What is an Exception

An exception is an occurrence that causes the program's execution to be interrupted. In other words, when a Python script meets a condition that it is unable to handle, it throws an exception. Exceptions in Python are raised in the form of objects. When an exception occurs, an object containing information about the exception is initialized. In Python, an exception must be addressed or the program may terminate.

Exception Handling in Python

Exception handling in Python allows us to handle and recover from errors or exceptional situations that can occur during the execution of our code. These exceptional situations are known as exceptions. Python provides a mechanism to catch and manage exceptions to prevent our program from crashing when something unexpected happens.

Syntax for Exception Handing

The syntax for exception handling is as follows:

```
Try:
    #the code that can raise exception
except ExceptionA:
    #the code to execute if ExceptionA occurs
```

For example:

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

Output: Division by zero is not allowed

Multiple Exceptions

We can handle different types of exceptions separately by using multiple except blocks.

Syntax

```
try:
    # Code that might raise an exception
except ExceptionType1:
    # Handle ExceptionType1
except ExceptionType2:
    # Handle ExceptionType2
except ExceptionType3:
    # Handle ExceptionType3
```

Example

To manage multiple exceptions, simply stack one exception handling block on top of the other. Consider the following exception:
Exam

```
try:
        num1 = 10
        num2 = 2
        result = num1/num2
        print(result)
except ZeroDivisionError:
        print ("Sorry, division by zero not possible")
except NameError:
        print ("Some variable/s are not defined")
else:
        print("Program executed without an exception")
```

In the script above, both the “ZeroDivisionError” and “NameError” exceptions are handled. Therefore, if you set the value of num2 to 0, the “ZeroDivisionError” exception will occur. However if you try to divide the num1 by ‘b’, the “NameError” exception will occur since the variable “b” is not defined. Finally if none of the exception occurs, the statement in the **else** block will execute.

Another way to handle multiple exceptions is by using Exception object which is base class for all the exceptions. The Exception object can be used to handle all types of exceptions. Take a look at the following example.

```
try:
    num1 = 10
    num2 = 0
    result = num1/num2
    print(result)
except Exception:
    print ("Sorry, program cannot continue")
else:
    print("Program executed without an exception")
```

In the script above, all the different types of exceptions will be handled by code block for Exception object. Therefore, a generic message will be printed to the user. In the script above, the num2 contains zero. Therefore, the "ZeroDivisionError" exception will occur. The result will look like this:

```
Sorry, program cannot continue
```

Generic Exception Handling:

We can use a generic **except** block to catch any exception, but this should be used sparingly because it can make it harder to identify and fix issues.

```
try:
    # Code that might raise an exception
except:
    # Handle any exception
```

Else Block:

We can use an else block after the try and except blocks to specify code that should be executed if no exception occurs.

```
try:
    # Code that might raise an exception
except ExceptionType:
    # Handle the exception
else:
    # Code to run if no exception occurred
```

Finally Block:

We can use a **finally** block to specify code that should be executed regardless of whether an exception occurred or not. This is useful for cleanup operations like closing files or network connections.

try:

 # Code that might raise an exception

except ExceptionType:

 # Handle the exception

finally:

 # Code to run no matter what

Python File Handling

File handling or data handling is the process of performing various operations on various types of files. The most popular file operations are opening a file, reading the contents of a file, creating a file, writing data to a file, appending data to a file, etc.

1. Opening a File

To open a file in python the open function is used. It takes 3 parameters: The path to the file, the mode in which the file should be opened and the buffer size in number of lines. The third parameter is optional. The open function returns file object. The syntax of the open function is as follows:

file_object = open(file_name, file_mode, buffer_size)

Following table describe different types of modes along with their description:

Mode	Description
R	Opens file for read only
r+	Opens file for reading and writing
Rb	Only Read file in binary
rb+	Opens file to read and write in binary
W	Opens file to write only. Overwrites existing files with same Name
w+	Opens file for reading and writing
Wb	Opens file to read and write in binary. Overwrites existing files with

	same name
A	Opens a file for appending content at the end of the file
a+	Opens file for appending as well as reading content
Ab	Opens a file for appending content in binary
ab+	Opens a file for reading and appending content in binary

The file object returned by the ***open*** method has three main attributes:
1- name: returns the name of the file
2- mode: returns the mode with which the file was opened
3- closed: is the file closed or not

Example:

First create a file test.txt and place it in the root directory of the D drive.

```
file_object = open("D:/test.txt", "r+")

print(file_object.name)

print(file_object.mode)

print(file_object.closed)
```

In the script above, we open the test.txt file in the read and write mode. Next we print the name and mode of the file on the screen. Finally we print whether the file is closed or not.

Output:

```
D:/test.txt

r+

False
```

2. Close a File

To close an opened file, we can use ***close*** method.

Example:

```
file_object=open("D:/test.txt","r+")
print(file_object.name)
print(file_object.closed)
file_object.close()
print(file_object.closed)
```

In the script above, the test.txt file is opened in r+ mode. The name of the file is printed. Next we check if the file is opened using closed attribute, which returns false, since the file is open at the

moment. We then close the file using close method. We again check if the file is closed, which returns true since we have closed the file.

Output:

D:/test.txt

False

True

3. Writing Data to a File

To write data to a file, the ***write*** function is used. The content that is to be written to the file is passed as parameter to the write function.

Example:

```
file_object = open("D:/test1.txt", "w+")
file_object .write( "Welcome to Python.\n The best programming language! \n")
file_object .close()
```

In the script above, the file test.txt located at the root directory of D drive is opened. The file is opened for reading and writing. Next two lines of text have been passed to the write function. Finally the file is closed.
If we will go to root directory of D drive, we will see a new file test1.txt with the following contents:

Welcome to Python.
The best programming language!

4. Reading Data from a File:

To read data from a file in Python, the ***read*** function is used. The number of bytes to read from a file is passed as a parameter to the read function.

Example:

```
file_object = open("D:/test1.txt", "r+")
sen = file_object.read(12)
print("The file reads: "+sen)
file_object .close()
```

The script above reads the first 12 characters from the test1.txt file that we wrote in the last example.
Output:

The file reads: Welcome to P

To read the complete file, do not pass anything to the read function. The following script reads the complete test1.txt file and prints its total content.

```
file_object = open("D:/test1.txt", "r+")
sen = file_object.read()
print(sen)
file_object .close()
```

Output:

Welcome to Python.
The best programming language!

5. Renaming Python Files

We can rename and delete python files using Python ***os*** module. To rename a file, the rename function is used. The old name of the file is passed as first parameter while new name is passed as second parameter.

Example:

```
import os

os.rename( "D:/test1.txt", "D:/test2.txt" )
```

The above script renames file test1.txt to test2.txt

6. Delete Python Files

To delete a file in Python, the remove method is used.

Example:

```
import os
os.remove("D:/test.txt")
```

The above script deletes the test.txt file located at the root directory of D drive.

Recent Trends in Python

Python is a high-level, versatile, and easy-to-learn programming language. It is known for its simplicity and readability, making it an excellent choice for beginners and experienced developers alike. Python has wide range of applications, including web development, data analysis, machine learning, automation, scientific computing, artificial intelligence, and more. Its simplicity and extensive ecosystem make it a top choice for developers across various domains.

1. **Artificial Intelligence and Machine Learning**:
    - Python remained the language of choice for AI and machine learning. TensorFlow, PyTorch, and scikit-learn were popular libraries for deep learning and ML.
2. **Data Science and Analytics**:
    - Python continued to dominate the field of data science with libraries like Pandas, NumPy, and Jupyter notebooks for data analysis and visualization.
3. **Natural Language Processing (NLP)**:
    - NLP applications using Python were on the rise, thanks to libraries like spaCy and the widespread use of pre-trained language models like GPT-3 and BERT.
4. **Computer Vision**:
    - Python was widely used for computer vision applications, leveraging libraries such as OpenCV and deep learning frameworks like OpenAI's DALL-E and CLIP.
5. **Web Development**:
    - Frameworks like Django, Flask, and FastAPI were popular choices for web development. Many developers used Python for both server-side and client-side development.
6. **Serverless and Cloud Computing**:
    - Python was a common language for developing serverless functions and cloud-based applications on platforms like AWS Lambda, Google Cloud Functions, and Azure Functions.
7. **Blockchain and Cryptocurrency**:
    - Python was used in blockchain development for creating smart contracts, developing blockchain applications, and interacting with cryptocurrencies.
8. **DevOps and Automation**:
    - Python played a crucial role in DevOps and automation tasks, including infrastructure management, deployment automation, and continuous integration.
9. **Cybersecurity**:
    - Python was employed for tasks related to cybersecurity, such as penetration testing, vulnerability assessment, and security tool development.

10. **Quantitative Finance**:
    - Python was widely used in the finance industry for quantitative analysis, algorithmic trading, risk management, and portfolio optimization.
11. **Healthcare and Bioinformatics**:
    - Python was used in healthcare for tasks like medical imaging analysis, data analysis, and genomics research.
12. **IoT (Internet of Things)**:
    - Python was used in IoT projects for device communication, data analysis, and building IoT applications.
13. **Educational Tools and Platforms**:
    - Python was a popular language for creating educational resources and platforms, and it remained a common choice for teaching programming.
14. **3D and Game Development**:
    - Python, particularly with libraries like Pygame and Panda3D, was used for 3D graphics and game development.
15. **Quantum Computing**:
    - Python was adopted for quantum computing development and simulations, with libraries like Qiskit gaining traction.
16. **Type Hinting and Static Analysis**:
    - The use of type hinting with tools like **mypy** to enhance code quality and maintainability continued to grow.