Programming in Python (CSM210-2C)

Unit 3

Exception Handling in python

What is Exception ?

- Exception handling in Python is a crucial concept that allows a programmer to manage and respond to unexpected events or errors that may occur during the execution of a program.
- It helps ensure that a program can handle errors gracefully without crashing.
- Whenever an error occurs that makes Python unsure what to do next, it creates an exception object. If you write code that handles the exception, the program will continue running. If you don't handle the exception, the program will halt and show a traceback, which includes a report of the exception that was raised.

Basic Syntax

Python uses the **try...except** block to handle exceptions.

The basic syntax is as follows

```
try:
    # Code that may raise an exception
except ExceptionType:
    # Code that runs if the exception occurs
```

Example

```
try:
    num = int(input("Enter a number: "))
    print(f"The number you entered is {num}")
except ValueError:
```

```
print("Invalid input. Please enter a valid integer.")
```

Common Exception Types

1. NameError:

A NameError exception in Python occurs when you try to use a variable or function name that has not been defined or is not accessible in the current scope. Essentially, it means that the interpreter has encountered a name it doesn't recognize.

Common Causes of NameError

1. Typographical Errors: Misspelling a variable or function name.
2. Using Undefined Variables: Trying to access a variable that hasn't been assigned a value.
3. Scope Issues: Referencing a variable outside its defined scope.
4. Incorrect Import Statements: Not importing a module or function before using it.

Typographical Error

```
# Incorrect

value = 10
print(vale)  # NameError: name 'vale' is not defined
# Correct

value = 10
print(value)  # Output: 10
```

Using Undefined Variables

```
print(num)          # NameError: name 'num' is not defined
num = 5
print(num)  # Output: 5
```

Scope Issues

```
def func():
    x = 10

func()
print(x)  # NameError: name 'x' is not defined
```

Incorrect Import Statements

```
# Attempting to use math module without importing
result = math.sqrt(16)            # NameError: name 'math' is not defined

# Correct
import math
result = math.sqrt(16)
print(result)  # Output: 4.0
```

2. **IndexError:**

An IndexError in Python occurs when you try to access an index that is out of the range of a sequence, such as a list, tuple, or string.

This typically happens when you refer to an index that doesn't exist within the given sequence

Common Causes of IndexError

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

```
my_list = [1, 2, 3]
print(my_list[-4])  # IndexError: list index out of range

my_list = [1, 2, 3]
for i in range(len(my_list) + 1):  # This loop will go out of range
    print(my_list[i])  # IndexError: list index out of range on the last iteration
```

3. **TypeError:**

A TypeError in Python occurs when an operation or function is applied to an object of an inappropriate type.

This typically happens when you try to perform an action that is not supported for the data type you are working with.

Common Causes of TypeError

Unsupported Operation Between Different Types:

```
result = 'string' + 10  # TypeError: can only concatenate str (not "int") to str
```

Calling a Non-Callable Object:

```
my_list = [1, 2, 3]
my_list()  # TypeError: 'list' object is not callable
```

Incorrect Number of Arguments Passed to a Function:

```
def add(a, b):
    return a + b
add(1)  # TypeError: add() missing 1 required positional argument: 'b'
```

Using an Inappropriate Method or Attribute:

```
num = 5
num.append(10)  # TypeError: 'int' object has no attribute 'append'
```

4. **ValueError:**
   - A ValueError in Python occurs when a function receives an argument of the correct type but an inappropriate value.
   - This typically happens when the provided value does not conform to the expected range or format. Here's a detailed explanation with examples:

   Common Causes of ValueError

   Invalid Type Conversion:

   num = int("abc")

   # ValueError: invalid literal for int() with base 10: 'abc'
   Incorrect Number of Elements for Unpacking:

   a, b = [1, 2, 3]
   # ValueError: too many values to unpack (expected 2)
   Invalid Values in Functions:

   import math
   result = math.sqrt(-1)
   # ValueError: math domain error

5. **ImportError:**
   - An ImportError in Python occurs when an import statement fails to find the module definition or the specific name within a module.
   - This exception typically arises when there is an issue with the import paths or the availability of the module.

Common Causes of ImportError

Module Not Found: The module you're trying to import doesn't exist or isn't installed.

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

Importing from a Module that Doesn't Contain the Specified Name:

```
from math import non_existent_function

# ImportError: cannot import name 'non_existent_function'
```

Circular Imports: Two or more modules depend on each other.

```
# Module a.py
import b  # This will lead to a circular import if b.py imports a.py

# Module b.py
import a
```

6. ZeroDivisionError:

- A ZeroDivisionError in Python occurs when you attempt to divide a number by zero.
- This is a specific type of ArithmeticError and is raised to indicate that the mathematical operation of division by zero is undefined and cannot be performed.

Common Causes of ZeroDivisionError

Direct Division by Zero:

```
result = 10 / 0  # ZeroDivisionError: division by zero
```

Modulo by Zero:

```
result = 10 % 0  # ZeroDivisionError: integer division or modulo by zero
```

Division or Modulo in Expressions:

```
a = 10
```

```
b = 0

result = a / b  # ZeroDivisionError: division by zero
```

7. **KeyError:**
   - A KeyError in Python occurs when you try to access a dictionary key that does not exist.
   - This exception is raised to indicate that the specified key cannot be found in the dictionary. Here's a detailed explanation with examples:

Common Causes of KeyError

Accessing a Non-Existent Key:

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

print(my_dict['address'])  # KeyError: 'address'
```

Using Incorrect Key Names:

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

print(my_dict['Name'])  # KeyError: 'Name' (case-sensitive)
```

8. **FileNotFoundError:**
   - A FileNotFoundError in Python occurs when you try to open a file that does not exist.
   - This exception is raised to indicate that the specified file or directory could not be found. Here's a detailed explanation with examples:

Common Causes of FileNotFoundError

Incorrect File Path:

open('non_existent_file.txt', 'r') # FileNotFoundError: [Errno 2] No such file or directory: 'non_existent_file.txt'

File Not Present in the Specified Directory:

open('/path/to/non_existent_file.txt', 'r') # FileNotFoundError: [Errno 2] No such file or directory: '/path/to/non_existent_file.txt'

Errors in Python

In Python programming, errors can be broadly categorized into three types:

- Compile-time errors,
- Runtime errors,
- Logical errors.

Each type of error affects the program in different ways. Let's explore these types in detail:

1. Compile-time Errors

Compile-time errors occur when the code is being compiled. In Python, since it's an interpreted language, these errors are typically identified during the parsing or syntax analysis phase before execution begins.

Examples of Compile-time Errors:

- **Syntax Errors**: These occur when the code violates the syntax rules of the language.

  # Missing colon

  if True

    print("Hello") # SyntaxError: invalid syntax

- **Indentation Errors**: These occur when the code is not properly indented.

```
def my_function():
print("Hello")  # IndentationError: expected an indented block
```

2. Runtime Errors

Runtime errors occur during the execution of the program. These errors cause the program to terminate unexpectedly.

Examples of Runtime Errors:

- **ZeroDivisionError**: Occurs when attempting to divide by zero.

  ```
  result = 10 / 0  # ZeroDivisionError: division by zero
  ```

- **TypeError**: Occurs when an operation is applied to an object of inappropriate type.

  ```
  result = 'string' + 10  # TypeError: can only concatenate str (not "int") to str
  ```

- **FileNotFoundError**: Occurs when trying to open a file that does not exist.

  ```
  with open('non_existent_file.txt', 'r') as file:
      content = file.read()  # FileNotFoundError: [Errno 2] No such file or directory: 'non_existent_file.txt'
  ```

3. Logical Errors

Logical errors occur when the code does not produce the expected output. These errors do not cause the program to crash but result in incorrect behavior or output. Logical errors are often the most challenging to debug.

Examples of Logical Errors:

- **Incorrect Algorithm Implementation**: Using the wrong formula or approach.

```
def calculate_area(radius):
    return 2 * 3.14 * radius  # Incorrect formula for area of a circle (should be πr²)
print(calculate_area(5))  # Incorrect output
```

- **Off-by-One Errors**: Looping one too many or one too few times.

```
total = 0
for i in range(1, 10):  # Should be range(1, 11) to include 10
    total += i
print(total)  # Incorrect output
```

Handling and Preventing Errors

Compile-time Errors:

- **Syntax Checking**: Use Integrated Development Environments (IDEs) or code editors with syntax highlighting and error checking.
- **Code Reviews**: Conduct thorough code reviews to catch syntax and indentation errors early.

Runtime Errors:

- **Error Handling**: Use try...except blocks to handle potential runtime errors gracefully.

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero.")
```

Logical Errors:

- **Testing**: Write unit tests and perform thorough testing to catch logical errors.
- **Debugging**: Use debugging tools and techniques to trace the source of logical errors.
- **Code Reviews**: Peer reviews can help identify logical flaws in the code.

How to use exception handling in Python:

1. **The** try **block**: This block contains the code that might raise an exception.
2. **The** except **block**: This block handles the exception that is raised in the try block.
3. **The** else **block**: This block runs if no exception occurs in the try block.
4. **The** finally **block**: This block runs whether an exception occurs or not, typically used for cleanup actions.

```
try:        # Code that might raise an exception
    number = int(input("Enter a number: "))
    result = 10 / number
    print("Result:", result)
except ZeroDivisionError:
     print("Error: Division by zero is not allowed.")
except ValueError:
    # Handling the value error exception
    print("Error: Invalid input. Please enter a valid number.")
else:
    # Executed if no exception occurs
    print("No exceptions occurred.")
finally:
    # Always executed
    print("Execution completed.")
```

Key Points:

- The try block contains code that could potentially throw exceptions.
- The except block(s) handle specific exceptions.
- The else block executes if the try block succeeds without exceptions.
- The finally block executes no matter what, useful for resource cleanup.

Example

```
try:
    number = int(input("Enter a number: "))
    result = 10 / number
except (ZeroDivisionError, ValueError) as e:
    print("An error occurred:", e)
else:
    print("Result:", result)
finally:
    print("Execution completed.")
```

Example of nested Exception

```
try:
    number = int(input("Enter a number: "))
    try:
        result = 10 / number
```

```
        except ZeroDivisionError:
            print("Inner error: Division by zero is not allowed.")
        else:
            print("Inner result:", result)
    except ValueError:
        print("Outer error: Invalid input. Please enter a valid number.")
    finally:
        print("Execution completed.")
```

Basics of Raising Exceptions

Raising exceptions in Python is a way to signal that an error or exceptional condition has occurred in your code.

Exceptions can be raised using the raise keyword followed by the exception you want to raise.

Examples:

raise Exception("This is a generic exception")

Raising specific exceptions: Python provides several built-in exceptions that can be used to signify specific error conditions.

raise ValueError("This is a value error")

raise TypeError("This is a type error")

Custom Exceptions

You can define your own exceptions by creating a new class that inherits from the built-in Exception class or any of its subclasses.

Defining a custom exception:

```
class CustomError(Exception):
    pass


# Raising the custom exception
raise CustomError("This is a custom error")
```

```
try:
    raise ValueError("An error occurred")
except ValueError as e:
    print(f"Caught an error: {e}")
```

```
try:
    raise ValueError("An error occurred")
except ValueError as e:
    print(f"Caught an error: {e}")
finally:
    print("This runs no matter what")
```