EPAM Python Software Engineer Training

Lesson 2: Python exceptions

Contents

1 Course

1.1 Exceptions

In Python there are quite a few different standard exception classes derived from a base *Exception* class. A couple of exception classes are derived from a *BaseException* class. A common rule is that descendants of an *Exception* class are errors while all the others shall be treated as exceptional, but not erroneous situation. For example, *KeyboradInterrupt* is an exceptional situation caused by a *SIGINT* system signal, but this is not an error; the other example is a *SystemExit* which is raised when an application code explicitly tells an application to exit. Usually, these non-error exceptions have a special meaning to a Python interpreter or one of Python libraries and shell not be catched by a user code, hence, a difference.

1.2 Try block

Starting since Python version `2.7` there is a single `try .. except .. else .. finally` block where all of the last three keywords are optional, but at least one of `except` or `finally` must be present. A common sense for an `else` clause is for a code that shall be executed if no exception has occured; this helps to make a source code more structured by wrapping only a place where an exception is really expected into a `try..except` block.

Due to a semantical difference between error and non-error exceptions described above the below code snippet is a bad habit in Python as it will mask non-error exceptions leading to unexpected behavior:

```
try:
    # Do something
except:
    # General exception handling logic
```

A correct way to do this would be one of the below:

```
try:
    # Do something
except Exception:
    # General exception handling logic

try:
    # Do something
except:
    # General exception handling logic
    raise
```

I.e. one shall always take care that a non-error exception goes up the stack.

1.3 Advanced Exception Handling

Sometimes an exception can occur in a quite unexpected place that cannot be covered by a reasonably global exception hander. For example, in a separate simple thread where we do not expect any exceptions. To handle such incidents one might set a `sys.excepthook` to handle such cases.

A quite important thing for unexpected exceptions is to log a full traceback for them somehow. This information is returned by a `sys.exc_info` function inside an *except* clause. A Python *logging* module, for example, knows how to log tuples returned by *exc_info* properly.

1.4 Signal Handling

A good application shall know how to handle all or at a predefined set of system signals. Typically, at least a `SIGINT` shall be handled. In the easiest cases this can be achieved by handling a `KeyboardInterrupt` exception. However, this is not alway appropriate as an exception breaks a normal control flow that may lead to data loss. In more complex cases one shall override a default signal handler for expected system signals using a `signal` module. A common sense is to handle at least `SIGINT` and `SIGTERM` gracefully.

1.5 Exit cleanup

To explicitly terminate an application it is possible to call a `sys.exit` function at any place which will raise a `SystemExit` that is handled by a Python interpreter is such a way that all threads are terminated and an application exits without a traceback.

If some clean up tasks are required that cannot be achieved by a `finally` block, one can use an `atexit` module to register exit handlers. They are called if an application terminates gracefully, but are not executed if an application is killed by a signal or forcefully terminated using `os.exit`. Nevertheless, a `finally` clause would not be executed in these cases too.

2 Tasks

2.1 File Copy

Write a simple program that reads content from one file an writes it to yet another file. All possible I/O and OS errors shall be handled gracefully (e.g. nonexisting input file, insufficient permissions etc) and an appropriate diagnostic information shall be printed to standard error. If a read of an input file fails - not subsequent write shall be done. An output file shall be written only if it does not exist, otherwise an error shall occur (think of concurrency problems associated with this part of a task).

An application shall return an appropriate exit code identifying success or failure do fulfill a requested operation.

2.2 Advanced File Copy

A task is the same as above, but a keyboard interruption shall be handled gracefully printing a message `Operation terminated by user`. In order to better visualize user interaction, file content shall be read/written line by line with a `time.sleep` for one second between each line and printing a diagnostic information like below (each dot means one line):

```
Copying a file "aaa.txt" into "bbb.txt"
.............................
Operation complete
```

In case of user initiated termination all data that was written into a file up to data shall be preserved.

2.3 Timer

Write a program that waits for a user input and prints a dot every second until a user either hit `Enter` or is killed with a signal. In case of a `Ctrl-C` a message `User input cancelled` shall be printed on a new line. If a user input was received an application shall print it back on a new line.

> ***Hint***
>
> Use a system timer to implement this task without threads.

2.4 Signals

Write a simple program that logs all received signals to the standard input. Every minute an application shall print a number of received signals up to data.

Besides, a `SIGHUP` signal shall reset a received signals counter to zero. A `SIGINT` and `SIGTERM` signals shall terminate a program gracefully.

At exit a program shall print a total number of received signals and a number of resets. Use an `atexit` module to do this part of a task.