Python Week -7

Assignment Solution

PW SKILLS

Theory Assignment:

1. In Python, errors can be broadly classified into three types: syntax errors, runtime errors (exceptions), and logical errors. Syntax errors occur when the Python interpreter encounters incorrect syntax in the code, preventing it from parsing and executing the program. Runtime errors, also known as exceptions, occur during the execution of the program when something unexpected happens, such as division by zero or trying to access a non-existent file. Logical errors, on the other hand, occur when the program runs without any syntax or runtime errors but produces incorrect results due to flawed logic.

2. Exceptions in Python are events that occur during the execution of a program that disrupt the normal flow of instructions. They can be raised explicitly using the `raise` statement or implicitly by the Python interpreter when an error condition is encountered. Exceptions are handled using try-except blocks, where the code that might raise an exception is placed inside the try block, and the handling code is placed inside the except block.

3. Syntax errors in Python occur when the interpreter encounters code that violates the language syntax rules. These errors prevent the interpreter from parsing the code and generating bytecode for execution. When a syntax error is encountered, the interpreter raises a SyntaxError exception and provides information about the location and nature of the error, allowing developers to correct it before running the program.

4. Built-in exceptions in Python are predefined classes that represent various error conditions that can occur during program execution. Examples of commonly occurring built-in exceptions include ValueError (raised when an operation or function receives an argument of the correct type but with an inappropriate value), TypeError (raised when an operation or function is applied to an object of an inappropriate type), and FileNotFoundError (raised when attempting to open or manipulate a file that does not exist).

5. Exceptions can be forcefully raised in Python using the `raise` statement. By explicitly raising exceptions, developers can signal abnormal conditions in their code and handle them appropriately. Additionally, the `assert` statement can be used to raise an AssertionError if a specified condition evaluates to False. For **example**:

```
x = 10
if x > 5:
    raise ValueError("x should not be greater than 5")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[1], line 3
      1 x = 10
      2 if x > 5:
----> 3     raise ValueError("x should not be greater than 5")

ValueError: x should not be greater than 5
```

6. The purpose of the `try...except` block in Python is to handle exceptions gracefully and prevent them from terminating the program abruptly. The `try` block contains the code that may raise an exception, while the `except` block contains the code to handle the exception if it occurs. By using `try...except` blocks, developers can anticipate and respond to potential errors, ensuring the robustness and reliability of their programs.

7. Catching exceptions using the `try...except` block involves placing the code that may raise an exception inside the `try` block and providing one or more `except` blocks to handle specific types of exceptions. If an exception occurs in the `try` block, the interpreter searches for a matching `except` block. If a match is found, the code inside the corresponding `except` block is executed to handle the exception. If no matching `except` block is found, the exception propagates up the call stack until it is caught or the program terminates.

Here's an example:

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Division by zero is not allowed")
```

```
Division by zero is not allowed
```

8. The `finally` clause in Python exception handling is used to define cleanup actions that must be executed regardless of whether an exception occurs or not. It is typically used to release external resources such as files or network connections or to perform cleanup tasks like closing database connections. The `finally` block is executed even if an exception is raised in the `try` block or if the program exits the `try` block through a `return` statement. Here's an example:

```python
try:

```
    file = open("example.txt", "r")
    # Perform operations on the file
except FileNotFoundError:
    print("File not found")
finally:
    file.close()  # Close the file regardless of exceptions
```

9. Multiple exceptions can be handled in Python using the `try...except` block by providing multiple `except` blocks, each handling a specific type of exception. The `except` blocks are evaluated sequentially, and the first block that matches the raised exception type is executed.

Here's an example:

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Division by zero")
except ValueError:
    print("Invalid value")
```

```
Division by zero
```

10. The `else` clause in Python exception handling is executed if no exception occurs in the `try` block. It provides a way to specify code that should run only when no exceptions are raised. The `else` clause complements the `try...except` block by separating the code that may raise exceptions from the code that should execute only in the absence of exceptions.

Here's an example:

```
try:
    result = 10 / 2
except ZeroDivisionError:
    print("Division by zero")
else:
    print("Division successful")
```

```
Division successful
```

11. Operator overloading in Python allows custom classes to define their own behavior for built-in operators such as `+`, `-`, `*`, and `/`. It enables developers to extend the functionality of operators to work with custom objects by defining special methods (magic methods) within the class.

12. Magic methods, also known as dunder methods (double underscore methods), play a crucial role in operator overloading in Python. These methods are invoked implicitly by the interpreter to implement specific behavior for built-in operators when used with custom objects. Examples of commonly used magic methods for operator overloading include `__add__`, `__sub__`, `__mul__`, and `__div__`.

13. Implementing operator overloading in a custom Python class involves defining special methods within the class to handle specific operators. For example, to overload the addition and subtraction operators, you would define the `__add__` and `__sub__` methods, respectively.

Here's an example:

In [8]:
```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        if isinstance(other, Vector):
            return Vector(self.x + other.x, self.y + other.y)
        else:
            raise TypeError("Unsupported operand type for addition")

    def __sub__(self, other):
        if isinstance(other, Vector):
            return Vector(self.x - other.x, self.y - other.y)
        else:
            raise TypeError("Unsupported operand type for subtraction")

    def __repr__(self):
        return f"Vector({self.x}, {self.y})"

v1 = Vector(1, 2)
v2 = Vector(3, 4)

result_add = v1 + v2  # Addition using operator overloading
result_sub = v1 - v2  # Subtraction using operator overloading

print(result_add)
print(result_sub)
```

```
Vector(4, 6)
Vector(-2, -2)
```

14. Comparison operators (`__eq__`, `__ne__`, `__lt__`, `__gt__`, `__le__`, `__ge__`) and assignment operators (`__iadd__`, `__isub__`, `__imul__`, etc.) contribute to operator overloading in Python by allowing custom classes to define behavior for equality, comparison, and in-place modification operations. These operators enable developers to make custom objects work seamlessly with Python's built-in operators.

15. Operator overloading in Python offers several advantages, including improved code readability by allowing intuitive actions on custom objects, increased expressiveness by incorporating mathematical and logical operations directly into class definitions, and enhanced maintainability by providing a clear and concise way to define custom behavior for operators. Additionally, operator overloading facilitates the development of domain-specific languages and promotes code reuse through the use of familiar syntax and semantics.

Programming Assignments:

Please refer to this Link: Week-07.ipynb