Why do we need the “finally” clause in Python?

153

41

I am not sure why we need `finally` in `try...except...finally` statements. In my opinion, this code block

```
try:
    run_code1()
except TypeError:
    run_code2()
other_code()
```

is the same with this one using `finally`:

```
try:
    run_code1()
except TypeError:
    run_code2()
finally:
    other_code()
```

Am I missing something?

python exception-handling try-finally

share improve this question

edited Dec 4 '17 at 12:30 Eugene Yarmash 75.2k ●21 ●161 ●244

asked Jul 18 '12 at 23:44 RNA 75.7k ●9 ●34 ●51

add a comment

11 Answers

active oldest votes

214

It makes a difference if you return early:

```
try:
    run_code1()
except TypeError:
    run_code2()
    return None   # The finally block is run before the method returns
finally:
    other_code()
```

Compare to this:

```
try:
    run_code1()
except TypeError:
    run_code2()
    return None

other_code()  # This doesn't get run if there's an exception.
```

Other situations that can cause differences:

- If an exception is thrown inside the except block.
- If an exception is thrown in `run_code1()` but it's not a `TypeError`.
- Other control flow statements such as `continue` and `break` statements.

share improve this answer

edited Jul 19 '12 at 0:01

answered Jul 18 '12 at 23:46 Mark Byers 549k ●116 ●1290 ●1308

try: #x = Hello + 20 x = 10 + 20 except: print 'I am in except block' x = 20 + 30 else: print 'I am in else block' x += 1 finally: print 'Finally x = %s' %(x) – Abhijit Sahu Oct 2 '17 at 6:32

What if there is an exception in `finally` code? – Hrvoje T Apr 12 at 9:24

add a comment

41

You can use `finally` to make sure files or resources are closed or released regardless of whether an exception occurs, *even if you don't catch the exception.* (Or if you don't catch that *specific* exception.)

```
myfile = open("test.txt", "w")

try:
    myfile.write("the Answer is: ")
    myfile.write(42)   # raises TypeError, which will be propagated to caller
finally:
    myfile.close()     # will be executed before TypeError is propagated
```

In this example you'd be better off using the `with` statement, but this kind of structure can be used for other kinds of resources.

A few years later, I wrote a blog post about an abuse of `finally` that readers may find amusing.

share improve this answer

edited Apr 28 at 18:49

answered Jul 18 '12 at 23:51 kindall 119k ●15 ●170 ●229

add a comment

11

They are not equivalent. Finally code is run no matter what else happens. It is useful for cleanup code that has to run.

share improve this answer

answered Jul 18 '12 at 23:46 Antimony 25k ●4 ●63 ●78

6 `Finally code is run no matter what else happens` ... unless there's an infinite loop. Or a powercut. Or `os._exit()`. Or... – Mark Byers Jul 18 '12 at 23:49

2 ... a segfault. Or `SIGABRT`. – Sven Marnach Jul 18 '12 at 23:52

2 @Mark Actually, sys.exit throws a normal exception. But yes, anything that causes the process to terminate immediately will mean that nothing else executes. – Antimony Jul 18 '12 at 23:52

@Antimony: Thanks. Changed to `os._exit`. – Mark Byers Jul 18 '12 at 23:53

add a comment

7

The code blocks are not equivalent. The `finally` clause will also be run if `run_code1()` throws an exception other than `TypeError`, or if `run_code2()` throws an exception, while `other_code()` in the first version wouldn't be run in these cases.

share improve this answer

answered Jul 18 '12 at 23:46 Sven Marnach 310k ●66 ●709 ●670

add a comment

6

In your first example, what happens if `run_code1()` raises an exception that is not `TypeError`? ... `other_code()` will not be executed.

Compare that with the `finally:` version: `other_code()` is guaranteed to be executed regardless of any exception being raised.

share improve this answer

answered Jul 18 '12 at 23:47 mhawke 55.4k ●7 ●48 ●72

add a comment

3

`finally` is for defining **"clean up actions"**. The `finally` clause is executed in any event before leaving the `try` statement, whether an exception (even if you do not handle it) has occurred or not.

I second @Byers's example.

share improve this answer

answered May 15 '15 at 17:59 kakhkAtion 1,079 ●11 ●18

add a comment

2

Finally can also be used when you want to run "optional" code before running the code for your main work and that optional code may fail for various reasons.

In the following example, we don't know precisely what kind of exceptions `store_some_debug_info` might throw.

We could run:

```
try:
  store_some_debug_info()
except Exception:
  pass
do_something_really_important()
```

But, most linters will complain about catching too vague of an exception. Also, since we're choosing to just `pass` for errors, the `except` block doesn't really add value.

```
try:
  store_some_debug_info()
finally:
  do_something_really_important()
```

The above code has the same effect as the 1st block of code but is more concise.

share improve this answer

answered Feb 17 '17 at 17:14 Brad Johnson 315 ●3 ●15

add a comment

1

Perfect example is as below:

```
try:
    #x = Hello + 20
    x = 10 + 20
except:
    print 'I am in except block'
    x = 20 + 30
else:
    print 'I am in else block'
    x += 1
finally:
    print 'Finally x = %s' %(x)
```

share improve this answer

answered Oct 2 '17 at 6:33 Abhijit Sahu 62 ●1 ●7

add a comment

1

As explained in the documentation, the `finally` clause is intended to define clean-up actions that must be executed **under all circumstances**.

> If `finally` is present, it specifies a ‘cleanup’ handler. The `try` clause is executed, including any `except` and `else` clauses. If an exception occurs in any of the clauses and is not handled, the exception is temporarily saved. The `finally` clause is executed. If there is a saved exception it is re-raised at the end of the `finally` clause.

An example:

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
...         print("result is", result)
...     finally:
...         print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

As you can see, the `finally` clause is executed in any event. The `TypeError` raised by dividing two strings is not handled by the `except` clause and therefore re-raised after the `finally` clause has been executed.

In real world applications, the finally clause is useful for releasing external resources (such as files or network connections), regardless of whether the use of the resource was successful.

share improve this answer

answered Dec 3 '17 at 13:20 Eugene Yarmash 75.2k ●21 ●161 ●244

add a comment

1

To add to the other answers above, the `finally` clause executes no matter what whereas the `else` clause executes only if an exception was not raised.

For example, writing to a file with no exceptions will output the following:

```
file = open('test.txt', 'w')

try:
    file.write("Testing.")
    print("Writing to file.")
except IOError:
    print("Could not write to file.")
else:
    print("Write successful.")
finally:
    file.close()
    print("File closed.")
```

OUTPUT:

```
Writing to file.
Write successful.
File closed.
```

If there is an exception, the code will output the following, (note that a deliberate error is caused by keeping the file read-only.

```
file = open('test.txt', 'r')

try:
    file.write("Testing.")
    print("Writing to file.")
except IOError:
    print("Could not write to file.")
else:
    print("Write successful.")
finally:
    file.close()
    print("File closed.")
```

OUTPUT:

```
Could not write to file.
File closed.
```

We can see that the `finally` clause executes regardless of an exception. Hope this helps.

share improve this answer

answered Dec 13 '17 at 6:46 captainblack 701 ●2 ●14 ●33

add a comment

0

Using delphi professionally for some years taught me to safeguard my cleanup routines using finally. Delphi pretty much enforces the use of finally to clean up any resources created before the try block, lest you cause a memory leak. This is also how Java, Python and Ruby works.

```
resource = create_resource
try:
  use resource
finally:
  resource.cleanup
```

and resource will be cleaned up regardless of what you do between try and finally. Also, it won't be cleaned up if execution never reaches the `try` block. (i.e. `create_resource` itself throws an exception) It makes your code "exception safe".

As to why you actually need a finally block, not all languages do. In C++ where you have automatically called destructors which enforce cleanup when an exception unrolls the stack. I think this is a step up in the direction of cleaner code compared to try...finally languages.

```
{
  type object1;
  smart_pointer<type> object1(new type());
} // destructors are automagically called here in LIFO order so no finally required.
```

share improve this answer

answered Mar 16 at 6:12 nurettin 7,092 ●3 ●40 ●59