BACHELOR OF COMPUTER APPLICATIONS

SEMESTER 5

DCA3104

PYTHON PROGRAMMING

Unit 11

Exception Handling

Table of Contents

1. INTRODUCTION

In the previous units, you learned about variables and the parameters to define them. You understood all the keywords reserved in Python for specific actions and the different data types that store data in various forms, such as string, alphanumeric, integers, decimals, etc.

You got to know how Python is an object-oriented programming language. In Python, everything is classified as an object that has attributes and methods. These objects are stored as a collection in classes. We execute program codes through the methods or functions in Python that are associated with classes. The units so far also covered the topics of data structures, iterable and non-iterable elements, and the various operators used in Python. Data structures or sequences allow programmers to store collections of data and retrieve them easily. You have studied in-depth some of the most common data structures like lists, strings, tuples, sets, and dictionaries.

Conditional statements (branching) and looping are also a fundamental part of Python programming. Conditional statements define the conditions or parameters that the program has to evaluate before code execution in a sequential manner. On the other hand, loops are used to execute code as long as the conditions are satisfied. Unlike other programming languages, Python gives the programmer the flexibility to nest various conditionals and loops within each other, without any restrictions.

In this unit, you will learn about errors or exceptions. You will get to know why these errors arise and about the different types of errors. You will also learn about what to do when the exceptions come up and how you can solve them.

1.1 Learning Objectives:

At the end of this unit, you will:

- *Have a better understanding of code and program errors in Python.*
- *Know the different types of errors, and when and why they occur.*
- *Be able to use the try-except and try-except-else statement in exception handling.*
- *Understand multiple exceptions and know how to use the try and finally blocks.*
- *Learn how to raise exceptions and customise them.*

2. EXCEPTION HANDLING INTRODUCTION

In professional scenarios, programmers deal with vast amounts of codes that might be difficult to handle. Due to errors in code, the code script may not work as expected, run partially, abruptly end, or even "crash" and not run at all.

In programming, **error handling** involves the techniques and practices used to test and identify errors. Identifying these errors and rectifying them is known as **debugging**. During code execution, if an error arises, the program stops the program execution and shows a built-in error message known as an **exception**. Exception handling is a crucial part of programming. A front-end user may not understand an exception if it shows up and will not know how to proceed. This makes the programmers look unprofessional. Apart from this, if errors exist in a code it can expose an application, web page or server to security threats. Therefore, it is a programmer's responsibility to predict potential errors in the program and insert code that will take necessary actions in such a way that the front-end user does not receive an exception.

STUDY NOTE

When errors are predicted and addressed, they are known as **handled errors**. Otherwise, they are referred to as **unhandled errors**.

SELF-ASSESSMENT QUESTIONS - 1

1. Identifying and correcting errors is known as _____.
2. If there is an error in a program, Python throws an _____.
3. Techniques and practices used to test and identify errors is known as error handling. [True/False]
4. A programmer should always predict potential error and add code to deal with them. [True/False]
5. In Python, when program errors are anticipated and rectified they are called:

   a) corrected errors b) handled errors

   c) unhandled errors d) None of the above

3. COMMON ERRORS

In Python programming, an error may be caused due to several reasons and can be of different types. A **SyntaxError** occurs when the rules or syntax for coding program instructions are violated. They arise when a keyword is omitted or misspelt, the branching or looping structure is incomplete, inconsistency in indenting and variables, improper use of operators, functions, function arguments, and more.

Run-time errors occur during program execution if a value is processed not acceptable in the programming language. For example, if you try to divide a value by zero it will throw a run-time error as the value of zero is not mathematically determined. These errors also arise when a resource is accessed in a way that is not allowed. For example, if you call a variable or function that does not exist. Run-time errors cause the program to end abruptly or "crash".

STUDY NOTE

Logical errors are more difficult to predict and correct as Python does not throw any exceptions for these errors.

Logical errors or bugs arise when a program seems to run properly, but produces unexpected or wrong results. To avoid logical errors, the program must be tested repeatedly, and the results should be thoroughly scrutinized, verified, and validated.

3.1. Name Error

A **NameError** is one of the most common types of errors that arise in Python programming. Python throws a name error if you try to use a variable or a function name that is not defined or is not available in the local or global scope. There are a few instances when a NameError arise.

(i) **When the variable or function name is misspelt**

When you declare a variable or a function, Python stores the value with the exact name you have declared. It does not have the ability to assess misspelt words and relies on your variable spellings.

STUDY NOTE

Unlike other programming languages, Python does not require the program code to be enclosed in brackets.

```
books = ["Sherlock Holmes", "The Order", "The Alchemist"]
# Misspelled variable name
print(boooks)

#Output:
Traceback (most recent call last):
  File "main.py", line 3, in <module>
print(boooks)
NameError: name 'boooks' is not defined
```

In the above example, the variable "books" has been misspelt as "boooks" in the print statement. The error can be easily rectified by correcting the spelling:

```
books = ["Sherlock Holmes", "The Order", "The Alchemist"]
print(books)

#Output:
Sherlock Holmes
The Order
The Alchemist
```

(ii) **When a function is called before it is declared**

Python reads code in a sequential manner, i.e., from top to bottom. Hence, functions must be declared before they are called in a program.

```
books = ["Sherlock Holmes", "The Order", "The Alchemist"]
get_books(books)
# Defining a function
def get_books(books):
for b in books:
print(b)
```

```
#Output:
Traceback (most recent call last):
 File "<string>", line 3, in <module>
NameError: name 'get_books' is not defined
```

We try to call the get_books() in line three. However, we did not define it before using it in the program.

Correct Code:

```
# Defining a function
def get_books(books):
for b in books:
print(b)
books = ["Sherlock Holmes", "The Order", "The Alchemist"]
get_books(books)

#Output:
Sherlock Holmes
The Order
The Alchemist
```

(iii) **When we have not defined a variable**

It can be easy to forget to define a variable in large programs. However, Python cannot work with variables until they are declared and throws a NameError.

Let's look at a program that prints out a list of books:

STUDY NOTE

In Python, the exact line that has the error along with the type of error is shown in the exception message.

```
for b in books:
print(b)

#Output:
Traceback (most recent call last):
File "<string>", line 1, in <module>
NameError: name 'books' is not defined
```

we have not declared the variable "books" before using it.

Correct Code:

```
books = ["Sherlock Holmes", "The Order", "The Alchemist"]
for b in books:
print(b)
Output:
Sherlock Holmes
The Order
The Alchemist
```

(iv) **When a string is not defined properly**

Strings in Python should be enclosed in single or double quotation marks ('...' or "..."), otherwise, it is considered a part of the program.

```
print(Order)
#Output:
Traceback (most recent call last):
```

```
File "<string>", line 1, in <module>
NameError: name 'Order' is not defined

# "Order" is treated as a variable in the above code.
Correct Code:

print('Order')
#Output:
Order
```

(v) **When a variable is declared out of scope**

Variables have two scopes: local and global. Local variables are those that can only be accessed within the class or function they are defined in. On the other hand, global variables can be accessed throughout the program. If we try to access a local variable outside its class or function, the program throws a NameError.

```
def get_books():
books = ["Sherlock Holmes", "The Order", "The Alchemist"]
for b in books:
print(b)

get_books()
print(len(books))

#Output:
Sherlock Holmes
The Order
The Alchemist
Traceback (most recent call last):
  File "<string>", line 7, in <module>
NameError: name 'books' is not defined
```

Although the list of books is printed successfully, the last line throws an error. This is because we have declared "books" inside the get_books() function as a local variable.

We can correct this by declaring the variable in our main program:

```
books = ["Sherlock Holmes", "The Order", "The Alchemist"]
def get_books():
 for b in books:
    print(b)

get_books()
print(len(books))

#Output:

Sherlock Holmes
The Order
The Alchemist
3
```

3.2. Indentation Error

Indentation is a crucial part of programming in Python as the code is arranged using whitespaces. Python follows the PEP8 whitespace ethics where there should be 4 whitespaces used between any alternative or iteration. In Python, a block of code should start with indentation and the ending code after it should be non-indented. Some of the reasons an Indentation Error may arise are:

STUDY NOTE

Unlike other programming languages, Python does not require the program code to be enclosed in brackets.

- When you use both spaces and tabs in the written code.
- When code in compound statements like conditionals and loops are not indented properly.

Example:

```
# Wrong indentation
site = 'pro'
if site == 'pro':
print('Using Python')
else:
print('Please use proper indents')

#Output:

File "<string>", line 3
   print('Logging in to EduCBA!')
   ^
IndentationError: expected an indented block
```

Although the Syntax of the above code is correct, the if and else statement blocks are not indented properly.

Correct Code:

```
# Correct indentation
site = 'pro'
if site == 'pro':
   print('Using Python')
else:
   print('Please use proper indents')

#Output:

Using Python
```

3.3. Io Error

The **IOError** is associated with the input-output operations in Python. These exceptions arise if you try to open a file that does not exist (by using the open() function) or when there

is an error in the print statement. The I/O reasons for failure may be: “Disk full” or “File not found”.

The IOError exception follows the standard format of:

IOError: [Errno 1] No such file or directory: <file_name>

Here the error name is followed by the error number (that has occurred in a single program) and the reason for the error. IOError shows the common reason as "No such file or directory", which can mean that the file we have asked for does not exist or the file location is incorrect. The exception ends with the file name so we can identify and correct the file we have passed.

Let us look at an example of how an IOError is raised:

```
import sys
file = open('mefile.txt')
lines = file.readline()
slines = int(lines.strip())
#Output:
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'myfile.txt'
```

In the above code, we try to perform three operations on a file. We first try to access the file. Next, the readline() method should read the content of the file and at last strip() should remove the characters that are at the start and end of the sentences. As expected, the above code will throw an IOError as "mefile" does not exist.

3.4. EOF Error

The EOFError exception is raised in Python when the input() (or raw_input() in Python 2) return an end-of-file (EOF) without reading any input. This happens when we ask the user for input but it is not provided in the input box.

STUDY NOTE

We handle the IOError and EOFError exceptions by using the **try** and **except** keywords in Python.

Example:

```
n = num(input())
print(n * 15)

#Output:
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
EOFError: EOF when reading a line
```

Activity I

Correct the common errors in the following code:

```
nums = [(1,5), (4,3), (2,4), (9,3)]

print("The original list of tuples:")

print(nums)

print("\nSum of all the elements of each tuple stored inside the said list of tuples:")

print(test(numms))


for post in blog_posts:

total_likes = total_likes + post['Likes']
```

SELF-ASSESSMENT QUESTIONS - 2

6. _____ errors arise when the rules of code formatting are not followed in Python.
7. Logical errors are also known as _____.
8. Variables have _____ and _____ scope.
9. A NameError arises when a variable or function name is misspelt, or they are used before declaration. [True/False]
10. Python follows the PEP8 whitespace ethics where there should be 3 whitespaces used between any alternative or iteration. [True/False]
11. Which of the following errors will arise if a file cannot be found?

    a) EOFError b) IOError c) NameError d) ImportError
12. The following code will give which type of errors?

```
def get_flr():
    flr = ["Rose", "Lily", "Tulip"]
    for f in flr:
    print(f)

get_flr()
print(len(flr))
```

A. Name and Indentation
B. Name and IO
C. Only Name
D. Only Indentation

4. TRY-EXCEPT STATEMENT

You are now familiar with the most common type of errors that occur in Python programming. But how are these errors resolved in Python?

Many programming languages like Python have a construct to handle and deal with errors automatically, known as **Exception Handling**. Errors are resolved by saving the state of the code execution at the instance the error occurred and interrupting the normal flow of the program to execute a piece of code known as the exception handler.

In python, we can use the try-except statement to handle exceptions. The code that can potentially raise an exception is placed inside the try clause. On the other hand, the code that can handle these exceptions are placed in the except clause. By using the try-except statements, we can choose what operations should be performed once an exception is caught. Each try statement must have a following except statement.

STUDY NOTE

Most programming languages like C++, Objective-C, PHP, Java, Ruby, Python, etc. have built-in support for exception handling.

Example:

```
# import module sys to get the type of exception
import sys
randomList = ['a', 0, 2]
for entry in randomList:
    try:
        print("The entry is", entry)
        r = 1/int(entry)
        break
    except:
        print("Oops!", sys.exc_info()[0], "occurred.")
        print("Next entry.")
        print()
print("The reciprocal of", entry, "is", r)
#Output:
```

```
The entry is a
Oops! <class 'ValueError'> occurred.
Next entry.
The entry is 0
Oops! <class 'ZeroDivisionError'>occured.
Next entry.
The entry is 2
The reciprocal of 2 is 0.5
```

In the above program, we use the for loop to loop through the values in the list " randomList". As mentioned, the code that may raise an exception is placed under the *try* block.

The *except* block is skipped if no exception arises and the loop continues to the next value. However, if an exception occurs it is caught by the *except* block and the mentioned action is executed.

We use the **exc_info()** function to print the exception name. "a" causes a ValueError as it is a character and 0 causes a ZeroDivisionError as it is mathematically invalid.

4.1. Try-Except-Else Statement

Sometimes you may wish to execute specific blocks of code if no exceptions arise when the *try* clause runs. In such cases, you can use the *else* clause with the *try* statement. The statement follows the syntax:

```
try:
<code>
except:
<code>
else:
<code>
```

Example:

```
# printing reciprocal of even numbers
try:
    num = int(input("Enter a number: "))
    assert num % 2 == 0
except:
    print("Not an even number!")
else:
    reciprocal = 1/num
    print(reciprocal)
#Output:
# For an odd number
Enter a number: 1
Not an even number!
# For an even number
Enter a number: 6
0.16
# If 0 is passed
Enter a number: 0
Traceback (most recent call last):
  File "<string>", line 7, in <module>
    reciprocal = 1/num
ZeroDivisionError: division by zero
```

In the above code, the code block inside else is not handled by the preceding except clause. Hence, if 0 value is passed, the program throws a ZeroDivisionError.

STUDY NOTE

The preceding except clauses do not handle the exceptions that may arise in the else clause.

Activity II

The given code has a few bugs. Add a try-except clause so the code runs without any errors. If a blog post didn't get any likes, a 'Likes' key should be added to that dictionary with a value of 0.

```
blog_posts = [{'Photos': 3, 'Likes': 21, 'Comments': 2}, {'Likes': 13, 'Comments': 2, 'Shares': 1}, {'Photos': 5, 'Likes': 33, 'Comments': 8, 'Shares': 3}, {'Comments': 4, 'Shares': 2}, {'Photos': 8, 'Comments': 1, 'Shares': 1}, {'Photos': 3, 'Likes': 19, 'Comments': 3}]


total_likes = 0


for post in blog_posts:

total_likes = total_likes + post['Likes']
```

SELF-ASSESSMENT QUESTIONS - 3

13. What is the minimum number of except statements a try-except block should have?
    A. more than zero
    B. one
    C. more than one
    D. zero

14. The else clause of a try-except-else statement is executed when an exception occurs [True/False]

15. The following Syntax is of the try-except-else statement. [True/False]

```
try:
    # Do something
except:
    # Do something
else:
    # Do something
```

16. If an exception arises in the else clause, it's preceding except clause will handle it. [True/False]

17. Python has a construct to handle and deal with errors automatically, known as _____.

5. MULTIPLE EXCEPTIONS

A piece of code does not always throw only one exception. Multiple errors may arise while coding and all of these have to be predicted and handled without making the code complicated or lengthy. You can do this by using multiple except clauses with the try statement.

Each exception needs to be handled in specific ways and each except clause should catch only a particular exception. Although any number of except statements can be used in the code, only one of them is executed if an exception occurs.

Example:

```
try:
  <try block>
  pass
except ValueError:
  <handle ValueError exception>
  pass
except (TypeError, ZeroDivisionError):
  # handle multiple exceptions
  <TypeError and ZeroDivisionError>
  pass
except:
  <handle all other exceptions>
  pass
```

The above code is a pseudo-code where the first except block will handle any ValueError exceptions, while the second except block will handle both TypeError and ZeroDivisionError. The third block will handle all other exceptions.

5.1. Try And Finally Block

While programming, we may be connected to external resources such as remote data centre through a network, or we may be working with an external file or GUI (Graphical User Interface). In these cases, it is essential that we clean up these resources before the program halts (whether it ran successfully or not).

STUDY NOTE

The *finally* clause is executed no matter what (Whether an exception is raised or not) and is used to release external resources.

We use the *finally* statement with the try clause to close a file or disconnect from a network.

It follows the syntax:

```
try:
< try block >
except :
< except block >
:
:
:
finally :
< finally block >
Example:

try:
  f = open("test.txt",encoding = 'utf-8')
  # perform file operations
finally:
  f.close()
```

The above finally block closes the "test.txt" file at the end of the program.

Activity III

Write a multiple exception code that will evaluate TypeError, KeyError, and IndexError for a given list.

SELF-ASSESSMENT QUESTIONS - 4

18. The following syntax is that of the try-finally statement. [True/False]

```
try:
    <block>
finally:
    <block>
except:
    <block>
```

19. Multiple exceptions be handled by only one block of statement. [True/False]

20. _____ clause is always executed in the program.

21. The output of the given code is _____.

```
def a():
  try:
    print(1)
  finally:
    print(2)
a()
```

22. The output of the following code is:

```
def a():
    try:
        f(x, 4)
    finally:
        print('after f')
        print('after f?')
    a()
```

A. after f?
B. after f
C. no output
D. error

6. RAISING EXCEPTIONS

Generally in Python, exceptions are raised automatically when errors occur and are caught during run-time. However, we can manually raise exceptions by using the raise keyword. The raise keyword can be used in the following manner:

```
# raising the KeyboardInterrupt exception
>>> raise KeyboardInterrupt
Traceback (most recent call last):
...
KeyboardInterrupt

# raising the MemoryError exception
>>> raise MemoryError("This is an argument")
Traceback (most recent call last):
...
MemoryError: This is an argument

# raising the ValueError exception
try:
  a = int(input("Enter a positive integer: "))
  if a <= 0:
    raise ValueError("Not a positive number!")
  else:
    print(a)
except ValueError as ve:
  print(ve)

Enter a positive integer: -2
Not a positive number!
```

The third code will raise a ValueError with the sentence "Not a positive number!" is a number less than 0 is entered.

The raise keyword can also be used in conjugation with the for and the while loops with the syntax:

if (< conditional statement >) :

raise Exception (< custom statement >)

If the defined conditional statement returns a "True" value, the user-defined exception with the custom statement is thrown at the user. The statement can include any data type and variable.

Example:

```
def getMonth(m):
  if m<1 or m>12:
    raise ValueError("Invalid")
  print(m)
getMonth(20)
#Output:
Traceback (most recent call last):
 File "<string>", line 5, in <module>
 File "<string>", line 3, in getMonth
ValueError: Invalid
```

The above code throws the custom error as the value passed as an argument is greater than 12.

6.1. Custom Exceptions

Programmers can define their own custom exceptions by creating a new class.

```
>>> class CustomError(Exception):
... pass
...
>>> raise CustomError
Traceback (most recent call last):
...
__main__.CustomError
>>> raise CustomError("An error occurred")
Traceback (most recent call last):
...
__main__.CustomError: An error occurred
```

In the above example, we have created a new exception called CustomError that is derived from the Exception class. To raise this exception, we use the *raise* keyword with a custom message.

Any custom exception class can do what a built-in class can. Most programmers create a base class and derive exception classes from them.

STUDY NOTE

It is a good practice to save all the custom exceptions that your program will raise in a separate file that can be accessed.

Example:

```
# define Python custom exceptions
class Error(Exception):
    """Base class for other exceptions"""
    pass
class ValueTooSmallError(Error):
    """Raised when the input value is too small"""
    pass


class ValueTooLargeError(Error):
    """Raised when the input value is too large"""
    pass
# you need to guess this number
```

```
number = 10
# user guesses a number until he/she gets it right
while True:
   try:
      i_num = int(input("Enter a number: "))
      if i_num< number:
         raise ValueTooSmallError
      elifi_num> number:
         raise ValueTooLargeError
      break
   except ValueTooSmallError:
      print("This value is too small, try again!")
      print()
   except ValueTooLargeError:
      print("This value is too large, try again!")
      print()
print("Congratulations! You guessed it correctly.")
#Output:
Enter a number: 12
This value is too large, try again!
Enter a number: 0
This value is too small, try again!
Enter a number: 8
This value is too small, try again!
Enter a number: 10
Congratulations! You guessed it correctly.
```

The above program asks users to guess a stored number till they get it right. To help them, hints are provided to tell them if the number they have guessed is too small or too large.

The defined base class in the program is "Error". The exceptions "ValueTooSmallError" and "ValueTooLargeError" are derived from "Error"

Activity IV

Define a custom exception class which takes a string message as attribute. Hint: To define a custom exception, we need to define a class inherited from Exception.

SELF-ASSESSMENT QUESTIONS - 5

23. We use the _____ keyword to create custom exceptions.
24. User-defined exception classes are derived from a _____.
25. When using the raise keyword with a loop, the custom exception is raised if the return value is "True". [True/False]
26. The given code throws a TypeError. [True/False]

    2 + '5'
27. If the following code “t[23]” is given, the error will be:
    A. NameError
    B. ValueError
    C. TypeError
    D. IndexError

7. SUMMARY

- Error handling is the process of using practices to test and identify errors. When an error arises, the program throws an exception. The process of rectifying errors is called debugging.
- Potential errors should always be considered before code execution.
- A syntax error occurs when the rules of code formatting are not followed. Run-time errors raise during run-time if a processed value is invalid.
- Logical errors are also called bugs. They raise when the program runs without throwing an exception but gives unexpected or wrong results. These are the most difficult to detect.
- NameError arises when a variable or function name is misspelt, a function is called before it is declared; when we don't define a variable; when a string is not defined properly; when a variable is declared out of scope.
- Variables have two scopes: local and global. Local variables can only be used within the class they are defined in. Global variables can be used anywhere in the program.
- IndentationError arises when the code is not indented properly. Python follows the PEP8 whitespace ethics where there should be 4 whitespaces used between any alternative or iteration.
- IOError arises when there is an error in the input-output operations. If you try to open a file that does not exist or the file location is wrong, an IOError exception is thrown.
- EOFError is raised when the input() or raw_input() function reaches its end-of-file (EOF).
- We use the try-except statement to provide construct to handle exceptions. The potential error code is placed in the try clause and the code that will handle the error is placed in the except clause.
- We can use the else statement to execute specific blocks of code if no exceptions arise when the try clause runs. We can use multiple *except* clauses to execute and deal with specific errors.
- The *finally* clause is used to disconnect from external networks and close external files or GUIs.

- We can manually raise exceptions by using the *raise* keyword. The keyword can also be used in loops and is executed when the value is "True".
- Programmers can define their own custom exceptions by creating a new base class and deriving exception classes from it.

8. GLOSSARY

- **Class -** An object constructor, or a "blueprint" for creating objects.
- **Exception -** A message thrown by the program when an error occurs.
- **Debugging** - The process of identifying and correcting errors.
- **Bugs -** Logical errors that give unexpected results.
- **Local variable -** Variable that can only be used in the class it is defined in.
- **Global variable -** Variable that can be used anywhere in the program.
- **IndexError -** Error rise when an index of a sequence is out of range.
- **ValueError -** Error rise when a function gets an argument of the correct data type but improper value.
- **TypeError** - Error rise when an operation or function is applied to an incorrect data type.
- **ZeroDivisionError** - Error rise when the divisor in a division or modulo is zero.
- **MemoryError -** Error rise when an operation runs out of memory.

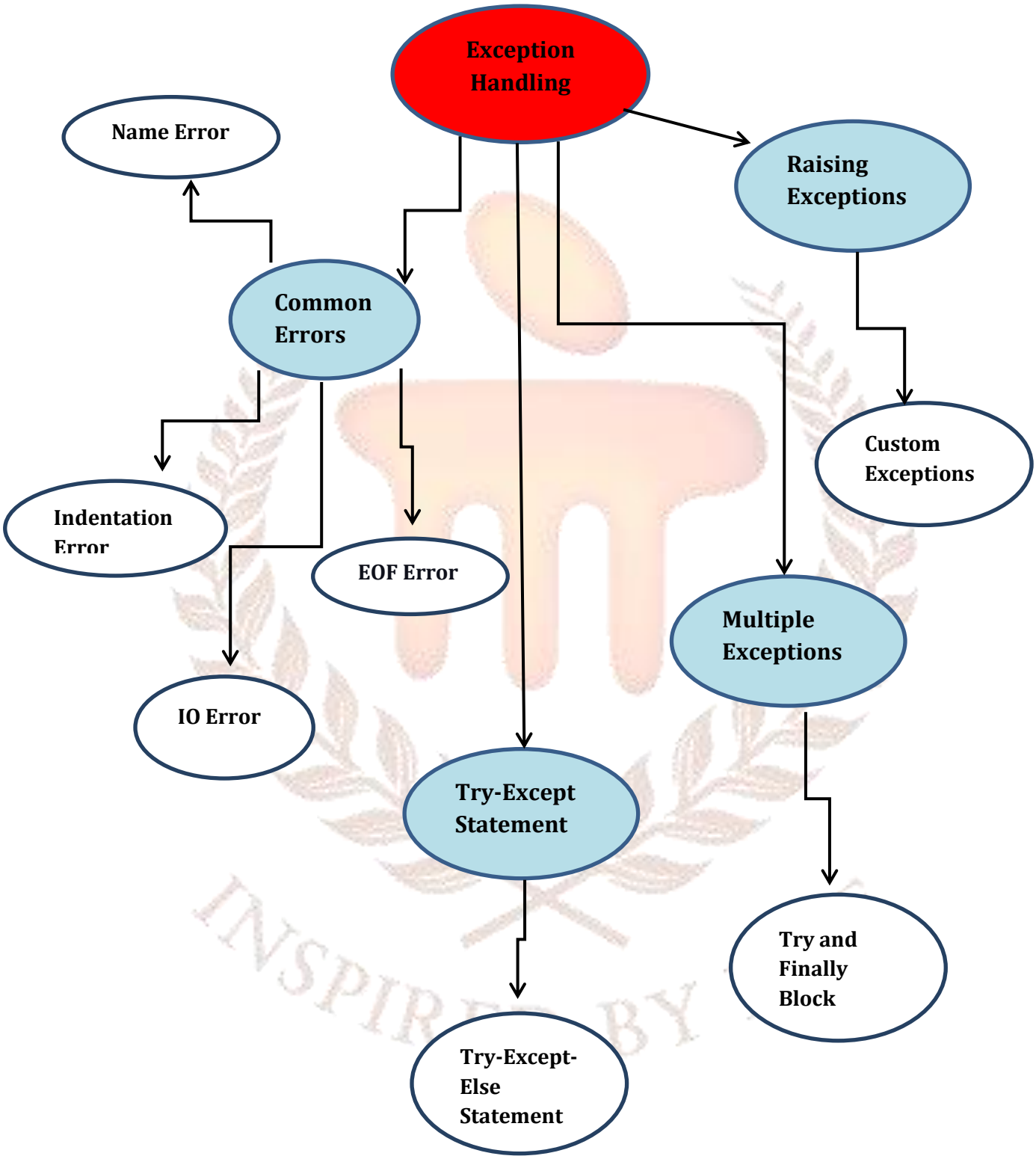

Fig 1: Conceptual Map

9. CASE STUDY

Error free library log recorder using Python

A library is upgrading its inventory system by making the complete process online. The store owner decides that the best language to make software will be Python for its ease of learning, light-weight structure, and various functions and methods integrated into the library. The library attendant wants to create an app that records the list of books that are available in the library without any duplication, to track the complete number of books received, rented or damaged to return and books that should be available for use, such that new books can be added, removed, and updated in the list in the future. There were many books which will be already available so the tracking is bit though due to duplication.

Along with that, the library attendant needs a method through which the sales of all items can be compared every month without any errors in numbers and registers. That is, there should be an error handling involves the techniques and practices used to test and identify errors. Identifying these errors and debugging them. Based on the results obtained from the comparison of these lists, the library attendant plans to decide whether to continue receiving the book in the library or not.

The program should also provide the complete activities of individual book that was used the most in a month and one that was used at the least. It should also let the library attendant to remove various items at a time as their availability and demand changes according to the best seller.

Imagine yourself as a part of the team that has been tasked to create such a program. Discuss and derive an answer for the following questions.

Questions:

1. What kind of exception handling are you going to use? To build such a program, discuss the advantages and disadvantages of the type you are using.
2. What approaches would you use in the software to ensure that it meets all of the library attendant's expectations?

10. TERMINAL QUESTIONS

SHORT ANSWER QUESTIONS

Q1. What is an exception?

Q2. Can we add other statements in between 'try', 'except', and 'finally' blocks?

Q3. Is it possible to use a 'try' block without the 'except' and 'finally' blocks?

Q4. List the type of errors?

Q5. What is Indention error?

LONG ANSWER QUESTIONS

Q1. Write a Python code to execute exception handling for the following code: (use the try-except statement)

```
a = 10

b = 0

print("Result of Division: " + str(a/b))
```

Q2. Write a Python code that will ask the users to "Enter numerator number" and "Enter denominator number", and use multiple except blocks to raise ZeroDivisionError and ValueError. Another except block should handle other errors.

Q3. Write a Python exception handling code that will accept a float integer and inverse its value. After program execution, it should print the message: "There may or may not have been an exception." (use the try-finally statement)

Q4. What is the standard format of IO error exception along with an example?

11. ANSWERS

SELF-ASSESSMENTS QUESTIONS

1. Debugging
2. Exception
3. True
4. True
5. B. handled errors. When errors are predicted and addressed, they are known as handled errors. Otherwise, they are referred to as unhandled errors.
6. Syntax
7. Bugs
8. Local, global
9. True
10. False
11. B. IOError. IOErrors are associated with input-output operations.
12. A. Name and Indentation.
13. 13) A. more than zero. There has to be at least one except statement.
14. False
15. True
16. False
17. Exception Handling
18. False
19. True
20. Finally
21. 1 2. No error occurs in the try block so 1 is printed. Then the finally block is executed and 2 is printed.
22. D. error. 'f' is not defined.
23. Raise
24. base class
25. True
26. True

27. A. NameError. 't' is not defined.

TERMINAL QUESTIONS

SHORT ANSWER QUESTIONS

Answer 1: Exceptions are abnormal conditions or errors that are sometimes encountered in a program. They disrupt the flow of the program and it is important to them. Not handling the exceptions can result in unwanted or incomplete outputs and even termination of the program.

Answer 2: Adding other statements like 'if', 'else', 'for', etc. is not recommended as the ‘try’, ‘catch’, and ‘finally’ blocks make up one exception handling unit. They follow the syntax:

```
try:
< try block >
except :
< except block >
:
:
:
finally :
< finally block >
```

Answer 3: Using a ‘try’ block without the ‘catch’ and ‘finally’ blocks will result in a compilation error. It is necessary that a 'try' block is followed by an 'except' block or a 'finally' block, if not both. The code that can potentially raise an exception is placed inside the try clause. On the other hand, the code that can handle these exceptions are placed in the except clause. If neither is present, the flow of exception handling is undisrupted.

Answer 4: Different types of errors are:

- Name Error
- Indentation Error
- IO Error
- EOF error

Answer 5: Indentation is a crucial part of programming in Python as the code is arranged using whitespaces. Python follows the PEP8 whitespace ethics where there should be 4 whitespaces used between any alternative or iteration. In Python, a block of code should start with indentation and the ending code after it should be non-indented. Some of the reasons an IndentationError may arise are:

- When you use both spaces and tabs in the written code.
- When code in compound statements like conditionals and loops are not indented properly.

LONG ANSWER QUESTIONS:

Answer 1: Answer Code:

```
# try block
try:
   a = 10
   b = 0
   print("Result of Division: " + str(a/b))
except:
print("You have divided a number by zero, which is not allowed.")
```

Answer 2: Answer Code:

```
# try block
try:
   a = int(input("Enter numerator number: "))
   b = int(input("Enter denominator number: "))
```

```
        print("Result of Division: " + str(a/b))
# except block handling division by zero
except(ZeroDivisionError):
print("You have divided a number by zero, which is not allowed.")
# except block handling wrong value type
except(ValueError):
print("You must enter integer value")
# generic except block
except:
  print("Oops! Something went wrong!")
```

Answer 3: Answer Code:

```
try:
  x = float(input("Your number: "))
  inverse = 1.0 / x
finally:
  print("There may or may not have been an exception.")
print("The inverse: ", inverse)
```

Answer 4: The IOError exception follows the standard format of:

IOError: [Errno 1] No such file or directory: <file_name>

Here the error name is followed by the error number (that has occurred in a single program) and the reason for the error. IOError shows the common reason as "No such file or directory", which can mean that the file we have asked for does not exist or the file location is incorrect. The exception ends with the file name so we can identify and correct the file we have passed.

Example:

```
import sys
file = open('mefile.txt')
lines = file.readline()
slines = int(lines.strip())
Output:
```

```
import sys
file = open('myfile.txt')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

12. SUGGESTED BOOKS AND E-REFERENCES

BOOKS:

- Sharma, P. (2017). Programming in Python. John Wiley & Sons. (Third ed.).
- Vashisht, R. (2013). Exception Handling in Python. Dover Publishing.
- Müller, A. C., Guido, Sarah (2016). Introduction to Machine Learning with Python: A Guide for Data Scientists. O'Reilly Media, Inc.

E-REFERENCES:

- Python Exception, viewed on 31 March 2021, < https://www.javatpoint.com/python-exception-handling >
- Error Handling, viewed on 31 March 2021, < https://www.studytonight.com/python/introduction-to-error-exception-python >
- Errors and Exceptions, viewed on 31 March 2021, < https://www.python-course.eu/python3_exception_handling.php >
- Exception Handling with Python viewed on 31 March 2021 < https://hub.packtpub.com/exception-handling-python/ >
- Beginner Guide to Exception and Exception Handling in Python viewed on 31 March 2021, < https://towardsdatascience.com/exception-handling-in-python-85f49801b131 >