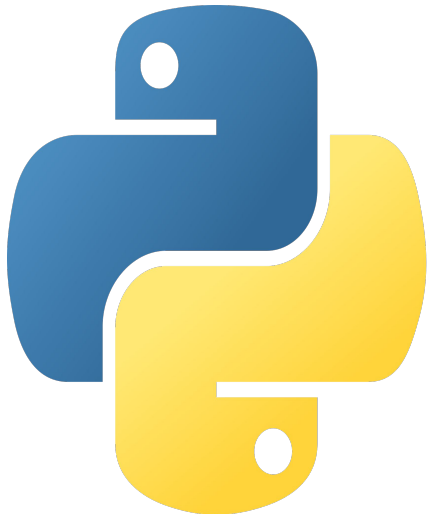

Exception Handling

Syntax Error

- Syntax errors are detected when we have not followed the rules of the particular programming language while writing a program. These errors are also known as parsing errors. On encountering a syntax error, the interpreter does not execute the program unless we rectify the errors, save and rerun the program.

Exception

- Exceptions are mistakes discovered during execution. Exceptions are triggered whenever there is a mistake in a program.
- if a statement or expression is syntactically correct, there might arise an error during its execution.
- For example, trying to open a file that does not exist, division by zero and so on. Such types of errors might disrupt the normal execution of the program and are called exceptions.

The EAFP Principle in Python

- Python's approach to exception handling is based on the EAFP principle, which stands for **"Easier to Ask for Forgiveness than Permission"** This approach encourages writing code that assumes something will work and handles exceptions if it doesn't, rather than checking for conditions beforehand.
- EAFP Approach

```
try:
    result = 10 / 0 # Attempt a division by zero
except ZeroDivisionError as e:
    print(f"An error occurred: {e}")
```

- In this code, we attempt to divide by zero without checking if the denominator is zero. If a ZeroDivisionError occurs, we catch it and handle it gracefully.
- The alternative, non-Pythonic approach (**LBYL - "Look Before You Leap"**), would involve checking for zero before performing the division. Python's EAFP approach is often preferred because it leads to more concise and readable code.

Built - in Exception

- Commonly occurring exceptions are usually defined in the compiler/interpreter. These are called built-in exceptions.
- Python’s standard library is an extensive collection of built-in exceptions that deals with the commonly occurring errors (exceptions) by providing the standardized solutions for such errors.

Built - in Exception

- ZeroDivisionError: Raised when you try to divide by zero.
- TypeError: Raised when an operation is performed on an inappropriate data type.
- ValueError: Raised when a function receives an argument of the correct data type but with an inappropriate value.
- IndexError: Raised when trying to access an index that is out of range for a sequence (e.g., a list or a string).
- KeyError: Raised when trying to access a non-existent dictionary key.
- FileNotFoundError: Raised when trying to open or manipulate a file that doesn't exist.
- ImportError: Raised when an imported module cannot be found.

Built - in Exception

- ZeroDivisionError: Raised when you try to divide by zero.

```
try:
    result = 10 / 0 # Attempt a division by zero
except ZeroDivisionError as e:
    print(f"Error: {e}")
```

Built - in Exception

- TypeError: Raised when an operation is performed on an inappropriate data type.

```
try:
    result = "5" + 3 # Attempt to concatenate a string and an integer
except TypeError as e:
    print(f"Error: {e}")
```

Built - in Exception

- ValueError: Raised when a function receives an argument of the correct data type but with an inappropriate value.

```
try:
    num = int("ten") # Attempt to convert an invalid string to an integer
except ValueError as e:
    print(f"Error: {e}")
```

Built - in Exception

- IndexError: Raised when trying to access an index that is out of range for a sequence (e.g., a list or a string).

```
my_list = [1, 2, 3]
try:
    element = my_list[3] # Access an index that is out of range
except IndexError as e:
    print(f"Error: {e}")
```

Built - in Exception

- KeyError: Raised when trying to access a non-existent dictionary key.

```
my_list = [1, 2, 3]
try:
    element = my_list[3] # Access an index that is out of range
except IndexError as e:
    print(f"Error: {e}")
```

Built - in Exception

- FileNotFoundError: Raised when trying to open or manipulate a file that doesn't exist.

```
try:
    with open("nonexistent.txt", "r") as file:
        content = file.read()
except FileNotFoundError as e:
    print(f"Error: {e}")
```

Built - in Exception

- ImportError: Raised when an imported module cannot be found.

```
try:
    import non_existent_module # Attempt to import a module that doesn't exist
except ImportError as e:
    print(f"Error: {e}")
```

Built - in Exception

- KeyboardInterruption – Called on pressing Ctrl+C keys
- SyntaxError: Raised for syntax errors.
- IOError: It is raised when the file specified in a program statement cannot be opened.
- IndentationError: It is raised due to incorrect indentation in the program code.
- OverFlowError: It is raised when the result of a calculation exceeds the maximum limit for numeric data type.
- FileExistsError: This is Python's built-in Exception thrown when creating a file or directory already in the file system.
- AttributeError: occurs when an object does not have an attribute being referenced, such as calling a method that does not exist on an object.

Exception Handling

- An exception is a Python object that represents an error. When an error occurs during the execution of a program, an exception is said to have been raised. Such an exception needs to be handled by the programmer so that the program does not terminate abnormally. Therefore, while designing a program, a programmer may anticipate such erroneous situations that may arise during its execution and can address them by including appropriate code to handle that exception.
- Exception handling in Python is a process of resolving errors that occur in a program. This involves catching exceptions, understanding what caused them, and then responding accordingly.

Exception Handling

```
try:
   # code that may cause an exception
except ExceptionType as e:
   # code to handle the exception
else:
   # code to execute if no exceptions were raised
finally:
   # code that will always be executed, regardless of exceptions
```

Exception Handling - multiple except block

You can have multiple except blocks to handle different types of exceptions. Python will execute the first matching except block it encounters.

```
try:
    user_input = int(input("Enter a number: "))
    result = 10 / user_input
except ZeroDivisionError as e:
    print("Error: Cannot divide by zero")
except ValueError as e:
    print("Error: Invalid input. Please enter a valid number.")
```

Exception Handling - else block

Python allows you to use the else and finally clauses in conjunction with try and except blocks.

The else block is executed when no exception occurs within the try block. It is often used for code that should run only if no exceptions were raised.

```
try:
    file = open("example.txt", "r")
except FileNotFoundError as e:
    print("Error: File not found")
else:
    content = file.read()
    print("File content:", content)
```

Exception Handling - finally block

The finally block is always executed, whether an exception was raised or not. It is commonly used for cleanup code, such as closing files or releasing resources.

```
try:
    file = open("example.txt", "r")
except FileNotFoundError as e:
    print("Error: File not found")
else:
    content = file.read()
    print("File content:", content)
finally:
    file.close() # Always close the file, even if an exception occurred
```

Exception Handling - raise statement

In addition to handling exceptions raised by Python or third-party libraries, you can also raise your own exceptions when specific conditions are met. This can be useful for signaling errors or exceptional situations within your code.

raise SomeException("Error message")

SomeException is the type of exception you want to raise. It can be any built-in or custom exception type.

"Error message" is an optional message that provides additional information about the exception.

When to Raise Exceptions

You should raise exceptions in your code when you encounter situations where the normal flow of your program cannot proceed due to an error or exceptional condition. Raising custom exceptions with descriptive error messages can help make your code more readable and maintainable.

Best practices for Exception Handling

- Keep try blocks small and focused to properly handle exceptions
- Catch specific exceptions instead of generic Exception class to differentiate errors
- Print custom error messages from except blocks upon failures
- Use finally clause to execute sections of cleanup code reliably
- Define custom exception classes to match application scenarios
- Use try-except blocks only where needed.
- Don’t wrap your entire code in a massive try-except block; limit it to potential error-prone sections.
- Don’t overuse try-except blocks in business logic to avoid hiding real issues
- Avoid using except: without specifying the exception type, as it can catch unintended errors.
- Use logging to record exceptions for later analysis.

Best practices for Exception Handling

Error Messages and Traceback

- When an exception occurs, Python provides valuable information about the error. You can access this information through the exception object. It's good practice to include meaningful error messages in your exceptions. For example:

```
def divide(x, y):
    if y == 0:
        raise ValueError("Division by zero is not allowed")
    return x / y

try:
    result = divide(10, 0)
except ValueError as e:
    print(f"Error: {e}")
```

Best practices for Exception Handling

Logging Exceptions for Debugging

- Logging is an essential part of debugging and monitoring the behavior of your applications. Python's logging module allows you to log exceptions and other information during program execution.

```
import logging
logging.basicConfig(filename='app.log', level=logging.ERROR)
def some_function():
    try:
        # Code that might raise an exception
    except Exception as e:
        logging.error(f"An error occurred: {e}", exc_info=True)

some_function()
```

Exception Handling - Example

1. Handling invalid user input:

```
while True:
    try:
        number = int(input("Please enter a number: "))
    except ValueError:
        print("You did not enter a valid number!")
        continue
    else:
        print(f"The square of your number is {number**2}")
        break
```

Exception Handling - Example

2. Catching different exception types:

```
import sys


try:
   f = open("data.txt")
   x = 10/0
except FileNotFoundError:
   print("The file was not found!")
except ZeroDivisionError:
   print("Cannot divide by 0!")
except:
   print(f"Unknown error: {sys.exc_info()[1]}")
```

Exception Handling - Example

3. Defining custom exception class:

```
class NegativeNumberError(Exception):
  pass

def calculate_square(num):
  if num < 0:
    raise NegativeNumberError("Negative numbers unacceptable")
  return num**2

num = -6
try:
  result = calculate_square(num)
except NegativeNumberError as e:
  print(e)
```