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

User-Defined Exceptions in Python

Some commonly raised Exceptions are ArithmeticError, AttributeError, ImportError, IOError, FileNotFoundError, etc.

Python detects all the critical errors that occur during Compile-time and Runtime. It stops the program's execution if the error occurs and raises an exception. Some commonly raised Exceptions are ArithmeticError, AttributeError, ImportError, IOError, FileNotFoundError.

**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.

### Catching Exception

As a usual exception, this exception can be handled using the **try-except** block.

**Code:**

try:

raise JustException("Raise an Exception")

except Exception as e:

print("Exception Raised")

**Output:**

Raise an Exception

Exception Raised

**Explanation:**

In the try block, we are raising the exception with the message. This message is printed from the \_\_init\_\_ function. In except block, we print the message rather than ending the program in an undesired state.

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

* **Classic exceptions**
* These exceptions can only be declared in the interfaces of methods or function modules using **EXCEPTIONS** and can be raised within such a procedure using the statements [**RAISE**](javascript:call_link('abapraise_exception.htm')) or [**MESSAGE RAISING**](javascript:call_link('abapmessage_raising.htm')). The procedure caller can use the addition **EXCEPTIONS** of the statements [**meth( ... )**](javascript:call_link('abapcall_method_static_short.htm')) or [**CALL FUNCTION**](javascript:call_link('abapcall_function.htm')) to assign return codes for the system field **sy-subrc** to the exceptions the caller wants to handle and evaluate them after the call.
* **Class-Based Exceptions**
* These exceptions are defined by exception classes, from which an exception object can be created when an exception is raised (if a handler uses the addition **INTO** in **CATCH**). A class-based exception can either cancel the current context or allow for a resume. Exceptions are raised using the statement **RAISE EXCEPTION** and handled using **CATCH** in a **TRY** control structure. Class-based exceptions can be raised in any procedures and can be further propagated by any procedures.

The coexistence of the two exception concepts is regulated as follows:

* Classic and class-based exceptions cannot be declared together in the interface of a procedure. Within a processing block, either only classic or only class-based exceptions can be raised.
* For reasons of interoperability, within a processing block class-based exceptions can be handled and evaluate the return values of function modules and methods using classic exceptions.
* **Example**
* The following source code shows the definition of an exception class, its declaration, and the raising in a method as well as its handling using **CATCH** after the call of the method in a **TRY** block.
* **CLASS cx\_application\_error DEFINITION  
    INHERITING FROM cx\_static\_check.  
  ENDCLASS.**
* **CLASS application DEFINITION.  
    PUBLIC SECTION.  
      METHODS do\_something  
        RAISING cx\_application\_error.  
  ENDCLASS.**
* **CLASS application IMPLEMENTATION.  
    METHOD do\_something.  
      ...  
      RAISE EXCEPTION TYPE cx\_application\_error.  
      ...  
    ENDMETHOD.  
  ENDCLASS.**
* **...**
* **... oref TYPE REF TO application.  
    
  ...  
    
  TRY.  
      oref->do\_something( ).  
      CATCH cx\_application\_error.  
        ...  
  ENDTRY.**

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

## [**Abstract**](https://peps.python.org/pep-3134/#abstract)

This PEP 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.

## [Explicit Exception Chaining](https://peps.python.org/pep-3134/#explicit-exception-chaining)

The \_\_cause\_\_ attribute on exception objects is always initialized to None. It is set by a new form of the raise statement:

raise EXCEPTION from CAUSE

which is equivalent to:

exc **=** EXCEPTION

exc**.**\_\_cause\_\_ **=** CAUSE

raise exc

In the following example, a database provides implementations for a few different kinds of storage, with file storage as one kind. The database designer wants errors to propagate as DatabaseError objects so that the client doesn’t have to be aware of the storage-specific details, but doesn’t want to lose the underlying error information.

class DatabaseError**(**Exception**):**

pass

class FileDatabase(Database):

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

try:

self.file = open(filename)

except IOError, exc:

raise DatabaseError('failed to open') from exc

If the call to open() raises an exception, the problem will be reported as a DatabaseError, with a \_\_cause\_\_ attribute that reveals the IOError as the original cause.

## [Enhanced Reporting](https://peps.python.org/pep-3134/#enhanced-reporting)

The default exception handler will be modified to report chained exceptions. The chain of exceptions is traversed by following the \_\_cause\_\_ and \_\_context\_\_ attributes, with \_\_cause\_\_ taking priority. In keeping with the chronological order of tracebacks, the most recently raised exception is displayed last; that is, the display begins with the description of the innermost exception and backs up the chain to the outermost exception. The tracebacks are formatted as usual, with one of the lines:

The above exception was the direct cause of the following exception**:**

or

During handling of the above exception**,** another exception occurred**:**

between tracebacks, depending whether they are linked by \_\_cause\_\_ or \_\_context\_\_ respectively. Here is a sketch of the procedure:

def print\_chain(exc):

if exc.\_\_cause\_\_:

print\_chain(exc.\_\_cause\_\_)

print '\nThe above exception was the direct cause...'

elif exc.\_\_context\_\_:

print\_chain(exc.\_\_context\_\_)

print '\nDuring handling of the above exception, ...'

print\_exc(exc)

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

## **Raising Exceptions**

The [raise](https://docs.python.org/3/reference/simple_stmts.html#raise) statement allows the programmer to force a specified exception to occur. For example:

>>>

>>> raise NameError('HiThere')

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: HiThere

The sole argument to [raise](https://docs.python.org/3/reference/simple_stmts.html#raise) indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from [Exception](https://docs.python.org/3/library/exceptions.html#Exception)). If an exception class is passed, it will be implicitly instantiated by calling its constructor with no arguments:

raise ValueError # shorthand for 'raise ValueError()'

If you need to determine whether an exception was raised but don’t intend to handle it, a simpler form of the [raise](https://docs.python.org/3/reference/simple_stmts.html#raise) statement allows you to re-raise the exception:

>>>

>>> try:

... raise NameError('HiThere')

... except NameError:

... print('An exception flew by!')

... raise

...

An exception flew by!

Traceback (most recent call last):

File "<stdin>", line 2, in <module>

NameError: HiThere

A [try](https://docs.python.org/3/reference/compound_stmts.html#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

A class in an [except](https://docs.python.org/3/reference/compound_stmts.html#except) clause is compatible with an exception if it is the same class or a base class thereof (but not the other way around — an except clause listing a derived class is not compatible with a base class). For example, the following code will print B, C, D in that order:

class B(Exception):

pass

class C(B):

pass

class D(C):

pass

for cls in [B, C, D]:

try:

raise cls()

except D:

print("D")

except C:

print("C")

except B:

print("B")

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

This is **because a string is a sequence**; we must assume that error messages are always more than two characters long!

## **The Simplest Way**

To capture the exceptions we use a code block called try-except. We put the piece of code that is suspected to be a source of error inside a try block then capture and design the response inside the except block. Check the following example where function calcArea is defined to receive a value from the users for theradius of a circle and returns the calculated area of the circle.

|  |  |
| --- | --- |
| def calcArea(radius): | |
|  | | pi = 3.1416 | |
|  | | * radius = float(radius) | |
|  | | * area = pi \* radius \*\* 2 | |
|  | | * return area | |
|  | |  | |
|  | | * for radius in [5, "a", 4, 8, "b", 0]: | |
|  | | * try: | |
|  | | * + area = calcArea(radius) | |
|  | | * + print("Area for radius {} = {}\n".format(radius, area)) | |
|  | | * except Exception as e: | |
|  | | * + print("Something is wrong with {}\n".format(radius)) | |

# **Understanding Exceptions as a Class**

I have mentioned several times in this post that Exceptions are classes but haven’t got much detail into that rather than showing some applications. Let’s try to do that in this section.

**In Python, all the Exceptions derive from the class called BaseException.**

Below BaseException all the built-in exceptions are arranged as a hierarchy. The main four subclasses under BaseException are:

* SystemExit
* KeyboardInterrupt
* GeneratorExit
* Exception

The exceptions that we commonly care about or want to take action upon are the ones under the Exception subclass. Exception subclass again contains several groups of subclasses.

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.