In this tutorial you'll learn how to handle error conditions in Python from a whole system point of view. Error handling is a critical aspect of design, and it crosses from the lowest levels (sometimes the hardware) all the way to the end users. If you don't have a consistent strategy in place, your system will be unreliable, the user experience will be poor, and you'll have a lot of challenges debugging and troubleshooting.

The key to success is being aware of all these interlocking aspects, considering them explicitly, and forming a solution that addresses each point.

# Status Codes vs. Exceptions

There are two main error handling models: status codes and exceptions. Status codes can be used by any programming language. Exceptions require language/runtime support.

Python supports exceptions. Python and its standard library use exceptions liberally to report on many exceptional situations like IO errors, divide by zero, out of bounds indexing, and also some not so exceptional situations like end of iteration (although it is hidden). Most libraries follow suit and raise exceptions.

That means your code will have to handle the exceptions raised by Python and libraries anyway, so you may as well raise exceptions from your code when necessary and not rely on status codes.

# Quick Example

Before diving into the inner sanctum of Python exceptions and error handling best practices, let's see some exception handling in action:

```
01  def f():
02
03      return 4 / 0
04
05
06
07  def g():
08
09      raise Exception("Don't call us. We'll call you")
10
11
12
13  def h():
14
15      try:
16
17          f()
18
19      except Exception as e:
20
21          print(e)
22
23      try:
24
25          g()
26
27      except Exception as e:
28
29          print(e)
```

Here is the output when calling `h()`:

```
1  h()
2
3  division by zero
4
5  Don't call us. We'll call you
```

# Python Exceptions

Python exceptions are objects organized in a class hierarchy.

Here is the whole hierarchy:

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StandardError
      |    +-- BufferError
      |    +-- ArithmeticError
      |    |    +-- FloatingPointError
      |    |    +-- OverflowError
      |    |    +-- ZeroDivisionError
      |    +-- AssertionError
      |    +-- AttributeError
      |    +-- EnvironmentError
      |    |    +-- IOError
      |    |    +-- OSError
      |    |         +-- WindowsError (Windows)
      |    |         +-- VMSError (VMS)
      |    +-- EOFError
      |    +-- ImportError
      |    +-- LookupError
      |    |    +-- IndexError
      |    |    +-- KeyError
      |    +-- MemoryError
      |    +-- NameError
      |    |    +-- UnboundLocalError
      |    +-- ReferenceError
      |    +-- RuntimeError
      |    |    +-- NotImplementedError
      |    +-- SyntaxError
      |    |    +-- IndentationError
      |    |         +-- TabError
      |    +-- SystemError
      |    +-- TypeError
      |    +-- ValueError
      |         +-- UnicodeError
      |              +-- UnicodeDecodeError
      |              +-- UnicodeEncodeError
      |              +-- UnicodeTranslateError
      +-- Warning
```

```
083
084             +-- DeprecationWarning
085
086             +-- PendingDeprecationWarning
087
088             +-- RuntimeWarning
089
090             +-- SyntaxWarning
091
092             +-- UserWarning
093
094             +-- FutureWarning
095
096     +-- ImportWarning
097
098     +-- UnicodeWarning
099
100     +-- BytesWarning
```

There are several special exceptions that are derived directly from `BaseException`, like `SystemExit`, `KeyboardInterrupt` and `GeneratorExit`. Then there is the `Exception` class, which is the base class for `StopIteration`, `StandardError` and `Warning`. All the standard errors are derived from `StandardError`.

When you raise an exception or some function you called raises an exception, that normal code flow terminates and the exception starts propagating up the call stack until it encounters a proper exception handler. If no exception handler is available to handle it, the process (or more accurately the current thread) will be terminated with an unhandled exception message.

## Raising Exceptions

Raising exceptions is very easy. You just use the `raise` keyword to raise an object that is a sub-class of the `Exception` class. It could be an instance of `Exception` itself, one of the standard exceptions (e.g. `RuntimeError`), or a subclass of `Exception` you derived yourself. Here is a little snippet that demonstrates all cases:

```
01  # Raise an instance of the Exception class itself
02
03  raise Exception('Ummm... something is wrong')
04
05
06
07  # Raise an instance of the RuntimeError class
08
09  raise RuntimeError('Ummm... something is wrong')
10
11
12
13  # Raise a custom subclass of Exception that keeps the timestamp the exception was created
14
15  from datetime import datetime
16
17
18
19  class SuperError(Exception):
20
21      def __init__(self, message):
22
23          Exception.__init__(message)
24
25          self.when = datetime.now()
26
27
28
29
30
31  raise SuperError('Ummm... something is wrong')
```

## Catching Exceptions

You catch exceptions with the `except` clause, as you saw in the example. When you catch an exception, you have three options:

- Swallow it quietly (handle it and keep running).
- Do something like logging, but re-raise the same exception to let higher levels handle.
- Raise a different exception instead of the original.

## Swallow the Exception

You should swallow the exception if you know how to handle it and can fully recover.

For example, if you receive an input file that may be in different formats (JSON, YAML), you may try parsing it using different parsers. If the JSON parser raised an exception that the file is not a valid JSON file, you swallow it and try with the YAML parser. If the YAML parser failed too then you let the exception propagate out.

```
01  import json
02
03  import yaml
04
05
06
07  def parse_file(filename):
08
09      try:
10
11          return json.load(open(filename))
12
13      except json.JSONDecodeError
14
15          return yaml.load(open(filename))
```

Note that other exceptions (e.g. file not found or no read permissions) will propagate out and will not be caught by the specific except clause. This is a good policy in this case where you want to try the YAML parsing only if the JSON parsing failed due to a JSON encoding issue.

If you want to handle *all* exceptions then just use `except Exception`. For example:

```
1  def print_exception_type(func, *args, **kwargs):
2
3      try:
4
5          return func(*args, **kwargs)
6
7      except Exception as e:
8
9          print type(e)
```

Note that by adding `as e`, you bind the exception object to the name `e` available in your except clause.

## Re-Raise the Same Exception

To re-raise, just add `raise` with no arguments inside your handler. This lets you perform some local handling, but still lets upper levels handle it too. Here, the `invoke_function()` function prints the type of exception to the console and then re-raises the exception.

```
01  def invoke_function(func, *args, **kwargs):
02
03      try:
04
05          return func(*args, **kwargs)
06
07      except Exception as e:
08
09          print type(e)
10
11          raise
```

## Raise a Different Exception

There are several cases where you would want to raise a different exception. Sometimes you want to group multiple different low-level exceptions into a single category that is handled uniformly by higher-level code. In order cases, you need to transform the exception to the user level and provide some application-specific context.

# Finally Clause

Sometimes you want to ensure some cleanup code executes even if an exception was raised somewhere along the way. For example, you may have a database connection that you want to close once you're done. Here is the wrong way to do it:

```
def fetch_some_data():

    db = open_db_connection()

    query(db)

    close_db_Connection(db)
```

If the `query()` function raises an exception then the call to `close_db_connection()` will never execute and the DB connection will remain open. The `finally` clause always executes after a try all exception handler is executed. Here is how to do it correctly:

```
 def fetch_some_data():

     db = None

     try:

         db = open_db_connection()

         query(db)

     finally:

         if db is not None:

             close_db_connection(db)
```

The call to `open_db_connection()` may not return a connection or raise an exception itself. In this case there is no need to close the DB connection.

When using `finally`, you have to be careful not to raise any exceptions there because they will mask the original exception.

# Context Managers

Context managers provide another mechanism to wrap resources like files or DB connections in cleanup code that executes automatically even when exceptions have been raised. Instead of try-finally blocks, you use the `with` statement. Here is an example with a file:

```
def process_file(filename):

     with open(filename) as f:

         process(f.read())
```

Now, even if `process()` raised an exception, the file will be closed properly immediately when the scope of the `with` block is exited, regardless of whether the exception was handled or not.

# Logging

Logging is pretty much a requirement in non-trivial, long-running systems. It is especially useful in web applications where you can treat all exceptions in a generic way: Just log the exception and return an error message to the caller.

When logging, it is useful to log the exception type, the error message, and the stacktrace. All this information is available via the `sys.exc_info` object, but if you use the `logger.exception()` method in your exception handler, the Python logging system will extract all the relevant information for you.

This is the best practice I recommend:

```
import logging

logger = logging.getLogger()



def f():

    try:

        flaky_func()

    except Exception:

        logger.exception()

        raise
```

If you follow this pattern then (assuming you set up logging correctly) no matter what happens you'll have a pretty good record in your logs of what went wrong, and you'll be able to fix the issue.

If you re-raise, make you sure you don't log the same exception over and over again at different levels. It is a waste, and it might confuse you and make you think multiple instances of the same issue occurred, when in practice a single instance was logged multiple times.

The simplest way to do it is to let all exceptions propagate (unless they can be handled confidently and swallowed earlier) and then do the logging close to the top level of your application/system.

## Sentry

Logging is a capability. The most common implementation is using log files. But, for large-scale distributed systems with hundreds, thousands or more servers, this is not always the best solution.

To keep track of exceptions across your whole infrastructure, a service like sentry is super helpful. It centralizes all exception reports, and in addition to the stacktrace it adds the state of each stack frame (the value of variables at the time the exception was raised). It also provides a really nice interface with dashboards, reports and ways to break down the messages by multiple projects. It is open source, so you can run your own server or subscribe to the hosted version.

## Dealing With Transient Failure

Some failures are temporary, in particular when dealing with distributed systems. A system that freaks out at the first sign of trouble is not very useful.

If your code is accessing some remote system that is not responding, the traditional solution is timeouts, but sometimes not every system is designed with timeouts. Timeouts are not always easy to calibrate as conditions change.

Another approach is to fail fast and then retry. The benefit is that if the target is responding fast then you don't have to spend a lot of time in sleep condition and can react immediately. But if it failed, you can retry multiple times until you decide it is really unreachable and raise an exception. In the next section, I'll introduce a decorator that can do it for you.

## Helpful Decorators

Two decorators that can help with error handling are the `@log_error`, which logs an exception and then re-raises it, and the `@retry` decorator, which will retry calling a function several times.

## Error Logger

Here is a simple implementation. The decorator excepts a logger object. When it decorates a function and the function is invoked, it will wrap the call in a try-except clause, and if there was an exception it will log it and finally re-raise the exception.

```
def log_error(logger)

    def decorated(f):

        @functools.wraps(f)

        def wrapped(*args, **kwargs):

            try:

                return f(*args, **kwargs)

            except Exception as e:

                if logger:

                    logger.exception(e)

                raise

        return wrapped

    return decorated
```

Here is how to use it:

```
import logging

logger = logging.getLogger()



@log_error(logger)

def f():

    raise Exception('I am exceptional')
```



## Retrier

Here is a very good implementation of the @retry decorator.

```
import time

import math



# Retry decorator with exponential backoff

def retry(tries, delay=3, backoff=2):

  '''Retries a function or method until it returns True.



  delay sets the initial delay in seconds, and backoff sets the factor by which
```

```
    the delay should lengthen after each failure. backoff must be greater than 1,
    or else it isn't really a backoff. tries must be at least 0, and delay
    greater than 0.'''

    if backoff <= 1:
      raise ValueError("backoff must be greater than 1")

    tries = math.floor(tries)
    if tries < 0:
      raise ValueError("tries must be 0 or greater")

    if delay <= 0:
      raise ValueError("delay must be greater than 0")

    def deco_retry(f):

      def f_retry(*args, **kwargs):
        mtries, mdelay = tries, delay # make mutable

        rv = f(*args, **kwargs) # first attempt
        while mtries > 0:
          if rv is True: # Done on success
            return True

          mtries -= 1      # consume an attempt
          time.sleep(mdelay) # wait...
          mdelay *= backoff  # make future wait longer

          rv = f(*args, **kwargs) # Try again

        return False # Ran out of tries :-(

        return f_retry # true decorator -> decorated function

      return deco_retry  # @retry(arg[, ...]) -> true decorator
```

## Conclusion

Error handling is crucial for both users and developers. Python provides great support in the language and standard library for exception-based error handling. By following best practices diligently, you can conquer this often neglected aspect.