|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| C:\Users\Admin\Desktop\download.jpg | USMAN INSTITUTE OF TECHNOLOGY | | | | | |  |
|  |  |  |  |  |  |  |  |
|  | Department of Computer Science  CS121 Object Oriented Programming | | | | | |  |
|  |  | Lab # 13  Exception Handling | | | |  |  |
|  | Objective:  This experiment introduces the students to the concept of Exception Handling and user defined Exception classes | | | | | |  |
|  | **Name of Student:**  **Roll No: Sec.**  **Date of Experiment:** | | | | | |  |
|  | **Marks Obtained/Remarks:**  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  **Signature:** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | | | | | |  |

**Lab 13: Exception Handling**

An error that occurs at runtime is also called an exception. Exception Handling is a mechanism to deal to with an exception. When a run time error occurs, the default exceptional control flow is to stop the program and print the error message. The lengthy error message is called a stack traceback or traceback. It gives information on the statement that caused the error by tracing back to the function calls that led to this statement. The line numbers of the function calls are also displayed in the error message for tracing the errors.

**Python Exception Handling Syntax**

try:

<body>

except <ExceptionType>:

<handler>

Wrap the code that might raise n exception in a try clause. <body> contains the code that may raise an exception. When an exception occurs, the rest of the code in <body> is skipped. If the exception matches an exception type, the corresponding handler is executed <handler> is the code that processes the exception.

*Example 1:*

def main():

myList = [4,2,5,1,6]

try:

for i in range(10):

print(myList[i])

except IndexError:

print("Out of Index")

main()

**Catching Multiple Exceptions**

* def main():
* try:
* number1, number2 = eval(
* input("Enter two numbers, separated by a comma: "))
* result = number1 / number2
* print("Result is", result)
* except ZeroDivisionError:
* print("Division by zero!")
* except SyntaxError:
* print("A comma may be missing in the input")
* except:
* print("Something wrong in the input")
* main()

**Raising Exceptions**

An exception is raised from a function. When a function detects an error, it creates an object from an appropriate exception class. It then throws the exception to the caller of the function and the information pertaining to an exception is wrapped in an object.

*Example 2:*

class Circle:

def \_\_init\_\_(self,r=0):

self.setRadius(r)

def setRadius(self,r):

if r < 0:

raise ValueError()

else:

self.\_\_radius = r

def getRadius(self):

return self.\_\_radius

* def main():
* try:
* c1 = Circle()
* print(c1.getRadius())
* c1.setRadius(-4)
* print(c1.getRadius())
* except ValueError:
* print("Invalid Input")
* main()

**Using Exception Objects**

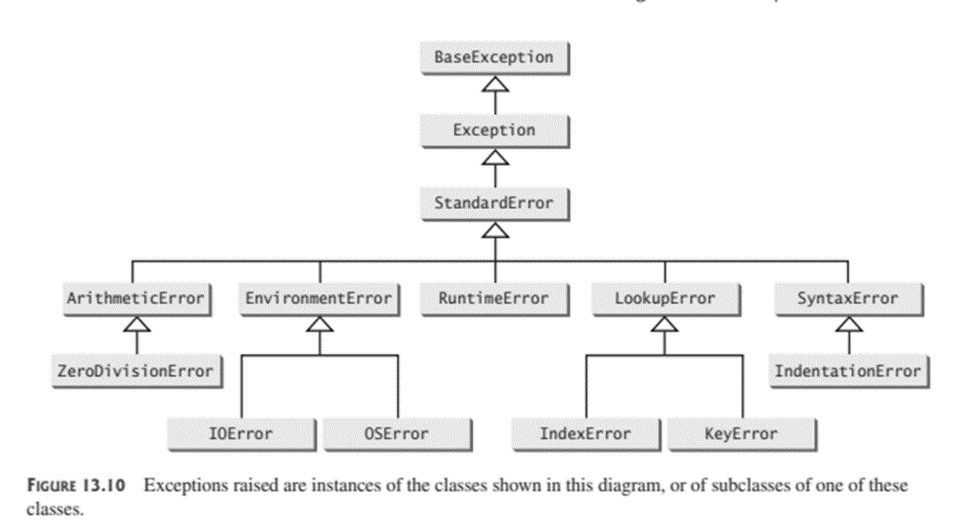
Exception object can be accessed in the except clause. To throw an exception, an exception object is created and is thrown using the raise keyword. So, an exception is wrapped in an object. When the except clause catches the exception, the exception object is assigned to a variable named ex. This object can now be used in the handler.

*Example 3:*

* class Circle:
* def \_\_init\_\_(self,r=0):
* self.setRadius(r)
* def setRadius(self,r):
* if r < 0:
* raise ValueError("Radius can not be negative")
* else:
* self.\_\_radius = r
* def getRadius(self):
* return self.\_\_radius
* def main():
* try:
* c1 = Circle()
* print(c1.getRadius())
* c1.setRadius(-4)
* print(c1.getRadius())
* except ValueError as ex:
* print(ex)
* main()

**User Defined Exceptions**

The BaseException class is the root of exception classes. All Python exception classes inherit directly or indirectly from BaseException. We can also define our own exception classes, derived from BaseException or from a subclass of BaseException.



*Example 4:*

* class NegativeRadiusException(BaseException):
* def \_\_init\_\_(self,r):
* super().\_\_init\_\_()
* self.radius = r
* class Circle:
* def \_\_init\_\_(self,r=0):
* self.setRadius(r)
* def setRadius(self,r):
* if r < 0:
* raise NegativeRadiusException(r)
* else:
* self.\_\_radius = r
* def getRadius(self):
* return self.\_\_radius
* def main():
* try:
* c1 = Circle()
* print(c1.getRadius())
* c1.setRadius(-4)
* print(c1.getRadius())
* except NegativeRadiusException as ex:
* print("Invalid Radius {} is entered".format(ex.radius))
* main()

# Student Exercise

Text

Description automatically generated

Text

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

1. Modify the Rational class Rational.py, to throw a RuntimeError exception if the denominator is 0.

**Code:**

def gcd(n,d):

    n1 = abs(n)

    n2 = abs(d)

    k = 1

    while k <= n1 and k <= n2:

        if n1%k==0 and n2%k==0:

            gcd = k

        k += 1

    return gcd

class Rational:

    def \_\_init\_\_(self,numerator=1,denominator=0):

        if denominator == 0:

            raise RuntimeError("Denominator cannot be zero")

        else:

            divisor = gcd(numerator,denominator)

            self.\_\_numerator = (1 if denominator > 0 else -1) \* int(numerator / divisor)

            self.\_\_denominator = int(abs(denominator) / divisor)

    def getNumerator(self):

        return self.\_\_numerator

    def getDenominator(self):

        return self.\_\_denominator

    def \_\_add\_\_(self,secondRational):

        n = self.\_\_numerator \* secondRational.getDenominator() + self.\_\_denominator \* secondRational.getNumerator()

        d = self.\_\_denominator \* secondRational.getDenominator()

        return Rational(n,d)

    def \_\_sub\_\_(self, secondRational):

        n = self.\_\_numerator \* secondRational.getDenominator() - self.\_\_denominator \* secondRational.getNumerator()

        d = self.\_\_denominator \* secondRational.getDenominator()

        return Rational(n,d)

    def \_\_mul\_\_(self, secondRational):

        n = self.\_\_numerator \* secondRational.getNumerator()

        d = self.\_\_denominator \* secondRational.getDenominator()

        return Rational(n,d)

    def \_\_float\_\_(self):

        return self.\_\_numerator / self.\_\_denominator

    def \_\_int\_\_(self):

        return int(self.\_\_float\_\_())

    def \_\_str\_\_(self):

        if self.\_\_denominator == 1:

            return str(self.\_\_numerator)

        else:

            return str(self.\_\_numerator) + "/" + str(self.\_\_denominator)

    def \_\_lt\_\_(self, secondRational):

        return self.\_\_cmp\_\_(secondRational) < 0.0

    def \_\_le\_\_(self, secondRational):

        return self.\_\_cmp\_\_(secondRational) <= 0.0

    def \_\_gt\_\_(self, secondRational):

        return self.\_\_cmp\_\_(secondRational) > 0.0

    def \_\_ge\_\_(self, secondRational):

        return self.\_\_cmp\_\_(secondRational) >= 0.0

    def \_\_cmp\_\_(self, secondRational):

        temp = self.\_\_sub\_\_(secondRational)

        if temp.getNumerator()>0:

            return 1

        elif temp.getNumerator()<0:

            return -1

        else:

            return 0

r1 = Rational(4,2)

r2 = Rational(2,3)

r3 = r1 + r2

print(r3)

r3 = r1 - r2

print(r3)

r3 = r1 \* r2

print(r3)

print(float(r3))

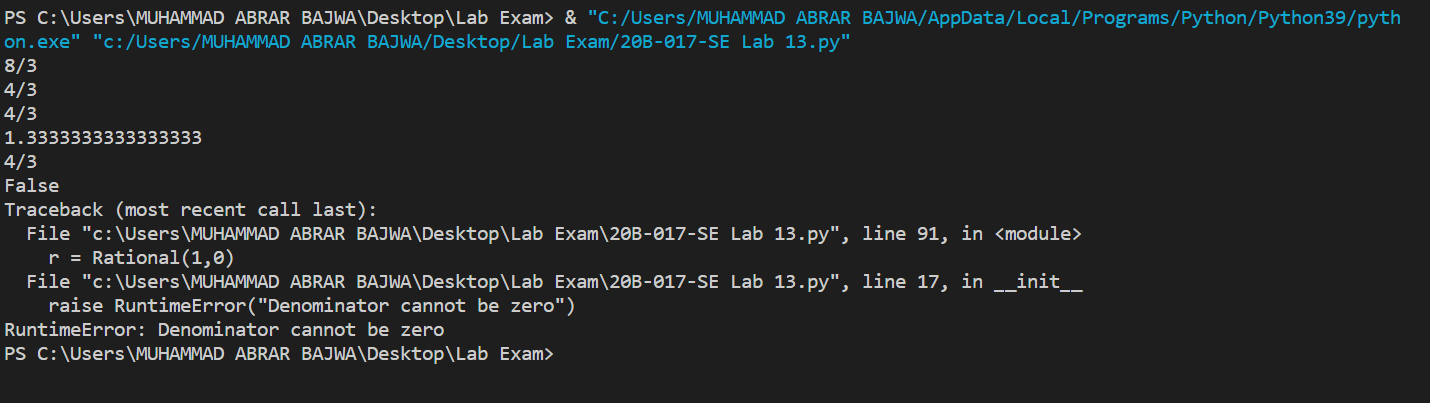
print(r3)

print(r1 < r2)

#for error

r = Rational(1,0)

**Output:**



1. Design a class named Triangle that extends the GeometricObject class. The Triangle class contains:

■ Three float data fields named side1, side2, and side3 to denote the three sides of the triangle.

■ A constructor that creates a triangle with the specified side1, side2, and side3 with default values 1.0.

■ The accessor methods for all three data fields.

■ A method named getArea() that returns the area of this triangle.

■ A method named getPerimeter() that returns the perimeter of this triangle.

■ A method named \_\_str\_\_() that returns a string description for the triangle.

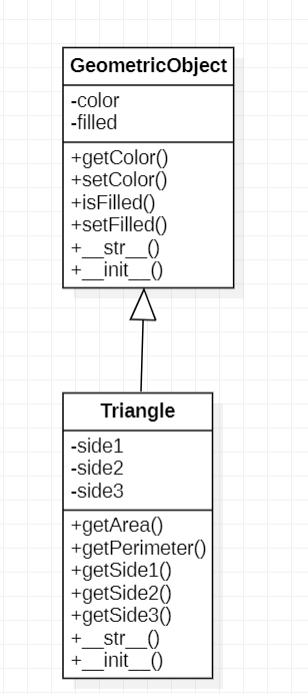
The \_\_str\_\_() method is implemented as follows:

return "Triangle: side1 = " + str(side1) + " side2 = " + str(side2) + " side3 = " + str(side3)

Draw the UML diagrams for the classes Triangle and GeometricObject. Implement the Triangle class. Write a test program that prompts the user to enter the three sides of the triangle, a color, and 1 or 0 to indicate whether the triangle is filled. The program should create a Triangle object with these sides and set the color and filled properties using the input. The program should display the

triangle’s area, perimeter, color, and True or False to indicate whether the triangle is filled or not.

**UML Diagram:**

****

**Code:**

class GeometricObject:

    def \_\_init\_\_(self,color="green",filled=True):

        self.\_\_color = color

        self.\_\_filled = filled

    def getColor(self):

        return self.\_\_color

    def setColor(self,color):

        self.\_\_color = color

    def isFilled(self):

        return self.\_\_filled

    def setFilled(self,filled):

        self.\_\_filled = filled

    def \_\_str\_\_(self):

        return "color: " + self.\_\_color + " and filled: " + str(self.\_\_filled)

class Triangle(GeometricObject):

    def \_\_init\_\_(self,side1=1.0,side2=1.0,side3=1.0):

        super().\_\_init\_\_()

        if side1 + side2 > side3 and side1 + side3 > side2 and side2 + side3 > side1:

            self.\_\_side1 = side1

            self.\_\_side2 = side2

            self.\_\_side3 = side3

        else:

            raise RuntimeError("The three given sides cannot form a triangle")

    def getSide1(self):

        return self.\_\_side1

    def getSide2(self):

        return self.\_\_side2

    def getSide3(self):

        return self.\_\_side3

    def getArea(self):

        s = (self.\_\_side1 + self.\_\_side2 + self.\_\_side3) / 2

        return (s\*(s-self.\_\_side1)\*(s-self.\_\_side2)\*(s-self.\_\_side3)) \*\* 0.5

    def getPerimeter(self):

        return self.\_\_side1 + self.\_\_side2 + self.\_\_side3

    def \_\_str\_\_(self):

        return f"Triangle\n\nSide1: " + str(self.\_\_side1) + " side2: " + str(self.\_\_side2) + " side3: " + str(self.\_\_side3)

**Output:**

**Graphical user interface, text

Description automatically generated**

1. Modify the Triangle class to throw a RuntimeError exception if the three given sides cannot form a triangle.

Define an exception class named TriangleError that extends RuntimeError. The TriangleError class contains the private data fields side1, side2, and side3 with accessor methods for the three sides of a triangle. Modify the Triangle class to throw a TriangleError exception if the three given sides cannot form a triangle.

**Code:**

class GeometricObject:

    def \_\_init\_\_(self,color="green",filled=True):

        self.\_\_color = color

        self.\_\_filled = filled

    def getColor(self):

        return self.\_\_color

    def setColor(self,color):

        self.\_\_color = color

    def isFilled(self):

        return self.\_\_filled

    def setFilled(self,filled):

        self.\_\_filled = filled

    def \_\_str\_\_(self):

        return "color: " + self.\_\_color + " and filled: " + str(self.\_\_filled)

class Triangle(GeometricObject):

    def \_\_init\_\_(self,side1=1.0,side2=1.0,side3=1.0):

        super().\_\_init\_\_()

        if side1 + side2 > side3 and side1 + side3 > side2 and side2 + side3 > side1:

            self.\_\_side1 = side1

            self.\_\_side2 = side2

            self.\_\_side3 = side3

        else:

            raise RuntimeError("The three given sides cannot form a triangle")

    def getSide1(self):

        return self.\_\_side1

    def getSide2(self):

        return self.\_\_side2

    def getSide3(self):

        return self.\_\_side3

    def getArea(self):

        s = (self.\_\_side1 + self.\_\_side2 + self.\_\_side3) / 2

        return (s\*(s-self.\_\_side1)\*(s-self.\_\_side2)\*(s-self.\_\_side3)) \*\* 0.5

    def getPerimeter(self):

        return self.\_\_side1 + self.\_\_side2 + self.\_\_side3

    def \_\_str\_\_(self):

        return f"Triangle\n\nSide1: " + str(self.\_\_side1) + " side2: " + str(self.\_\_side2) + " side3: " + str(self.\_\_side3)

class TriangleError(RuntimeError):

    def \_\_init\_\_(self,side1,side2,side3):

        self.\_\_side1 = side1

        self.\_\_side2 = side2

        self.\_\_side3 = side3

    def getSide1(self):

        return self.\_\_side1

    def getSide2(self):

        return self.\_\_side2

    def getSide3(self):

        return self.\_\_side3

    def \_\_str\_\_(self):

        return "Error! The three given sides cannot form a triangle"

tr = Triangle(2,1,1)

print(tr)

print("Area: ",tr.getArea())

print("Perimeter: ",tr.getPerimeter())

**Output:**

**Runtime Error**

Text

Description automatically generated