    def set_real(self, real):
        self._real = real

    def set_imag(self, imag):
        self._imag = imag

    # Add method
    def add(self, other):
        return Complex(self._real + other._real, self._imag +
            other._imag)

    # Subtract method
    def subtract(self, other):
        return Complex(self._real - other._real, self._imag -
            other._imag)

    # Multiply method
    def multiply(self, other):
        real = self._real * other._real - self._imag *
            other._imag
        imag = self._real * other._imag + self._imag *
            other._real
        return Complex(real, imag)

    # Modulus method
    def modulus(self):
        return (self._real ** 2 + self._imag ** 2) ** 0.5

    # Print details
    def print_complex(self):
        print(f"{self._real} + {self._imag}i")
        print(f"Modulus: {self.modulus():.2f}")

# Usage example
c1 = Complex(2, 3)  # Create object
c1.print_complex()  # Print details

c2 = Complex(1, 4)
c3 = c1.add(c2)
c3.print_complex()

c4 = c1.multiply(c2)
c4.print_complex()
```

Output:

```
2 + 3i
Modulus: 3.61
3 + 7i
Modulus: 7.62
-10 + 11i
```

```
Modulus: 14.87
```

3.7.5 Class Student

A Student class with name, age, and grades list.

```python
class Student:
    def __init__(self, name, age):
        self._name = name  # Protected attribute
        self._age = age    # Protected attribute
        self._grades = []  # Protected list

    # Accessors (Getters)
    def get_name(self):
        return self._name

    def get_age(self):
        return self._age

    def get_grades(self):
        return self._grades

    # Modifiers (Setters)
    def set_name(self, name):
        self._name = name

    def set_age(self, age):
        if age > 0:
            self._age = age
        else:
            print("Age must be positive.")

    # Add grade method
    def add_grade(self, grade):
        if 0 <= grade <= 100:
            self._grades.append(grade)
        else:
            print("Grade must be between 0 and 100.")

    # Average grade method
    def average_grade(self):
        if not self._grades:
            return 0
        return sum(self._grades) / len(self._grades)

    # Print details
    def print_student(self):
        print(f"Name: {self._name}, Age: {self._age}")
        print(f"Grades: {self._grades}")
        print(f"Average Grade: {self.average_grade():.2f}")

# Usage example
```

```python
stu = Student("Alice", 20)  # Create object
stu.add_grade(85)
stu.add_grade(92)
stu.print_student()  # Print details

# Use setters
stu.set_age(21)
stu.add_grade(78)

# Use getters and methods
print("New Age:", stu.get_age())
print("New Average:", f"{stu.average_grade():.2f}")
stu.print_student()  # Print updated details
```

Output:

```
Name: Alice, Age: 20
Grades: [85, 92]
Average Grade: 88.50
New Age: 21
New Average: 85.00
Name: Alice, Age: 21
Grades: [85, 92, 78]
Average Grade: 85.00
```

3.7.6 Class Circle

A Circle class with radius, calculating area and circumference.

```python
import math

class Circle:
    def __init__(self, radius):
        self._radius = radius  # Protected attribute

    # Accessors (Getters)
    def get_radius(self):
        return self._radius

    # Modifiers (Setters)
    def set_radius(self, radius):
        if radius > 0:
            self._radius = radius
        else:
            print("Radius must be positive.")

    # Area method
    def circle_area(self):
        return math.pi * (self._radius ** 2)

    # Circumference method
    def circle_circumference(self):
```

```
        return 2 * math.pi * self._radius

    # Print details
    def print_circle(self):
        print(f"Radius: {self._radius}")
        print(f"Area: {self.circle_area():.2f}")
        print(f"Circumference:
           {self.circle_circumference():.2f}")

# Usage example
circ = Circle(5)  # Create object
circ.print_circle()  # Print details

# Use setter
circ.set_radius(7)

# Use getter and methods
print("New Radius:", circ.get_radius())
print("New Area:", f"{circ.circle_area():.2f}")
circ.print_circle()  # Print updated details
```

Output:

```
Radius: 5
Area: 78.54
Circumference: 31.42
New Radius: 7
New Area: 153.94
Radius: 7
Area: 153.94
Circumference: 43.98
```

3.7.7 Class BankAccount

A BankAccount class with account number, balance, and methods for deposit/withdraw.

```
class BankAccount:
    def __init__(self, account_number, balance=0):
        self._account_number = account_number  # Protected
           attribute
        self._balance = balance                # Protected
           attribute

    # Accessors (Getters)
    def get_account_number(self):
        return self._account_number

    def get_balance(self):
        return self._balance

    # Deposit method
    def deposit(self, amount):
```

```
        if amount > 0:
            self._balance += amount
        else:
            print("Amount must be positive.")

    # Withdraw method
    def withdraw(self, amount):
        if 0 < amount <= self._balance:
            self._balance -= amount
        else:
            print("Invalid withdrawal amount.")

    # Print details
    def print_account(self):
        print(f"Account Number: {self._account_number}")
        print(f"Balance: {self._balance:.2f}")

# Usage example
acc = BankAccount("123456", 100)  # Create object
acc.print_account()  # Print details

# Deposit and withdraw
acc.deposit(50)
acc.withdraw(30)

# Use getters
print("New Balance:", acc.get_balance())
acc.print_account()  # Print updated details
```

Output:

```
Account Number: 123456
Balance: 100.00
New Balance: 120.0
Account Number: 123456
Balance: 120.00
```

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')0open('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')0open('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