

DOUG HELLMANN

Python, OpenStack, and Open Source

2009-06-19 BY DOUG

Python Exception Handling Techniques

Error reporting and processing through exceptions is one of Python's key features. Care must be taken when handling exceptions to ensure proper application cleanup while maintaining useful error reporting.

Error reporting and processing through exceptions is one of Python's key features. Unlike C, where the common way to report errors is through function return values that then have to be checked on every invocation, in Python a programmer can raise an exception at any point in a program. When the exception is raised, program execution is interrupted as the interpreter searches back up the stack to find a context with an exception handler. This search algorithm allows error handling to be organized cleanly in a central or high-level place within the program structure. Libraries may not need to do any exception handling at all, and simple scripts can frequently get away with wrapping a portion of the main program in an exception handler to print a nicely formatted error. Proper exception handling in more complicated situations can be a little tricky, though, especially in cases where the program has to clean up after itself as the exception propagates back up the stack.



RECENT POSTS

sphinxcontrib-spelling 5.2.0
2020-07-28

beagle 0.3.0
2020-07-07

sphinxcontrib-spelling 5.1.2
2020-06-22

sphinxcontrib-spelling 5.1.0
2020-06-05

beagle 0.2.2
2020-05-16

LINKS

[About Me](#)

[Python Module of the Week](#)

Throwing and Catching

The statements used to deal with exceptions are `raise` and `except`. Both are language keywords. The most common form of throwing an exception with `raise` uses an instance of an exception class.

```
1  #!/usr/bin/env python
2
3  def throws():
4      raise RuntimeError('this is the error message')
5
6  def main():
7      throws()
8
9  if __name__ == '__main__':
10     main()
```

The arguments needed by the exception class vary, but usually include a message string to explain the problem encountered.

If the exception is left unhandled, the default behavior is for the interpreter to print a full traceback and the error message included in the exception.

```
1  $ python throwing.py
2  Traceback (most recent call last):
3    File "throwing.py", line 10, in <module>
4      main()
5    File "throwing.py", line 7, in main
6      throws()
```

The Python 3 Standard Library by Example

The Python 2 Standard Library by Example

Long Form Posts

Presentations

Book Reviews

CONTINUE THE CONVERSATION

I don't have comments enabled, but you can find me on Twitter [@doughellmann](#).

ARCHIVES

Select Month ▼

CATEGORIES

Book Reviews

Long Form Posts

Posts

Presentations

Release

```
7 File "throwing.py", line 4, in throws
8     raise RuntimeError('this is the error message')
9 RuntimeError: this is the error message
```

For some scripts this behavior is sufficient, but it is nicer to catch the exception and print a more user-friendly version of the error.

```
1 #!/usr/bin/env python
2
3 import sys
4
5 def throws():
6     raise RuntimeError('this is the error message')
7
8 def main():
9     try:
10         throws()
11         return 0
12     except Exception, err:
13         sys.stderr.write('ERROR: %sn' % str(err))
14         return 1
15
16 if __name__ == '__main__':
17     sys.exit(main())
```

In the example above, all exceptions derived from `Exception` are caught, and just the error message is printed to `stderr`. The program follows the Unix convention of returning an exit code indicating whether there was an error or not.

```
$ python catching.py  
ERROR: this is the error message
```

Logging Exceptions

For daemons or other background processes, printing directly to `stderr` may not be an option. The file descriptor might have been closed, or it may be redirected somewhere that errors are hard to find. A better option is to use the logging module to log the error, including the full traceback.

```
1  #!/usr/bin/env python  
2  
3  import logging  
4  import sys  
5  
6  def throws():  
7      raise RuntimeError('this is the error message')  
8  
9  def main():  
10     logging.basicConfig(level=logging.WARNING)  
11     log = logging.getLogger('example')  
12     try:  
13         throws()  
14         return 0  
15     except Exception, err:  
16         log.exception('Error from throws():')  
17         return 1  
18  
19 if __name__ == '__main__':  
20     sys.exit(main())
```

In this example, the logger is configured to use the default behavior of sending its output to `stderr`, but that can easily be adjusted. Saving tracebacks to a log file can make it easier to debug problems that are otherwise hard to reproduce outside of a production environment.

```
1 $ python logging_errors.py
2 ERROR:example:Error from throws():
3 Traceback (most recent call last):
4   File "logging_errors.py", line 13, in main
5     throws()
6   File "logging_errors.py", line 7, in throws
7     raise RuntimeError('this is the error message')
8 RuntimeError: this is the error message
```

Cleaning Up and Re-raising

In many programs, simply reporting the error isn't enough. If an error occurs part way through a lengthy process, you may need to undo some of the work already completed. For example, changes to a database may need to be rolled back or temporary files may need to be deleted. There are two ways to handle cleanup operations, using a `finally` stanza coupled to the exception handler, or within an explicit exception handler that raises the exception after cleanup is done.

For cleanup operations that should always be performed, the simplest implementation is to use `try:finally`. The `finally` stanza is guaranteed to be run, even if the code inside the `try` block raises an exception.

```
1  #!/usr/bin/env python
2
3  import sys
4
5  def throws():
6      print 'Starting throws()'
7      raise RuntimeError('this is the error message')
8
9  def main():
10     try:
11         try:
12             throws()
13             return 0
14         except Exception, err:
15             print 'Caught an exception'
16             return 1
17     finally:
18         print 'In finally block for cleanup'
19
20 if __name__ == '__main__':
21     sys.exit(main())
```

This old-style example wraps a `try:except` block with a `try:finally` block to ensure that the cleanup code is called no matter what happens inside the main program.

```
$ python try_finally_oldstyle.py
Starting throws()
Caught an exception
In finally block for cleanup
```

While you may continue to see that style in older code, since Python 2.5 it has been possible to combine `try:except` and `try:finally` blocks into a single level. Since the newer style uses fewer levels of indentation and the resulting code is easier to read, it is being adopted quickly.

```
1  #!/usr/bin/env python
2
3  import sys
4
5  def throws():
6      print 'Starting throws()'
7      raise RuntimeError('this is the error message')
8
9  def main():
10     try:
11         throws()
12         return 0
13     except Exception, err:
14         print 'Caught an exception'
15         return 1
16     finally:
17         print 'In finally block for cleanup'
18
19  if __name__ == '__main__':
20     sys.exit(main())
```

The resulting output is the same:

```
$ python try_finally.py
Starting throws()
```

```
Caught an exception
In finally block for cleanup
```

Re-raising Exceptions

Sometimes the cleanup action you need to take for an error is different than when an operation succeeds. For example, with a database you may need to rollback the transaction if there is an error but commit otherwise. In such cases, you will have to catch the exception and handle it. It may be necessary to catch the exception in an intermediate layer of your application to undo part of the processing, then throw it again to continue propagating the error handling.

```
1  #!/usr/bin/env python
2  """Illustrate database transaction management using sqlite3.
3  """
4
5  import logging
6  import os
7  import sqlite3
8  import sys
9
10 DB_NAME = 'mydb.sqlite'
11 logging.basicConfig(level=logging.INFO)
12 log = logging.getLogger('db_example')
13
14 def throws():
15     raise RuntimeError('this is the error message')
16
17 def create_tables(cursor):
18     log.info('Creating tables')
19     cursor.execute("create table module (name text, description text)")
```



```
20
21 def insert_data(cursor):
22     for module, description in [('logging', 'error reporting and auditing'),
23                                 ('os', 'Operating system services'),
24                                 ('sqlite3', 'SQLite database access'),
25                                 ('sys', 'Runtime services'),
26                                 ]:
27         log.info('Inserting %s (%s)', module, description)
28         cursor.execute("insert into module values (?, ?)", (module, description))
29     return
30
31 def do_database_work(do_create):
32     db = sqlite3.connect(DB_NAME)
33     try:
34         cursor = db.cursor()
35         if do_create:
36             create_tables(cursor)
37             insert_data(cursor)
38             throws()
39     except:
40         db.rollback()
41         log.error('Rolling back transaction')
42         raise
43     else:
44         log.info('Committing transaction')
45         db.commit()
46     return
47
48 def main():
49     do_create = not os.path.exists(DB_NAME)
50     try:
51         do_database_work(do_create)
52     except Exception, err:
53         log.exception('Error while doing database work')
```

```
54         return 1
55     else:
56         return 0
57
58 if __name__ == '__main__':
59     sys.exit(main())
```

This example uses a separate exception handler in `do_database_work()` to undo the changes made in the database, then a global exception handler to report the error message.

```
1  $ python sqlite_error.py
2  INFO:db_example:Creating tables
3  INFO:db_example:Inserting logging (error reporting and auditing)
4  INFO:db_example:Inserting os (Operating system services)
5  INFO:db_example:Inserting sqlite3 (SQLite database access)
6  INFO:db_example:Inserting sys (Runtime services)
7  ERROR:db_example:Rolling back transaction
8  ERROR:db_example:Error while doing database work
9  Traceback (most recent call last):
10     File "sqlite_error.py", line 51, in main
11         do_database_work(do_create)
12     File "sqlite_error.py", line 38, in do_database_work
13         throws()
14     File "sqlite_error.py", line 15, in throws
15         raise RuntimeError('this is the error message')
16 RuntimeError: this is the error message
```

Preserving Tracebacks

Frequently the cleanup operation itself introduces another opportunity for an error condition in your program. This is especially the case when a system runs out of resources (memory, disk space, etc.). Exceptions raised from within an exception handler can mask the original error if they aren't handled locally.

```
1  #!/usr/bin/env python
2
3  import sys
4  import traceback
5
6  def throws():
7      raise RuntimeError('error from throws')
8
9  def nested():
10     try:
11         throws()
12     except:
13         cleanup()
14         raise
15
16  def cleanup():
17      raise RuntimeError('error from cleanup')
18
19  def main():
20     try:
21         nested()
22         return 0
23     except Exception, err:
24         traceback.print_exc()
25         return 1
26
27  if __name__ == '__main__':
28     sys.exit(main())
```

When `cleanup()` raises an exception while the original error is being processed, the exception handling machinery is reset to deal with the new error.

```
1 $ python masking_exceptions.py
2 Traceback (most recent call last):
3   File "masking_exceptions.py", line 21, in main
4     nested()
5   File "masking_exceptions.py", line 13, in nested
6     cleanup()
7   File "masking_exceptions.py", line 17, in cleanup
8     raise RuntimeError('error from cleanup')
9 RuntimeError: error from cleanup
```

Even catching the second exception does not guarantee that the original error message will be preserved.

```
1 #!/usr/bin/env python
2
3 import sys
4 import traceback
5
6 def throws():
7     raise RuntimeError('error from throws')
8
9 def nested():
10    try:
11        throws()
12    except:
```

```

13         try:
14             cleanup()
15         except:
16             pass # ignore errors in cleanup
17             raise # we want to re-raise the original error
18
19 def cleanup():
20     raise RuntimeError('error from cleanup')
21
22 def main():
23     try:
24         nested()
25         return 0
26     except Exception, err:
27         traceback.print_exc()
28         return 1
29
30 if __name__ == '__main__':
31     sys.exit(main())

```

Here, even though we have wrapped the `cleanup()` call in an exception handler that ignores the exception, the error in `cleanup()` hides the original error because only one exception context is maintained.

```

1 $ python masking_exceptionsCatch.py
2 Traceback (most recent call last):
3   File "masking_exceptionsCatch.py", line 24, in main
4     nested()
5   File "masking_exceptionsCatch.py", line 14, in nested
6     cleanup()
7   File "masking_exceptionsCatch.py", line 20, in cleanup

```

```
8     raise RuntimeError('error from cleanup')
9 RuntimeError: error from cleanup
```

A naive solution is to catch the original exception and retain it in a variable, then re-raise it explicitly.

```
1  #!/usr/bin/env python
2
3  import sys
4  import traceback
5
6  def throws():
7      raise RuntimeError('error from throws')
8
9  def nested():
10     try:
11         throws()
12     except Exception, original_error:
13         try:
14             cleanup()
15         except:
16             pass # ignore errors in cleanup
17         raise original_error
18
19  def cleanup():
20      raise RuntimeError('error from cleanup')
21
22  def main():
23     try:
24         nested()
25         return 0
26     except Exception, err:
```

```
27     traceback.print_exc()
28     return 1
29
30 if __name__ == '__main__':
31     sys.exit(main())
```

As you can see, this does not preserve the full traceback. The stack trace printed does not include the `throws()` function at all, even though that is the original source of the error.

```
1 $ python masking_exceptions_reraise.py
2 Traceback (most recent call last):
3   File "masking_exceptions_reraise.py", line 24, in main
4     nested()
5   File "masking_exceptions_reraise.py", line 17, in nested
6     raise original_error
7 RuntimeError: error from throws
```

A better solution is to re-raise the original exception *first*, and handle the clean up in a `try:finally` block.

```
1 #!/usr/bin/env python
2
3 import sys
4 import traceback
5
6 def throws():
7     raise RuntimeError('error from throws')
8
9 def nested():
```

```
10     try:
11         throws()
12     except Exception, original_error:
13         try:
14             raise
15         finally:
16             try:
17                 cleanup()
18             except:
19                 pass # ignore errors in cleanup
20
21 def cleanup():
22     raise RuntimeError('error from cleanup')
23
24 def main():
25     try:
26         nested()
27         return 0
28     except Exception, err:
29         traceback.print_exc()
30         return 1
31
32 if __name__ == '__main__':
33     sys.exit(main())
```

This construction prevents the original exception from being overwritten by the latter, and preserves the full stack in the traceback.

```
1 $ python masking_exceptions_finally.py
2 Traceback (most recent call last):
3   File "masking_exceptions_finally.py", line 26, in main
```



```
4     nested()
5     File "masking_exceptions_finally.py", line 11, in nested
6         throws()
7     File "masking_exceptions_finally.py", line 7, in throws
8         raise RuntimeError('error from throws')
9 RuntimeError: error from throws
```

The extra indentation levels aren't pretty, but it gives the output we want. The error reported is for the original exception, including the full stack trace.

See also

Errors and Exceptions

The standard library documentation tutorial on handling errors and exceptions in your code.

PyMOTW: exceptions

Python Module of the Week article about the exceptions module.

exceptions module

Standard library documentation about the exceptions module.

PyMOTW: logging

Python Module of the Week article about the logging module.

logging module

Standard library documentation about the logging module.

PREVIOUS

← **PyMOTW: gettext**

NEXT

PyMOTW: robotparser →

Proudly powered by WordPress