Python AWS Training

Python

Authorized & published by Summitworks Technologies Inc

SummitWorks™
GLOBAL SOLUTION ARCHITECTS

Exception Management

- Python Exceptions
  - Types of Python Exceptions
  - Handling Exceptions with try/except/finally
  - Triggering Exceptions with raise
  - Defining New Exception Types
  - Implementing Exception Handling in Functions, Methods and Classes
- Live Examples

What is an Exception?

- An exception is an error that happens during execution of a program. When error occurs, Python generate an exception that can be handled, which avoids your program to crash.

What is an Exception?

- 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 can't 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 would terminate and come out.

Example

- **>>>** l = [1,2,3]
- **>>>** l['apples'] o/p: TypeError: list indices must be integers, **not** str

```
>>> n = int(input("Please enter a number: "))
 Please enter a number: 23.5
 Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10:
'23.5'
```

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.
```

Important points about syntax:

- 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.

The *except* clause with no exceptions:

- You can also use the except statement with no exceptions defined as follows:

```
try:
You do your operations here;
.....................
except:
If there is any exception, then execute this block.
.....................
else:
If there is no exception then execute this block.
```

Problem

- This kind of a **try-except** statement catches all the exceptions that occur. Using this kind of try-except statements is not considered a good programming practice though, because it catches all exceptions and does not make the programmer identify the root cause of the problem that may occur.

The *except* clause with multiple exceptions:

- You can also use the same *except* statement to handle multiple exceptions as follows:

```
try:
You do your operations here;
.....................
except(Exception1[, Exception2[,...ExceptionN]]]):
If there is any exception from the given exception list,
then execute this block.
.....................
else:
If there is no exception then execute this block.
```

Raise

- You can explicitly throw an exception in Python using “raise” statement. raise will cause an exception to occur and thus execution control will stop in case it is not handled.

```
raise Exception_class<value>
```

Explanation

- To raise an exception, raise statement is used. It is followed by exception class name.
- Exception can be provided with a value that can be given in the parenthesis. (here, Hello)
- To access the value "as" keyword is used. "e" is used as a reference variable which stores the value of the exception.

As-keyword.

- We can name a variable within an except statement. We use the as-keyword for this. Here we name the IOError "err" and can use it within the clause.

```
try:
f = open("does-not-exist")
except IOError as err:
# We can use IOError as an instance.
print("Error:", err)
print("Number:", err.errno)

Output Error: [Errno 2] No such file or directory: 'does-not-exist'
Number: 2
```

The try---finally clause:

- You can use a **finally:** block along with a **try:** block. The finally block is a place to put any code that must execute, whether the try-block raised an exception or not. The syntax of the try-finally statement is this:

```
try:
You do your operations here;
.....................
Due to any exception, this may be skipped.
finally:
This would always be executed.
.....................
```

Try except finally else

```
>>> try:
        ... f1(1) ...
 except:
        ... print "caught an exception“
 ... else:
        ... print "no exception" ...
finally:
         ... print "the end"
```

Custom Exceptions

- We can add user-defined exceptions by creating a new class in Python. The trick here is to derive the custom exception class from the base exception class. Most of the built-in exceptions use the same technique to enforce their exceptions.
- Example:

```
>>> class UserDefinedError(Exception):
...     pass
>>> raise UserDefinedError
Traceback (most recent call last):
...
__main__.UserDefinedError
>>> raise UserDefinedError("An error occurred")
Traceback (most recent call last):
...
__main__.UserDefinedError: An error occurred
```

List Of Python Built-In Exceptions

Exception	Cause of Error
AssertionError	If the assert statement fails.
AttributeError	When an attribute assignment or the reference fails.
EOFError	If there is no input or the file pointer is at EOF.
Exception	It is the base class for all exceptions.
EnvironmentError	For errors that occur outside the Python environment.
FloatingPointError	When floating point operation fails.
GeneratorExit	If a generator's <close()> method gets called.
ImportError	When the imported module is not available.
IOError	If an input/output operation fails.
IndexError	When the index of a sequence is out of range.

List Of Python Built-In Exceptions.

Exception	Cause of Error
KeyboardInterrupt	When the user hits an interrupt key (Ctrl+c or delete).
MemoryError	If an operation runs out of memory.
NameError	When a variable is not available in local or global scope.
NotImplementedError	If an abstract method isn't available.
OSError	When a system operation fails.
OverflowError	If the result of an arithmetic operation exceeds the range.
ReferenceError	When a weak reference proxy accesses a garbage collected reference.
RuntimeError	If the generated error doesn't fall under any category.
StandardError	It is a base class for all built-in exceptions except <StopIteration> and <SystemExit>.
StopIteration	The <next()> function has no further item to be returned.
SyntaxError	For errors in Python syntax.
IndentationError	When indentation is not proper.

List Of Python Built-In Exceptions

Exception	Cause of Error
SystemExit	The <sys.exit()> function raises it.
TypeError	When a function is using an object of the incorrect type.
UnboundLocalError	If the code using an unassigned reference gets executed.
UnicodeError	For a Unicode encoding or decoding error.
ValueError	When a function receives invalid values.
ZeroDivisionError	If the second operand of division or modulo operation is zero.

Assert

- When it encounters an assert statement, Python evaluates the accompanying expression, which is hopefully true. If the expression is false, Python raises an *AssertionError exception*.

syntax

```
assert Expression[, Arguments]
```

- If the assertion fails, Python uses ArgumentExpression as the argument for the AssertionError.
- AssertionError exceptions can be caught and handled like any other exception using the try-except statement, but if not handled, they will terminate the program and produce a traceback.

Use of Assertions

- The easiest way to think of an assertion is to liken it to a **raise-if** statement (or to be more accurate, a raise-if-not statement). An expression is tested, and if the result comes up false, an exception is raised.

- Programmers often place assertions at the start of a function to check for valid input, and after a function call to check for valid output.

Exception hierarchy

- In Python, all exceptions must be instances of a class that derives from BaseException. In a try statement with an except clause that mentions a particular class

https://docs.python.org/3/library/exceptions.html#exception-hierarchy

Points to remember while working on exception handling

- **Never use exceptions for flow-control:** When used for flow-control, exceptions are like goto. There might be a few esoteric cases in which they're appropriate, but 99.99% of the time they are not.
- Handle exceptions at the level that knows how to handle them
- Do not expose implementation details with exceptions
- Document the exceptions thrown by your code

Handle exceptions at the level that knows how to handle them

- By their very nature, exceptions can propagate up a hierarchy and can be caught at multiple levels. A question rises - where is it appropriate to catch and handle an exception ?
- The best place is that piece of code that can handle the exception. For some exceptions, like programming errors (e.g. IndexError, TypeError, NameError etc.), exceptions are best left to the programmer / user, because "handling" them will just hide real bugs.
- If you have a complete application, it is not appropriate to just fail with an exception, of course. It's better to demonstrate a polite error message (or dialog) and thoroughly log the exception itself, which can later help in the investigation of the bug.
- So before writing try/except, always ask yourself - "is this the right place to handle this exception ? do I have enough information to handle it here ?".
- This is also the reason why you should be extremely careful with except: clauses that catch everything. These will not only catch the exceptions you intended, but all of them.

Do not expose implementation details with exceptions

- We can achieve this by using customized exceptions
- class StuffCachingError(Exception): pass

➢You didn't expose the internal implementation to outside code. Now users of do_stuff can install handlers for StuffCachingError, without knowing you use files under the hood.

➢If the user wants to investigate the error in depth, you've also provided the original exception in the error message, and he can do so.

```
def do_stuff():
    try:
        cache = open(filename)
        # do stuff with cache
    except IOError, e:
        raise StuffCachingError('Caching error: %s' % e)
Benefits:
```

Document the exceptions thrown by your code

- What it expects from its argument or the state of the world when invoked
- The result(s) it returns
- Its side effects on the world
- The exceptions it throws

Any Queries