**Interview Questions**

1. **What is Exception?can we handle Exception? How?**

An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.

When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

**Catching Exceptions in Python**

In Python, exceptions can be handled using a try statement.

The critical operation which can raise an exception is placed inside the try clause. The code that handles the exceptions is written in the except clause.

We can thus choose what operations to perform once we have caught the exception.

1. **What is Exception Handling?**

The functions described in this chapter will let you handle and raise Python exceptions. It is important to understand some of the basics of Python exception handling. When a function must fail because some function it called failed, it generally doesn’t set the error indicator; the function it called already set it. It is responsible for either handling the error and clearing the exception or returning after cleaning up any resources it holds (such as object references or memory allocations); it should not continue normally if it is not prepared to handle the error. If returning due to an error, it is important to indicate to the caller that an error has been set. If the error is not handled or carefully propagated, additional calls into the Python/C API may not behave as intended and may fail in mysterious ways.

1. **What is custom Exception?**

Python has numerous [built-in exceptions](https://www.programiz.com/python-programming/exceptions) that force your program to output an error when something in the program goes wrong.

However, sometimes you may need to create your own custom exceptions that serve your purpose.

In Python, users can define custom exceptions by creating a new class. This exception class has to be derived, either directly or indirectly, from the built-in Exception class. Most of the built-in exceptions are also derived from this class.

if any arguments are passed to the Class constructor, we will take the first argument passed, and assign it to an attribute in the object called message. If no arguments are passed, None will be assigned to the message attribute.

1. **How to raise the exception?**

An exception appears during program execution and changes its normal flow due to an error. An exception arises on account of an error. The main cause of an exception is a logical error. Like many other programming languages, Python provides several built-in exceptions, i.e., ZeroDivisionError, ImportError, EOFError, etc.; for instance, the ZeroDivisionError exception is raised when a number is divided by zero. The Python exceptions are handled by the try statement. We define a try block and put the code vulnerable code inside this block, which can raise an exception. Next, after the try block, we define an except block to deal with the exception. Besides all these built-in exceptions, sometimes we need to raise or throw an exception when we encounter a specific situation. The Python developer can throw a user-defined exception easily. We use the raise keyword for raising or throwing an exception. This article explains the Python raise keyword usage for throwing the exception with examples.

1. **Where we throw the exception?**

In Python, you should freely throw them whenever there is no point executing the rest of the code in your function, and/or none of the returns correctly reflect what happened in the function. Note that returning early from a function is different: returning early means you already figured out the answer and don't need the rest of the code to figure out the answer. I am saying that exceptions should be thrown when the answer is not known, and the rest of the code to determine the answer cannot be reasonably run.

1. **Where we catch the exception?**

In Python, exceptions can be handled using a try statement.

The critical operation which can raise an exception is placed inside the try clause. The code that handles the exceptions is written in the except clause.

We can thus choose what operations to perform once we have caught the exception. Here is a simple example.

# import module sys to get the type of exception

import sys

randomList = ['a', 0, 2]

for entry in randomList:

try:

print("The entry is", entry)

r = 1/int(entry)

break

except:

print("Oops!", sys.exc\_info()[0], "occurred.")

print("Next entry.")

print()

print("The reciprocal of", entry, "is", r)

**Output**

The entry is a

Oops! <class 'ValueError'> occurred.

Next entry.

The entry is 0

Oops! <class 'ZeroDivisionError'> occured.

Next entry.

The entry is 2

The reciprocal of 2 is 0.5

In this program, we loop through the values of the randomList list. As previously mentioned, the portion that can cause an exception is placed inside the try block.

If no exception occurs, the except block is skipped and normal flow continues(for last value). But if any exception occurs, it is caught by the except block (first and second values).

Here, we print the name of the exception using the exc\_info() function inside sys module. We can see that a causes ValueError and 0 causes ZeroDivisionError.

Since every exception in Python inherits from the base Exception class, we can also perform the above task in the following way:

# import module sys to get the type of exception

import sys

randomList = ['a', 0, 2]

for entry in randomList:

try:

print("The entry is", entry)

r = 1/int(entry)

break

except Exception as e:

print("Oops!", e.\_\_class\_\_, "occurred.")

print("Next entry.")

print()

print("The reciprocal of", entry, "is", r)

This program has the same output as the above program.

## Catching Specific Exceptions in Python

In the above example, we did not mention any specific exception in the except clause.

This is not a good programming practice as it will catch all exceptions and handle every case in the same way. We can specify which exceptions an except clause should catch.

A try clause can have any number of except clauses to handle different exceptions, however, only one will be executed in case an exception occurs.

We can use a tuple of values to specify multiple exceptions in an except clause. Here is an example pseudo code.

try:

# do something

pass

except ValueError:

# handle ValueError exception

pass

except (TypeError, ZeroDivisionError):

# handle multiple exceptions

# TypeError and ZeroDivisionError

pass

except:

# handle all other exceptions

pass

1. **Difference Between Exception and Error?**

The syntax error exception occurs when the code does not conform to Python keywords, naming style, or programming structure. The interpreter sees the invalid syntax during its parsing phase and raises a **SyntaxError** exception. The program stops and fails at the point where the syntax error happened. That’s why syntax errors are exceptions that can’t be handled.

Here’s an example code block with a syntax error (note the absence of a colon after the “if” condition in parentheses):

a = 10

b = 20

if (a < b)

print('a is less than b')

c = 30

print (c)

The interpreter picks up the error and points out the line number. Note how it doesn’t proceed after the syntax error:

File "test.py", line 4

if (a < b)

^

SyntaxError: invalid syntax

Process finished with exit code 1

On the other hand, an exception happens when the code has no syntax error but encounters other error situations. These conditions can be addressed within the code—either in the current function or in the calling stack. In this sense, exceptions are not fatal. A Python program can continue to run if it gracefully handles the exception.

Here is an example of a Python code that doesn’t have any syntax errors. It’s trying to run an arithmetic operation on two string variables:

a = 'foo'

b = 'bar'

print (a % b)

The exception raised is TypeError:

Traceback (most recent call last):

File test.py", line 4, in print (a % b) TypeError: not all arguments converted during string formatting Process finished with exit code 1

Python throws the TypeError exception when there are wrong data types. Similar to TypeError, there are several built-in exceptions like:

1. ModuleNotFoundError
2. ImportError
3. MemoryError
4. OSError
5. SystemError
6. ... And so on

Errors are the problems in a program due to which the program will stop the execution. On the other hand, exceptions are raised when some internal events occur which changes the normal flow of the program

1. **Difference Between Finally and Else?**

*[finally:]*

This code **always** executes after the other blocks, even if there was an uncaught exception (that didn’t cause a crash, obviously) or a return statement in one of the other blocks. Code executed in this block is just like normal code: if there is an exception, it will not be automatically caught (and probably stop the program). This block is also optional.

*[else:]*

This code is executed only if no exceptions were raised in the *try*block. Code executed in this block is just like normal code: if there is an exception, it will not be automatically caught (and probably stop the program). Notice that if the *else* block is executed, then the *except* block is not, and vice versa. This block is optional.

1. **Explain try block?**

*try:*

The code with the exception that you want to catch. If an exception is raised, control flow leaves this block immediately and goes to the *except* block

1. **Explain except block?**

*except [(Exception[, Exception])] [as VAR]:*

This code is executed only if an exception was raised in the *try* block. Code executed in this block is just like normal code: if there is an exception, it will not be automatically caught (and probably stop the program).

You can optionally name specific types of exceptions in the *except* statements, in which case the block will only be executed if one of the named exceptions was the one raised in *try*. When naming multiple exceptions, use a tuple. It is legal to have multiple *except* statements, each of which names different types of exceptions. If no exceptions are named in the *except* statement, it will catch all exceptions; useful to put after *except* statements that name exceptions.

Including the *as* expression with the *except* statement allows you to store the exception object raised in the *try* block in a variable and use it within this block. To store an exception in a variable when none are named, name “Exception” instead (i.e. except Exception as e). Only one variable can be listed after *as*.

This block is **required**(even if all it has is *pass*).

1. **Explain finally block?**

Explain finally bock:

Python provides a keyword finally, which is always executed after try and except blocks. The finally block always executes after normal termination of try block or after try block terminates due to some exception.

This clause is executed no matter what, and is generally used to release external resources.

For example, we may be connected to a remote data center through the network or working with a file or a Graphical User Interface (GUI).

In all these circumstances, we must clean up the resource before the program comes to a halt whether it successfully ran or not. These actions (closing a file, GUI or disconnecting from network) are performed in the finally clause to guarantee the execution.

Syntax:

try:

# Some Code....

except:

# optional block

# Handling of exception (if required)

else:

# execute if no exception

finally:

# Some code .....(always executed)

1. **Explain else block?**

The code enters the else block only if the try clause does not raise an exception. Else block will execute only when no exception occurs.