MASTER OF COMPUTER APPLICATIONS

SEMESTER 1

PYTHON PROGRAMMING

Unit 7

Exception Handling

Table of Contents

1. INTRODUCTION

In Unit 6, we navigated the intricate world of File I/O in Python, unraveling the processes behind reading from and writing to files, understanding the various file modes for different applications, and embracing the with statement for efficient resource management. This journey equipped us with the necessary skills to handle text and binary files, ensuring that our Python programs can interact with external data sources smoothly and efficiently.

Now, as we step into Unit 7, we're delving into the realm of Exception Handling—a crucial skill set that every Python programmer must master. In any programming journey, encountering errors is inevitable. These errors, if unhandled, can disrupt the flow of your program, leading to unexpected crashes or, worse, making your code vulnerable to security threats. Exception Handling empowers us to anticipate potential pitfalls in our code and elegantly navigate around them, ensuring a seamless experience for the end-user and safeguarding our programs from unforeseen disruptions.

To effectively conquer this unit, it's essential to adopt a hands-on approach. Start by familiarizing yourself with common Python errors, their causes, and the best practices to avoid them. Then, gradually progress to implementing try-except blocks in your code, carefully observing how they can prevent your program from terminating abruptly due to errors. Challenge yourself with exercises that involve multiple exceptions and learn to raise your exceptions to handle specific error scenarios uniquely. By actively engaging with these concepts through coding exercises and real-world applications, you'll build a robust foundation in Exception Handling, making your Python programs more resilient and reliable.

1.1 Learning Objectives

After studying this unit, you will be able to:

- ❖ *Identify common types of errors in Python such as syntax errors, runtime errors, and logical errors.*
- ❖ *Describe the concept and importance of exception handling in Python programming.*
- ❖ *Explain the use of the try-except block to handle exceptions.*
- ❖ *Apply the try-except block in Python code to manage errors gracefully.*
- ❖ *Analyze the impact of unhandled exceptions on program flow and user experience.*
- ❖ *Differentiate between various error types and appropriate handling mechanisms.*

- *Evaluate scenarios where custom exception handling is required and implement it using Python.*

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.

INSPIRED BY LIFE

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".

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 NameError 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.

1. **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.

```
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
```

2. **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
```

3. 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:

```
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
```

4. 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
```

5. **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:

- 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('Using Python')
  ^
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.

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
```

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.

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.

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).

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.

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)
```

Output:

```
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.

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
    elif_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".

7. SUMMARY

Wrapping up our journey through Unit 7 on Exception Handling, it's been quite an enlightening path, hasn't it? We started off by understanding that, in the world of programming, not everything goes as planned. Errors are bound to happen, and that's perfectly okay. What's crucial, though, is how we deal with these errors to ensure our programs are robust and user-friendly.

We kicked things off by diving into the realm of common errors. Remember how we talked about syntax errors being like grammatical mistakes in programming? They're the kind of errors that Python points out to us, like missing colons or misspelled keywords. Then we moved on to runtime errors, which are a bit sneakier since they pop up when our program is running. These include trying to divide by zero or accessing a variable that hasn't been defined yet. And let's not forget about those logical errors, where everything runs smoothly, but the output just isn't what we expected. It's like telling your GPS to take you to the beach, and you end up at a library instead – not quite the plan!

But here's where the real magic happens – exception handling. We learned that Python, being the considerate language it is, offers us a way to anticipate and manage these errors without letting our program crash. Using try-except blocks, we can wrap potentially problematic code and decide exactly how to handle unexpected situations. It's like having a safety net that catches you when you're learning to trapeze, ensuring you can try again without fear.

We also explored the versatility of these blocks, remembering how we can specify different types of exceptions for more precise control. And for those times when we want our program to clean up or release resources, no matter what happens, the finally block is our trusty friend.

Then, we took a step further into the world of custom exceptions. By defining our own exception classes, we can tailor our error handling to fit the unique needs of our applications. It's like creating custom tools for a specialized job – they just work better.

Through all this, the key takeaway is that exception handling is not just about preventing crashes; it's about creating a seamless and intuitive experience for the users of our programs. It's about taking those inevitable errors and turning them into opportunities for grace and resilience. So, as we move forward in our programming adventures, let's remember to handle exceptions not just as a best practice, but as a fundamental part of crafting beautiful, robust software.

8. GLOSSARY

Exception Handling: The process of responding to exceptional conditions that occur during the execution of a program, such as errors or anomalies, in a controlled manner.

SyntaxError: An error that occurs when the parser detects a syntax violation in the code. Common causes include misspelled keywords, missing colons, or incorrect indentation.

Run-time Error: Errors that occur during the execution of the program, often due to operations that are mathematically incorrect or operations on incompatible types.

Logical Error: Errors that occur when a program runs without crashing but produces incorrect or unexpected results due to flaws in the logic.

NameError: This error is raised when the program tries to access a variable or a function that has not been defined or is out of scope.

IndentationError: An error caused by incorrect indentation, leading to blocks of code that are not correctly structured.

IOError: Refers to input/output errors, such as trying to open a file that doesn't exist or attempting to write to a file that is not writable.

EOFError: An error that occurs when the input() function hits an end-of-file condition (EOF) without reading any data.

try-except Block: A control structure used to handle exceptions in a program. The try block contains the code that might throw an exception, and the except block contains the code that executes if an exception occurs.

else Block: Used in conjunction with try-except blocks to specify a block of code to be executed if no exceptions were raised in the try block.

finally Block: A block of code that is always executed after the try and except blocks, regardless of whether an exception was raised or not, often used for cleanup tasks.

Multiple Exceptions: Handling more than one specific exception type using multiple except clauses, allowing for more granular control over exception handling.

raise Keyword: Used to manually raise an exception in Python. It allows the programmer to force a specified exception to occur.

Custom Exceptions: User-defined exception classes that inherit from Python's built-in Exception class, allowing for creating specific exception types for certain error conditions.

Assert Statement: A debugging aid that tests a condition as an internal self-check in the program. If the condition is False, an AssertionError is raised.

9. SELF-ASSESSMENT QUESTIONS

1. An error that occurs at runtime, disrupting the normal flow of a program, is known as an _________.
2. The _________ block in Python is used to catch and handle exceptions.
3. A _________ error occurs when the syntax of the Python code is incorrect.
4. The _________ block is always executed in a try-except statement, regardless of whether an exception occurred or not.
5. Python uses the _________ keyword to manually raise an exception.
6. A _________ error in Python is raised when a variable that has not been defined is referenced.
7. Custom exceptions in Python are created by defining a new class that inherits from the _________ class.
8. The process of identifying and removing errors from a program is known as _________.
9. To handle multiple exceptions with a single except clause, the exceptions must be placed in a _________.
10. Using the _________ statement, a developer can assert that a certain condition is true.
11. What type of error is caused by operations that are mathematically incorrect?
    A. SyntaxError
    B. NameError
    C. Run-time Error
    D. Logical Error
12. Which keyword is used in conjunction with try to handle exceptions?
    A. catch
    B. except
    C. finally
    D. error
13. What does the EOFError stand for?
    A. End Of File Error
    B. Execution Of Function Error
    C. External Output Failure Error

INSPIRED BY LIFE

D. None of the above

14. Which block is used to execute code when no exception is raised in the try block?

A. else

B. finally

C. except

D. error

15. Which of the following is NOT a built-in exception in Python?

A. ValueError

B. ZeroDivisionError

C. FileNotFoundError

D. OutOfBoundsError

16. Which function is used to get the type of an exception?

A. type()

B. isinstance()

C. exc_info()

D. get_error()

17. What is the output of raising a ValueError in Python?

A. A crash report

B. An error message specific to the ValueError

C. The program immediately exits

D. None of the above

18. How can multiple exceptions be handled in Python?

A. By nesting try-except blocks

B. By using multiple except blocks after a single try block

C. By separating exceptions with a semicolon in a single except block

D. By using a single except block without specifying exception types

19. What is the purpose of the assert statement in Python?

A. To raise a SyntaxError

B. To terminate the program

C. To ensure a condition returns True

D. To handle exceptions silently

20. Which statement about custom exceptions is true?

A. Custom exceptions cannot be caught by try-except blocks

B. Custom exceptions are primarily used for debugging

C. Custom exceptions must inherit from the built-in Exception class

D. Custom exceptions are discouraged in Python programming

10. TERMINAL QUESTIONS

1. Explain the difference between SyntaxError and NameError in Python with examples.
2. Describe how the try-except-else statement works with a real-life analogy.
3. Discuss the importance of exception handling in software development and how it impacts user experience.
4. How do logical errors differ from runtime errors, and what strategies can be used to identify them?
5. Explain the role of the finally block in exception handling and provide a scenario where it is particularly useful.
6. Write a Python program that handles a ZeroDivisionError exception.
7. Create a Python function that uses a try-except block to catch a ValueError.
8. Write a Python script that opens a file and uses exception handling to print an error message if the file doesn't exist (FileNotFoundError).
9. Develop a Python program that demonstrates the use of multiple except clauses.
10. Write a Python program that uses the assert statement to validate user input as a positive number.
11. Create a custom exception class in Python for handling an OutOfRangeError when a number is not within an expected range.
12. Implement a Python program that handles both an IOError and an EOFError gracefully.
13. Write a Python script that raises a KeyboardInterrupt exception and handles it with an informative message to the user.
14. Design a Python function that demonstrates the use of the finally block to close a database connection, regardless of whether an exception occurred.
15. Construct a Python program that uses a try-except block to handle a specific TypeError.
16. Write a Python program that prompts the user for an integer and uses exception handling to catch and print a message for non-integer inputs.
17. Develop a Python program that uses the in keyword to avoid a KeyError in a dictionary access.
18. Create a Python script that handles the indentation error and demonstrates the correct way to write an if-else block.

19. Implement a Python function that catches a custom TooManyRequestsError when a limit is exceeded and suggests waiting.
20. Write a Python script that simulates connecting to a server and raises a custom exception ServerNotRespondingError if the connection fails.

INSPIRED BY LIFE

11. ANSWERS

11.1. Self-Assessment Questions

1. Exception
2. except
3. SyntaxError
4. finally
5. raise
6. NameError
7. Exception
8. debugging
9. tuple
10. assert
11. C) Run-time Error
12. B) except
13. A) End Of File Error
14. A) else
15. D) OutOfBoundsError
16. C) exc_info()
17. B) An error message specific to the ValueError
18. B) By using multiple except blocks after a single try block and D) By using a single except block without specifying exception types
19. C) To ensure a condition returns True
20. C) Custom exceptions must inherit from the built-in Exception class

11.2. Terminal Questions

1. Refer to section 2. COMMON ERRORS for the explanation and examples of SyntaxError and NameError.
2. An analogy and explanation can be found in section 3. TRY-EXCEPT STATEMENT.
3. The importance and impact are discussed in the introductory section 1. EXCEPTION HANDLING INTRODUCTION.
4. Differences and identification strategies are covered in section 2. COMMON ERRORS.

5. The role and scenarios for the finally block are explained in the section titled TRY AND FINALLY BLOCK.
6. ZeroDivisionError exception handling: See section 3. TRY-EXCEPT STATEMENT for handling exceptions with a try-except block.
7. Catching a ValueError: The method is detailed in section 3. TRY-EXCEPT STATEMENT, showing how to use try-except for handling specific exceptions like ValueError.
8. FileNotFoundError handling: This type of exception handling is discussed in the IO ERROR subsection of the 2. COMMON ERRORS section.
9. Demonstrating multiple except clauses: Refer to section 4. MULTIPLE EXCEPTIONS for an example and explanation on how to use multiple except clauses in a single try-except block.
10. Assert statement usage: The concept of assertion is implicitly covered in the exception handling sections, particularly 3. TRY-EXCEPT STATEMENT. While not explicitly mentioned, the use of assert follows similar principles to exception handling.
11. Creating a custom exception class for OutOfRangeError: Look into the CUSTOM EXCEPTIONS part under 5. RAISING EXCEPTIONS for guidance on defining and raising custom exceptions.
12. Handling IOError and EOFError: These exceptions are specifically discussed in the IO ERROR and EOF ERROR subsections of the 2. COMMON ERRORS section.
13. Handling a KeyboardInterrupt exception: While not directly covered in the shared content, raising and handling exceptions such as KeyboardInterrupt can be inferred from the 5. RAISING EXCEPTIONS section.
14. Use of finally block for resource management: The TRY AND FINALLY BLOCK section provides insights into how the finally block is used for cleanup activities, like closing a database connection.
15. Handling a TypeError: General guidance on handling exceptions, including TypeError, can be found in 3. TRY-EXCEPT STATEMENT.
16. Catching non-integer inputs: This application of exception handling is related to catching ValueError, as discussed in 3. TRY-EXCEPT STATEMENT.

17. Avoiding a KeyError with in keyword: While not explicitly described in the content, using the in keyword to check for the presence of a key before accessing it in a dictionary follows basic Python syntax and is related to the principles discussed in the NAME ERROR section of 2. COMMON ERRORS.
18. Correcting an IndentationError: Guidance on proper indentation and avoiding IndentationError is provided in the INDENTATION ERROR subsection of 2. COMMON ERRORS.
19. Custom exception for TooManyRequestsError: The creation and handling of custom exceptions are discussed in the CUSTOM EXCEPTIONS part under 5. RAISING EXCEPTIONS.
20. Raising ServerNotRespondingError for connection failures: Similar to question 19, this involves defining and raising a custom exception, as explained in the CUSTOM EXCEPTIONS section of 5. RAISING EXCEPTIONS.

INSPIRED BY LIFE