Q1. What are the two latest user-defined exception constraints in Python 3.X?

The root class for all exceptions is the new exception Exception. From this, two additional classes are derived, StandardError, which is the root class for all standard exceptions, and SystemExit. It is recommended that user-defined exceptions in new code be derived from Exception, although for backward compatibility reasons, this is not required. Eventually this rule will be tightened.

SystemExit is derived from Exception because while it is an exception, it is not an error.

Most standard exceptions are direct descendants of StandardError. Some related exceptions are grouped together using an intermediate class derived from StandardError; this makes it possible to catch several different exceptions in one except clause, without using the tuple notation.

Sometimes we must enforce constraints on the values that specific program variables can take or save the program from running into an undesired state. In such situations, Python allows programmers to create User-defined Exceptions. To create a User-defined Exception, we need to create a class directly or indirectly derived from the built-in Exception class.

Let us see how we can create our exception in python:

Creating Exception Class:

Our Exception class should Implement Exceptions to raise exceptions. In this exception class, we can either put pass or give an implementation. Let us define init function.

Code:

class JustException(Exception):

def \_\_init\_\_(self, message):

print(message)

In the example above, JustException is a user-defined Exception. This class implements the Exception class. We have defined the \_\_init\_\_ function that takes the message of the String type as a parameter and prints the message. This JustException class can implement everything like a normal class in Python.

Raising User-defined Exception

When a certain condition is met, we use the raise keyword to throw an exception. To raise a User-defined Exception, we can do the following:

Code:

raise JustException("Raise an Exception")

The raise keyword raises the exception mentioned after it. In this case, JustException is raised.

Q2. How are class-based exceptions that have been raised matched to handlers?

while True:

try:

x = int(input("Please enter a number: "))

break

except ValueError:

print("Oops! That was no valid number. Try again...")

A try statement may have more than one except clause, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding try clause, not in other handlers of the same try statement. An except clause may name multiple exceptions as a parenthesized tuple, for example:

except (RuntimeError, TypeError, NameError):

... pass

Q3. Describe two methods for attaching context information to exception artefacts.

During the handling of one exception (exception A), it is possible that another exception (exception B) may occur. In today’s Python (version 2.4), if this happens, exception B is propagated outward and exception A is lost. In order to debug the problem, it is useful to know about both exceptions. The \_\_context\_\_ attribute retains this information automatically.

Sometimes it can be useful for an exception handler to intentionally re-raise an exception, either to provide extra information or to translate an exception to another type. The \_\_cause\_\_ attribute provides an explicit way to record the direct cause of an exception.

In today’s Python implementation, exceptions are composed of three parts: the type, the value, and the traceback. The sys module, exposes the current exception in three parallel variables, exc\_type, exc\_value, and exc\_traceback, the sys.exc\_info() function returns a tuple of these three parts, and the raise statement has a three-argument form accepting these three parts. Manipulating exceptions often requires passing these three things in parallel, which can be tedious and error-prone. Additionally, the except statement can only provide access to the value, not the traceback. Adding the \_\_traceback\_\_ attribute to exception values makes all the exception information accessible from a single place.

PEP 344 proposes three standard attributes on exception instances: the \_\_context\_\_ attribute for implicitly chained exceptions, the \_\_cause\_\_ attribute for explicitly chained exceptions, and the \_\_traceback\_\_ attribute for the traceback. A new raise ... from statement sets the \_\_cause\_\_ attribute.

Q4. Describe two methods for specifying the text of an exception object's error message.

add\_note(note)¶

Add the string note to the exception’s notes which appear in the standard traceback after the exception string. A TypeError is raised if note is not a string.

New in version 3.11.

\_\_notes\_\_

A list of the notes of this exception, which were added with add\_note(). This attribute is created when add\_note() is called.

New in version 3.11.

Q5. Why do you no longer use string-based exceptions?

Exceptions can be class objects or string objects. While traditionally, most exceptions have been string objects, in Python 1.5, all standard exceptions have been converted to class objects, and users are encouraged to do the same. The source code for those exceptions is present in the standard library module exceptions; this module never needs to be imported explicitly.

For backward compatibility, when Python is invoked with the -X option, most of the standard exceptions are strings2.7. This option may be used to run code that breaks because of the different semantics of class based exceptions. The -X option will become obsolete in future Python versions, so the recommended solution is to fix the code.

Two distinct string objects with the same value are considered different exceptions. This is done to force programmers to use exception names rather than their string value when specifying exception handlers. The string value of all built-in exceptions is their name, but this is not a requirement for user-defined exceptions or exceptions defined by library modules.

For class exceptions, in a try statement with an except clause that mentions a particular class, that clause also handles any exception classes derived from that class (but not exception classes from which it is derived). Two exception classes that are not related via subclassing are never equivalent, even if they have the same name.

The built-in exceptions listed below can be generated by the interpreter or built-in functions. Except where mentioned, they have an ``associated value'' indicating the detailed cause of the error. This may be a string or a tuple containing several items of information (e.g., an error code and a string explaining the code). The associated value is the second argument to the raise statement. For string exceptions, the associated value itself will be stored in the variable named as the second argument of the except clause (if any). For class exceptions, that variable receives the exception instance. If the exception class is derived from the standard root class Exception, the associated value is present as the exception instance's args attribute, and possibly on other attributes as well.

User code can raise built-in exceptions. This can be used to test an exception handler or to report an error condition ``just like'' the situation in which the interpreter raises the same exception; but beware that there is nothing to prevent user code from raising an inappropriate error.

The following exceptions are only used as base classes for other exceptions. When string-based standard exceptions are used, they are tuples containing the directly derived classes.

Exception:

The root class for exceptions. All built-in exceptions are derived from this class. All user-defined exceptions should also be derived from this class, but this is not (yet) enforced. The str() function, when applied to an instance of this class (or most derived classes) returns the string value of the argument or arguments, or an empty string if no arguments were given to the constructor. When used as a sequence, this accesses the arguments given to the constructor (handy for backward compatibility with old code). The arguments are also available on the instance's args attribute, as a tuple.