Day 10 of Training at Ansh Info Tech

Topics Covered

- **Exception Handling in Python**
- **File Handling in Python**
- **Random Module in Python**
- **50 Questions on Exception Handling**
- **25 Questions on File Handling**
- **15+ Questions on Random Module**
- **Rock, Paper, Scissors Game Using Python**

```
# prompt: Exception Handling in Python in detail

# try:
#   # Code that may raise an exception
# except Exception as e:
#   # Code to handle the exception
#   print(f"An exception occurred: {e}")
# finally:
#   # Code that will always be executed, regardless of whether an exception is raised
#   print("The 'finally' block always runs")

# # Example 1: Handling a specific exception type
# try:
#   # Code that may raise a ValueError
# except ValueError as e:
#   # Code to handle the ValueError exception
#   print(f"A ValueError occurred: {e}")

# # Example 2: Handling multiple exception types
# try:
#   # Code that may raise a ValueError or IndexError
# except (ValueError, IndexError) as e:
#   # Code to handle either a ValueError or IndexError exception
#   print(f"Either a ValueError or IndexError occurred: {e}")

# # Example 3: Using the 'else' clause
# try:
#   # Code that may raise an exception
# except Exception as e:
#   # Code to handle the exception
#   print(f"An exception occurred: {e}")
# else:
#   # Code to execute if no exception is raised
#   print("No exception was raised")

# # Example 4: Using the 'finally' clause
# try:
#   # Code that may raise an exception
# except Exception as e:
#   # Code to handle the exception
#   print(f"An exception occurred: {e}")
# finally:
#   # Code that will always be executed, regardless of whether an exception is raised
#   print("The 'finally' block always runs")
```

```
  File "<ipython-input-5-4a8c37b3d4f0>", line 5
    except Exception as e:
    ^
IndentationError: expected an indented block after 'try' statement on line 3
```

```
# prompt: types of exception

# Built-in Exceptions
print("Built-in Exceptions:")
print("- ArithmeticError")
print("- AssertionError")
print("- AttributeError")
print("- EOFError")
print("- FloatingPointError")
print("- GeneratorExit")
print("- ImportError")
print("- IndexError")
print("- KeyError")
print("- KeyboardInterrupt")
print("- MemoryError")
print("- NameError")
print("- NotImplementedError")
print("- OSError")
print("- OverflowError")
print("- RuntimeError")
print("- StopIteration")
print("- SyntaxError")
print("- SystemError")
print("- SystemExit")
print("- TypeError")
print("- UnboundLocalError")
print("- ValueError")
print("- ZeroDivisionError")

# User-Defined Exceptions
print("\nUser-Defined Exceptions:")
class MyException(Exception):
  pass

try:
  raise MyException("This is a user-defined exception")
except MyException as e:
  print(f"MyException occurred: {e}")
```

Double-click (or enter) to edit

Introduction to Exception Handling

Exception handling in Python is a mechanism to respond to runtime errors, preventing the program from crashing and allowing the program to handle errors gracefully. It helps in debugging, maintaining clean code, and providing user-friendly error messages.

Key Concepts

1. Exception: An exception is an error that occurs during the execution of a program. When an exception is raised, the normal flow of the program is interrupted.

2. Try Block: The code that might raise an exception is placed inside a try block.

3. Except Block: The code that handles the exception is placed inside an except block.

4. Else Block: The code inside the else block is executed if no exceptions are raised.

5. Finally Block: The code inside the finally block is executed regardless of whether an exception is raised or not.

6. Raise: Used to raise an exception manually.

try:

```
# code that may raise an exception
```

except ExceptionType:

```
# code that runs if the exception occurs
```

else:

```
# code that runs if no exception occurs
```

finally:

```
# code that runs no matter what
```

Common Built-in Exceptions

1. IndexError
2. KeyError
3. ValueError
4. TypeError
5. ZeroDivisionError
6. FileNotFoundError
7. IOError
8. ImportError
9. AttributeError
10. RuntimeError

Example 1: Handling Division by Zero

```
def divide(a, b):
    try:
        result = a / b
    except ZeroDivisionError:
        return "Cannot divide by zero!"
    else:
        return result
    finally:
        print("Execution of divide function complete.")

print(divide(10, 2))  # Output: 5.0
print(divide(10, 0))  # Output: Cannot divide by zero!
```

```
Execution of divide function complete.
5.0
Execution of divide function complete.
Cannot divide by zero!
```

```
print(10/0)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-4-fe01563e1bc6> in <cell line: 1>()
----> 1 print(10/0)

ZeroDivisionError: division by zero
```

Example 2: Handling File Operations

```
def read_file(file_path):
    try:
        with open(file_path, 'r') as file:
            data = file.read()
    except FileNotFoundError:
        return "File not found!"
    except IOError:
        return "Error reading file!"
    else:
        return data
    finally:
        print("Execution of read_file function complete.")

print(read_file("existing_file.txt"))  # Output: (contents of the file)
print(read_file("nonexistent_file.txt"))  # Output: File not found!
```

```
Execution of read_file function complete.
File not found!
Execution of read_file function complete.
File not found!
```

Example 3: Handling Multiple Exceptions

```
def process_input(value):
    try:
        result = int(value)
    except ValueError:
        return "Invalid input! Please enter a number."
    except TypeError:
        return "Invalid type! Please enter a valid input."
    else:
        return f"Valid input: {result}"
    finally:
        print("Execution of process_input function complete.")

print(process_input("10"))  # Output: Valid input: 10
print(process_input("abc"))  # Output: Invalid input! Please enter a number.
print(process_input(None))   # Output: Invalid type! Please enter a valid input.
```

```
Execution of process_input function complete.
Valid input: 10
Execution of process_input function complete.
```

```
Invalid input! Please enter a number.
Execution of process_input function complete.
Invalid type! Please enter a valid input.
```

Example 4: Custom Exception

```
class NegativeValueError(Exception):
    def __init__(self, value):
        self.value = value
        self.message = f"Negative value error: {value}"
        super().__init__(self.message)

def check_positive(value):
    try:
        if value < 0:
            raise NegativeValueError(value)
        return "Value is positive."
    except NegativeValueError as e:
        return str(e)
    finally:
        print("Execution of check_positive function complete.")

print(check_positive(10))   # Output: Value is positive.
print(check_positive(-5))   # Output: Negative value error: -5
```

```
Execution of check_positive function complete.
Value is positive.
Execution of check_positive function complete.
Negative value error: -5
```

Best Practices for Exception Handling

Catch Specific Exceptions: Always catch specific exceptions instead of a generic Exception to handle errors more precisely.

Use Finally Block: Ensure that necessary cleanup (e.g., closing files or releasing resources) is performed by using the finally block.

Avoid Silent Failures: Do not use empty except blocks; always provide some logging or error message.

Log Exceptions: Use logging to record exceptions for future debugging and monitoring.

Use Custom Exceptions: Define custom exceptions for specific error conditions in your application to provide more meaningful error handling.

1. IndexError

Scenario: Accessing an invalid index in a list.

```
def get_list_element(lst, index):
    try:
        return lst[index]
    except IndexError as e:
        return f"IndexError: {e}"

my_list = [1, 2, 3]
print(get_list_element(my_list, 2))  # Output: 3
print(get_list_element(my_list, 5))  # Output: IndexError: list index out of range
```

```
3
IndexError: list index out of range
```

2. KeyError

Scenario: Accessing a non-existent key in a dictionary.

```
def get_dict_value(d, key):
    try:
        return d[key]
    except KeyError as e:
        return f"KeyError: {e}"

my_dict = {'a': 1, 'b': 2}
print(get_dict_value(my_dict, 'a'))  # Output: 1
print(get_dict_value(my_dict, 'c'))  # Output: KeyError: 'c'
```

```
1
KeyError: 'c'
```

3. ValueError Scenario: Converting an invalid string to an integer.

```
def convert_to_int(value):
    try:
        return int(value)
    except ValueError as e:
        return f"ValueError: {e}"

print(convert_to_int("123"))  # Output: 123
print(convert_to_int("abc"))  # Output: ValueError: invalid literal for int() with base 10: 'abc'
```

```
123
ValueError: invalid literal for int() with base 10: 'abc'
```

4. TypeError Scenario: Performing an invalid operation on incompatible types.

```
def add_numbers(a, b):
    try:
        return a + b
    except TypeError as e:
        return f"TypeError: {e}"

print(add_numbers(10, 5))
print(add_numbers(10, "five"))
```

```
15
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

5. ZeroDivisionError Scenario: Dividing a number by zero.

```
def divide(a, b):
    try:
        return a / b
    except ZeroDivisionError as e:
        return f"ZeroDivisionError: {e}"

print(divide(10, 2))
print(divide(10, 0))
```

6. FileNotFoundError Scenario: Trying to open a non-existent file.

```
def read_file(file_path):
    try:
        with open(file_path, 'r') as file:
            return file.read()
    except FileNotFoundError as e:
        return f"FileNotFoundError: {e}"

print(read_file("existing_file.txt"))
print(read_file("nonexistent_file.txt"))
```

7. IOError Scenario: Error occurs during input/output operation.

```
def write_file(file_path, content):
    try:
        with open(file_path, 'w') as file:
            file.write(content)
    except IOError as e:
        return f"IOError: {e}"

print(write_file("/path/to/readonly_file.txt", "Some content"))
```

```
IOError: [Errno 2] No such file or directory: '/path/to/readonly_file.txt'
```

8. ImportError Scenario: Importing a non-existent module.

```
try:
    import non_existent_module
except ImportError as e:
    print(f"ImportError: {e}")
```

```
ImportError: No module named 'non_existent_module'
```

9. AttributeError Scenario: Accessing an invalid attribute of an object.

```
class MyClass:
    def __init__(self, value):
        self.value = value

obj = MyClass(10)
try:
    print(obj.non_existent_attribute)
except AttributeError as e:
    print(f"AttributeError: {e}")
```

```
AttributeError: 'MyClass' object has no attribute 'non_existent_attribute'
```

10. RuntimeError Scenario: General runtime error not covered by other categories.

```
def raise_runtime_error():
    try:
        raise RuntimeError("This is a runtime error")
    except RuntimeError as e:
        return f"RuntimeError: {e}"

print(raise_runtime_error())
```

```
RuntimeError: This is a runtime error
```

PRACTICE QUESTIONS

1. **ATM Withdrawal**: Write a function that simulates an ATM withdrawal. The function should check if the account balance is sufficient for the withdrawal amount and raise an exception if not. Handle scenarios where the input withdrawal amount is not a number or is negative.
2. **User Login System**: Create a function that simulates a user login system. It should raise an exception if the username or password is incorrect and handle cases where the input values are empty strings.
3. **Online Shopping Cart**: Write a function that adds items to an online shopping cart. Handle scenarios where the item is out of stock, the item ID is invalid, or the quantity requested is more than the available stock.
4. **Temperature Conversion**: Implement a function that converts temperatures between Fahrenheit and Celsius. Raise an exception if the input is not a number and handle this error gracefully.

5. **Student Grades**: Write a function that calculates the average grade of a list of student grades. Handle scenarios where the list might be empty, the grades might not be valid numbers, or some grades might be missing.
6. **Email Sending Service**: Simulate an email sending service function that raises an exception if the email address is invalid, the server is unreachable, or the sending fails. Handle these exceptions and provide meaningful error messages.
7. **Online Reservation System**: Create a function that makes an online reservation for a hotel room. Handle cases where the room is already booked, the reservation date is in the past, or the input date format is incorrect.
8. **Bank Account Transfer**: Write a function that transfers money between two bank accounts. Raise an exception if the source account has insufficient funds or if the transfer amount is invalid. Handle these exceptions appropriately.
9. **File Parsing**: Write a function that parses a configuration file. Handle scenarios where the file is missing, the file format is incorrect, or required configuration keys are missing.
10. **Currency Converter**: Create a function that converts an amount from one currency to another using a predefined exchange rate. Handle scenarios where the input amount is not a number, the currency code is invalid, or the exchange rate is missing.
11. **Product Price Checker**: Write a function that checks the price of a product from an online store's API. Handle cases where the product ID is invalid, the API is unreachable, or the response data is malformed.
12. **Flight Booking System**: Implement a function that books a flight ticket. Handle scenarios where the flight is fully booked, the passenger details are incomplete, or the payment processing fails.
13. **Quiz Application**: Write a function that grades a multiple-choice quiz. Handle cases where the answer key is missing, the student's answers contain invalid choices, or the quiz data is corrupted.
14. **Library Management System**: Create a function that issues a book to a library member. Handle scenarios where the book is not available, the member's subscription is expired, or the member ID is invalid.
15. **Inventory Management**: Write a function that updates the inventory levels of products in a warehouse. Handle cases where the product ID is invalid, the update quantity is negative, or the database connection fails.

Certainly! Here are 25 more scenario-based questions to practice exception handling in Python:

16. **Weather Forecast Application**: Write a function that retrieves the weather forecast from an API. Handle scenarios where the city name is invalid, the API key is incorrect, or the API service is down.
17. **Social Media Post Scheduler**: Create a function that schedules a post on a social media platform. Handle cases where the post content is empty, the scheduled time is in the past, or the API request fails.
18. **Online Examination System**: Write a function that submits answers for an online exam. Handle scenarios where the answer format is incorrect, the exam has already ended, or the student ID is invalid.
19. **File Upload Service**: Implement a function that uploads a file to a server. Handle cases where the file size exceeds the limit, the file format is unsupported, or the server is unreachable.
20. **IoT Device Monitoring**: Create a function that monitors the status of IoT devices. Handle scenarios where the device ID is invalid, the device is offline, or the response data is malformed.
21. **Online Course Enrollment**: Write a function that enrolls a student in an online course. Handle cases where the course is full, the enrollment period has ended, or the student information is incomplete.
22. **Library Book Return**: Implement a function that processes the return of a borrowed library book. Handle scenarios where the book ID is invalid, the return date is past due, or the library system is offline.
23. **E-commerce Checkout**: Create a function that processes an e-commerce checkout. Handle cases where the payment method is invalid, the shipping address is incomplete, or the cart is empty.
24. **Healthcare Appointment Booking**: Write a function that books an appointment with a healthcare provider. Handle scenarios where the provider is fully booked, the appointment date is in the past, or the patient information is missing.
25. **Remote Server Command Execution**: Implement a function that executes commands on a remote server. Handle cases where the server is unreachable, the command execution fails, or the authentication credentials are incorrect.
26. **Data Import Tool**: Create a function that imports data from a CSV file. Handle scenarios where the file format is incorrect, the file contains missing values, or the data types are inconsistent.
27. **Bank Loan Application**: Write a function that processes a bank loan application. Handle cases where the applicant's credit score is too low, the loan amount is too high, or required documents are missing.
28. **Real-time Stock Price Fetcher**: Implement a function that fetches real-time stock prices from an API. Handle scenarios where the stock symbol is invalid, the API request times out, or the response data is incomplete.

29. **Email Subscription Service**: Create a function that subscribes a user to an email list. Handle cases where the email address is invalid, the subscription service is down, or the user is already subscribed.
30. **User Profile Update**: Write a function that updates a user's profile information. Handle scenarios where the new data is invalid, the user ID does not exist, or the database update fails.
31. **Event Ticket Booking**: Implement a function that books tickets for an event. Handle cases where the event is sold out, the ticket quantity is invalid, or the payment fails.
32. **Product Review Submission**: Create a function that submits a product review. Handle scenarios where the review content is empty, the rating is out of range, or the product ID is invalid.
33. **Chat Application Message Sending**: Write a function that sends a message in a chat application. Handle cases where the recipient ID is invalid, the message content is empty, or the server is unreachable.
34. **User Registration System**: Implement a function that registers a new user. Handle scenarios where the username is already taken, the password is too weak, or required fields are missing.
35. **Data Backup Service**: Create a function that backs up data to a remote server. Handle cases where the backup file is too large, the server is unreachable, or the authentication fails.
36. **Online Quiz Timer**: Write a function that manages the timer for an online quiz. Handle scenarios where the time format is incorrect, the timer starts late, or the timer malfunctions.
37. **Multi-language Translation Service**: Implement a function that translates text into multiple languages using an API. Handle cases where the text is too long, the language code is invalid, or the API request fails.
38. **Restaurant Reservation System**: Create a function that makes a reservation at a restaurant. Handle scenarios where the restaurant is fully booked, the reservation time is invalid, or the customer information is incomplete.
39. **Automatic Billing System**: Write a function that processes automatic billing for subscriptions. Handle cases where the payment method is declined, the billing cycle is incorrect, or the user account is inactive.
40. **Cloud Storage File Deletion**: Implement a function that deletes a file from cloud storage. Handle scenarios where the file ID is invalid, the file does not exist, or the server is unreachable.
41. **Fitness Tracker Data Sync**: Create a function that syncs data from a fitness tracker to an app. Handle cases where the device is disconnected, the data format is invalid, or the sync fails.
42. **Online Store Inventory Update**: Write a function that updates the inventory of an online store. Handle scenarios where the product ID is invalid, the update quantity is negative, or the database connection fails.
43. **Payment Gateway Integration**: Implement a function that integrates with a payment gateway to process transactions. Handle cases where the transaction is declined, the payment details are invalid, or the gateway is unreachable.
44. **Content Management System (CMS) Post Creation**: Create a function that creates a new post in a CMS. Handle scenarios where the post content is empty, the post title is missing, or the server is down.
45. **IoT Device Configuration**: Write a function that configures settings on an IoT device. Handle cases where the device ID is invalid, the configuration data is incorrect, or the device is unreachable.
46. **Customer Feedback Form**: Implement a function that processes customer feedback from a form. Handle scenarios where the feedback content is empty, the rating is out of range, or the form submission fails.
47. **Inventory Reorder System**: Create a function that triggers reorders for low inventory items. Handle cases where the reorder quantity is invalid, the supplier ID is incorrect, or the database update fails.
48. **Insurance Claim Submission**: Write a function that submits an insurance claim. Handle scenarios where the claim details are incomplete, the claim amount is invalid, or the claim submission fails.
49. **Online Learning Platform Assignment Submission**: Implement a function that allows students to submit assignments on an online learning platform. Handle cases where the assignment file is too large, the submission deadline has passed, or the file format is unsupported.
50. **Smart Home Device Control**: Create a function that controls smart home devices (e.g., lights, thermostat). Handle scenarios where the device ID is invalid, the command is unsupported, or the device is offline.

Double-click (or enter) to edit

1. ATM Withdrawal: Write a function that simulates an ATM withdrawal. The function should check if the account balance is sufficient for the withdrawal amount and raise an exception if not. Handle scenarios where the input withdrawal amount is not a number or is negative.

```
def checkBal(withdraw_amount):
  balance = 1000
  try:
    if(withdraw_amount<0):
      raise ValueError
    if(withdraw_amount<=balance):
      balance -= withdraw_amount
      print("Ho gya")
    else:
      raise Exception
  except ValueError:
    print("Value Error")
  except TypeError:
    print("Type Error")
  except:
    print("Insufficient Balance")
  finally:
    print("Finally")
    print(balance)

checkBal(500)
checkBal(-50)
checkBal(1500)
checkBal("Abcd")
```

```
Ho gya
Finally
500
Value Error
Finally
1000
Insufficient Balance
Finally
1000
Type Error
Finally
1000
```

2. User Login System: Create a function that simulates a user login system. It should raise an exception if the username or password is incorrect and handle cases where the input values are empty strings.

```
data = {"user1":"pass1","user2":"pass2"}

def login(username,password):
  if len(username) == 0 or len(password) == 0:
    raise Exception("Empty String")
  if username in data and password == data[username]:
    print("Login Successful")
  else:
    raise Exception("Invalid Credentials")

try:
  login("user1","pass1")
except Exception as e:
  print(e)
```

```
Login Successful
```

3. Online Shopping Cart: Write a function that adds items to an online shopping cart. Handle scenarios where the item is out of stock, the item ID is invalid, or the quantity requested is more than the available stock.

```
def addItem(item_id,quantity):
  inventory = {1:10,2:20,3:30}
  try:
    if quantity<0:
      raise Exception("Quantity Cannot be Negative")
    elif item_id not in inventory:
      raise Exception("Item Not Found")
    elif item_id in inventory and inventory[item_id]>=quantity:
      inventory[item_id] -= quantity
      print("Item Added")
    elif item_id in inventory:
      raise Exception("Item Required Quantity Not Available")
  except Exception as e:
    print(e)

addItem(1, -5)
addItem(4, 10)
addItem(2, 10)
addItem(1, 15)
```

```
Quantity Cannot be Negative
Item Not Found
Item Added
Item Required Quantity Not Available
```

4. Temperature Conversion: Implement a function that converts temperatures between Fahrenheit and Celsius. Raise an exception if the input is not a number and handle this error gracefully.

```
def convertCtoF(temp):
  try:
    F = (temp * 9/5) + 32
    print(F)
  except Exception as e:
    print(e)

def convertFtoC(temp):
  try:
    C = (temp - 32) * 5/9
    print(C)
  except Exception as e:
    print(e)

convertCtoF(100)
convertCtoF("abc")
convertFtoC(212)
```

```
212.0
unsupported operand type(s) for /: 'str' and 'int'
100.0
```

5. Student Grades: Write a function that calculates the average grade of a list of student grades. Handle scenarios where the list might be empty, the grades might not be valid numbers, or some grades might be missing.

```
def calcAverage(grades):
  try:
    if(len(grades) == 0):
      raise Exception("Empty List")
    else:
      sum = 0
      for grade in grades:
        sum += grade
      average = sum/len(grades)
      print(average)
  except Exception as e:
    print(e)

calcAverage([])
calcAverage([10,20,30])
calcAverage([10,20,"abc"])
```

```
Empty List
20.0
```

```
unsupported operand type(s) for +=: 'int' and 'str'
```

```
# Question 6
# Email Sending Service: Simulate an email sending service function
# that raises an exception if the email address is invalid, the server
# is unreachable, or the sending fails. Handle these exceptions and
# provide meaningful error messages.

import re

class EmailError(Exception): pass
class InvalidEmailError(EmailError): pass
class ServerUnreachableError(EmailError): pass
class SendingFailedError(EmailError): pass

def send_email(email, message):
    try:
        if not re.match(r"[^@]+@[^@]+\.[^@]+", email):
            raise InvalidEmailError("Invalid email address.")
        if not simulate_server_reachable():
            raise ServerUnreachableError("Email server is unreachable.")
        if not simulate_send_success():
            raise SendingFailedError("Failed to send the email.")
        print("Email sent successfully.")
    except InvalidEmailError as e:
        print(f"Error: {e}")
    except ServerUnreachableError as e:
        print(f"Error: {e}")
    except SendingFailedError as e:
        print(f"Error: {e}")

def simulate_server_reachable():
    return True  # Simulated reachable server

def simulate_send_success():
    return True  # Simulated successful send

# Test
send_email("test@example.com", "Hello!")

# Question 7
# Online Reservation System: Create a function that makes an online
# reservation for a hotel room. Handle cases where the room is already
# booked, the reservation date is in the past, or the input date format
# is incorrect.

from datetime import datetime

class ReservationError(Exception): pass
class RoomBookedError(ReservationError): pass
class InvalidDateError(ReservationError): pass
class PastDateError(ReservationError): pass

def make_reservation(room, date):
    try:
        res_date = datetime.strptime(date, "%Y-%m-%d")
        if res_date < datetime.now():
            raise PastDateError("Reservation date is in the past.")
        if simulate_room_booked(room, res_date):
            raise RoomBookedError("Room is already booked.")
        print("Reservation successful.")
    except ValueError:
        print("Error: Invalid date format.")
    except PastDateError as e:
        print(f"Error: {e}")
    except RoomBookedError as e:
        print(f"Error: {e}")

def simulate_room_booked(room, date):
    return False  # Simulated room availability

# Test
make_reservation(101, "2024-07-01")

# Question 8
# Bank Account Transfer: Write a function that transfers money
# between two bank accounts. Raise an exception if the source account
# has insufficient funds or if the transfer amount is invalid.
# Handle these exceptions appropriately.
```

```
class TransferError(Exception): pass
class InsufficientFundsError(TransferError): pass
class InvalidAmountError(TransferError): pass

class BankAccount:
    def __init__(self, balance):
        self.balance = balance

def transfer_money(src, dst, amount):
    try:
        if amount <= 0:
            raise InvalidAmountError("Amount must be positive.")
        if src.balance < amount:
            raise InsufficientFundsError("Insufficient funds.")
        src.balance -= amount
        dst.balance += amount
        print("Transfer successful.")
    except InvalidAmountError as e:
        print(f"Error: {e}")
    except InsufficientFundsError as e:
        print(f"Error: {e}")

# Test
src = BankAccount(100)
dst = BankAccount(50)
transfer_money(src, dst, 30)

# Question 9
# File Parsing: Write a function that parses a configuration file.
# Handle scenarios where the file is missing, the file format is
# incorrect, or required configuration keys are missing.

import json

class ConfigError(Exception): pass
class FileMissingError(ConfigError): pass
class IncorrectFormatError(ConfigError): pass
class MissingKeyError(ConfigError): pass

def parse_config(file_path):
    try:
        with open(file_path, 'r') as file:
            config = json.load(file)
        required_keys = ["host", "port"]
        for key in required_keys:
            if key not in config:
                raise MissingKeyError(f"Missing required key: {key}")
        print("Configuration parsed successfully.")
        return config
    except FileNotFoundError:
        print("Error: Configuration file is missing.")
    except json.JSONDecodeError:
        print("Error: Configuration file format is incorrect.")
    except MissingKeyError as e:
        print(f"Error: {e}")

# Test
parse_config("config.json")

# Question 10
# Currency Converter: Create a function that converts an amount from
# one currency to another using a predefined exchange rate. Handle
# scenarios where the input amount is not a number, the currency code
# is invalid, or the exchange rate is missing.

class ConversionError(Exception): pass
class InvalidAmountError(ConversionError): pass
class InvalidCurrencyCodeError(ConversionError): pass

exchange_rates = {
    "USD": 1.0,
    "EUR": 0.85,
    "JPY": 110.0
}

def convert_currency(amount, from_currency, to_currency):
```

```
    try:
        if not isinstance(amount, (int, float)):
            raise InvalidAmountError("Amount must be a number.")
        if from_currency not in exchange_rates or to_currency not in exchange_rates:
            raise InvalidCurrencyCodeError("Invalid currency code.")
        converted_amount = (amount / exchange_rates[from_currency]) * exchange_rates[to_currency]
        print(f"Converted amount: {converted_amount}")
        return converted_amount
    except InvalidAmountError as e:
        print(f"Error: {e}")
    except InvalidCurrencyCodeError as e:
        print(f"Error: {e}")

# Test
convert_currency(100, "USD", "EUR")
```

```python
# Question 11
# Product Price Checker: Write a function that checks the price of a
# product from an online store's API. Handle cases where the product ID
# is invalid, the API is unreachable, or the response data is malformed.

import requests

class PriceCheckError(Exception): pass
class InvalidProductIDError(PriceCheckError): pass
class APIUnreachableError(PriceCheckError): pass
class MalformedResponseError(PriceCheckError): pass

def check_product_price(product_id):
    try:
        response = requests.get(f"http://api.example.com/products/{product_id}")
        response.raise_for_status()
        data = response.json()
        if 'price' not in data:
            raise MalformedResponseError("Response data is malformed.")
        print(f"Product price: {data['price']}")
    except requests.exceptions.RequestException:
        print("Error: API is unreachable.")
    except MalformedResponseError as e:
        print(f"Error: {e}")

# Test
check_product_price(12345)

# Question 12
# Flight Booking System: Implement a function that books a flight
# ticket. Handle scenarios where the flight is fully booked, the
# passenger details are incomplete, or the payment processing fails.

class BookingError(Exception): pass
class FlightFullyBookedError(BookingError): pass
class IncompletePassengerDetailsError(BookingError): pass
class PaymentProcessingError(BookingError): pass

def book_flight(flight_id, passenger_details, payment_info):
    try:
        if not all(key in passenger_details for key in ["name", "passport"]):
            raise IncompletePassengerDetailsError("Passenger details are incomplete.")
        if simulate_flight_full(flight_id):
            raise FlightFullyBookedError("Flight is fully booked.")
        if not simulate_payment_process(payment_info):
            raise PaymentProcessingError("Payment processing failed.")
        print("Flight booked successfully.")
    except IncompletePassengerDetailsError as e:
        print(f"Error: {e}")
    except FlightFullyBookedError as e:
        print(f"Error: {e}")
    except PaymentProcessingError as e:
        print(f"Error: {e}")

def simulate_flight_full(flight_id):
    return False  # Simulated flight availability

def simulate_payment_process(payment_info):
    return True  # Simulated payment processing

# Test
book_flight(123, {"name": "John Doe", "passport": "A1234567"}, {"card_number": "4111111111111111"})

# Question 13
# Quiz Application: Write a function that grades a multiple-choice
# quiz. Handle cases where the answer key is missing, the student's
# answers contain invalid choices, or the quiz data is corrupted.

class QuizError(Exception): pass
class MissingAnswerKeyError(QuizError): pass
class InvalidAnswerChoiceError(QuizError): pass
class CorruptedQuizDataError(QuizError): pass

def grade_quiz(quiz_data, student_answers):
    try:
        if 'answer_key' not in quiz_data:
            raise MissingAnswerKeyError("Answer key is missing.")
```

```
        for question, answer in student_answers.items():
            if answer not in quiz_data['answer_key'][question]:
                raise InvalidAnswerChoiceError(f"Invalid answer choice: {answer}")
        score = sum(quiz_data['answer_key'][q] == a for q, a in student_answers.items())
        print(f"Quiz score: {score}")
    except MissingAnswerKeyError as e:
        print(f"Error: {e}")
    except InvalidAnswerChoiceError as e:
        print(f"Error: {e}")
    except KeyError:
        print("Error: Quiz data is corrupted.")

# Test
quiz_data = {
    "answer_key": {
        "q1": "a",
        "q2": "b",
        "q3": "c"
    }
}
student_answers = {
    "q1": "a",
    "q2": "b",
    "q3": "d"
}
grade_quiz(quiz_data, student_answers)

# Question 14
# Library Management System: Create a function that issues a book to a
# library member. Handle scenarios where the book is not available, the
# member's subscription is expired, or the member ID is invalid.

class LibraryError(Exception): pass
class BookNotAvailableError(LibraryError): pass
class SubscriptionExpiredError(LibraryError): pass
class InvalidMemberIDError(LibraryError): pass

def issue_book(book_id, member_id):
    try:
        if not simulate_member_exists(member_id):
            raise InvalidMemberIDError("Member ID is invalid.")
        if simulate_subscription_expired(member_id):
            raise SubscriptionExpiredError("Member's subscription is expired.")
        if not simulate_book_available(book_id):
            raise BookNotAvailableError("Book is not available.")
        print("Book issued successfully.")
    except InvalidMemberIDError as e:
        print(f"Error: {e}")
    except SubscriptionExpiredError as e:
        print(f"Error: {e}")
    except BookNotAvailableError as e:
        print(f"Error: {e}")

def simulate_member_exists(member_id):
    return True  # Simulated member check

def simulate_subscription_expired(member_id):
    return False  # Simulated subscription check

def simulate_book_available(book_id):
    return True  # Simulated book availability

# Test
issue_book(123, "M001")

# Question 15
# Inventory Management: Write a function that updates the inventory
# levels of products in a warehouse. Handle cases where the product ID
# is invalid, the update quantity is negative, or the database
# connection fails.

class InventoryError(Exception): pass
class InvalidProductIDError(InventoryError): pass
class NegativeQuantityError(InventoryError): pass
class DatabaseConnectionError(InventoryError): pass

def update_inventory(product_id, quantity):
```

```
    try:
        if not simulate_product_exists(product_id):
            raise InvalidProductIDError("Product ID is invalid.")
        if quantity < 0:
            raise NegativeQuantityError("Quantity cannot be negative.")
        if not simulate_database_update(product_id, quantity):
            raise DatabaseConnectionError("Failed to connect to the database.")
        print("Inventory updated successfully.")
    except InvalidProductIDError as e:
        print(f"Error: {e}")
    except NegativeQuantityError as e:
        print(f"Error: {e}")
    except DatabaseConnectionError as e:
        print(f"Error: {e}")

def simulate_product_exists(product_id):
    return True  # Simulated product check

def simulate_database_update(product_id, quantity):
    return True  # Simulated database update

# Test
update_inventory(123, 50)

# Question 16
# Weather Forecast Application: Write a function that retrieves the
# weather forecast from an API. Handle scenarios where the city name is
# invalid, the API key is incorrect, or the API service is down.

import requests

class WeatherError(Exception): pass
class InvalidCityError(WeatherError): pass
class InvalidAPIKeyError(WeatherError): pass
class APIDownError(WeatherError): pass

def get_weather_forecast(city_name):
    try:
        response = requests.get(f"http://api.weather.com/forecast/{city_name}")
        response.raise_for_status()
        data = response.json()
        if 'error' in data:
            if data['error'] == 'invalid_api_key':
                raise InvalidAPIKeyError("API key is incorrect.")
            if data['error'] == 'city_not_found':
                raise InvalidCityError("City name is invalid.")
        print(f"Weather forecast for {city_name}: {data['forecast']}")
    except requests.exceptions.RequestException:
        print("Error: API service is down.")
    except InvalidAPIKeyError as e:
        print(f"Error: {e}")
    except InvalidCityError as e:
        print(f"Error: {e}")

# Test
get_weather_forecast("New York")

# Question 17
# Social Media Post Scheduler: Create a function that schedules a post
# on a social media platform. Handle cases where the post content is
# empty, the scheduled time is in the past, or the API request fails.

import datetime

class PostSchedulerError(Exception): pass
class EmptyContentError(PostSchedulerError): pass
class PastScheduledTimeError(PostSchedulerError): pass
class APIRequestFailedError(PostSchedulerError): pass

def schedule_post(content, scheduled_time):
    try:
        if not content:
            raise EmptyContentError("Post content is empty.")
        if scheduled_time < datetime.datetime.now():
            raise PastScheduledTimeError("Scheduled time is in the past.")
        if not simulate_api_request(content, scheduled_time):
            raise APIRequestFailedError("API request failed.")
```

```
        print("Post scheduled successfully.")
    except EmptyContentError as e:
        print(f"Error: {e}")
    except PastScheduledTimeError as e:
        print(f"Error: {e}")
    except APIRequestFailedError as e:
        print(f"Error: {e}")

def simulate_api_request(content, scheduled_time):
    return True  # Simulated API request

# Test
schedule_post("Hello, world!", datetime.datetime(2024, 7, 1, 12, 0))

# Question 18
# Online Examination System: Write a function that submits answers for
# an online exam. Handle scenarios where the answer format is incorrect,
# the exam has already ended, or the student ID is invalid.

class ExamError(Exception): pass
class InvalidAnswerFormatError(ExamError): pass
class ExamEndedError(ExamError): pass
class InvalidStudentIDError(ExamError): pass

def submit_exam_answers(student_id, answers):
    try:
        if not isinstance(answers, dict):
            raise InvalidAnswerFormatError("Answer format is incorrect.")
        if not simulate_student_exists(student_id):
            raise InvalidStudentIDError("Student ID is invalid.")
        if simulate_exam_ended():
            raise ExamEndedError("Exam has already ended.")
        print("Answers submitted successfully.")
    except InvalidAnswerFormatError as e:
        print(f"Error: {e}")
    except InvalidStudentIDError as e:
        print(f"Error: {e}")
    except ExamEndedError as e:
        print(f"Error: {e}")

def simulate_student_exists(student_id):
    return True  # Simulated student ID check

def simulate_exam_ended():
    return False  # Simulated exam status check

# Test
submit_exam_answers("S001", {"q1": "a", "q2": "b"})

# Question 19
# File Upload Service: Implement a function that uploads a file to a
# server. Handle cases where the file size exceeds the limit, the file
# format is unsupported, or the server is unreachable.

class FileUploadError(Exception): pass
class FileSizeExceededError(FileUploadError): pass
class UnsupportedFileFormatError(FileUploadError): pass
class ServerUnreachableError(FileUploadError): pass

def upload_file(file_path):
    try:
        if not simulate_file_exists(file_path):
            raise FileNotFoundError("File not found.")
        if simulate_file_size(file_path) > 10 * 1024 * 1024:
            raise FileSizeExceededError("File size exceeds the limit.")
        if not simulate_file_format_supported(file_path):
            raise UnsupportedFileFormatError("File format is unsupported.")
        if not simulate_server_reachable():
            raise ServerUnreachableError("Server is unreachable.")
        print("File uploaded successfully.")
    except FileNotFoundError as e:
        print(f"Error: {e}")
    except FileSizeExceededError as e:
        print(f"Error: {e}")
    except UnsupportedFileFormatError as e:
        print(f"Error: {e}")
    except ServerUnreachableError as e:
```

```
        print(f"Error: {e}")

def simulate_file_exists(file_path):
    return True  # Simulated file existence check

def simulate_file_size(file_path):
    return 5 * 1024 * 1024  # Simulated file size

def simulate_file_format_supported(file_path):
    return True  # Simulated file format support

def simulate_server_reachable():
    return True  # Simulated server reachability

# Test
upload_file("example.txt")

# Question 20
# IoT Device Monitoring: Create a function that monitors the status of
# IoT devices. Handle scenarios where the device ID is invalid, the
# device is offline, or the response data is malformed.

class DeviceMonitoringError(Exception): pass
class InvalidDeviceIDError(DeviceMonitoringError): pass
class DeviceOfflineError(DeviceMonitoringError): pass
class MalformedResponseError(DeviceMonitoringError): pass

def monitor_device(device_id):
    try:
        response = requests.get(f"http://api.iot.com/devices/{device_id}/status")
        response.raise_for_status()
        data = response.json()
        if 'status' not in data:
            raise MalformedResponseError("Response data is malformed.")
        if data['status'] == 'offline':
            raise DeviceOfflineError("Device is offline.")
        print(f"Device status: {data['status']}")
    except requests.exceptions.RequestException:
        print("Error: API service is unreachable.")
    except InvalidDeviceIDError as e:
        print(f"Error: {e}")
    except DeviceOfflineError as e:
        print(f"Error: {e}")
    except MalformedResponseError as e:
        print(f"Error: {e}")

# Test
monitor_device("D001")

# Question 21
# Online Course Enrollment: Write a function that enrolls a student in
# an online course. Handle cases where the course is full, the
# enrollment period has ended, or the student information is incomplete.

class EnrollmentError(Exception): pass
class CourseFullError(EnrollmentError): pass
class EnrollmentPeriodEndedError(EnrollmentError): pass
class IncompleteStudentInfoError(EnrollmentError): pass

def enroll_in_course(course_id, student_info):
    try:
        if not all(key in student_info for key in ["name", "email"]):
            raise IncompleteStudentInfoError("Student information is incomplete.")
        if simulate_course_full(course_id):
            raise CourseFullError("Course is full.")
        if simulate_enrollment_period_ended(course_id):
            raise EnrollmentPeriodEndedError("Enrollment period has ended.")
        print("Enrollment successful.")
    except IncompleteStudentInfoError as e:
        print(f"Error: {e}")
    except CourseFullError as e:
        print(f"Error: {e}")
    except EnrollmentPeriodEndedError as e:
        print(f"Error: {e}")

def simulate_course_full(course_id):
    return False  # Simulated course availability
```

```
def simulate_enrollment_period_ended(course_id):
    return False  # Simulated enrollment period status

# Test
enroll_in_course(101, {"name": "Alice", "email": "alice@example.com"})

# Question 22
# Library Book Return: Implement a function that processes the return
# of a borrowed library book. Handle scenarios where the book ID is
# invalid, the return date is past due, or the library system is
# offline.

class BookReturnError(Exception): pass
class InvalidBookIDError(BookReturnError): pass
class PastDueDateError(BookReturnError): pass
class LibrarySystemOfflineError(BookReturnError): pass

def return_book(book_id, return_date):
    try:
        if not simulate_book_exists(book_id):
            raise InvalidBookIDError("Book ID is invalid.")
        if return_date < datetime.date.today():
            raise PastDueDateError("Return date is past due.")
        if not simulate_library_system_online():
            raise LibrarySystemOfflineError("Library system is offline.")
        print("Book returned successfully.")
    except InvalidBookIDError as e:
        print(f"Error: {e}")
    except PastDueDateError as e:
        print(f"Error: {e}")
    except LibrarySystemOfflineError as e:
        print(f"Error: {e}")

def simulate_book_exists(book_id):
    return True  # Simulated book existence check

def simulate_library_system_online():
    return True  # Simulated library system status

# Test
return_book(456, datetime.date(2024, 6, 30))

# Question 23
# E-commerce Checkout: Create a function that processes an e-commerce
# checkout. Handle cases where the payment method is invalid, the
# shipping address is incomplete, or the cart is empty.

class CheckoutError(Exception): pass
class InvalidPaymentMethodError(CheckoutError): pass
class IncompleteShippingAddressError(CheckoutError): pass
class EmptyCartError(CheckoutError): pass

def checkout(cart, payment_method, shipping_address):
    try:
        if not cart:
            raise EmptyCartError("The cart is empty.")
        if not payment_method:
            raise InvalidPaymentMethodError("Payment method is invalid.")
        if not shipping_address:
            raise IncompleteShippingAddressError("Shipping address is incomplete.")
        if not simulate_payment_process(payment_method):
            raise InvalidPaymentMethodError("Payment processing failed.")
        print("Checkout successful.")
    except EmptyCartError as e:
        print(f"Error: {e}")
    except InvalidPaymentMethodError as e:
        print(f"Error: {e}")
    except IncompleteShippingAddressError as e:
        print(f"Error: {e}")

def simulate_payment_process(payment_method):
    return True  # Simulated payment processing

# Test
checkout(["item1", "item2"], {"card_number": "4111111111111111"}, {"address": "123 Main St"})
```

```
# Question 24
# Healthcare Appointment Booking: Write a function that books an
# appointment with a healthcare provider. Handle scenarios where the
# provider is fully booked, the appointment date is in the past, or the
# patient information is missing.

class AppointmentError(Exception): pass
class ProviderFullyBookedError(AppointmentError): pass
class PastAppointmentDateError(AppointmentError): pass
class MissingPatientInfoError(AppointmentError): pass

def book_appointment(provider_id, appointment_date, patient_info):
    try:
        if not all(key in patient_info for key in ["name", "dob"]):
            raise MissingPatientInfoError("Patient information is missing.")
        if appointment_date < datetime.datetime.now():
            raise PastAppointmentDateError("Appointment date is in the past.")
        if simulate_provider_fully_booked(provider_id, appointment_date):
            raise ProviderFullyBookedError("Provider is fully booked.")
        print("Appointment booked successfully.")
    except MissingPatientInfoError as e:
        print(f"Error: {e}")
    except PastAppointmentDateError as e:
        print(f"Error: {e}")
    except ProviderFullyBookedError as e:
        print(f"Error: {e}")

def simulate_provider_fully_booked(provider_id, appointment_date):
    return False  # Simulated provider availability

# Test
book_appointment("P001", datetime.datetime(2024, 7, 1, 14, 0), {"name": "Bob", "dob": "1980-01-01"})

# Question 25
# Remote Server Command Execution: Implement a function that executes
# commands on a remote server. Handle cases where the server is
# unreachable, the command execution fails, or the authentication
# credentials are incorrect.

class CommandExecutionError(Exception): pass
class ServerUnreachableError(CommandExecutionError): pass
class CommandFailedError(CommandExecutionError): pass
class AuthenticationError(CommandExecutionError): pass

def execute_remote_command(server, command, credentials):
    try:
        if not simulate_server_reachable(server):
            raise ServerUnreachableError("Server is unreachable.")
        if not authenticate(credentials):
            raise AuthenticationError("Authentication failed.")
        if not simulate_command_execution(server, command):
            raise CommandFailedError("Command execution failed.")
        print("Command executed successfully.")
    except ServerUnreachableError as e:
        print(f"Error: {e}")
    except AuthenticationError as e:
        print(f"Error: {e}")
    except CommandFailedError as e:
        print(f"Error: {e}")

def simulate_server_reachable(server):
    return True  # Simulated server reachability

def authenticate(credentials):
    return True  # Simulated authentication

def simulate_command_execution(server, command):
    return True  # Simulated command execution

# Test
execute_remote_command("server.example.com", "ls -l", {"username": "admin", "password": "password"})

# Question 31
# Event Ticket Booking: Implement a function that books tickets for an
# event. Handle cases where the event is sold out, the ticket quantity is
# invalid, or the payment fails.
```

```
class TicketBookingError(Exception): pass
class EventSoldOutError(TicketBookingError): pass
class InvalidTicketQuantityError(TicketBookingError): pass
class PaymentFailedError(TicketBookingError): pass

def book_event_tickets(event_id, ticket_quantity, payment_info):
    try:
        if ticket_quantity <= 0:
            raise InvalidTicketQuantityError("Ticket quantity is invalid.")
        if simulate_event_sold_out(event_id):
            raise EventSoldOutError("Event is sold out.")
        if not simulate_payment_process(payment_info):
            raise PaymentFailedError("Payment failed.")
        print("Tickets booked successfully.")
    except InvalidTicketQuantityError as e:
        print(f"Error: {e}")
    except EventSoldOutError as e:
        print(f"Error: {e}")
    except PaymentFailedError as e:
        print(f"Error: {e}")

def simulate_event_sold_out(event_id):
    return False  # Simulated event availability

def simulate_payment_process(payment_info):
    return True  # Simulated payment processing

# Test
book_event_tickets(101, 2, {"card_number": "4111111111111111", "cvv": "123"})

# Question 32
# Product Review Submission: Create a function that submits a product
# review. Handle scenarios where the review content is empty, the rating
# is out of range, or the product ID is invalid.

class ReviewSubmissionError(Exception): pass
class EmptyReviewContentError(ReviewSubmissionError): pass
class InvalidRatingError(ReviewSubmissionError): pass
class InvalidProductIDError(ReviewSubmissionError): pass

def submit_product_review(product_id, review_content, rating):
    try:
        if not review_content:
            raise EmptyReviewContentError("Review content is empty.")
        if not (1 <= rating <= 5):
            raise InvalidRatingError("Rating is out of range.")
        if not simulate_valid_product_id(product_id):
            raise InvalidProductIDError("Product ID is invalid.")
        print("Review submitted successfully.")
    except EmptyReviewContentError as e:
        print(f"Error: {e}")
    except InvalidRatingError as e:
        print(f"Error: {e}")
    except InvalidProductIDError as e:
        print(f"Error: {e}")

def simulate_valid_product_id(product_id):
    return True  # Simulated product ID validation

# Test
submit_product_review(1234, "Great product!", 5)

# Question 33
# Chat Application Message Sending: Write a function that sends a message
# in a chat application. Handle cases where the recipient ID is invalid,
# the message content is empty, or the server is unreachable.

class MessageSendingError(Exception): pass
class InvalidRecipientIDError(MessageSendingError): pass
class EmptyMessageContentError(MessageSendingError): pass
class ServerUnreachableError(MessageSendingError): pass

def send_chat_message(recipient_id, message_content):
    try:
        if not recipient_id:
            raise InvalidRecipientIDError("Recipient ID is invalid.")
        if not message_content:
```

```
            raise EmptyMessageContentError("Message content is empty.")
        if not simulate_server_reachable():
            raise ServerUnreachableError("Server is unreachable.")
        print("Message sent successfully.")
    except InvalidRecipientIDError as e:
        print(f"Error: {e}")
    except EmptyMessageContentError as e:
        print(f"Error: {e}")
    except ServerUnreachableError as e:
        print(f"Error: {e}")

def simulate_server_reachable():
    return True  # Simulated server reachability

# Test
send_chat_message("user123", "Hello, how are you?")

# Question 34
# User Registration System: Implement a function that registers a new
# user. Handle scenarios where the username is already taken, the password
# is too weak, or required fields are missing.

class UserRegistrationError(Exception): pass
class UsernameTakenError(UserRegistrationError): pass
class WeakPasswordError(UserRegistrationError): pass
class MissingRequiredFieldsError(UserRegistrationError): pass

def register_user(username, password, email):
    try:
        if not username or not password or not email:
            raise MissingRequiredFieldsError("Required fields are missing.")
        if simulate_username_taken(username):
            raise UsernameTakenError("Username is already taken.")
        if len(password) < 6:
            raise WeakPasswordError("Password is too weak.")
        print("User registered successfully.")
    except MissingRequiredFieldsError as e:
        print(f"Error: {e}")
    except UsernameTakenError as e:
        print(f"Error: {e}")
    except WeakPasswordError as e:
        print(f"Error: {e}")

def simulate_username_taken(username):
    return False  # Simulated username availability

# Test
register_user("newuser", "password123", "newuser@example.com")

# Question 35
# Data Backup Service: Create a function that backs up data to a remote
# server. Handle cases where the backup file is too large, the server is
# unreachable, or the authentication fails.

class DataBackupError(Exception): pass
class BackupFileTooLargeError(DataBackupError): pass
class BackupServerUnreachableError(DataBackupError): pass
class BackupAuthenticationError(DataBackupError): pass

def backup_data(file_path, server, credentials):
    try:
        if simulate_file_too_large(file_path):
            raise BackupFileTooLargeError("Backup file is too large.")
        if not simulate_server_reachable(server):
            raise BackupServerUnreachableError("Backup server is unreachable.")
        if not simulate_authentication(credentials):
            raise BackupAuthenticationError("Authentication failed.")
        print("Data backed up successfully.")
    except BackupFileTooLargeError as e:
        print(f"Error: {e}")
    except BackupServerUnreachableError as e:
        print(f"Error: {e}")
    except BackupAuthenticationError as e:
        print(f"Error: {e}")

def simulate_file_too_large(file_path):
    return False  # Simulated file size check
```

```
def simulate_server_reachable(server):
    return True  # Simulated server reachability

def simulate_authentication(credentials):
    return True  # Simulated authentication

# Test
backup_data("backup.zip", "backup.server.com", {"username": "admin", "password": "password"})

# Question 36
# Online Quiz Timer: Write a function that manages the timer for an
# online quiz. Handle scenarios where the time format is incorrect, the
# timer starts late, or the timer malfunctions.

import time

class QuizTimerError(Exception): pass
class IncorrectTimeFormatError(QuizTimerError): pass
class TimerStartError(QuizTimerError): pass
class TimerMalfunctionError(QuizTimerError): pass

def start_quiz_timer(duration):
    try:
        if not isinstance(duration, int) or duration <= 0:
            raise IncorrectTimeFormatError("Time format is incorrect.")
        start_time = time.time()
        if time.time() - start_time > 1:
            raise TimerStartError("Timer starts late.")
        # Simulate timer running
        for _ in range(duration):
            time.sleep(1)
            if simulate_timer_malfunction():
                raise TimerMalfunctionError("Timer malfunction.")
        print("Quiz timer ended successfully.")
    except IncorrectTimeFormatError as e:
        print(f"Error: {e}")
    except TimerStartError as e:
        print(f"Error: {e}")
    except TimerMalfunctionError as e:
        print(f"Error: {e}")

def simulate_timer_malfunction():
    return False  # Simulated timer malfunction

# Test
start_quiz_timer(5)

# Question 37
# Multi-language Translation Service: Implement a function that translates
# text into multiple languages using an API. Handle cases where the text
# is too long, the language code is invalid, or the API request fails.

import requests

class TranslationError(Exception): pass
class TextTooLongError(TranslationError): pass
class InvalidLanguageCodeError(TranslationError): pass
class APIRequestFailedError(TranslationError): pass

def translate_text(text, language_code):
    try:
        if len(text) > 1000:
            raise TextTooLongError("Text is too long.")
        if not simulate_valid_language_code(language_code):
            raise InvalidLanguageCodeError("Language code is invalid.")
        response = requests.post("http://api.translate.com/translate", data={"text": text, "lang": language_code}
        if response.status_code != 200:
            raise APIRequestFailedError("API request failed.")
        translation = response.json().get("translation")
        print(f"Translated text: {translation}")
    except TextTooLongError as e:
        print(f"Error: {e}")
    except InvalidLanguageCodeError as e:
        print(f"Error: {e}")
    except APIRequestFailedError as e:
        print(f"Error: {e}")
```

```
def simulate_valid_language_code(language_code):
    valid_codes = ["en", "es", "fr", "de", "zh"]
    return language_code in valid_codes

# Test
translate_text("Hello, world!", "es")

# Question 38
# Restaurant Reservation System: Create a function that makes a reservation
# at a restaurant. Handle scenarios where the restaurant is fully booked,
# the reservation time is invalid, or the customer information is incomplete.

class ReservationError(Exception): pass
class RestaurantFullyBookedError(ReservationError): pass
class InvalidReservationTimeError(ReservationError): pass
class IncompleteCustomerInfoError(ReservationError): pass

def make_reservation(restaurant_id, reservation_time, customer_info):
    try:
        if not reservation_time or reservation_time < 0 or reservation_time > 24:
            raise InvalidReservationTimeError("Reservation time is invalid.")
        if not customer_info.get("name") or not customer_info.get("contact"):
            raise IncompleteCustomerInfoError("Customer information is incomplete.")
        if simulate_restaurant_fully_booked(restaurant_id, reservation_time):
            raise RestaurantFullyBookedError("Restaurant is fully booked.")
        print("Reservation made successfully.")
    except InvalidReservationTimeError as e:
        print(f"Error: {e}")
    except IncompleteCustomerInfoError as e:
        print(f"Error: {e}")
    except RestaurantFullyBookedError as e:
        print(f"Error: {e}")

def simulate_restaurant_fully_booked(restaurant_id, reservation_time):
    return False  # Simulated restaurant availability

# Test
make_reservation(101, 19, {"name": "John Doe", "contact": "123-456-7890"})

# Question 39
# Automatic Billing System: Write a function that processes automatic billing
# for subscriptions. Handle cases where the payment method is declined, the
# billing cycle is incorrect, or the user account is inactive.

class BillingError(Exception): pass
class PaymentDeclinedError(BillingError): pass
class IncorrectBillingCycleError(BillingError): pass
class InactiveAccountError(BillingError): pass

def process_automatic_billing(user_account, payment_info, billing_cycle):
    try:
        if billing_cycle not in ["monthly", "yearly"]:
            raise IncorrectBillingCycleError("Billing cycle is incorrect.")
        if not simulate_active_account(user_account):
            raise InactiveAccountError("User account is inactive.")
        if not simulate_payment_process(payment_info):
            raise PaymentDeclinedError("Payment method is declined.")
        print("Automatic billing processed successfully.")
    except IncorrectBillingCycleError as e:
        print(f"Error: {e}")
    except InactiveAccountError as e:
        print(f"Error: {e}")
    except PaymentDeclinedError as e:
        print(f"Error: {e}")

def simulate_active_account(user_account):
    return True  # Simulated account status check

# Test
process_automatic_billing("user123", {"card_number": "4111111111111111", "cvv": "123"}, "monthly")

# Question 40
# Cloud Storage File Deletion: Implement a function that deletes a file from
# cloud storage. Handle scenarios where the file ID is invalid, the file does
# not exist, or the server is unreachable.
```

```
class FileDeletionError(Exception): pass
class InvalidFileIDError(FileDeletionError): pass
class FileNotFoundError(FileDeletionError): pass
class ServerUnreachableError(FileDeletionError): pass

def delete_cloud_file(file_id):
    try:
        if not simulate_valid_file_id(file_id):
            raise InvalidFileIDError("File ID is invalid.")
        if not simulate_file_exists(file_id):
            raise FileNotFoundError("File does not exist.")
        if not simulate_server_reachable():
            raise ServerUnreachableError("Server is unreachable.")
        print("File deleted successfully.")
    except InvalidFileIDError as e:
        print(f"Error: {e}")
    except FileNotFoundError as e:
        print(f"Error: {e}")
    except ServerUnreachableError as e:
        print(f"Error: {e}")

def simulate_valid_file_id(file_id):
    return True  # Simulated file ID validation

def simulate_file_exists(file_id):
    return True  # Simulated file existence check

# Test
delete_cloud_file("file123")

# Question 41
# Fitness Tracker Data Sync: Create a function that syncs data from a fitness
# tracker to an app. Handle cases where the device is disconnected, the data
# format is invalid, or the sync fails.

class DataSyncError(Exception): pass
class DeviceDisconnectedError(DataSyncError): pass
class InvalidDataFormatError(DataSyncError): pass
class SyncFailedError(DataSyncError): pass

def sync_fitness_data(device_id, data):
    try:
        if simulate_device_disconnected(device_id):
            raise DeviceDisconnectedError("Device is disconnected.")
        if not simulate_valid_data_format(data):
            raise InvalidDataFormatError("Data format is invalid.")
        if not simulate_sync_process(data):
            raise SyncFailedError("Sync failed.")
        print("Fitness data synced successfully.")
    except DeviceDisconnectedError as e:
        print(f"Error: {e}")
    except InvalidDataFormatError as e:
        print(f"Error: {e}")
    except SyncFailedError as e:
        print(f"Error: {e}")

def simulate_device_disconnected(device_id):
    return False  # Simulated device connectivity status

def simulate_valid_data_format(data):
    return True  # Simulated data format validation

def simulate_sync_process(data):
    return True  # Simulated sync process

# Test
sync_fitness_data("device123", {"steps": 1000, "calories": 500})

# Question 42
# Online Store Inventory Update: Write a function that updates the inventory
# of an online store. Handle scenarios where the product ID is invalid, the
# update quantity is negative, or the database connection fails.

class InventoryUpdateError(Exception): pass
class InvalidProductIDError(InventoryUpdateError): pass
class NegativeQuantityError(InventoryUpdateError): pass
class DatabaseConnectionError(InventoryUpdateError): pass
```

```
def update_inventory(product_id, quantity):
    try:
        if not simulate_valid_product_id(product_id):
            raise InvalidProductIDError("Product ID is invalid.")
        if quantity < 0:
            raise NegativeQuantityError("Update quantity is negative.")
        if not simulate_database_connection():
            raise DatabaseConnectionError("Database connection failed.")
        print("Inventory updated successfully.")
    except InvalidProductIDError as e:
        print(f"Error: {e}")
    except NegativeQuantityError as e:
        print(f"Error: {e}")
    except DatabaseConnectionError as e:
        print(f"Error: {e}")

def simulate_valid_product_id(product_id):
    return True  # Simulated product ID validation

def simulate_database_connection():
    return True  # Simulated database connection status

# Test
update_inventory(101, 50)

# Question 43
# Payment Gateway Integration: Implement a function that integrates with a
# payment gateway to process transactions. Handle cases where the transaction
# is declined, the payment details are invalid, or the gateway is unreachable.

class PaymentGatewayError(Exception): pass
class TransactionDeclinedError(PaymentGatewayError): pass
class InvalidPaymentDetailsError(PaymentGatewayError): pass
class GatewayUnreachableError(PaymentGatewayError): pass

def process_payment(amount, payment_details):
    try:
        if not simulate_valid_payment_details(payment_details):
            raise InvalidPaymentDetailsError("Payment details are invalid.")
        if simulate_transaction_declined():
            raise TransactionDeclinedError("Transaction declined.")
        if not simulate_gateway_reachable():
            raise GatewayUnreachableError("Gateway is unreachable.")
        print("Payment processed successfully.")
    except InvalidPaymentDetailsError as e:
        print(f"Error: {e}")
    except TransactionDeclinedError as e:
        print(f"Error: {e}")
    except GatewayUnreachableError as e:
        print(f"Error: {e}")

def simulate_valid_payment_details(payment_details):
    return True  # Simulated payment details validation

def simulate_transaction_declined():
    return False  # Simulated transaction declined

def simulate_gateway_reachable():
    return True  # Simulated gateway reachability

# Test
process_payment(100, {"card_number": "4111111111111111", "cvv": "123"})

# Question 44
# Content Management System (CMS) Post Creation: Create a function that
# creates a new post in a CMS. Handle scenarios where the post content
# is empty, the post title is missing, or the server is down.

class PostCreationError(Exception): pass
class EmptyPostContentError(PostCreationError): pass
class MissingPostTitleError(PostCreationError): pass
class ServerDownError(PostCreationError): pass

def create_cms_post(title, content):
    try:
        if not title:
```

```
            raise MissingPostTitleError("Post title is missing.")
        if not content:
            raise EmptyPostContentError("Post content is empty.")
        if not simulate_server_available():
            raise ServerDownError("Server is down.")
        print("CMS post created successfully.")
    except MissingPostTitleError as e:
        print(f"Error: {e}")
    except EmptyPostContentError as e:
        print(f"Error: {e}")
    except ServerDownError as e:
        print(f"Error: {e}")

def simulate_server_available():
    return True  # Simulated server availability

# Test
create_cms_post("New Post", "This is the content of the new post.")

# Question 45
# IoT Device Configuration: Write a function that configures settings on an
# IoT device. Handle cases where the device ID is invalid, the configuration
# data is incorrect, or the device is unreachable.

class IoTDeviceError(Exception): pass
class InvalidDeviceIDError(IoTDeviceError): pass
class IncorrectConfigurationDataError(IoTDeviceError): pass
class DeviceUnreachableError(IoTDeviceError): pass

def configure_iot_device(device_id, config_data):
    try:
        if not simulate_valid_device_id(device_id):
            raise InvalidDeviceIDError("Device ID is invalid.")
        if not simulate_valid_config_data(config_data):
            raise IncorrectConfigurationDataError("Configuration data is incorrect.")
```