Contents

1. Errors and Exception

Error in Python can be of two types i.e. Syntax errors and Exceptions.

Syntax error: This error is caused by the wrong syntax in the code. It leads to the termination of the program.

```
a = 10
if(a > 5)
print("a is greater than 5")
```

```
Output:
   if(a > 5)
          ^
SyntaxError: invalid syntax
```

```
a = 10
if(a > 5):
print("a is greater than 5")
```

```
   print("a is greater than 5")
   ^
IndentationError: expected an indented block
```

Exceptions: Exceptions are raised when the program is syntactically correct, but the code results in an error. This error does not stop the execution of the program, however, it changes the normal flow of the program.

Example:

```
marks = 10000
a = marks / 0
print(a)
```

```
  File "e:\python_programming\test.py", line 31, in <module>
    b = a/0
ZeroDivisionError: division by zero
```

Here are some of the most common types of exceptions in Python:

- **SyntaxError:** This exception is raised when the interpreter encounters a syntax error in the code, such as a misspelled keyword, a missing colon, or an unbalanced parenthesis.
- **TypeError**: This exception is raised when an operation or function is applied to an object of the wrong type, such as adding a string to an integer.
- **NameError**: This exception is raised when a variable or function name is not found in the current scope.
- **IndexError**: This exception is raised when an index is out of range for a list, tuple, or other sequence types.
- **KeyError**: This exception is raised when a key is not found in a dictionary.
- **ValueError**: This exception is raised when a function or method is called with an invalid argument or input, such as trying to convert a string to an integer when the string does not represent a valid integer.
- **AttributeError**: This exception is raised when an attribute or method is not found on an object, such as trying to access a non-existent attribute of a class instance.
- **IOError**: This exception is raised when an I/O operation, such as reading or writing a file, fails due to an input/output error.
- **ZeroDivisionError**: This exception is raised when an attempt is made to divide a number by zero.
- **ImportError**: This exception is raised when an import statement fails to find or load a module.

Code	Exceptions
print(10 + "Hello")	print(10 + "Hello") TypeError: unsupported operand type(s) for +: 'int' and 'str'
print(hello())	print(hello()) NameError: name 'hello' is not defined
numbers = [1, 2, 3, 4] print(numbers[4])	print(numbers[4]) IndexError: list index out of range

2. Catching and handling exceptions

Try and except statements are used to catch and handle exceptions in Python. Statements that can raise exceptions are kept inside the try clause and the statements that handle the exception are written inside except clause.

Python try...except Block

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

Example:

```
a = 10
b = int(input("Enter the denominator"))
try:
    result = a / b
    print(result)
except:
    print("Error: denominator cannot be 0.")
```

```
Enter the denominator0
Error: denominator cannot be 0.
```

Catching Specific Exceptions in Python

Example:

```
a = 10
numbers = [1, 2, 3, 4]
b = int(input("Enter the denominator"))
index = int(input("Enter the index"))
try:
```

```
    result = a / b
    print(result)
    data = numbers[index]
    print(data)
except ZeroDivisionError:
    print("Error: denominator cannot be 0.")


except IndexError:
    print("Error: index out of bound.")
```

```
Enter the denominator2
Enter the index5
5.0
Error: index out of bound.
```

Python try with else clause

```
Syntax:
try:
    # code that may cause exception
except:
    # code to run when exception occurs
else:
    # code to run if try runs without errors
```

Example:

```
a = 10
b = int(input("Enter the denominator: "))
try:
    result = a / b
except ZeroDivisionError:
```

```
    print("Error: denominator cannot be 0.")
else:
    print(result)
```

```
Enter the denominator: 5
2.0
```

Python try...finally

```
Syntax:
try:
    # code that may cause exception
except:
    # code to run when exception occurs
else:
    # code that always executes
```

Example:

```
a = 10
b = int(input("Enter the denominator: "))
try:
    result = a / b
except ZeroDivisionError:
    print("Error: denominator cannot be 0.")
else:
    print(result)
finally:
    print("Execution Completed.")
```

```
Enter the denominator: 0
Error: denominator cannot be 0.
Execution Completed.
```

```
Enter the denominator: 5
```

```
2.0
Execution Completed.
```

3. User-defined exceptions

In Python, we can define custom exceptions by creating a new class that is derived from the built-in Exception class.

Here's the syntax to define custom exceptions,

```
class CustomError(Exception):
    ...
    pass

try:
   ...
   raise CustomError()

except CustomError:
    ...
```

Example:

```
class InvalidAgeException(Exception):
    pass

try:
    age = int(input("Enter your age: "))
    if age < 18:
        raise InvalidAgeException()
    else:
        print("You can vote")
```

```
except InvalidAgeException:
    print("You cannot vote")
```

```
Enter your age: 10
You cannot vote
```

Example:

```
class InvalidAgeException(Exception):
    def __init__(self, message = "Age cannot be less than 18."):
        super().__init__(message)

age = int(input("Enter your age: "))
if age < 18:
    raise InvalidAgeException
```

```
Enter your age: 10
Traceback (most recent call last):
  File "e:\python_programming\tempCodeRunnerFile.py", line 8, in <module>
    raise InvalidAgeException
__main__.InvalidAgeException: Age cannot be less than 18.
```

Example:

```
class InvalidAgeException(Exception):
    def __init__(self, message = "Age cannot be less than 18."):
        super().__init__(message)

try:
    age = int(input("Enter your age: "))
    if age < 18:
        raise InvalidAgeException()

except InvalidAgeException as e:
```

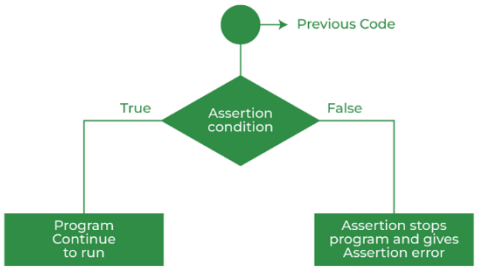
```
    print(e)
```

```
Enter your age: 10
Age cannot be less than 18.
```

4. Debugging programs with the assert statement

Python Assertions are the debugging tools that help in the smooth flow of code. Assertions are mainly assumptions that a programmer knows or always wants to be true and hence puts them in code so that failure of these doesn't allow the code to execute further.

The assert statement in Python is used for debugging purposes. It allows you to test if a condition in your code returns True. If the condition evaluates to False, an AssertionError exception is raised, which helps in identifying bugs or issues in your program.



Syntax :

```
assert condition, error_message(optional)
```

```
a = 10
b = 0
assert b != 0
print(a / b)
```

```
Traceback (most recent call last):
  File "e:\python_programming\assert_1.py", line 3, in <module>
    assert b != 0
AssertionError
```

Example:

```
def calculate_average(numbers):
    assert len(numbers) > 0, "List is Empty"
    return sum(numbers) / len(numbers)

numbers = [1, 2, 3, 4, 5]
print(calculate_average(numbers))
```

```
3.0
```

Assertions can be used to validate the accuracy of data, ensuring that it satisfies particular criteria. By placing assertions at critical points in the code, we can rapidly identify problems and isolate their source.

Q. Write a function for calculating the discounted price from the original price by asserting the value of the discount percentage in the range [0,100].

```
def calculate_discounted_price(original_price, discount):
    assert discount >= 0 and discount <= 100, "Invalid discount percentage!"
    discounted_price = original_price - (original_price * (discount / 100))
    return discounted_price

price = 1000
discount_percentage = 120

final_price = calculate_discounted_price(price, discount_percentage)
print(f"The final price after a {discount_percentage}% discount is: Rs. {final_price}")
```

```
    final_price = calculate_discounted_price(price, discount_percentage)
  File "e:\python_programming\assert_3.py", line 2, in calculate_discounted_price
    assert discount >= 0 and discount <= 100, "Invalid discount percentage!"
AssertionError: Invalid discount percentage!
```

5. Logging the exceptions

To log an exception in Python we can use a logging module and through that, we can log the error. The logging module provides a set of functions for simple logging and the following purposes.

- DEBUG
- INFO
- WARNING
- ERROR
- CRITICAL

Logging an exception in Python with an error can be done in the logging.exception() method. This function logs a message with level ERROR on this logger. This method should only be called from an exception handler.

```
import logging


try:
    print(a)
except:
    logging.exception("Error occured while printing")
```

```
ERROR:root:Error occured while printing
Traceback (most recent call last):
  File "e:\python_programming\logging_1.py", line 4, in <module>
    print(a)
NameError: name 'a' is not defined
```

```
import logging

logging.basicConfig(filename='test.log', level=logging.DEBUG)

def div(a, b):
    try:
        return a / b
    except Exception:
        logging.exception(f"An error occurred with a, b = {a}, {b}")

a = 20
b = 0

result = div(a, b)
if result is not None:
    logging.debug(f"{a}/{b} = {result}")
else:
    logging.debug(f"Division failed for {a}/{b}")
```

6. Introduction to file handling

Python supports file handling and allows users to handle files i.e., to read and write files, along with many other file handling options, to operate on files.

Python File Open:

Before performing any operation on the file like reading or writing, first, we have to open that file. For this, we should use Python's inbuilt function open() but at the time of opening, we have to specify the mode, which represents the purpose of the opening file.

```
Syntax: file_object = open(filename, mode)
```

Modes of file:

- r: open an existing file for a read operation.
- w: open an existing file for a write operation. If the file already contains some data, then it will be overridden but if the file is not present then it creates the file as well.
- a: open an existing file for append operation. It won't override existing data.
- r+: To read and write data into the file. This mode does not override the existing data, but you can modify the data starting from the beginning of the file.
- w+: To write and read data. It overwrites the previous file if one exists, it will truncate the file to zero length or create a file if it does not exist.
- a+: To append and read data from the file. It won't override existing data.

Functions to write and read from a file:

- write():
- writelines()
- read()
- readline()
- readlines()

Function to close a file:

- fclose()

Example:

Writing to a file	Reading from a file
f = open('myfile.txt', 'w') f.write('From File\nHello world') f.close()	f = open('myfile.txt', 'r') print(f.read()) f.close()

Using writelines() and readlines()

f = open('myfile.txt', 'w') names = ["Ram\n", "Hari\n", "Shyam\n"] f.writelines(names) f.close()	f = open('myfile.txt',"r") lines = f.readlines() for line in lines: print(line.strip()) f.close()

Append in a file:

```
f = open('myfile.txt', 'a')
names = ["Krishna\n", "Shiva\n", "Sita\n"]
f.writelines(names)
f.close()
```

With in File Handling:

The with statement in Python is used to wrap the execution of a block of code. It's commonly used in file handling to ensure that files are properly closed after their suite finishes, even if an exception is raised. With open() automatically closes the file, so we don't have to use the close() function.

Basic Syntax:

```
with open('filename.txt', 'mode') as file:
    # Perform file operations
```

```
with open('myfile.txt',"r") as f:
    lines = f.readlines()
    for line in lines:
        print(line.strip())
```

Note: If you're not using the with statement, you should manually close the file.

Example: You have a text file named source.txt that contains a list of numbers, with each number on a separate line. You need to write a Python script that reads the numbers from source.txt and then separates them into even and odd numbers. The even numbers should be written to a file named even.txt, and the odd numbers should be written to a file named odd.txt.

```
even_num = []
odd_num = []
with open('source.txt','r') as src_file:
    str_numbers = src_file.readlines()
    numbers = list(map(int,str_numbers))
    for number in numbers:
```

```
        if number % 2 == 0:
            even_num.append(str(number) + '\n')
        else:
            odd_num.append(str(number) + '\n')

with open('even.txt','w') as even_file:
    even_file.writelines(even_num)

with open('odd.txt','w') as odd_file:
    odd_file.writelines(odd_num)
```

Working with binary files:

```
with open('myfile.bin','wb') as myfile:
    binary_data = b'\x48\x65\x6c\x6c\x6f'
    myfile.write(binary_data)

with open('myfile.bin','rb') as myfile:
    binary_data_from_file = myfile.read()
    print(binary_data_from_file)
```

Q. What is the output of the program?

```
with open('myfile.bin','wb') as myfile:
    binary_data = b"\x50\x79\x74\x68\x6f\x6e\x20\x69\x73\x20\x65\x61\x73\x79"
    myfile.write(binary_data)

with open('myfile.bin','rb') as myfile:
    binary_data_from_file = myfile.read()
    print(binary_data_from_file)
```

Exception Handling in File:

```
try:
    with open('myfile.txt',"r") as f:
        lines = f.readlines()
        for line in lines:
            print(line.strip())

except FileNotFoundError:
    print("Error File not found")
except PermissionError:
    print("Error You do not have permission")
except IOError:
    print("Error IO error occurs")
except Exception as e:
    print("An error occured",e)
```

Example: **Using Pickle Module**

```
import pickle

class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f'{self.name} {self.age}'

person = Person("Alice", 30)

with open('person.pkl', 'wb') as file:
```

```
    pickle.dump(person, file)


with open('person.pkl', 'rb') as file:
    person_from_file = pickle.load(file)


print(person_from_file)
```

Random File Access:

Random file access refers to the ability to read from or write to any part of a file without having to sequentially read or write through the entire file. This is particularly useful for applications where you need to frequently access different parts of a file.

For random file access in Python, the following functions and methods are essential:

- seek(offset, whence) : Moves the file pointer.
- tell() : Returns the current position

Example:

myfile.txt has I love python programming

```
with open('myfile.txt',"r") as f:
    f.seek(7)
    data = f.read(6)
    position = f.tell()
    print(data)
    print(f"File pointer is at {position}")
```

Output:

```
python
File pointer is at 13
```

7. Python libraries and Maths