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.

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.

```
#!/usr/bin/env python

def throws():
    raise RuntimeError('this is the error message')

def main():
    throws()
```

```
if __name__ == '__main__':
    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.

```
$ python throwing.py
Traceback (most recent call last):
  File "throwing.py", line 10, in <module>
    main()
  File "throwing.py", line 7, in main
    throws()
  File "throwing.py", line 4, in throws
    raise RuntimeError('this is the error message')
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.

```
#!/usr/bin/env python

import sys

def throws():
    raise RuntimeError('this is the error message')

def main():
    try:
        throws()
        return 0
    except Exception, err:
        sys.stderr.write('ERROR: %sn' % str(err))
        return 1

if __name__ == '__main__':
    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.

```
#!/usr/bin/env python

import logging
import sys

def throws():
    raise RuntimeError('this is the error message')

def main():
    logging.basicConfig(level=logging.WARNING)
    log = logging.getLogger('example')
    try:
        throws()
        return 0
    except Exception, err:
        log.exception('Error from throws():')
        return 1

if __name__ == '__main__':
    sys.exit(main())
```

In this example, the logger is configured to 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.

```
$ python logging_errors.py
ERROR:example:Error from throws():
```

```
Traceback (most recent call last):
  File "logging_errors.py", line 13, in main
    throws()
  File "logging_errors.py", line 7, in throws
    raise RuntimeError('this is the error message')
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.

```
#!/usr/bin/env python

import sys

def throws():
    print 'Starting throws()'
    raise RuntimeError('this is the error message')

def main():
    try:
        try:
            throws()
            return 0
        except Exception, err:
            print 'Caught an exception'
            return 1
    finally:
        print 'In finally block for cleanup'

if __name__ == '__main__':
    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.

```
#!/usr/bin/env python

import sys

def throws():
    print 'Starting throws()'
    raise RuntimeError('this is the error message')

def main():
    try:
        throws()
        return 0
    except Exception, err:
        print 'Caught an exception'
        return 1
    finally:
        print 'In finally block for cleanup'

if __name__ == '__main__':
    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.

```
#!/usr/bin/env python
"""Illustrate database transaction management using sqlite3.
"""

import logging
import os
import sqlite3
import sys

DB_NAME = 'mydb.sqlite'
logging.basicConfig(level=logging.INFO)
log = logging.getLogger('db_example')

def throws():
    raise RuntimeError('this is the error message')

def create_tables(cursor):
    log.info('Creating tables')
    cursor.execute("create table module (name text, description text)")

def insert_data(cursor):
    for module, description in [('logging', 'error reporting and auditing'),
                                ('os', 'Operating system services'),
                                ('sqlite3', 'SQLite database access'),
                                ('sys', 'Runtime services'),
                                ]:
        log.info('Inserting %s (%s)', module, description)
        cursor.execute("insert into module values (?, ?)", (module, description))
    return

def do_database_work(do_create):
    db = sqlite3.connect(DB_NAME)
    try:
        cursor = db.cursor()
        if do_create:
            create_tables(cursor)
        insert_data(cursor)
        throws()
    except:
        db.rollback()
        log.error('Rolling back transaction')
        raise
    else:
        log.info('Committing transaction')
        db.commit()
    return

def main():
    do_create = not os.path.exists(DB_NAME)
    try:
        do_database_work(do_create)
    except Exception, err:
        log.exception('Error while doing database work')
        return 1
    else:
        return 0

if __name__ == '__main__':
    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.

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

```
#!/usr/bin/env python

import sys
import traceback

def throws():
    raise RuntimeError('error from throws')

def nested():
    try:
        throws()
    except:
        cleanup()
        raise

def cleanup():
    raise RuntimeError('error from cleanup')

def main():
    try:
        nested()
        return 0
    except Exception, err:
        traceback.print_exc()
        return 1

if __name__ == '__main__':
    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.

```
$ python masking_exceptions.py
Traceback (most recent call last):
  File "masking_exceptions.py", line 21, in main
    nested()
  File "masking_exceptions.py", line 13, in nested
    cleanup()
  File "masking_exceptions.py", line 17, in cleanup
    raise RuntimeError('error from cleanup')
RuntimeError: error from cleanup
```

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

```
#!/usr/bin/env python

import sys
import traceback

def throws():
    raise RuntimeError('error from throws')

def nested():
    try:
        throws()
    except:
        try:
            cleanup()
        except:
            pass # ignore errors in cleanup
        raise # we want to re-raise the original error

def cleanup():
    raise RuntimeError('error from cleanup')

def main():
    try:
        nested()
        return 0
    except Exception, err:
        traceback.print_exc()
        return 1

if __name__ == '__main__':
    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.

```
$ python masking_exceptions_catch.py
Traceback (most recent call last):
  File "masking_exceptions_catch.py", line 24, in main
    nested()
  File "masking_exceptions_catch.py", line 14, in nested
    cleanup()
  File "masking_exceptions_catch.py", line 20, in cleanup
    raise RuntimeError('error from cleanup')
RuntimeError: error from cleanup
```

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

```
#!/usr/bin/env python

import sys
import traceback

def throws():
    raise RuntimeError('error from throws')

def nested():
    try:
        throws()
    except Exception, original_error:
        try:
            cleanup()
        except:
            pass # ignore errors in cleanup
        raise original_error

def cleanup():
    raise RuntimeError('error from cleanup')

def main():
    try:
        nested()
        return 0
    except Exception, err:
        traceback.print_exc()
        return 1

if __name__ == '__main__':
    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.

```
$ python masking_exceptions_reraise.py
Traceback (most recent call last):
  File "masking_exceptions_reraise.py", line 24, in main
    nested()
  File "masking_exceptions_reraise.py", line 17, in nested
    raise original_error
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.

```
#!/usr/bin/env python

import sys
import traceback

def throws():
    raise RuntimeError('error from throws')

def nested():
    try:
        throws()
    except Exception, original_error:
        try:
            raise
        finally:
            try:
                cleanup()
            except:
                pass # ignore errors in cleanup

def cleanup():
    raise RuntimeError('error from cleanup')

def main():
    try:
        nested()
        return 0
    except Exception, err:
        traceback.print_exc()
        return 1

if __name__ == '__main__':
    sys.exit(main())
```

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

```
$ python masking_exceptions_finally.py
Traceback (most recent call last):
  File "masking_exceptions_finally.py", line 26, in main
    nested()
  File "masking_exceptions_finally.py", line 11, in nested
    throws()
  File "masking_exceptions_finally.py", line 7, in throws
    raise RuntimeError('error from throws')
RuntimeError: error from throws
```

The extra indention 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.

2009-06-19 doug Long Form Posts python