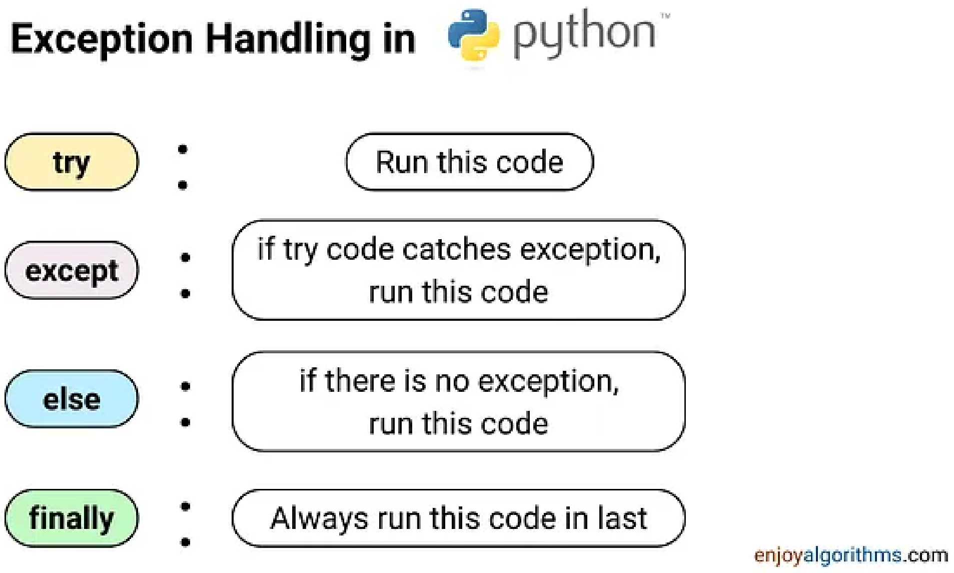
Exception Handling :



Exception Handling in python

try : Run this code

except : if try code catches exception, run this code

else : if there is no exception, run this code

finally : Always run this code in last

enjoyalgorithms.com

```
try:
# Code that may raise an exception
except ExceptionType:
# Code to handle the exception
else:
# Code to execute if no exception occurs
finally:
# Code that always runs, regardless of an exception
```

Exception Handling in Python

Exception handling in Python is a way to handle errors and other exceptional conditions in your code in a graceful manner. When something goes wrong in a program, an exception is raised. If not handled properly, the program crashes. However, Python provides a structured way to catch and handle these exceptions to ensure that the program can either recover or at least fail gracefully.

Key Concepts of Exception Handling

1. **Exceptions**: An exception is an event that occurs during the execution of a program that disrupts its normal flow. Examples of exceptions include division by zero, file not found, or invalid input.
2. **Raising Exceptions**: You can raise exceptions in your code using the `raise` keyword. This is useful when you want to explicitly trigger an error based on certain conditions.
3. **Handling Exceptions**: You can handle exceptions using the `try`, `except`, `else`, and `finally` blocks.
4. **Custom Exceptions**: You can define your own exceptions by creating custom exception classes.

Raising Exceptions

You can raise exceptions manually in your code using the `raise` keyword followed by the exception you want to raise.

Example:

```
pythonCopy code
def check_age(age):
    if age < 18:
        raise ValueError("Age must be 18 or older.")
    return "Age is valid."

try:
    print(check_age(16))
except ValueError as e:
    print(f"Error: {e}")
```

Exception Hierarchy

All exceptions in Python are derived from the base class `BaseException`. Some common exceptions include:

- **Exception**: The base class for most exceptions.
- **ArithmeticError**: The base class for arithmetic errors like `ZeroDivisionError`.
- **ValueError**: Raised when a function gets an argument of the correct type but with an invalid value.
- **TypeError**: Raised when an operation or function is applied to an object of inappropriate type.
- **IOError**: Raised when an input/output operation fails.
- **KeyError**: Raised when a dictionary key is not found.

Basic Syntax of Exception Handling

Python uses the `try`, `except`, `else`, and `finally` blocks to handle exceptions.

Syntax:

```
pythonCopy code
try:
    # Code that may raise an exception
except ExceptionType as e:
    # Code that runs if the exception occurs
else:
    # Code that runs if no exception occurs
finally:
    # Code that runs no matter what (optional)
```

- `try`: Contains code that might throw an exception.
- `except`: Contains code that runs when an exception is raised in the `try` block.
- `else`: Contains code that runs if no exception is raised in the `try` block.

- `finally`: Contains code that always runs, whether an exception was raised or not (e.g., closing files).

Example: Handling Exceptions

```
pythonCopy code
try:
    num1 = int(input("Enter a number: "))
    num2 = int(input("Enter another number: "))
    result = num1 / num2
except ZeroDivisionError:
    print("Error: Division by zero is not allowed.")
except ValueError:
    print("Error: Invalid input. Please enter a valid number.")
else:
    print(f"The result is: {result}")
finally:
    print("This block is always executed, whether an exception occurred or not.")
```

Multiple Exceptions

You can catch multiple exceptions by specifying them in a tuple.

```
pythonCopy code
try:
    num = int(input("Enter a number: "))
    result = 100 / num
except (ValueError, ZeroDivisionError) as e:
    print(f"Error: {e}")
```

The `finally` Block

The `finally` block is used for cleaning up resources, such as closing files or releasing locks, regardless of whether an exception occurred.

```
pythonCopy code
try:
    f = open('somefile.txt', 'r')
    # Some file operations
except FileNotFoundError:
    print("File not found.")
finally:
    f.close()  # Ensures that the file is closed no matter wh
at
```

Creating Custom Exceptions

You can create your own exceptions by defining a class that inherits from `Exception`.

Example:

```
pythonCopy code
class CustomError(Exception):
    pass

def check_positive(number):
    if number < 0:
        raise CustomError("Number must be positive.")

try:
    check_positive(-5)
except CustomError as e:
    print(f"Custom Error: {e}")
```

`tryexceptelse`

The `else` block runs only if no exceptions were raised in the `try` block.

```
pythonCopy code
try:
    result = 10 / 2
except ZeroDivisionError:
    print("Cannot divide by zero.")
else:
    print("Division successful.")
```

Common Built-in Exceptions

Here are some commonly encountered built-in exceptions in Python:

- `AttributeError`: Raised when an invalid attribute reference is made.
- `IndexError`: Raised when a sequence index is out of range.
- `KeyError`: Raised when a dictionary key is not found.
- `TypeError`: Raised when an operation is applied to an object of inappropriate type.
- `ValueError`: Raised when a function receives an argument of the correct type but inappropriate value.
- `FileNotFoundError`: Raised when a file or directory is requested but doesn't exist.
- `IOError`: Raised when an I/O operation fails.

Nested Exception Handling

You can nest `try-except` blocks within each other. This is useful when multiple layers of error handling are needed.

Example:

```
pythonCopy code
try:
    try:
        num = int(input("Enter a number: "))
        result = 10 / num
    except ValueError:
        print("Invalid input.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
```

Best Practices for Exception Handling

1. **Catch Specific Exceptions**: Always catch the most specific exceptions possible, instead of catching everything with a bare `except`.
2. **Use Finally for Cleanup**: Use the `finally` block to clean up resources, such as closing files or releasing locks.
3. **Avoid Overuse of Exceptions**: Don't use exceptions for normal control flow; reserve them for exceptional cases.
4. **Log Exceptions**: Log exceptions to help with debugging instead of just printing them.

Example: Logging Exceptions

```
pythonCopy code
import logging

try:
    num = int(input("Enter a number: "))
    result = 100 / num
except ZeroDivisionError as e:
    logging.error("Attempted division by zero")
except ValueError as e:
    logging.error("Invalid input provided")
```

Here is a summary table of common Python errors and their descriptions:

Error Type	Description	Example Cause
`SyntaxError`	Raised when there is invalid syntax in the code.	Missing colon in an `if` statement.
`IndentationError`	Raised when there is incorrect indentation.	Extra or missing indentation in a block of code.
`NameError`	Raised when a variable or function name is not defined.	Using a variable that hasn't been initialized.
`TypeError`	Raised when an operation is performed on an inappropriate type.	Adding a string to an integer.
`ValueError`	Raised when a function receives an argument of correct type but inappropriate value.	Converting a non-numeric string to an integer.
`IndexError`	Raised when accessing an index out of range in a sequence.	Accessing the 5th element of a list with only 3 items.
`KeyError`	Raised when a dictionary key is not found.	Accessing a non-existent key in a dictionary.
`AttributeError`	Raised when an invalid attribute reference is made.	Accessing an undefined attribute of an object.
`ZeroDivisionError`	Raised when attempting to divide by zero.	Dividing a number by zero.
`ImportError`	Raised when an import statement fails to find the module.	Importing a non-existent module.
`ModuleNotFoundError`	A subclass of `ImportError`, raised when a module cannot be found.	Importing a module that doesn't exist.
`FileNotFoundError`	Raised when a file operation fails due to the file not existing.	Trying to open a non-existent file.
`IOError`	Raised when an I/O operation fails.	Attempting to read from a file that is closed.

`OverflowError`	Raised when the result of an arithmetic operation is too large.	Calculating an extremely large exponential.
`MemoryError`	Raised when an operation runs out of memory.	Creating a very large list that exceeds memory.
`StopIteration`	Raised when there are no further items in an iterator.	Using `next()` on an iterator beyond its limit.
`AssertionError`	Raised when an `assert` statement fails.	Failing an assertion check.
`RuntimeError`	Raised when an error occurs that does not belong to any specific category.	Exceeding the recursion depth.
`NotImplementedError`	Raised when an abstract method is called in a base class.	Calling an unimplemented method in a subclass.
`EOFError`	Raised when `input()` reaches the end of the file unexpectedly.	Pressing `Ctrl+D` (Unix) or `Ctrl+Z` (Windows) during input.
`KeyboardInterrupt`	Raised when the user interrupts program execution with `Ctrl+C`.	Stopping a running program with `Ctrl+C`.

1. SyntaxError

- **Description:** Raised when the code contains invalid syntax.
- **Example 1:** Missing colon in an `if` statement.

```
if True
    print("This will raise a SyntaxError")
```

- **Example 2:** Using an assignment operator (`=`) instead of an equality operator (`==`) in a conditional statement.

```
pythonCopy code
if x = 5:  # Should be 'if x == 5:'
    print("This will raise a SyntaxError")
```

2. IndentationError

- **Description:** Raised when the code is not properly indented.
- **Example 1:** Missing indentation in a block of code.

```
def my_function():
print("This will raise an IndentationError")  # This line
should be indented
```

- **Example 2:**

```
for i in range(5):
print(i)  # Should be indented inside the loop
```

3. NameError

- **Description:** Raised when a variable or function name is not defined.
- **Example 1:** Using a variable that hasn't been initialized.

```
print(x)  # x is not defined, this will raise a NameError
```

- **Example 2:** Calling a function that has not been declared.

```
my_function()  # This will raise a NameError if 'my_function' is not defined
```

4. TypeError

- **Description:** Raised when an operation is applied to an object of inappropriate type.
- **Example 1:** Adding a string and an integer.

```
result = "Hello" + 5  # This will raise a TypeError
```

- **Example 2:** Passing the wrong type of argument to a function.

```
def add_numbers(a, b):
    return a + b
add_numbers(5, "10")  # This will raise a TypeError
```

5. ValueError

- **Description:** Raised when a function receives an argument of the correct type but inappropriate value.
- **Example 1:** Converting a non-numeric string to an integer.

```
number = int("abc")  # This will raise a ValueError
```

- **Example 2:** Providing an out-of-range value for a specific function.

```
import math
result = math.sqrt(-1)  # This will raise a ValueError
```

6. IndexError

- **Description:** Raised when trying to access an index that is out of range in a sequence (like a list or tuple).
- **Example 1:** Accessing an out-of-range index in a list.

```
my_list = [1, 2, 3]
print(my_list[5])  # This will raise an IndexError
```

- **Example 2:** Accessing an invalid index in a tuple.

```
my_tuple = (10, 20, 30)
print(my_tuple[4])  # This will raise an IndexError
```

7. KeyError

- **Description:** Raised when trying to access a key that does not exist in a dictionary.
- **Example 1:** Accessing a non-existent key in a dictionary.

```
my_dict = {'a': 1, 'b': 2}
print(my_dict['c'])  # This will raise a KeyError
```

- **Example 2:** Attempting to retrieve a key that isn't present in a dictionary.

```
my_dict = {'name': 'John'}
print(my_dict['age'])  # This will raise a KeyError
```

8. AttributeError

- **Description:** .
- **Example 1:** Accessing a non-existent method of a string.

```
my_string = "Hello"
my_string.append(" World")  # This will raise an Attribute
Error
```

- **Example 2:** Trying to access an undefined attribute of an object.

```
class MyClass:
    pass
obj = MyClass()
print(obj.attribute)  # This will raise an AttributeError
```

9. ZeroDivisionError

- **Description:** Raised when attempting to divide by zero.
- **Example 1:** Dividing an integer by zero.

```
result = 10 / 0  # This will raise a ZeroDivisionError
```

- **Example 2:** Performing integer division by zero.

```
pythonCopy code
result = 10 // 0  # This will raise a ZeroDivisionError
```

10. ImportError / ModuleNotFoundError

- **Description:** Raised when an import statement fails to find a module.
- **Example 1:** Importing a module that doesn’t exist.

```
import non_existent_module  # This will raise a ModuleNotFoundError
```

- **Example 2:** Importing a non-existent function from a valid module.

```
pythonCopy code
from math import nonexistent_function  # This will raise an ImportError
```

11. FileNotFoundError

- **Description:** Raised when a file operation fails due to the file not existing.
- **Example 1:** Trying to open a non-existent file.

```
file = open("non_existent_file.txt")  # This will raise a FileNotFoundError
```

- **Example 2:** Reading a file that has been deleted.

```
import os
os.remove("myfile.txt")
file = open("myfile.txt")  # This will raise a FileNotFoundError
```

12. IOError

- **Description:** Raised when an I/O operation fails (e.g., file read/write).
- **Example 1:** Trying to read a closed file.

```
pythonCopy code
file = open("file.txt", "r")
file.close()
file.read()  # This will raise an IOError
```

- **Example 2:** Trying to write to a read-only file.

```
pythonCopy code
file = open("file.txt", "r")
file.write("Hello")  # This will raise an IOError
```

Type Errors:

1. Unsupported Operation Between Two Types

In this case, you are trying to use an operation (like `+`) between incompatible types, such as trying to add a string and an integer.

Example:

```
pythonCopy code
name = "John"
age = 25
print(name + age)  # Trying to add a string and an integer.
```

Output:

```
pythonCopy code
TypeError: can only concatenate str (not "int") to str
```

Solution: Convert the integer to a string before adding:

```
pythonCopy code
print(name + str(age))
```

2. Calling a Non-Callable Object

This error occurs when you mistakenly try to call a variable or object as if it were a function, but it's not.

Example:

```
pythonCopy code
name = "John"
print(name())  # Trying to call a string like a function.
```

Output:

```
pythonCopy code
TypeError: 'str' object is not callable
```

Solution: Remove the parentheses `()` if you are not trying to call a function.

```
pythonCopy code
print(name)
```

3. Incorrect List Index Type

In Python, list indices must be integers. If you try to access a list using a non-integer value, it will raise a `TypeError`.

Example:

```
pythonCopy code
fruits = ["apple", "banana", "cherry"]
index = "1"
print(fruits[index])  # Using a string as a list index.
```

Output:

```
pythonCopy code
TypeError: list indices must be integers or slices, not str
```

Solution: Convert the string index to an integer before using it.

```
pythonCopy code
print(fruits[int(index)])
```

4. Iterating Over a Non-Iterable Object

This happens when you try to loop through an object that can't be iterated over (e.g., trying to loop through a number).

Example:

```
pythonCopy code
number = 12345
for digit in number:  # Trying to iterate over an integer.
    print(digit)
```

Output:

```
pythonCopy code
TypeError: 'int' object is not iterable
```

Solution: Convert the number to a string (or list) if you want to iterate through its digits.

```
pythonCopy code
for digit in str(number):
    print(digit)
```

5. Passing an Argument of the Wrong Type to a Function

When a function expects arguments of a specific type, passing the wrong type can raise a `TypeError`.

Example:

```
pythonCopy code
def subtract_numbers(a, b):
    return a - b
```

```
result = subtract_numbers("5", 3)  # Passing a string instead
of a number.
```

Output:

```
pythonCopy code
TypeError: unsupported operand type(s) for -: 'str' and 'int'
```

Solution: Ensure that the arguments passed to the function are of the correct type.

```
pythonCopy code
result = subtract_numbers(int("5"), 3)
```

Value Error :

What is a `ValueError` in Python?

A `ValueError` occurs when a function receives an argument of the correct data type but an inappropriate or invalid value. It can also happen during unpacking of iterable objects when there is a mismatch in the number of elements.

Common Reasons for `ValueError`

1. Invalid Argument

`ValueError` commonly occurs when you pass an invalid argument to a function. For example, the `float()` function can convert a number to a float, but if you pass a string that cannot be converted, Python raises a `ValueError`.

Example:

```
pythonCopy code
x = 25
y = "abc"  # This is an invalid argument for float()

print(float(x))  # Works fine
print(float(y))  # Raises ValueError
```

Output:

```
pythonCopy code
25.0
ValueError: could not convert string to float: 'abc'
```

Solution: Make sure to pass valid arguments to functions.

```
pythonCopy code
try:
    print(float(x))
    print(float(y))
except ValueError:
    print("Error: Cannot convert the string to a float.")
```

2. Incorrect Use of Math Module

Sometimes, `ValueError` is raised when you pass an invalid value to a function in the math module. For instance, the `math.factorial()` function only accepts non-negative integers. Passing a negative number will cause a `ValueError`.

Example:

```
pythonCopy code
import math
```

```
print(math.factorial(-5))  # Raises ValueError because the input is negative
```

Output:

```
pythonCopy code
ValueError: factorial() not defined for negative values
```

Solution: Validate your input before calling the function.

```
pythonCopy code
import math

n = 3
if n >= 0:
    print(math.factorial(n))
else:
    print("Error: Factorial is not defined for negative numbers.")
```

3. Unpacking an Iterable Object

When unpacking iterable objects like lists, tuples, or sets, if the number of variables doesn't match the number of elements in the iterable, a `ValueError` will be raised.

Example:

```
pythonCopy code
my_list = [1, 2, 3]
a, b, c, d = my_list  # Trying to unpack 4 values from a list with only 3 elements
```

Output:

```
pythonCopy code
ValueError: not enough values to unpack (expected 4, got 3)
```

Solution: Ensure the number of variables matches the number of items in the iterable.

```
pythonCopy code
my_list = [1, 2, 3]
a, b, c = my_list  # Correct number of variables

print(a, b, c)
```

index Error:

What Causes an `IndexError` in Python?

An `IndexError` in Python is raised when you attempt to access an index in a sequence (like a list or string) that does not exist. Here are common scenarios that cause this error:

1. **Accessing a Non-Existent Index**: This occurs when you try to access an index that is beyond the valid range of indices for a sequence. Python sequences are zero-indexed, meaning the first element has an index of 0, the second has an index of 1, and so on.

   Example:

   ```
   my_list = [1, 2, 4]
   print(my_list[4])  # This will raise an IndexError because
   ```

```
index 4 is out of range.
```

Output:

2. **Empty List**: Attempting to access any index in an empty list will raise an `IndexError` because there are no elements in the list.

   Example:

   ```
   empty_list = []
   print(empty_list[0])  # This will raise an IndexError beca
   use the list is empty.
   ```

   Output:

3.

   0.

   Example:

   ```
   my_string = "Example"
   print(my_string[-10])  # This will raise an IndexError bec
   ause -10 is out of range for this string.
   ```

   Output:

Modifying a List While Iterating Over It

Situation: If you modify a list (e.g., by deleting elements) while iterating over it, you might inadvertently cause an `IndexError`.

Example:

```
pythonCopy code
my_list = [1, 2, 3, 4]
for i in range(len(my_list)):
    print(my_list[i])
    my_list.pop()  # Reduces the list size

# IndexError: list index out of range
```

Key Errors :

Example 1: Accessing a Non-Existent Key in a Dictionary

This is the most common scenario where a `KeyError` occurs.

```
pythonCopy code
my_dict = {'name': 'Alice', 'age': 25}

# Trying to access a key that does not exist
print(my_dict['gender'])  # KeyError: 'gender'
```

- **Explanation**: The dictionary `my_dict` does not have the key `'gender'`, so trying to access it results in a `KeyError`.

Example 2: Using `.pop()` on a Non-Existent Key

The `pop()` method can also raise a `KeyError` if the specified key is not present in the dictionary.

```
pythonCopy code
my_dict = {'name': 'Alice', 'age': 25}
```

```
# Trying to pop a key that does not exist
my_dict.pop('gender')  # KeyError: 'gender'
```

- **Explanation**: The key `'gender'` does not exist in the dictionary, so calling `pop()` with this key causes a `KeyError`.

Example 3: Using `.get()` or `in` to Avoid KeyError (Safe Access)

A `KeyError` can be avoided using the `.get()` method, which returns `None` or a default value if the key is not found, or by checking if the key exists first.

```
pythonCopy code
my_dict = {'name': 'Alice', 'age': 25}

# Safe access using .get()
gender = my_dict.get('gender', 'Not specified')
print(gender)  # Output: Not specified

# Safe access using 'in'
if 'gender' in my_dict:
    print(my_dict['gender'])
```

4o

Attribute Error :

Understanding `AttributeError` in Python

An `AttributeError` in Python is raised when an invalid attribute reference is made. This error typically occurs when trying to access or call an attribute or method that does not exist for a given object.

Here are some common causes of `AttributeError` with examples:

1. Accessing a Non-Existent Attribute or Method

When you try to call a method or access an attribute that doesn't exist for the data type or object, an `AttributeError` will be raised.

Example 1:

```
# Python program to demonstrate AttributeError

X = 10

# Raises an AttributeError because integers do not have an 'append' method
X.append(6)
```

Output:

```
plaintextCopy code
Traceback (most recent call last):
  File "/path/to/your/script.py", line 5, in <module>
    X.append(6)
AttributeError: 'int' object has no attribute 'append'
```

2. Incorrect Method Name Due to Typographical Error

Python is case-sensitive, so a typo or incorrect case in method names will lead to an `AttributeError`.

Example 2:

```
# Python program to demonstrate AttributeError
# Raises an AttributeError as there is no method 'fst' for st
rings
string = "The famous website is { }".fst("geeksforgeeks")
print(string)
```

Output:

```
plaintextCopy code
Traceback (most recent call last):
  File "/path/to/your/script.py", line 3, in <module>
    string = "The famous website is { }".fst("geeksforgeeks")
AttributeError: 'str' object has no attribute 'fst'
```

3. Accessing a Non-Existent Attribute in a User-Defined Class

An `AttributeError` can occur in user-defined classes if you attempt to access an attribute that hasn't been defined.

Example 3:

```
# Python program to demonstrate AttributeError

class Geeks:
    def __init__(self):
        self.a = 'GeeksforGeeks'

# Driver's code
obj = Geeks()

print(obj.a)

# Raises an AttributeError because 'b' is not defined in the
Geeks class
```

```
print(obj.b)
```

Output:

```
plaintextCopy code
GeeksforGeeks
Traceback (most recent call last):
  File "/path/to/your/script.py", line 17, in <module>
    print(obj.b)
AttributeError: 'Geeks' object has no attribute 'b'
```

How to Fix `AttributeError`

Here are some approaches to handle and prevent `AttributeError`:

1. **Verify Attribute Names**: Ensure that the attribute or method names are correct and exist for the object you are working with.
2. **Check Object Type**: Confirm that the object is of the type you expect and that it supports the attribute or method you are trying to access.
3. **Debug and Fix Typographical Errors**: Ensure that method names and attribute references are spelled correctly and use the correct case.
4. **Verify Indentation**: Check that methods and attributes are correctly defined with proper indentation in classes.

Name Error :

Understanding `NameError` in Python

A `NameError` in Python is raised when an identifier (such as a variable or function) is not defined in the local or global scope. Here are some common causes of `NameError` and how to address them:

1. Misspelled Built-in Functions

A `NameError` can occur if you mistakenly misspell a built-in function or keyword. Python will not recognize the misspelled name and will raise a `NameError`.

Example 1:

```
pythonCopy code
geek = input()
prnt(geek)  # Misspelled 'print' as 'prnt'
```

Output:

```
plaintextCopy code
NameError: name 'prnt' is not defined
```

Fix: Ensure that built-in functions and keywords are spelled correctly.

```
pythonCopy code
geek = input()
print(geek)  # Correct spelling
```

2. Using Undefined Variables

A `NameError` will be raised if you attempt to use a variable that has not been defined.

Example 2:

```
pythonCopy code
geeky = input()
```

```
print(geek)  # 'geek' is not defined
```

Output:

```
plaintextCopy code
NameError: name 'geek' is not defined
```

Fix: Make sure that variables are defined before they are used.

```
pythonCopy code
geek = input()
print(geek)  # Correct variable name
```

3. Defining Variable After Usage

In Python, code is executed from top to bottom. If you use a variable before it is defined, Python will raise a `NameError`.

Example 3:

```
pythonCopy code
print(geek)  # 'geek' is used before it is defined
geek = "GeeksforGeeks"
```

Output:

```
plaintextCopy code
NameError: name 'geek' is not defined
```

Fix: Define variables before you use them.

```
pythonCopy code
geek = "GeeksforGeeks"
print(geek)  # Variable is defined before usage
```

4. Incorrect Usage of Scope

A `NameError` can also occur when a variable is defined within a local scope (e.g., inside a function) and then accessed outside that scope.

Example 4:

```
pythonCopy code
def assign():
    geek = "GeeksforGeeks"  # Local variable

assign()
print(geek)  # 'geek' is not accessible outside the function
```

Output:

```
plaintextCopy code
NameError: name 'geek' is not defined
```

Fix: Ensure that variables are defined in the correct scope where they are intended to be used. If you need to access a variable globally, define it outside of any functions or pass it as a parameter.

```
pythonCopy code
def assign():
    global geek
    geek = "GeeksforGeeks"  # Define 'geek' globally

assign()
```

```
print(geek)  # 'geek' is accessible here
```

Alternative Fix:

```
pythonCopy code
def assign():
    local_geek = "GeeksforGeeks"  # Local variable
    return local_geek

geek = assign()  # Assign the return value to a global variab
le
print(geek)  # 'geek' is accessible here
```

Importance of Handling Runtime Errors

1. **Prevent Program Crashes:**
    - **Reason:** Runtime errors can cause a program to terminate unexpectedly if not handled.
    - **Impact:** Proper error handling prevents abrupt program crashes, ensuring a smoother user experience.
2. **Improve User Experience:**
    - **Reason:** Users might encounter unexpected situations or provide invalid input.
    - **Impact:** Gracefully handling errors allows you to provide informative messages or fallback mechanisms, improving the overall user experience.
3. **Debugging and Maintenance:**

    - **Reason:** Unhandled errors make it difficult to identify the source of problems.
    - **Impact:** Handling errors helps in logging meaningful error messages, making debugging and maintenance easier.
4. **Ensure Data Integrity:**
    - **Reason:** Runtime errors might affect the integrity of data being processed.
    - **Impact:** Error handling can include mechanisms to rollback changes or safeguard data, ensuring that operations remain reliable.
5. **Security:**
    - **Reason:** Unhandled errors might expose sensitive information or lead to security vulnerabilities.
    - **Impact:** Proper handling can prevent unauthorized access or exposure of critical information.

In Python exception handling, the execution flow is governed by the presence and placement of `try`, `except`, `else`, and `finally` blocks. Here's how each block is executed and the rules that determine their execution:

1. **`try` Block**
    - **Execution**: The `try` block is executed first. This is where you place code that may raise an exception.
    - **Purpose**: To monitor and catch potential exceptions.
2. **`except` Block(s)**
    - **Execution**: If an exception occurs in the `try` block, Python looks for an `except` block that matches the type of the exception.
    - **Purpose**: To handle specific exceptions that arise from the `try` block.
3. **`else` Block**

- **Execution**: The `else` block is executed if no exceptions occur in the `try` block.
- **Purpose**: To run code that should only execute if the `try` block is successful and does not raise an exception.

4. `finally` **Block**

    - **Execution**: The `finally` block is executed after the `try`, `except`, and `else` blocks have finished executing, regardless of whether an exception was raised or not.
    - **Purpose**: To perform cleanup actions such as closing files or releasing resources, ensuring that this code runs regardless of whether an exception occurred.

Execution Flow:

1. **No Exception Raised**:
    - `try` → `else` → `finally`
2. **Exception Raised and Handled**:
    - `try` → (Matching `except`) → `finally`
3. **Exception Raised and Not Handled**:
    - `try` → (No matching `except`) → `finally`
    - If the exception is not handled by any `except` block, it will propagate up to higher levels of the call stack.
4. **Exception During** `finally` **Execution**:
    - If an exception occurs in the `finally` block, it will override any previous exceptions. The `finally` block executes last, so its exceptions might terminate the program if not handled elsewhere.

Purpose of each block :

1. `try` Block

Purpose:

The `try` block contains the code that might raise an exception. It's the section where you write code that you want to monitor for errors.

Syntax:

```
pythonCopy code
try:
    # Code that may raise an exception
    ...
```

Usage:

- Place the code that you suspect might cause an error inside the `try` block.
- Use it when you want to handle potential exceptions that might be raised by this code.

2. `except` Block

Purpose:

The `except` block handles exceptions raised by the code in the `try` block. You can specify different types of exceptions to handle different errors differently.

Syntax:

```
pythonCopy code
try:
    # Code that may raise an exception
    ...
except ExceptionType:
    # Code to handle the exception
    ...
```

Examples:

```
pythonCopy code
try:
    x = 1 / 0
except ZeroDivisionError:
    print("Cannot divide by zero")

try:
    data = [1, 2]
    print(data[3])
except IndexError:
    print("Index out of range")
```

2. Handling Multiple Exceptions

Handles multiple types of exceptions with separate `except` blocks:

```
pythonCopy code
try:
    # Code that may raise an exception
    value = int("not_a_number")
except ValueError:
    print("A ValueError occurred.")
except TypeError:
    print("A TypeError occurred.")
```

3. Handling Multiple Exceptions in One Block

Handles multiple types of exceptions in a single `except` block:

```
pythonCopy code
try:
    # Code that may raise an exception
    value = int("not_a_number")
except (ValueError, TypeError):
    print("A ValueError or TypeError occurred.")
```

Sure! Here's a brief explanation for each case where you might need a `try` block:

1. **User Input**

   Reason: Users might enter invalid data.

   Example: Handling errors when converting input to integers or floats.

   ```
   pythonCopy code
   try:
       age = int(input("Enter your age: "))
   except ValueError:
       print("Invalid input. Please enter a number.")
   ```

2. **File Operations**

   Reason: File operations may fail due to file not found, permission issues, etc.

   Example: Handling errors when opening or reading a file.

```
pythonCopy code
try:
    with open('file.txt', 'r') as file:
        content = file.read()
except FileNotFoundError:
    print("File not found.")
except IOError:
    print("Error reading the file.")
```

3. **Network Requests**

   Reason: Network operations may fail due to connectivity issues or server errors.

   Example: Handling errors when making HTTP requests.

```
pythonCopy code
import requests

try:
    response = requests.get('https://example.com')
    response.raise_for_status()
except requests.RequestException as e:
    print(f"Network error: {e}")
```

4. **Database Queries**

   Reason: Database operations may fail due to connection issues or invalid queries.

   Example: Handling errors when connecting to a database or executing a query.

```
pythonCopy code
import sqlite3
```

```
try:
    conn = sqlite3.connect('database.db')
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM table")
except sqlite3.DatabaseError as e:
    print(f"Database error: {e}")
finally:
    conn.close()
```

5. **Type Conversions**

   Reason: Type conversion operations might fail if the data is not in the expected format.

   Example: Handling errors when converting data types.

   ```
   pythonCopy code
   try:
       number = int("not_a_number")
   except ValueError:
       print("Conversion failed. The input is not a valid number.")
   ```

6. **Calculations**

   Reason: Mathematical operations may raise errors, such as division by zero.

   Example: Handling errors during arithmetic calculations.

   ```
   pythonCopy code
   try:
       result = 10 / 0
   except ZeroDivisionError:
   ```

```
    print("Cannot divide by zero.")
```

7. **External API Calls**

   Reason: External API calls may fail due to server issues or invalid responses.

   Example: Handling errors when interacting with APIs.

   ```
   pythonCopy code
   import requests

   try:
       response = requests.get('https://api.example.com/data')
       response.raise_for_status()
   except requests.HTTPError:
       print("API request failed.")
   ```

8. **Resource Allocation**

   Reason: Resource allocation, such as memory or file handles, might fail.

   Example: Handling errors when allocating or deallocating resources.

   ```
   pythonCopy code
   try:
       large_list = [0] * (10**10)
   except MemoryError:
       print("Memory allocation failed.")
   ```

9. **Dynamic Code Execution**

   Reason: Executing dynamic code might lead to syntax or runtime errors.

   Example: Handling errors when executing code generated at runtime.

```
pythonCopy code
try:
    exec("print('Hello'")
except SyntaxError:
    print("Error in dynamically executed code.")
```

10. **Data Parsing**

    Reason: Parsing data from strings or files might fail if the format is incorrect.

    Example: Handling errors when parsing JSON or XML.

```
pythonCopy code
import json

try:
    data = json.loads('{"name": "John", "age": "thirty"}')
except json.JSONDecodeError:
    print("Error decoding JSON.")
```

3. `else` Block

What to Write Inside the `else` Block:

- **Post-Processing:** Perform any actions that depend on the successful execution of the `try` block, such as processing results or further computations.
- **Success Messages:** Output success messages or results that indicate the successful completion of operations.

Purpose:

The `else` block runs if no exceptions were raised in the `try` block. It is useful for code that should run only if the `try` block succeeds.

Syntax:

```
pythonCopy code
try:
    # Code that may raise an exception
    ...
except ExceptionType:
    # Code to handle the exception
    ...
else:
    # Code to run if no exception occurs
    ...
```

Usage:

- Use `else` for code that should execute only if the `try` block is successful.
- It helps in separating the code that handles exceptions from the code that should run when no errors occur.

Example:

```
pythonCopy code
try:
    result = 10 / 2
except ZeroDivisionError:
    print("Cannot divide by zero")
else:
    print(f"Result is {result}")
```

4. `finally` Block

Purpose:

- **Resource Cleanup**
- **Guaranteed Execution**
- **Avoid Resource Leaks**
- **Maintain Code Integrity**
- **Finalization Tasks**

The `finally` block contains code that always runs, regardless of whether an exception occurred or not. It is often used for cleanup actions.

Syntax:

```
pythonCopy code
try:
    # Code that may raise an exception
    ...
except ExceptionType:
    # Code to handle the exception
    ...
else:
    # Code to run if no exception occurs
    ...
finally:
    # Code that always runs
    ...
```

Usage:

- Use `finally` for cleanup actions that must occur whether an exception is raised or not, such as closing files or releasing resources.

- It is executed no matter what happens in the `try`, `except`, or `else` blocks.

Example:

```
pythonCopy code
try:
    file = open("example.txt", "r")
    data = file.read()
except FileNotFoundError:
    print("File not found")
else:
    print("File read successfully")
finally:
    file.close()  # Ensure the file is closed whether or not
an error occurred
```

Summary of Usage:

- `try`: Place code that might raise exceptions here.
- `except`: Handle specific exceptions raised in the `try` block.
- `else`: Run code if no exceptions were raised in the `try` block.
- `finally`: Run code regardless of whether an exception was raised or not, typically for cleanup actions.

block syntax :

Example 1: Basic Nesting with `try`, `except`, `else`, and `finally`

```
pythonCopy code
try:
    print("Outer try block")
    try:
        x = 10 / 2  # Inner try block
    except ZeroDivisionError:
        print("Inner except block: Division by zero")
    else:
        print("Inner else block: No exception in inner try")
finally:
    print("Outer finally block: Always executes")
```

Example 1: Nested `try` Blocks

```
pythonCopy code
try:
    print("Outer try block")
    try:
        print("Inner try block")
        x = 1 / 0  # This will raise a ZeroDivisionError
    except ZeroDivisionError:
        print("Inner except block")
    else:
        print("Inner else block")
    finally:
        print("Inner finally block")
except Exception as e:
    print(f"Outer except block: {e}")
finally:
    print("Outer finally block")
```

Raising Exceptions in Python

In Python, you can explicitly raise exceptions using the `raise` keyword. This is useful when you want to indicate that an error has occurred in a specific part of your program. Raising exceptions allows you to control the flow of your program and provide informative error messages.

How to Raise an Exception

Syntax:

```
raise ExceptionType("Error message")
```

Example 1: Raising a Built-in Exception

You can raise built-in exceptions like `ValueError`, `TypeError`, etc., using the `raise` statement.

Example:

```
def divide(a, b):
    if b == 0:
        raise ZeroDivisionError("Division by zero is not allowed!")
    return a / b

try:
    result = divide(10, 0)
except ZeroDivisionError as e:
    print(e)
```

Explanation:

In this example, the `divide` function raises a `ZeroDivisionError` if the second parameter is zero. The `try` block catches this exception and prints an error message.

Example 2: Raising a Built-in Exception with Custom Message

You can also raise built-in exceptions with a custom error message.

Example:

```
def get_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative!")
    return age

try:
    age = get_age(-5)
except ValueError as e:
    print(e)
```

Explanation:

In this example, the
`get_age` function raises a `ValueError` if the age provided is negative. The `try` block catches this exception and prints the error message.

Exception Chaining in Python allows you to handle one exception and then raise a new exception while preserving the original traceback. This helps in maintaining the context of the original error while handling or raising a new exception.

Why Use Exception Chaining

1. **Preserve Context**: It retains information about the original exception, making debugging easier.
2. **Improve Error Handling**: Allows you to handle exceptions at different levels and raise more specific exceptions.

Syntax for Exception Chaining

You can use the `raise ... from ...` syntax to chain exceptions. This syntax specifies that the new exception was caused by a previous exception.

Example

Basic Example of Exception Chaining

Here's an example where an original exception is handled and a new, more specific exception is raised:

```
pythonCopy code
def divide_numbers(a, b):
    try:
        return a / b
    except ZeroDivisionError as e:
        raise ValueError("Invalid value for division") from e

try:
    result = divide_numbers(10, 0)
except ValueError as e:
    print(f"Handled ValueError: {e}")
    print(f"Original exception: {e.__cause__}")
```

Custom Exception Example

Custom Exception Definition:

```
class MyCustomError(Exception):
    def __init__(self, message):
        self.message = message
        super().__init__(self.message)
```

Using the Custom Exception:

```
def check_value(x):
    if x < 0:
        raise MyCustomError("Negative value is not allowed!")
    return x

try:
    result = check_value(-1)
except MyCustomError as e:
    print(f"Error: {e}")
```

Explanation

- **Custom Exception Definition:** `MyCustomError` inherits from the base `Exception` class. It takes a `message` parameter to describe the error.
- **Using the Custom Exception:** In the `check_value` function, if the input `x` is negative, it raises `MyCustomError` with an appropriate message. The `try` block catches this custom exception and prints the error message.

This example demonstrates how to create a simple custom exception and use it to handle specific errors in your program.

4o mini

Custom Exception Example 2

Custom Exception Definition:

```
pythonCopy code
class AgeTooYoungError(Exception):
    def __init__(self, age):
        self.age = age
        self.message = f"Age {self.age} is too young. Must be
18 or older."
```

```
        super().__init__(self.message)
```

Using the Custom Exception:

```
pythonCopy code
def check_age(age):
    if age < 18:
        raise AgeTooYoungError(age)
    return "Age is valid."

try:
    result = check_age(15)
except AgeTooYoungError as e:
    print(f"Error: {e}")
else:
    print(result)
```