Q1. Describe three applications for exception processing.

An Exception is an error that happens during the execution of a program. Whenever there is an error, Python generates an exception that could be handled. It basically prevents the program from getting crashed.

Many a time, there are valid as well as invalid exceptions. Exceptions are convenient in many ways for handling errors and special conditions in a program. When you think that you have a code which can produce an error, you can use exception handling technique.

You can raise an exception in your program by using the raise exception statement. Raising an exception breaks current code execution and returns the exception back until it is handled.

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.

If you have some suspicious code that may raise an exception, you can defend your program by placing the suspicious code in a try: block. After the try: block, include an except: statement, followed by a block of code which handles the problem as elegantly as possible.

Here are the reasons for using exceptions in Python:

* Exception handling allows you to separate error-handling code from normal code.
* An exception is a Python object which represents an error.
* As with code comments, exceptions helps you to remind yourself of what the program expects.
* It clarifies the code and enhances readability.
* Allows you to stimulate consequences as the error-handling takes place at one place and in one manner.
* An exception is a convenient method for handling error messages.
* In Python, you can raise an exception in the program by using the raise exception method.
* Raising an exception helps you to break the current code execution and returns the exception back to expection until it is handled.
* Processing exceptions for components which can’t handle them directly.

Q2. What happens if you don't do something extra to treat an exception?

When an exception occurred, if you don't handle it, the program terminates abruptly and the code past the line that caused the exception will not get executed.

Q3. What are your options for recovering from an exception in your script?

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

If you have some suspicious code that may raise an exception, you can defend your program by placing the suspicious code in a try: block. After the try: block, include an except: statement, followed by a block of code which handles the problem as elegantly as possible.

A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions. You can also provide a generic except clause, which handles any exception. After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception. The else-block is a good place for code that does not need the try: block's protection.

Q4. Describe two methods for triggering exceptions in your script.

The 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 indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from BaseException, such as Exception or one of its subclasses). 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 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

Q5. Identify two methods for specifying actions to be executed at termination time, regardless of whether or not an exception exists.

In Python, keywords else and finally can also be used along with the try and except clauses. While the except block is executed if the exception occurs inside the try block, the else block gets processed if the try block is found to be exception free.

Syntax:

try:

#statements in try block

except:

#executed when error in try block

else:

#executed if try block is error-free

finally:

#executed irrespective of exception occured or not

Finally block always executes irrespective of an exception being thrown or not. The final keyword allows you to create a block of code that follows a try-catch block.

Finally, clause is optional. It is intended to define clean-up actions which should be that executed in all conditions.

try:

raise KeyboardInterrupt

finally:

print 'welcome, world!'

Output

Welcome, world!

KeyboardInterrupt

Finally, clause is executed before try statement.