Error_handling

September 17, 2024

1 Increasing program robustness through handling exceptions

1.0.1 Error handling philosophy in Python

There is a clear distinction between code that handles normal execution and code that handles error conditions. A program is much more readable if there is a highly visible separation between the two. This is done in Python with the concept of Exceptions.

Early programming languages required errors to be indicated as special values in returns from function calls and long sequences of if-statements, before coming to the code that actually does the work. Both of these mingle normal execution with problem handling. The code becomes hard to read and hard to maintain.

C introduced the concepts of setjump and longjump for error handling. Difficult to use, and costly performancewise. Mostly used to bail out when memory or disk are exhausted.

C++ and Java used the concept of exceptions, that are *thrown* in one part of the program and *caught* somewhere in the call stack. Much better than C, but still requiring lots of declarations and incuring limitations.

Python has refined the exception concept and made it easy to use. There is no performance penalty for using exceptions. Everything is slow in Python and exceptions are not slower than anything else.

This makes Exceptions a first class member of the Python language. **Use them!** Even the interpreter uses them for regular tasks. When an iterator is exhausted, a *StopIteration* exception is raised.

Exceptions make it easy to separate code in the main execution path from error handling.

There is no excuse for showing an end user a traceback. Use exceptions to catch errors, log problems and give the user a friendly error message.

1.0.2 Maintaining program control with error handlers

The Python interpreter raises many exceptions when errors are encountered. Here are a few examples:

- IndexError - You are using a list index that doesn't exist
- KeyError - You are trying to access an element in a dict that doesn't exist
- OSError - You have a problem with a system resource, usually in the file system
- NotImplementedError - The functionality has not yet been implemented
- ImportError - Can't find the module you are trying to import

- StopIteration - calling next() on an iterator when it is exhausted

If an exception is not caught, the program exits. The exception is printed, together with a traceback showing where in the program execution flow the error occurred.

1.0.3 Detecting errors and raising exceptions

You handle errors in a try/except clause, where you can have different handlers for different exceptions. Exceptions form a hierarchy and an except clause for a class higher up in the hierarchy. For instance, *ArithmeticError* will catch *OverflowError*, *ZeroDivisionError* and *FloatingPointError*.

[1]:
```
import ipywidgets as widgets
from IPython.display import display
w = widgets.IntSlider(min=0, max=2)
display(w)
```

```
IntSlider(value=0, max=2)
```

[2]:
```
def get_from_list(index):
    my_list = ['Error_handling.ipynb', 'bar']
    return my_list[index]

def try_except_example():
    try:
        filename = get_from_list(w.value)
        with open(filename, 'r') as fp:
            lines = fp.readlines()
        if lines:
            print(len(lines))
    except IndexError:
        print('Sorry, the list index is too large.')
        return
    except OSError as e:
        print('We can access the attributes of the exception object.')
        print(e.args, e.errno)
        # print(dir(e))
    except (ImportError, NotImplementedError):
        raise NotImplementedError('We still need to build the module')
    except:
        print('Some other exception happened.')  # Normally very bad style to␣
 ↪catch all exceptions like this
        raise  # Raise the same exception again. May be caught at some other␣
 ↪level of the program
    else:
        print('Everything went well.')
    finally:
        print('This is executed after all try, except and else code has been␣
 ↪run.')
        print('It is useful for cleanup code that should always be run.')
```

```
try_except_example()
```

```
269
Everything went well.
This is executed after all try, except and else code has been run.
It is useful for cleanup code that should always be run.
```

1.0.4 Raising user-defined exceptions

Exceptions are regular Python classes. They form an inheritance hierarchy with *BaseException* as the root. When you want to generate your own exception, you make a class that inherits from one of the built in exceptions. *Exception* is normally the one you should be inheriting from, unless you are making a specialized version of one of the existing exceptions.

1.0.5 Reducing code complexity with context managers and the *with* statement

There are many instances when you always have to do something after executing some code. It can be things like closing a file, releasing a resource or commiting writes to a database.

Python has a mechanism for doing this:

```
with Class() as variable:
    do things
```

- It will instantiate the class, assign the created object to *variable* and call the *___enter___()* method of the object.
- Then it will execute the *do things*.
- When execution leaves the block, either by normal control flow, or by an exception being raised, the *___exit___()* method of the object will be called. This allows us to do whatever cleanup that is required.

There are a number of tricks in the standard library to make context managers both simpler and more powerful:

https://docs.python.org/3/library/contextlib.html

[3]:
```
import sys
import time
import datetime

class LicenseError(Exception):
    pass

class LicenseInUseError(LicenseError):
    pass

class LicenseExpiredError(LicenseError):
    def __init__(self, expiry_date, *args, **kwargs):
        super().__init__(args, kwargs)
```

```
        self.expiry_date = expiry_date

class License:
    ''' Context manager for single process use license.'''
    def __init__(self, key):
        self.key = key

    def __enter__(self):
        print('Entering __enter__')
        with open('license') as license_file:
            lines = license_file.readlines()
            if lines[0].rstrip() != self.key:
                raise LicenseError()
            expiry = datetime.date.fromisoformat(lines[1].rstrip())
            if expiry < datetime.date.today():
                raise LicenseExpiredError(expiry.isoformat())
        try:
            with open('lock', 'x') as _:
                pass
        except FileExistsError:
            raise LicenseInUseError()

        print('Entering context')

    def __exit__(self, exception_type, exception_value, traceback):
        # Parameters are None if no exception was raised
        print(exception_type)
        try:
            os.remove('lock')
        except OSError:
            pass
        # If we return True from __exit__, the exeption will be supressed
        # Otherwise it will be propagated, as normal

def do_super_cool_calculation(t):
    print('Calculating')
    time.sleep(t)
    return t

def license_example():
    license_key = 'xyzzy'
    try:
        with License(license_key) as l:
            return do_super_cool_calculation(4)
    except LicenseExpiredError as e:
        print('The license expired:', e.expiry_date)
    except LicenseInUseError:
```

```
        print('Another process is using the license')
    except LicenseError:
        print('There is a problem with your license. Please check that you have␣
 ↪the right license key.')
        sys.exit(1)
    else:
        print('Execution done.')  # Never reached because of return above!
    finally:
        print('All done.')  # Always reached, even with return above!
import os

print(license_example())
```

```
Entering __enter__
Entering context
Calculating
None
All done.
4
```