MODULE 6

INTERMEDIATE ELEMENTS OF PYTHON 3

MODULE OVERVIEW

Python is a versatile programming language that offers powerful tools and features to handle exceptions, manage file input and output (I/O), and utilize recursion. These concepts are fundamental to writing robust, efficient, and maintainable code. This overview provides a brief explanation of each concept and its significance in Python programming. Exceptions are events that occur during the execution of a program, disrupting the normal flow of code. File I/O is a fundamental operation in programming that allows reading data from files or writing data to files. Recursion is a powerful technique where a function calls itself during its execution.

MODULE LEARNING OBJECTIVES

By the end of this module:

- Understand the concept of exceptions in Python and their significance in error handling.
- Learn to use try-except blocks to gracefully handle exceptions and prevent program crashes.
- Use the else and finally clauses to execute specific code in different exception scenarios.
- Learn to read data from files using various methods, such as read(), readline(), and readlines().
- Grasp the concept of recursion and its use in solving complex problems through self-referential functions.
- Learn to implement recursion to traverse and manipulate complex data structures effectively.

LEARNING CONTENTS

6.1 Exceptions

In Python, exceptions are events that occur during the execution of a program, leading to the interruption of the normal flow of code. When an exceptional situation arises, Python raises an exception, which can be caught and handled by the programmer. This mechanism allows you to gracefully handle errors and prevent the program from crashing abruptly.

The basic syntax for handling exceptions in Python is the "try-except" block:

Example

```
try:
  # Code that may raise an exception
  # ...
except SomeExceptionType:
  # Code to handle the exception
  # ...
```

Here's a breakdown of the key components:

- `try`: The block of code that is monitored for exceptions. If an exception occurs within this block, Python will jump to the corresponding except block.
- `except`: The block of code that will be executed if an exception of the specified type occurs in the try block.
- `SomeExceptionType`: The specific type of exception you want to catch. You can replace SomeExceptionType with the actual exception you want to handle. For example, ZeroDivisionError, ValueError, FileNotFoundError, etc. If you want to catch any exception, you can use a generic except clause without specifying any exception type.

Example

```
try:
   x = 10 / 0
except ZeroDivisionError:
   print("Error: Division by zero occurred.")
```

This will raise a `ZeroDivisionError`

Output: Error: Division by zero occurred.

You can use multiple except blocks to handle different types of exceptions, or a single except block to handle multiple exception types:

Example

```
try:
   # Some code that may raise an exception
   # ...
except ValueError:
   # Handle ValueError
   # ...
except FileNotFoundError:
   # Handle FileNotFoundError
   # ...
except (TypeError, IndexError):
   # Handle both TypeError and IndexError
   # ...
except Exception as e:
   # Catch any other exceptions not explicitly handled above
   # e contains information about the exception
```

Python has a wide range of built-in exceptions to handle various error situations. Here are some of the most used built-in exceptions in Python:

Exception	Description
SyntaxError	Raised when the Python parser encounters a syntax error.
IndentationError	Raised when there is an incorrect indentation in the code.

NameError	Raised when a variable or name is not found in the local or global scope.
TypeError	Raised when an operation or function is applied to an object of inappropriate type.
IndexError	Raised when an index is out of range (e.g., accessing a non-existing element in a list).
ValueError	Raised when a function receives an argument of the correct type but an inappropriate value.
KeyError	Raised when a dictionary key is not found.
AttributeError	Raised when an attribute reference or assignment fails.
ZeroDivisionError	Raised when division or modulo operation is performed with zero as the denominator.
FileNotFoundError	Raised when a file or directory is requested but cannot be found.
IOError	Raised when an input/output operation fails, such as reading or writing to a file.
ImportError	Raised when an import statement fails to find the module or name being imported.
StopIteration	Raised by iterators to signal that the iteration is completed.
KeyboardInterrupt	Raised when the user interrupts the execution, usually by pressing Ctrl+C.
AssertionError	Raised when an assert statement fails
OverflowError	Raised when the result of an arithmetic operation is too large to be expressed within the available memory.
MemoryError	Raised when an operation runs out of memory.
SystemError	Raised when an internal error occurs in the Python interpreter.
ArithmeticError	Base class for arithmetic-related exceptions.
Exception	Base class for all built-in exceptions.

6.2 File Input and Output

In Python, file input and output operations are essential for reading data from files and writing data to files. File input and output (I/O) in Python refers to the process of reading data from files and writing data to files. It allows Python programs to interact with external files, enabling them to persist data beyond the runtime of the program.

6.2.1 File input

Reading data from files is known as file input. It involves opening a file, reading its content, and processing the data in the Python program. Python provides built-in functions and methods to handle file operations easily.

1. **Opening a File**

   To work with files, you need to open them first. Python provides the open() function for this purpose. The open() function takes two arguments: the file path and the mode in which the file should be opened (read, write, append, etc.). The file path can be either an absolute or a relative path. To open a file in Python, you can use the built-in open() function. The open() function takes two arguments: the file path and the mode in which you want to open the file. The mode specifies whether you want to read, write, or append to the file, among other options.

   Syntax

```
file = open(file_path, mode)
```

   Parameters:

   - file_path: The file path, either an absolute path or a relative path to the file you want to open.
   - mode: The mode in which you want to open the file. It is a string containing one or more characters specifying the access mode.

   Here are some common file modes:

   - **"r"**: Read mode - Opens the file for reading (default mode).
   - **"w"**: Write mode - Opens the file for writing. If the file exists, it truncates (clears) its content. If the file does not exist, it creates a new file.
   - **"a"**: Append mode - Opens the file for writing, but it does not truncate the existing content. Instead, it appends new data to the end of the file.
   - **"b"**: Binary mode - Opens the file in binary mode, used for non-text files like images, audio, etc.
   - **"t"**: Text mode - Opens the file in text mode, used for text files (default mode).

   Example - Opening a file in read mode and reading its content:

   Example

```
file_path = "example.txt"

# Open the file in read mode
with open(file_path, "r") as file:
    content = file.read()
    print(content)
```

2. **Reading from a File**
   To read a file in Python, you can use the open() function to open the file in read mode ("r"). Once the file is open, you can use various methods to read its content.

   Here are three common methods for reading a file:

   a. **read**(): Reads the entire content of the file as a single string.
   Example

```
file_path = "example.txt"

with open(file_path, "r") as file:
   content = file.read()
print(content)
```

Open the file in read mode

   b. **readline**(): Reads one line at a time from the file. Each call to readline() returns the next line in the file.
   Example

```
file_path = "example.txt"

with open(file_path, "r") as file:
   line1 = file.readline()
   line2 = file.readline()

print(line1)
print(line2)
```

Open the file in read mode
Read the first line
Read the second line
# ... and so on

   c. **readlines**(): Reads all lines from the file and stores them in a list. Each line is an element in the list.
   Example

```
file_path = "example.txt"

with open(file_path, "r") as file:
   lines = file.readlines()

for line in lines:
   print(line)
```

Open the file in read mode
Print the lines read from the file

   When using these methods, it's essential to remember the following:
   - Make sure the file exists and the file path is correct.
   - Handle exceptions, as file operations may raise IOError if there are any issues with file access or reading.
   - Always close the file after reading (using the with statement automatically takes care of this).

3. **Writing to a File**
   To write data to a file in Python, you need to open the file in write mode ("w") or append mode ("a") using the open() function. In write mode, if the file already exists, its content will be truncated (deleted), and the new data will be written from the beginning. In append mode, new

data will be added to the end of the existing content without deleting it.
Here are two examples demonstrating how to write to a file:

a. Writing a single line to a file:

To write a single line to a file in Python, you can use the open() function in write mode ("w") and then use the write() method to write the line to the file. Here's an example:

Example

```
file_path = "example.txt"

with open(file_path, "w") as file:
    file.write("Hello, World!\n")
```

Open the file in write mode and write a single line

In this example, the file named "example.txt" will be created (if it doesn't exist) or truncated (if it already exists), and the line "Hello, World!" will be written to the file with a newline character ("\n") at the end.

After running this code, the content of "example.txt" will be:

```
Hello, World!
```

If the file already existed and had some content, this code will overwrite the previous content and replace it with the new line. Keep in mind that when using the write() method, you need to include the newline character ("\n") if you want to write a new line to the file. If you don't include it, multiple calls to write() will write the text continuously without any line breaks.

b. Writing multiple lines to a file:

To write multiple lines to a file in Python, you can use the open() function in write mode ("w") or append mode ("a") along with the writelines() method to write a list of lines to the file. Here's an example for both write mode and append mode:

b.1 Writing multiple lines to a file using write mode ("w")

Example

```
file_path = "example.txt"

lines = ["Line 1\n", "Line 2\n", "Line 3\n"]

with open(file_path, "w") as file:
    file.writelines(lines)
```

A list of lines to write to the file

Open the file in write mode and write the lines

After running this code, the content of "example.txt" will be:

```
Line 1
Line 2
Line 3
```

In this case, if the file "example.txt" already existed, it will be truncated (its content will be deleted), and the new lines will be written.

b.2 Writing multiple lines to a file using append mode ("a"):

```
file_path = "example.txt"
lines = ["Line 4\n", "Line 5\n"]
with open(file_path, "a") as file:
    file.writelines(lines)
```

A list of lines to write to the file

Open the file in append mode and write the lines

After running the append mode code, the content of "example.txt" will be:

```
Line 1
Line 2
Line 3
Line 4
Line 5
```

In this case, if the file "example.txt" already existed, the new lines will be added to the end of the existing content without deleting it. Remember to include the newline character ("\n") at the end of each line in the lines list if you want each line to be written as a separate line in the file. Otherwise, all the lines will be written continuously without any line breaks.

4. **Closing a File**

Closing a file is an essential step after you finish reading from or writing to it in Python. Closing a file ensures that all the resources associated with the file, such as the file descriptor, memory buffers, and operating system resources, are released properly.

There are two common ways to close a file in Python:

a. **Using the close() method:**

You can explicitly call the close() method on the file object to close the file.

Example

```
file_path = "example.txt"

file = open(file_path, "r")

# File processing code here
# ...
file.close()
```

Open the file in read mode

Close the file explicitly

However, manually calling close() is prone to errors, especially if an exception occurs during file processing. To ensure proper file closure, it is better to use the second approach.

b. **Using the with statement:**

The with statement provides a more elegant way to handle file operations. It automatically takes care of closing the file when the indented block of code is exited, even if an exception occurs.

Example

```
file_path = "example.txt"

#
with open(file_path, "r") as file:
   # File processing code here
   # ...
```

Open the file in read mode
using the 'with' statement
The file is automatically
closed after the 'with' block

Using the with statement is recommended because it ensures proper file closure and reduces the risk of resource leaks. It also makes the code cleaner and more readable. Remember, after closing a file, you cannot perform any more file operations on that specific file object. If you need to read from or write to the file again, you must open it again using the open() function.

Always remember to close files after use, especially when dealing with a large number of files or performing file operations within loops. Proper file handling helps prevent issues like file locking, resource exhaustion, and data corruption.

6.3 Recursion

Python uses a concept called recursion where a function calls itself to solve a problem. Certain sorts of issues, especially those that can be divided into smaller, related subproblems, can be solved in this fashion in a strong and elegant way. Recursion occurs when a function calls itself repeatedly with decreasing inputs until it hits a base case, at which point it stops.

When using recursion, you need to define a base case (or termination condition) that stops the recursive calls and prevents infinite recursion. If the base case is not reached, the function will keep calling itself with smaller inputs until it reaches the base case.

Here's a simple example of a recursive function to calculate the factorial of a number:

Example

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)

result = factorial(5)
print(result)
```

Base case: factorial of 0 and 1 is 1

Recursive call to solve a smaller subproblem

Calculates 5! (5 factorial)

Output: 120 (since 5! = 5 * 4 * 3 * 2 * 1 = 120)

In this example, the factorial function is defined using recursion. The base case checks if n is 0 or 1, and in those cases, it returns 1 since the factorial of 0 and 1 is 1. For any other value of n, it returns n multiplied by the factorial of n - 1, which is the recursive call with a smaller input.

It's important to be cautious when using recursion because it can lead to stack overflow errors if not implemented correctly. In some cases, using iteration or other techniques may be more efficient and easier to understand than recursion. However, recursion can be a powerful tool for certain problems and is a fundamental concept in computer science.