**Q1. Describe three applications for exception processing.**

Integrating with external systems can introduce additional problems such as network errors involving firewalls, incorrect authentication or credentials, or system failures. An application that handles exceptions properly is critical to the success of any application.

Many errors can be anticipated at design time and addressed with specific actions to help ensure positive customer interaction. But unexpected errors may still occur and are just as important to address promptly.

|  |  |
| --- | --- |
| **Condition** | **Description** |
| When | The specified when rule returns true. |
| Queue When | If the specified when rule returns true, the request is queued and a Pega-specific SOAP fault that includes the queue ID of the request is returned. Useful when processing errors are due to temporary issues, such as item locking. |
| Mapping Error | An error occurs while mapping incoming data from the request message to the clipboard. |
| Security Error | Authentication fails. |
| Service error | A valid instance of the service activity cannot be found. |

Handling an exception

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.

Syntax

Here is simple syntax of *try....except...else* blocks −

try:

You do your operations here;

......................

except *ExceptionI*:

If there is ExceptionI, then execute this block.

except *ExceptionII*:

If there is ExceptionII, then execute this block.

......................

else:

If there is no exception then execute this block.

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

If you don't handle an exception, it will propagate up the call stack up to the interpreter, which will then display a traceback and exit.

import sys

try:

import feedparser

except:

print "Error: Cannot import feedparser.\n"

sys.exit(1)

# Exception Handling

## Basic form of handling exceptions

The standard way to handle exceptions is to use the try…except block. It’s pretty much like try…catch block in many other programming languages, if you have such a background.

The try clause includes the code that potentially raises an exception. If everything works well in the try clause, no code in the except clause will be executed. The try…except block is completed and the program will proceed.

However, if an exception is raised in the try clause, Python will stop executing any more code in that clause, and pass the exception to the except clause to see if this particular error is handled there.

Let’s take a look at a trivial example of the most basic form of exception handling:

|  |
| --- |
|  |
|  | >>> def divide\_twelve(number): |
|  | ... try: |
|  | ... print(f"Result: {12/number}") |
|  | ... except ZeroDivisionError: |
|  | ... print("You can't divide 12 by zero.") |
|  | ... |
|  | >>> # Use the function |
|  | >>> divide\_twelve(6) |
|  | Result: 2.0 |
|  | >>> divide\_twelve(0) |
|  | You can't divide 12 by zero.  **Q3. What are your options for recovering from an exception in your script?**  **Try Catch**  For anticipated exceptions, one may use try catch blocks.  try{  }catch($e){  }  **\_setRecovery**  \_setRecovery is no longer recommended and may be deprecated soon. Use the callback functions instead  The relevant APIs are:  \_setRecovery(fn) \_removeRecovery()  Any function can be assigned to a script as a recovery function. Once set, if there is an error during execution, the recovery function will be called before the script stops.  \_navigateTo("http://sahi.co.in/demo/");  function myRecoveryFn(){  \_alert("In myRecoveryFn."); // This statement will be alerted in case of script error.  }  \_setRecovery(myRecoveryFn); // Set the myRecoveryFn as recovery function.  \_click(\_link("Link Test")); // Works normally  \_click(\_link("Bad Link"));  // This statement fails and causes myRecoveryFn to be called.  \_alert("done");  // This statement will not be called, because script failed in the last statement.  The recovery function can be removed via \_removeRecovery();  \_setRecovery is no longer recommended and may be deprecated soon. Use the callback functions instead  **Taking screenshots**  Use \_focusWindow() and \_takeScreenShot() (available since Sahi Pro V4.3)  function onScriptError($e){  \_focusWindow();  \_takeScreenShot();  }  onScriptFailure = onScriptError;  **Q4. Describe two methods for triggering exceptions in your script.**  To avoid such a scenario, there are two methods to handle Python exceptions: Try – This method catches the exceptions raised by the program. Raise – Triggers an exception manually using custom exceptions.  One of the struggles developers face is how to catch all Python exceptions. Developers often categorize exceptions as coding mistakes that lead to errors when running the program. Some developers still fail to distinguish between errors and exceptions.  In the case of Python application development, a python program terminates as soon as it encounters an unhandled error. So, to establish the difference between errors and exceptions, there are two types of errors:   * Syntax errors * Logical errors (Exceptions)   First, let’s examine the syntax errors. Python syntax errors are caused by not following the proper structure (syntax) of the language. It is also known as a parsing error.  Here’s an example:  >>> ages = {      ‘jj’: 2,      ‘yoyo’: 4  }  print(f’JJ is {ages[“jj”]} years old.’)  The output:  JJ is 2 years old.  This is a simple code with no syntax error. Then we will add another variable tomtom:  >>> ages = {      ‘jj’: 2,      ‘yoyo’: 4      ‘tomtom’: 6  }  print(f’JJ is {ages[“jj”]} years old.’)  Upon inspection, you can see an invalid syntax on the second entry, yoyo, inside the array with a missing comma. Try to run this code, and you will get a traceback:  File “<pyshell>”, line 1      >>> ages = {       ^  SyntaxError: invalid syntax  As you notice, the traceback message didn’t pinpoint the exact line where the syntax error occurs. The Python interpreter only attempts to locate where the invalid syntax is. It only points to where it first noticed the problem. So, when you get a SyntaxError traceback, it means that you should visually inspect whether the interpreter is pointing to the right error.  In the example above, the interpreter encounters an invalid syntax inside an array called ages. Thus, it points out that there is something wrong inside the array.  This is a syntax error.  In most cases, a Python developer writes an impressive piece of code that is ready to execute. The program becomes a robust machine learning model, but during execution, Python throws up an unexpected error. Unfortunately, it is no longer the typical syntax error. Developers are now dealing with logical errors, also known as ***exceptions***. |

Here are common exceptions in Python:

* **IndexError** – You will get this error when an index is not found in a sequence. For instance, accessing the 6th index when the length of the list is only five(5).
* **IndentationError** – Happens when indentation is not specified correctly.
* **ValueError** – Occurs when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values.
* **IOError**– Developers often encounter this error when an input/output operation fails.
* **Arithmetic Error** – Occurs when numeric calculations fail.
* **Floating-point Error** – Happens when a floating-point calculation fails.
* **Assertion Error** – Occurs when there is assert statement failure.
* **Overflow Error** – Developers get this error if the result of an arithmetic operation is too large and becomes machine unreadable.
* **Type Error** – Happens when an incorrect type of function or operation is applied to an object.

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

**Python Exception Handling Mechanism**

Exception handling is managed by the following 5 keywords:

1. **try**
2. **catch**
3. **finally**
4. **throw**

**Python Try Statement**

A try statement includes keyword try, followed by a colon (:) and a suite of code in which exceptions may occur. It has one or more clauses.

Syntax:

try:

statement(s)

**The catch Statement**

Catch blocks take one argument at a time, which is the type of exception that it is likely to catch. These arguments may range from a specific type of exception which can be varied to a catch-all category of exceptions.

Rules for catch block:

* You can define a catch block by using the keyword catch
* Catch Exception parameter is always enclosed in parentheses
* It always represents the type of exception that catch block handles.
* An exception handling code is written between two {} curly braces.
* You can place multiple catch block within a single try block.
* You can use a catch block only after the try block.
* All the catch block should be ordered from subclass to superclass exception.

**Example:**

try

}

catch (ArrayIndexOutOfBoundsException e) {

System.err.printin("Caught first " + e.getMessage()); } catch (IOException e) {

System.err.printin("Caught second " + e.getMessage());

}

### Finally Statement in Python

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