

Error handling and logging



What could go wrong?

What could go wrong

- We made a programming mistake
- We get bad input
- There is an issue in some other part of the system



Lecture Overview

1. **Logging** — recording what your program does
2. **Exceptions & Tracebacks** — what goes wrong and how Python reports it
3. **Error Handling** — `try`, `except`, `else` and `finally`
4. **Raising Exceptions** — signalling errors to callers



Logging in Python



The logging Module: A Quick Start

```
1 import logging
2 logging.basicConfig(level=logging.INFO)
3 logging.info("Program started")
```

Logging Levels

- **DEBUG:** Detailed info
- **INFO:** Normal operations
- **WARNING:** Something unexpected
- **ERROR:** An error occurred
- **CRITICAL:** Serious error



Custom loggers

- Clearly label where messages originate
- Allow for very verbose logging in parts of the code

```
1 import logging
2 logging.basicConfig()
3 logger = logging.getLogger("my_code")
4 logger.setLevel(logging.DEBUG)
5 logger.debug("Debug level print for local logger working")
6
7 logging.info("Info level print for root logger still hidden")
```

```
DEBUG:my_code:Debug level print for local logger working
```



Create loggers

Create two loggers `lazy` and `eager` and make `lazy` print only critical messages, and `eager` print all messages.

Debugging Using Logs

- Start reading from both ends to find the error
- Search for words like “ERROR”
 - Don't write **Oops** instead of ERROR in your messages.
- Re-run with **DEBUG** level for more info



Exceptions, Tracebacks, and the Call Stack



What Are Exceptions?

- Exceptions are Python's way of signalling that something went wrong at runtime.
- When an exception is raised, normal execution stops and Python looks for a handler.
- If no handler is found, the program crashes and prints a traceback.

```
1 >>> int("hello")
2 ValueError: invalid literal for int() with base 10: 'hello'
```



Tracebacks

```
1 def caller():
2     div_by_zero()
3
4 def div_by_zero():
5     1 / 0
6
7 caller()
```

Traceback (most recent call last):

```
File "/private/tmp/call.py", line 7, in <module>
  caller()
```

```
File "/private/tmp/call.py", line 2, in caller
  div_by_zero()
```

```
File "/private/tmp/call.py", line 5, in div_by_zero
  1 / 0
```

```
~~^~~
```

ZeroDivisionError: division by zero



What Is the Call Stack?

- The stack is the chain of function calls leading the current point of execution.
- The innermost call is on top of the stack
- The traceback is the reverse view of the call stack from the outermost loop to the current call

```
Traceback (most recent call last):
  File "/private/tmp/call.py", line 7, in <module>
    caller()
  File "/private/tmp/call.py", line 2, in caller
    div_by_zero()
  File "/private/tmp/call.py", line 5, in div_by_zero
    1 / 0
    ~^~
ZeroDivisionError: division by zero
```



How do I read a Traceback?

Start from the bottom

```
File "/private/tmp/call.py", line 5, in div_by_zero
    1 / 0
    ~^~
```

```
ZeroDivisionError: division by zero
```

This is what tripped the code in the end. If this is deep in some library, work your way down from the top to see where in your code you went into the libraries (usually the mistake is on you and not on the library).



Trigger Some Exceptions

In a Python interpreter, try to trigger different exceptions:

- `1 / 0`
- `int("one")`
- `"hello"[42]`
- `{"a": 1}["b"]`
- `open("does_not_exist.txt")`

Read each traceback. What is the exception type? What does the message tell you?



Basics of Error Handling



Introduction to try and except

- Use `try` to run code that might fail.
- Use `except` to handle errors gracefully.

```
1 import logging
2
3 try:
4     s = input("Enter a number: ")
5     x = float(s)
6 except ValueError:
7     logging.error("%s is not a valid number!", s)
```



Handling Multiple Exceptions

```
1 import logging
2
3 try:
4     s = input("Enter a number: ")
5     num = float(s)
6     result = 1 / num
7 except ValueError:
8     logging.error("%s is not a valid number!", s)
9 except ZeroDivisionError:
10    logging.error("Can't divide by zero.")
```



Using else and finally

```
1 import logging
2 logging.basicConfig(level=logging.INFO)
3 try:
4     s = input("Enter a number to be squared: ")
5     num = float(s)
6 except ValueError:
7     logging.error("can't convert %s to a number!", s)
8 else:
9     logging.info(f"You entered: {num}, the square is {num**2}")
10 finally:
11     logging.info("I'll call it a day now.")
```



Context Managers: The Pythonic `finally`

Instead of `try/finally` for resource cleanup, prefer `with`:

```
1 # Instead of this:
2 try:
3     f = open("data.csv")
4     data = f.read()
5 finally:
6     f.close()
7
8 # Do this:
9 with open("data.csv") as f:
10     data = f.read()
```

The `with` statement guarantees cleanup — even if an exception
occurs.



Handle Some Errors

Write a function `safe_divide(a, b)` that:

- Returns `a / b`
- Catches `ZeroDivisionError` and prints a message instead of crashing
- Catches `TypeError` (e.g. `safe_divide("x", 2)`) and prints a different message



The “Silent Failure” Trap

```
1 try:
2     1/0
3 except:
4     pass # Don't do this!
```

Instead, catch specifically, log, and re-raise / handle:

```
1 import logging
2 try:
3     1/0
4 except ZeroDivisionError as e:
5     logging.exception("1/0 failed with ZeroDivisionError")
6     raise # or handle otherwise
```

Exit Codes: Escalating errors

- Programs return an `int` value to the caller, scripts rely on this
- `0` = success, non-zero = failure
- An unhandled exception automatically exits with code `1`
- If you catch exceptions, **you** need to set the exit code with `sys.exit()` (see [python docs](#))
- Only use `sys.exit()` at the top level (`__main__`) — functions should raise, not exit



Exit codes - example

```
1 import logging
2 import sys
3
4 try:
5     result = 1 / 0
6 except ZeroDivisionError:
7     logging.critical("1/0 failed")
8     sys.exit(1)
```



Raising Exceptions



Raising Built-in exceptions

Often, you can use a built-in exception for your scenario.

```
1 def div_one(val):
2     if float(val) == 0:
3         raise ZeroDivisionError(
4             "div_one will divide 1 by its argument. "
5             "Zero is a bad choice here."
6         )
7     return 1 / float(val)
8
9 print(div_one(1))
10 print(div_one(0))
```



Common Built-in Exceptions

Exception	When to use
<code>ValueError</code>	Right type, wrong value (e.g. <code>int("abc")</code>)
<code>TypeError</code>	Wrong type entirely (e.g. <code>len(42)</code>)
<code>KeyError</code>	Dictionary key not found
<code>FileNotFoundError</code>	File or directory does not exist
<code>RuntimeError</code>	Something went wrong at runtime, no better fit
<code>NotImplementedError</code>	Method exists but is not implemented yet



Raising exceptions from exceptions

Keeps the origin clear.

```
1 def div_one(val):
2     try:
3         num = float(val)
4     except ValueError as ve:
5         raise ValueError(
6             "div_one will divide 1 by its argument. "
7             "We need something convertible to float as argument."
8             ) from ve
9     return 1 / num
10
11 print(div_one("one"))
```

...

Raising exceptions from exceptions

The error message

```
Traceback (most recent call last):
  File "<python-input-0>", line 3, in div_one
    num = float(val)
ValueError: could not convert string to float: 'one'
```

The above exception was the direct cause of the following exception:

```
Traceback (most recent call last):
  File "<python-input-0>", line 11, in <module>
    print (div_one("one"))
          ~~~~~^~~~~~
  File "<python-input-0>", line 5, in div_one
    raise ValueError(
    ...<2 lines>...
    ) from ve
ValueError: div_one will divide 1 by its argument. We need something convertible to float as argument.
```



Logging Exceptions

Use `logger.exception()` inside an `except` block to log the message **and** the full traceback:

```
1 import logging
2
3 try:
4     int("not a number")
5 except ValueError:
6     logging.exception("Failed to convert input to int")
```

```
ERROR:root:Failed to convert input to int
Traceback (most recent call last):
  File "<python-input-0>", line 4, in <module>
    int("not a number")
    ~~~^^^^^^^^^^^^^^^^^^^^^^^^
ValueError: invalid literal for int() with base 10: 'not a number'
```

This bridges logging and error handling — record the problem *and* its context.



Exceptional Behavior

- Write a function `behave` that will raise a `BehaviorException`, when called with `"exceptional"` as argument.
- Write another function, that calls `behave` and catches this behavior. Make it log exceptional behavior.



Summary and Q&A



Key Takeaways

- Handle expected errors using `try/except`
- Use logging to record what's happening in your program
- Avoid silent failures
- Clear logs and error messages save time later



When to Handle vs. When to Let it Fail

- **Handle:** When you can recover and inform the user.
- **Let it fail:** When the program is in a bad state.

Nobody reads logs if things don't crash.



Resources for Further Learning

- [Python Docs: Errors and Exceptions](#)
- [Python Docs: logging module](#)



Shotgun Buffet



Raising Your Own Exceptions

```
1 class FunkyError(Exception):  
2     pass  
3  
4 raise FunkyError("Funky!")
```

```
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
FunkyError: Funky!
```

- Specific error conditions in your app
- Clearer debugging and error messages

Not an Error Yet? Use `warnings`

The `warnings` module sits between logging and exceptions:

```
1 import warnings
2
3 def compute(x):
4     if x < 0:
5         warnings.warn("x is negative, results may be unexpected",
6                       stacklevel=2)
7     return x ** 0.5
```

- Common in scientific libraries (NumPy, pandas)
- Visible by default, but can be silenced or turned into errors:

```
1 import warnings
2 warnings.filterwarnings("error") # treat warnings as exceptions
```

