Understanding Syntax Errors and Exception Handling in Python

There are 2 stages where error may happen in a program

* During compilation -> Syntax Error
* During execution -> Exceptions

Syntax Error

* Something in the program is not written according to the program grammar.
* Error is raised by the interpreter/compiler
* You can solve it by rectifying the program

Python syntax error demonstration with print statement

Explanation

* The code contains a classic Python 2 to Python 3 syntax error where the print statement is missing parentheses
* In Python 3, print must be called as a function with parentheses around its arguments
* This will raise a SyntaxError when executed in Python 3 environment
* The correct syntax should be print('Hello World') instead of print 'Hello World'
* This demonstrates how language version compatibility can cause unexpected errors in code execution

```
# Examples of sytnax error
```

Output

```
  Cell In[1], line 3
    print
```

Other examples of syntax error

* Leaving symbols like colon,brackets

* Misspelling a keyword
* Incorrect indentation
* empty if/else/loops/class/functions

This Python code attempts to print a message based on a conditional check of a variable's value.

Explanation

* The variable `a` is assigned the integer value of 5.
* An `if` statement checks if `a` is equal to 3.
* If the condition is true, it would print 'hello', but the condition will not be satisfied in this case.
* The code contains a syntax error due to a missing colon (`:`) at the end of the `if` statement.

```
a = 5
if a==3
```

Output

```
  Cell In[2], line 2
    if a==
```

This code attempts to print a message based on a conditional check of a variable's value.

Explanation

* The variable `a` is assigned the integer value of 5.
* The conditional statement uses `iff`, which is a syntax error; it should be `if`.
* If corrected, the code would check if `a` is equal to 3 and print 'hello' if true.
* As it stands, the condition will never be met, and nothing will be printed.

```
a = 5
iff a==3:
```

Output

```
  Cell In[3], line 2
    iff a=
```

This code snippet checks if a variable equals a specific value and attempts to print a message.

Explanation

* The variable `a` is assigned the integer value of 5.
* An `if` statement checks if `a` is equal to 3.
* If the condition is true, it would print 'hello', but in this case, it does not execute because `a` is 5.
* The indentation for the print statement is incorrect, which would cause a syntax error if the condition were met.

```
a = 5
```

```
if a==3:
```

Output

```
  Cell In[4], line 3
```

```
    print(
```

Understanding how IndexError occurs when accessing an out-of-bounds index in a list

Explanation

* The code attempts to access the 100th index of a list `L` that only contains three elements.
* Since list indices in Python are zero-based, valid indices for `L` are 0, 1, and 2.
* Attempting to access an index that exceeds the list's length results in an `IndexError`.
* This error indicates that the requested index is not available within the list's range.

```
# IndexError
```

```
# The Inde
```

Output

```
---------------------------------------------------------------------------IndexError
                           Traceback (most recent call last)Cell In[5], line 4
```

```
      1 #
```

Understanding the ModuleNotFoundError in Python when importing non-existent modules

Explanation

* The code attempts to import a module named `mathi`, which does not exist, leading to a `ModuleNotFoundError`.
* The `math` module is a standard library in Python, but the code incorrectly references a non-existent variant.
* The line `math.floor(5.3)` is intended to use the `floor` function to round down the number 5.3, but it will not execute due to the import error.
* This snippet highlights the importance of ensuring that the module names are correct to avoid runtime errors.

```
# ModuleNotFoundError
```

```
# The Modu
```

Output

```
---------------------------------------------------------------------------ModuleNotFoun
dError                        Traceback (most recent call last)Cell In[6], line 3
```

```
      1 #
```

Understanding how a KeyError is raised when accessing a missing dictionary key in Python

Explanation

* The code defines a dictionary `d` with a single key-value pair: `'name': 'madhu'`.
* It attempts to access the value associated with the key `'age'`, which does not exist in the dictionary.
* This results in a `KeyError`, indicating that the specified key is not found in the dictionary.
* KeyErrors are common when trying to retrieve values from dictionaries without checking for the existence of the key first.
* To avoid this error, one can use methods like `get()` or check for the key's presence using

the `in` keyword.

```
# KeyError
```
```
# The KeyE
```

Output

```
---------------------------------------------------------------------------KeyError
                          Traceback (most recent call last)Cell In[7], line 5
```
```
      1 #
```

Demonstrating Python's TypeError when combining incompatible data types

Explanation

* Attempting to add an integer (1) and a string ('a') results in a TypeError because Python doesn't support direct concatenation between different data types
* This error occurs at runtime when Python tries to execute the invalid operation and cannot find a suitable method to handle the combination
* The TypeError serves as a safeguard to prevent ambiguous operations that could lead to unexpected behavior in programs
* To fix this issue, you would need to convert one operand to match the other's type, such as converting the integer to string or vice versa
* Understanding this error helps developers write more robust code by anticipating type compatibility requirements in their operations

```
# TypeError
```
```
# The Type
inappropri
```

Output

```
---------------------------------------------------------------------------TypeError
                          Traceback (most recent call last)Cell In[8], line 3
```
```
      1 #
```

Understanding how ValueError is raised in Python when type conversion fails

Explanation

* The code attempts to convert the string 'a' into an integer using the `int()` function.
* Since 'a' is not a valid representation of an integer, Python raises a `ValueError`.
* This error indicates that the argument provided to the function is of an inappropriate type for the expected operation.
* Handling such errors can be done using try-except blocks to prevent program crashes.

```
# ValueError
```

```
# The Valu
```

Output

```
---------------------------------------------------------------------------ValueError
                           Traceback (most recent call last)Cell In[9], line 3
```

```
      1 #
```

Understanding the NameError in Python when referencing an undefined variable

Explanation

* The code attempts to print the value of the variable `k`, which has not been defined.
* When Python encounters an undefined variable, it raises a `NameError`.
* This error indicates that the interpreter cannot find a reference to the variable name in the current scope.
* To resolve this error, ensure that the variable is defined before it is used in the code.
* This snippet serves as a basic illustration of how Python handles variable scope and error reporting.

```
# NameError
```

```
# The Name
```

Output

```
---------------------------------------------------------------------------NameError
                            Traceback (most recent call last)Cell In[10], line 3
                                                                       1 #
```

Understanding the Cause of an AttributeError in Python Lists

Explanation

* The code attempts to call the `upper()` method on a list `L`, which is not a valid operation since `upper()` is a string method.
* This results in an `AttributeError`, indicating that the list object has no attribute named `upper`.
* The comment `# Stacktrace` suggests that the error message would provide a traceback of the error, helping to identify where the issue occurred.
* To fix this error, ensure that `upper()` is called on a string object instead of a list.

```
# AttributeError
                                                                       L = [1,2,3
```

Output

```
---------------------------------------------------------------------------AttributeErro
r                            Traceback (most recent call last)Cell In[11], line 3
                                                                       1 #
```

Exceptions

If things go wrong during the execution of the program(runtime). It generally happens when something unforeseen has happened.

* Exceptions are raised by python runtime
* You have to tackle is on the fly

Examples

* Memory overflow
* Divide by 0 -> logical error
* Database error

Understanding the Importance of Exception Handling in Python

Explanation

* The code snippet emphasizes the significance of managing errors in Python applications.
* It introduces the 'try-except' block as a fundamental structure for catching and handling exceptions.
* Proper exception handling prevents program crashes and allows for graceful error recovery.
* It encourages developers to anticipate potential errors and implement robust error management strategies.

```
# Why is it important to handle exceptions
```

```
# how to h
```

This code snippet demonstrates how to create and write to a text file in Python.

Explanation

* The `open` function is used to create a new file named 'sample.txt' in write mode ('w').
* The `with` statement ensures that the file is properly closed after its suite finishes, even if an error occurs.
* The `write` method is called on the file object `f` to write the string 'hello world' into the file.
* If 'sample.txt' already exists, it will be overwritten with the new content.

```
# let's create a file
```

```
with open(
```

This code demonstrates error handling while attempting to read a file in Python.

Explanation

* The `try` block attempts to open and read the contents of a file named 'sample1.txt'.
* The `with` statement ensures that the file is properly closed after its suite finishes, even if an

error occurs.
* If the file is not found or another error occurs, the `except` block catches the exception and prints a user-friendly message.
* This structure helps prevent crashes and allows for graceful error handling in file operations.

```
# try catch demo
try:
```

Output

```
sorry file not found
```

Handling exceptions while reading a file in Python to prevent crashes

Explanation

* The code attempts to open and read a file named 'sample.txt' in read mode.
* If the file is successfully opened, its contents are printed to the console.
* The code includes a reference to an undefined variable `m`, which will raise an exception if reached.
* The `except` block catches any exceptions that occur during the try block and prints the error message.
* This structure helps in gracefully handling errors without crashing the program.

```
# Catching specific exception
```

Output

```
hello world
name 'm' i
```

This code demonstrates how to catch and handle exceptions in Python while reading a file.

Explanation

* The code attempts to open and read a file named 'sample.txt' in read mode.
* If the file is successfully opened, it prints the contents of the file.
* The code also attempts to print a variable `m`, which is not defined, leading to an exception.
* The `except` block catches any exception that occurs and prints the exception's traceback for debugging purposes.
* The use of `Exception as e` allows for capturing any type of exception that may arise during the execution of the try block.

```
# Catching specific exception
```

Output

```
hello world
```

```
<built-in
```

Handling multiple exceptions in Python for robust error management

Explanation

* The code attempts to open and read a file named 'sample.txt', which may not exist.
* It includes specific exception handling for `FileNotFoundError`, `NameError`, and `ZeroDivisionError`, providing tailored error messages for each.
* A generic `Exception` catch is included at the end to handle any other unforeseen errors, ensuring that the program does not crash unexpectedly.
* The commented-out lines indicate potential errors that could be raised, demonstrating the importance of exception handling in maintaining program stability.
* This structure allows for graceful degradation of functionality, informing the user of issues without terminating the program abruptly.

```
# Catching specific exception
```

Output

```
hello world
```

```
list index
```

This code snippet demonstrates error handling while reading a file in Python.

Explanation

* The `try` block attempts to open a file named 'sample.txt' in read mode.
* If the file does not exist, a `FileNotFoundError` is caught, and a message 'file not found' is printed.
* Any other exceptions are caught by a general `Exception` handler, which prints 'Something went wrong'.
* If no exceptions occur, the `else` block executes, reading and printing the contents of the file.

```
# else
```

Output

```
hello world
```

This code handles file reading with error management and ensures cleanup execution.

Explanation

* The code attempts to open a file named 'sample.txt' in read mode.
* If the file is not found, it catches the `FileNotFoundError` and prints a specific error message.
* A general exception handler is included to catch any other unexpected errors, providing a fallback message.
* If the file opens successfully, its contents are read and printed.
* The `finally` block ensures that a message is printed regardless of whether an error occurred or not, guaranteeing that cleanup or final actions are always executed.

```
# Finally
```

Output

```
hello world
This will
```

This code handles file operations with error management and guaranteed execution of cleanup tasks.

Explanation

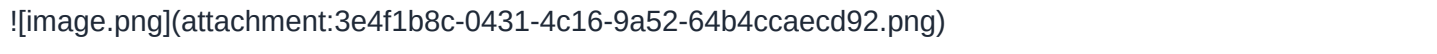
* The code attempts to open a file named 'sample1.txt' in read mode.
* If the file is not found, it catches the `FileNotFoundError` and prints a specific error message.
* A general exception handler is included to catch any other unforeseen errors and print a generic error message.
* If the file opens successfully, its contents are read and printed.
* The `finally` block ensures that a message is printed regardless of whether an error occurred or not, indicating that this part of the code will always execute.

```
# Finally
```

Output

```
file not found
```

```
This will
```



Understanding how to manually raise exceptions in Python for error handling

Explanation

* The `raise` keyword is used to trigger an exception intentionally in Python.
* This allows developers to create custom error messages or handle specific error conditions in their code.
* By passing values to the exception, additional context can be provided, making debugging easier.
* Raising exceptions can help maintain control over program flow and ensure that errors are handled appropriately.

```
# raise Exception
```

```
# In Pytho
```

This code snippet demonstrates how to manually raise a ModuleNotFoundError in Python.

Explanation

* The `raise` statement is used to trigger an exception in Python.
* `ModuleNotFoundError` is a built-in exception that indicates a module could not be found.
* The string 'Just a manual test' serves as a custom error message to provide context for the exception.

* This can be useful for testing error handling in code or for debugging purposes.

```
raise ModuleNotFoundError('Just a manual test')
```

Output

```
---------------------------------------------------------------------------ModuleNotFoun
dError                        Traceback (most recent call last)Cell In[2], line 1
                                                                              ----> 1 ra
```

Manually triggering a ZeroDivisionError in Python for testing purposes

Explanation

* The code raises a `ZeroDivisionError`, which is a built-in exception in Python that indicates an attempt to divide by zero.
* The message 'Again a manual test' provides context for the error, useful for debugging or logging.
* The comments suggest a comparison with Java exception handling, indicating the differences in syntax between the two languages.
* This snippet can be used in testing scenarios to simulate error conditions and verify error handling mechanisms.

```
raise ZeroDivisionError('Again a manual test')
                                                                              # Java
```

Output

```
---------------------------------------------------------------------------ZeroDivisionE
rror                          Traceback (most recent call last)Cell In[3], line 1
                                                                              ----> 1 ra
```

Python exception handling with bank account withdrawal validation

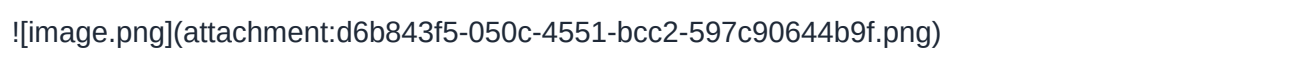
Explanation

* The code defines a Bank class with initialization and withdrawal methods that include input validation for negative amounts and insufficient funds
* It demonstrates proper exception handling using try-except blocks to catch and display error messages when invalid withdrawal operations occur
* The withdrawal method raises custom exceptions with descriptive error messages when withdrawal amounts are negative or exceed available balance
* The code shows how to handle multiple exception scenarios including invalid amount values and insufficient account balance conditions
* The program flow illustrates the difference between normal execution path (else block) and exception handling path (except block) when operations fail

```
class Bank:
```

Output

```
amount cannot be -ve
```



Custom exception handling in a banking application to manage withdrawal errors

Explanation

* Defines a custom exception class `MyException` that inherits from the built-in `Exception` class.
* The `Bank` class initializes with a `balance` and includes a `withdraw` method to handle withdrawal requests.

* The `withdraw` method checks if the withdrawal amount is negative or exceeds the current balance, raising `MyException` with appropriate messages if either condition is met.
* An instance of `Bank` is created with an initial balance, and a withdrawal attempt is made within a try-except block to catch and handle the custom exception without crashing the application.
* This structure allows for precise control over error handling, improving the robustness of the application.

```
class MyException(Exception):
    def __in
```

Output

```
amount cannot be -ve
```

Implementing a secure login system with custom error handling in Python

Explanation

* A custom exception class `SecurityError` is defined to handle security-related issues during login attempts.
* The `Google` class manages user credentials and device verification for secure login functionality.

* The `login` method checks if the provided device matches the registered device; if not, it raises a `SecurityError`.
* The try-except block captures the `SecurityError` and calls the `logout` method if an error occurs, ensuring proper error handling.
* The `finally` block executes regardless of the outcome, indicating that the database connection is closed after the login attempt.

```
class SecurityError(Exception):
```

Output

```
device not match
logout
```

Exception Handling

###`Q-1`: You are given a function definition. There might be several issues on execution of this function. You are asked to do exception handling for diffrent errors that this function goes in to `without altering this function`. And print error text.

Function parameters `l -> list, s -> could be anything`

```
def function(l: list, s, **args):
    last_e
```

Check for different function calls:-

```
function([1,2,1], 12)
function([
```

This Python function manipulates a list and handles multiple exceptions during execution.

Explanation

* The `function` takes a list `l`, a string `s`, and additional keyword arguments, modifying elements in the list based on the index derived from `s`.
* It sets the element at index `s` and `s + 10` to specific values and calculates the sum of the modified list.
* The result is adjusted by dividing by the last element of the list and multiplying by a parameter `p` from the keyword arguments.
* The function is wrapped in a try-except block to catch various exceptions like `IndexError`, `ValueError`, `TypeError`, `KeyError`, and `ZeroDivisionError`, printing the type and message of the exception if it occurs.

* The `finally` block ensures that "Thank you" is printed regardless of whether an exception occurred or not.

```
# Write code here
```

Output

```
Result: 452.0
Thank you
```

###`Q-2:` You are given a code snippet. There might be several issues on execution of this code. You are asked to do exception handling for diffrent errors, condition is what ever happens we need to execute last line printing correct result of `sum of elements`.

List have elemnts as any no of `key-pair dict with key as list index and value as any integer`,

`integers` and `numeric-strings`. There is always only one element in the dict.

```
l = [{0:2},2,3,4,'5', {5:10}]
```

```
# For calc
```

This code calculates the sum of various data types in a list while handling exceptions.

Explanation

* Initializes a list `l` containing integers, strings, and dictionaries.
* Sets a variable `s` to zero for accumulating the sum.
* Iterates through each element in the list using a for loop.
* Uses a try-except block to handle `TypeError` when adding non-integer types, attempting to retrieve values from dictionaries if encountered.
* Converts string elements to integers and adds them to the sum, finally printing the total sum.

```
# Write code here
```

```
l = [{0:2}
```

Output

```
26
```

`Q-3:`: File Handling with Exception handling

Write a program that opens a text file and write data to it as "Hello, Good Morning!!!". Handle exceptions that can be generated during the I/O operations. Do not show the success message on the main exception handling block (write inside the else block).

This code snippet demonstrates safe file writing with error handling in Python.

Explanation

* The code attempts to open a file named "text_file.txt" in write mode using a context manager to ensure proper resource management.
* If the file operation fails due to an IOError, it catches the exception and prints an error message.
* If the file is written successfully without any exceptions, it prints a success message indicating the operation was completed.
* The use of `with` ensures that the file is properly closed after the block of code is executed, even if an error occurs.

```
# write code here
```

```
try:
```

Output

```
File written successfully
```

`Q-4`: Number game program.

Write a number game program. Ask the user to enter a number. If the number is greater than number to be guessed, raise a **ValueTooLarge** exception. If the value is smaller the number to be guessed the, raise a **ValueTooSmall** exception and prompt the user to enter again. Quit the program only when the user enters the correct number. Also raise **GuessError** if user guess a number less than 1.

This code implements a number guessing game with custom exception handling for user input validation.

Explanation

* The code imports the `random` module to generate a random number between 1 and 100.
* Three custom exception classes (`ValueTooLarge`, `ValueTooSmall`, and `GuessError`) are defined to handle specific input errors.
* A loop prompts the user to enter a guess, checking if the input is valid and comparing it to the randomly generated number.
* If the guess is outside the range of 1-100, or incorrect, the corresponding exception is raised and handled, providing user feedback.
* The game continues until the user correctly guesses the number, at which point a success message is displayed.

```
# Write code here
```

```
import ran
```

Output

```
Input value is too small
```

```
Input valu
```

`Q-5:` Cast vote

Write a program that validate name and age as entered by the user to determine whether the person can cast vote or not. To handle the age, create **InvalidAge** exception and for name, create **InvalidName** exception. The name will be invalid when the string will be empty or name has only one word.

Example 1:

Input:

```
Enter the name:               goransh singh
```

```
Enter the
```

Output:

```
Goransh Singh  Congratulation !!! You can vote.
```

This code validates user input for name and age before accepting a vote.

Explanation

* Defines two custom exceptions, `InvalidAge` and `InvalidName`, each with a method to display an error message.
* Prompts the user to input their name and checks if it contains at least two words; raises `InvalidName` if not.
* Prompts the user to input their age and checks if it is below 18; raises `InvalidAge` if so.
* Uses try-except blocks to handle exceptions and display appropriate error messages.
* If no exceptions occur, it confirms the acceptance of the vote and thanks the user.

```
# Write code here
```

Output

```
Your vote is accepted successfully
```

```
Thank You
```

`Q-6`: Write a python function which infinitely prints natural numbers in a single line. Raise the **StopIteration** exception after displaying first 20 numnbers to exit from the program.

This code increments a number and prints it until a specified limit is reached.

Explanation

* The function `display(n)` takes an integer `n` as input and initializes a counter `i` to zero.
* A `while True` loop is used to continuously increment `n` and `i` until `i` reaches 21.
* When `i` equals 21, a `StopIteration` exception is raised to break the loop.
* The `except` block catches the `StopIteration` exception, terminating the loop gracefully.
* The `print` function outputs the incremented value of `n` on the same line, separated by spaces.

```
# Write code here
```

Output

```
101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120
```