Python Programming

1. EXCEPTIONS

Session 5

Python provides a very important feature to handle any unexpected error in your Python programs and to add debugging capabilities in them

- Exception Handling

> An exception is an unexpected event that occurs during program execution

```
#Syntax:

try:
    statement
except Exception as varname:
    statement

"""
Some specific exceptions (Lengthy but time-saving )-

ArithmeticError - Raised when an error occurs in numeric calculations

AssertionError  - Raised when an assert statement fails

AttributeError  - Raised when attribute reference or assignment fails

Exception - Base class for all exceptions

EOFError -  Raised when the input() method hits an "end of file" condition (EOF)

FloatingPointError -    Raised when a floating point calculation fails

GeneratorExit - Raised when a generator is closed (with the close() method)

ImportError -   Raised when an imported module does not exist

IndentationError -  Raised when indendation is not correct

IndexError -    Raised when an index of a sequence does not exist

KeyError -  Raised when a key does not exist in a dictionary

KeyboardInterrupt - Raised when the user presses Ctrl+c, Ctrl+z or Delete
```

```
LookupError -   Raised when errors raised cant be found

MemoryError -   Raised when a program runs out of memory

NameError - Raised when a variable does not exist

NotImplementedError -   Raised when an abstract method requires an inherited class
to override the method

OSError -   Raised when a system related operation causes an error

OverflowError - Raised when the result of a numeric calculation is too large

ReferenceError -    Raised when a weak reference object does not exist

RuntimeError -  Raised when an error occurs that do not belong to any specific
expections

StopIteration - Raised when the next() method of an iterator has no further values

SyntaxError -   Raised when a syntax error occurs

TabError -  Raised when indentation consists of tabs or spaces

SystemError -   Raised when a system error occurs

SystemExit -    Raised when the sys.exit() function is called

TypeError - Raised when two different types are combined

UnboundLocalError - Raised when a local variable is referenced before assignment

UnicodeError -  Raised when a unicode problem occurs

UnicodeEncodeError -    Raised when a unicode encoding problem occurs

UnicodeDecodeError -    Raised when a unicode decoding problem occurs

UnicodeTranslateError - Raised when a unicode translation problem occurs

ValueError -    Raised when there is a wrong value in a specified data type

ZeroDivisionError - Raised when the second operator in a division is zero

"""
```

The most common types of errors you'll encounter in Python are 1. syntax errors, 2. runtime errors, 3. logical errors, 4. name errors, 5. type errors, 6. index errors, and 7. attribute errors. Let's go through each with examples.

1. Syntax Errors

A syntax error occurs in Python when the interpreter is unable to parse the code due to the code violating Python language rules, such as inappropriate indentation, erroneous keyword usage, or incorrect operator use.

```
x = 10
if x == 10
print("x is 10")
```

Output

```
File "c:\Users\name\OneDrive\Desktop\demo.py", line 2
    If x == 10
              ^
SyntaxError: expected ':'
```

Solution

The `SyntaxError` occurs on `line 2` because the *if* statement is missing a colon `:` at the end of the line. The correct code should be:

```
x = 10
if x == 10:
   print("x is 10")
```

This code will execute correctly and print `x is 10` to the console because the `SyntaxError` has been fixed.

---

2. Runtime Errors

In Python, a runtime error occurs when the program is executing and encounters an unexpected condition that prevents it from continuing.

TYPES OF RUNTIME ERRORS

1.NameError

A `NameError` in Python is raised when the interpreter encounters a variable or function name that it cannot find in the current scope. This can happen for a variety of reasons, such as misspelling a variable or function name, using a variable or function before it is defined, or referencing a variable or function that is outside the current scope. Here's an example of a `NameError` in Python:

```
def calculate_sum(a, b):
    total = a + b
    return total

x = 5
y = 10
z = calculate_sum(x, w)
print(z)
```

Output

```
 Traceback (most recent call last):
    File "c:\Users\name\OneDrive\Desktop\demo.py", line 7, in <module>
        Z = calculate_sum(x, w)
                          ^
NameError: name 'w' is not defined
```

Solution:

This error message indicates that the interpreter could not find the variable `w` in the current scope. To fix this error, we need to correct the misspelling of the variable name to `y`

```
def calculate_sum(a, b):
    total = a + b
    return total

x = 5
y = 10
z = calculate_sum(x, y)
print(z)
```

Now the code will run without any errors, and the output will be `15`

2. TypeError

In Python, a `TypeError` is raised when an operation or function is applied to an object of an inappropriate type. This can happen when trying to perform arithmetic or logical operations on incompatible data types or when passing arguments of the wrong type to a function.

```
x = "10"
y = 5
Z = x + y
print(z)
```

Output

```
Traceback (most recent call last):
      File "c:\Users\name\OneDrive\Desktop\demo.py", line 3, in <module>
        Z = x + y
            ~~^~~
TypeError: can only concatenate str (not "int") to str
```

Solution:

This error message indicates that we cannot `concatenate` a `string` and an `integer` using the `+` operator. To fix this error, we need to convert the integer `y` to a string before concatenating it with `x` , like so:

```
x = "10"
y = 5
Z = x + str(y)
print(z)
```

Here, we have used the `str()` method to convert our integer to a string. Now the code will run without any errors, and the output will be `105` , which is the result of concatenating `x` and `y` as strings.

3. IndexError

An `IndexError` is raised in Python when we try to access an index of a sequence (such as a `string, list,` or `tuple`) that is out of range. This can happen when we try to access an element that doesn't exist in the sequence or when we try to access an element at an index that is greater than or equal to the length of the sequence. Here's an example of an `IndexError` in Python:

```
my_list = [100, 200, 300, 400, 500]
print(my_list[5])
```

Output

```
 Traceback (most recent call last):
      File "c:\Users\name\OneDrive\Desktop\demo.py", line 2, in <module>
        print(my_list[p
            ~~^~~
IndexError: list index out of range
```

Solution:

This error message indicates that we are trying to access an index that is outside the range of valid indices for the list. To fix this error, we need to make sure that we are only accessing valid indices of the list, like so:

```
my_list = [100, 200, 300, 400, 500]
print(my_list[4])
```

Now the code will run without any errors, and the output will be `500`, which is the element at `index 4` of the list.

4. AttributeError

In Python, an `AttributeError` is raised when you try to access an attribute or method of an object that does not exist or is not defined for that object. This can happen when you misspell the name of an attribute or method or when you try to access an attribute or method that is not defined for the type of object you are working with. Here's an example of an `AttributeError` in Python:

```
my_string = "Hello, world!"
my_string.reverse()
```

Output

```
Traceback (most recent call last):
    File "c:\Users\name\OneDrive\Desktop\demo.py", line 2, in <module>
        my_string.reverse()
        ^^^^^^^^^^^^^^^^^
AttributeError: 'str' object has no attribute 'reverse'
```

Solution

To fix this error, we need to use a different method or attribute that is defined for strings, like `[::-1]` to reverse the string:

```
my_string = "Hello, world!"
reversed_string = my_string[::-1]
print(reversed_string)
```

Now the code will run without any errors, and the output will be `!dlrow ,olleH`, which is the reversed string of `my_string`.

3. Logical Errors

A logical error occurs in Python when the code runs without any syntax or runtime errors but produces incorrect results due to flawed logic in the code. These types of errors are often caused by incorrect assumptions, an incomplete understanding of the problem, or the incorrect use of algorithms or formulas.

Unlike syntax or runtime errors, logical errors can be challenging to detect and fix because the code runs without producing any error messages. The results may seem correct, but the code might produce incorrect output in certain situations. Here is an example of a logical error in Python:

```
def calculate_factorial(n):
    result = 1
    for i in range(1, n):
          result = result * i
    return result

print(calculate_factorial(5))
```

Output

```
24
```

In this example, the function `calculate_factorial()` is designed to calculate the factorial of a given number `n`. So when we run it, let's say for `n = 5`, it runs without any problem but gives an output of `24` instead of `120`. The reason is a logical error in the code that causes it to produce incorrect results. The `for` loop is iterating from `1` to `n-1` instead of from `1` to `n`, causing the issue. This means that the `factorial` is being calculated incorrectly, resulting in an incorrect output.

Solution

To fix this logical error, we need to change the range of the `for` loop to include the number `n` itself. Here's the corrected code:

```
def calculate_factorial(n):
    result = 1
    for i in range(1, n+1):
         result = result * i
    return result

print(calculate_factorial(5))
```

Output

```
120
```

- **We handle these built-in and user-defined exceptions in Python using `try`, `except` and `finally` statements.**

2. Exception Handling

Python try...except Block

The `try...except` block is used to handle exceptions in Python. Here's the syntax of `try...except` block:

```
try:
    # code that may cause exception
except:
    # code to run when exception occurs
```

Here, we have placed the code that might generate an exception inside the `try` block. Every `try` block is followed by an `except` block.

When an exception occurs, it is caught by the `except` block. The `except` block cannot be used without the try block.

Example: Exception Handling Using try...except

```
try:
    numerator = 10
    denominator = 0

    result = numerator/denominator

    print(result)
except:
    print("Error: Denominator cannot be 0.")

# Output: Error: Denominator cannot be 0.
```

In the example, we are trying to divide a number by **0**. Here, this code generates an exception.

To handle the exception, we have put the code, `result = numerator/denominator` inside the `try` block. Now when an exception occurs, the rest of the code inside the `try` block is skipped.

The `except` block catches the exception and statements inside the `except` block are executed.

If none of the statements in the `try` block generates an exception, the `except` block is skipped.

Catching Specific Exceptions in Python

For each `try` block, there can be zero or more `except` blocks. Multiple `except` blocks allow us to handle each exception differently.

The argument type of each `except` block indicates the type of exception that can be handled by it. For example,

```
try:

    even_numbers = [2,4,6,8]
    print(even_numbers[5])

except ZeroDivisionError:
    print("Denominator cannot be 0.")

except IndexError:
    print("Index Out of Bound.")

# Output: Index Out of Bound
```

In this example, we have created a list named even_numbers.

Since the list index starts from **0**, the last element of the list is at index **3**. Notice the statement,

```
print(even_numbers[5])
```

Here, we are trying to access a value to the index **5**. Hence, `IndexError` exception occurs.

When the `IndexError` exception occurs in the `try` block,

- The `ZeroDivisionError` exception is skipped.
- The set of code inside the `IndexError` exception is executed.

Python try with else clause

In some situations, we might want to run a certain block of code if the code block inside `try` runs without any errors.

For these cases, you can use the optional `else` keyword with the `try` statement.

Let's look at an example:

```
# program to print the reciprocal of even numbers

try:
    num = int(input("Enter a number: "))
    assert num % 2 == 0
except:
    print("Not an even number!")
else:
    reciprocal = 1/num
    print(reciprocal)
```

Output

If we pass an odd number:

```
Enter a number: 1
Not an even number!
```

If we pass an even number, the reciprocal is computed and displayed.

```
Enter a number: 4
0.25
```

However, if we pass **0**, we get `ZeroDivisionError` as the code block inside `else` is not handled by preceding `except`.

```
Enter a number: 0
Traceback (most recent call last):
  File "<string>", line 7, in <module>
    reciprocal = 1/num
ZeroDivisionError: division by zero
```

> **Note**: Exceptions in the `else` clause are not handled by the preceding except clauses.

Python try...finally

In Python, the `finally` block is always executed no matter whether there is an exception or not.

The `finally` block is optional. And, for each `try` block, there can be only one `finally` block.

Let's see an example,

```
try:
    numerator = 10
    denominator = 0

    result = numerator/denominator

    print(result)
except:
    print("Error: Denominator cannot be 0.")

finally:
    print("This is finally block.")
```

Output

```
Error: Denominator cannot be 0.
This is finally block.
```

Activity: Handle Errors with the Try-Except Block in Python

1. Create a Try-Except Block.... with two exceptions
    - Within the try block, have your code (that is more likely to have errors)

    Code:

    > Create two variables x and y
    >
    > Set x to take an integer input from Users so it prompts user to enter a number
    >
    > Set y to hold value of 10 / input from User
    >
    > print the output

    Now, think of a possible error that can occur if you are prompting the user to input a value

    - User might enter an invalid integer Eg: Negative value
    - Hint: What error takes care of that?

    In the except block, print a statement to inform the User to enter a valid integer

    - User might enter the value zero. Math division does not divide by 0.
    - What error takes care of this?

    In the second except block, print a statement to inform the User that they cannot divide by zero.