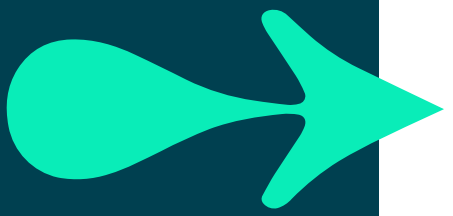
POWERING POTENTIAL

Python 3 Programming

Error Handling and Exceptions

ERROR HANDLING AND EXCEPTIONS



Contents

Writing to stderr

Don't forget that error messages should go to stderr

- Script errors are often redirected by the user
- Ordinarily print goes to stdout, but it can be changed

→ Syntax for using stderr with print changed at Python 3

- Using sys.stderr.write works on Python 2 and 3

```
$ myscript.py > out 2> err
```

Version neutral

```
import sys

if something_nasty:
    sys.stderr.write("Invalid types compared")
    exit(1)
```

```
    print("Invalid types compared", file=sys.stderr)
```

- See errno module, and os.strerror() for error number translation

Most people know what `stderr` is for - writing error messages, yet it is very common to see scripts where error messages are written to `stdout` instead. Python routes its own error messages to `stderr` but it cannot know which of your messages are errors, so it is up to you!

Using `stderr` is important because many tools use that stream to route error messages elsewhere, or to indicate if an error has occurred. For example, a common system administrator's trick is to redirect `stderr` (file descriptor 2) to a file, then test the size of that file after the script has run. If the file size is zero, then the script did not produce any errors and so it worked! You can argue this is flawed logic, the return value should be tested instead, nevertheless it is a common technique.

For more sophisticated error message routing, see the **`logging`** module (part of the standard library). For debugging and error reporting in web applications, see the **`cgitb`** module (should be bundled with your installation).

The syntax for routing **`print`** output to `stderr` changed at Python 3. Prior to that release, the old syntax was:

```
print >> sys.stderr, message
```

However, the syntax using:

```
sys.stderr.write (message)
```

did not change.

Standard error numbers are defined as literals in the errno module, and can be converted to strings using os.strerror(). For example,

error number 2 is `errno.ENOENT` which is "No such file or directory".

Controlling warnings

- **Warnings can be generated by Python and by user code**
  - Is a warning to be issued?
  - Where should the warning be sent?

  → Default: sys.stderr
- **The `warnings` standard module gives us control**
  - Generate user warnings with `warnings.warn()`
  - Sending and formatting uses functions which can be overridden
  - Warnings can be filtered by type, text, or category
- **Can be controlled through the -Wd command-line option**
  - This makes warnings visible that are usually ignored
  - DeprecationWarnings are not displayed unless turned on using -Wd or a warnings filter

By default, Python warnings are written to stderr, but they are controllable from a program in a number of ways. However, you don't see warnings very often, because the most common ones are, by default, ignored. We can also generate warnings (class `UserWarning`) from within a program. `DeprecationWarning`, `PendingDeprecationWarning`, and `ImportWarning` are ignored unless the -Wd option (or a filter) makes them visible. The filter to turn the -Wd option on within a program is:

```
warnings.simplefilter('default')
```

Warnings are technically part of exception handling, but controlled in a very different way. We can describe them in a filter using a regular expression, and some simple examples are shown on the next slide.
We shall see later where Warnings fit into the exception hierarchy.

Further warnings control can be achieved using a *context manager*, which is outside the scope of this course.

Warnings - examples

- Raise a non-fatal UserWarning

```
import warnings

warnings.warn('Oops')
print('Ending...')
```

```
warn.py:4: UserWarning: Oops
  warnings.warn('Oops')
Ending...
```

- Turn a warning into a fatal exception

```
import warnings

warnings.simplefilter('default')
warnings.filterwarnings('error', '.*')

import time
time.accept2dyear = True
time.asctime((11, 1, 1, 12, 34, 56, 4, 1, 0))

print('Ending...')
```

Equivalent to -Wd option

RegExp filter (all

This raises a DeprecationWarning

Will not be executed

The first example is very simple, sending a warning message from the user. Note that the program will complete, despite the warning message.

The second example sets `DeprecationWarnings` (and others) on, the equivalent to the `-Wd` command-line option. The filter then turns all warnings (the `'.*'` regular expression) into errors. The `time` methods are just a handy way of generating a `DeprecationWarning`.

Possible actions include:

error	turn matching warnings into exceptions
ignore	don't print matching warnings
always	print matching warnings
default	print the first occurrence of warnings for each location where it is issued
module	print the first occurrence of warnings for each module where it is issued
once	print only the first occurrence of warnings, regardless of location

Exception handling

- **Traditional error handling techniques include**
  - Returning a value from a function to indicate success or failure
  - Ignore the error
  - Log the error, but otherwise ignore it
  - Put an object into some kind of invalid state that can be tested
  - Aborting the program
- **In Python, an exception can be thrown**
  - An exception is represented by an object
  → Usually of a class derived from the **exception** superclass
  → Includes diagnostic attributes which may be printed
  - Throwing an exception transfers control
  - The function call stack is unwound until a handler capable of handling the exception object is found

For the majority of programmers, their least favorite chore is testing and handling error conditions. Often, there are many ways for something to fail, resulting in exception code that can dwarf the core program logic.
One of the traditional approaches to signaling errors is to return some kind of status value from a function. There are a number of problems with this:

Can lead to cascading, if else, if code that can be hard to read.
Overloaded operators do not have a 'spare' return value for exceptions.
Return values are too easy to ignore.

One technique is to have an object know whether or not it is in a good or bad state, which can then be queried. The problem with this is that, although it can handle constructor problems, it can lead to clumsy code. It also requires extra members for manipulating and representing the error, making the class less cohesive.
Aborting the program is a last resort. It is up to the caller to determine what the failure policy should be, and for the called component to detect any failure.
The Python solution is to represent exceptions as objects. As objects, they can be self describing, whereas something like a simple integer value – as used in other languages – is not very informative, as it says nothing about the details of an exception. The next feature

of Python's exception handling is that there is a separate path of control for exceptions than for normal function returns. Exceptions can be caught by defining a handler, but there is otherwise no need to detect an exception in a function, solely to return it to that function's caller so that it can do the same – you only catch exceptions you are interested in.

Exception syntax

- Unhandled exceptions terminate the program
- Trapping an exception:

```
try:
    code body
except [exception_list [as var]]:
    exception handler
else:
    statements if no exception
finally:
    final statements
```

Optional, not executed if an exception occurs

Optional, always* executed

- Although we use terms like "try block", there is no real or implied scope within each code section

The try block contains code to be tested. Should any of that code (which is often a function call) raise an exception then it can be trapped, and code executed in the except block to handle it. If an exception list is specified, then the handler code will only be executed if the exception is in the list, otherwise the stack will be unwound until a handler is found. Within the except block, the exception raised will be available if the optional as statement was used. The exception object has a number of useful attributes available, which vary depending on the subclass of exception.
The else block will only be executed if everything in the try block worked OK.
The finally block is guaranteed to be always executed, regardless of what happened in the code body (but see later).

Multiple exceptions

- **It is common to wish to trap more than one exception**
  - Each with its own handler
  - Or multiple exceptions with the same handler

```
filename = "foo"
try:
    f = open(filename)
except FileNotFoundError:
    errmsg = filename + " not found"
except (TypeError, ValueError):
    errmsg = "Invalid filename"
...

if errmsg != "":
    exit(errmsg)
```

For example, TypeError would be raised if `filename` was not a string.

Remember, `exit()` raises a SystemExit exception!

Statements within the **`try`** block, particularly when functions are called, could raise different exceptions depending on circumstances. In this case, it is possible to test for more than one exception in a structure not unlike a case statement, with multiple **`except`** tests.
The same handler code can be executed by supplying a tuple of exceptions to the **`except`**.

Exception arguments

- **Each exception has an arguments attribute**
  - Stored in a tuple
  - The number of elements, and their meaning varies
  - Other attributes may be available
- **Access the exception using the 'as' clause**

```
import sys
try:
    f = open("foo")
except FileNotFoundError as err:
    print("Could not open",
          err.filename, err.args[1],
          file=sys.stderr)

    print("Exception arguments:", err.args,
          file=sys.stderr)
```

```
Could not open foo No such file or directory
Exception arguments: (2, 'No such file or directory')
```

py3

When an exception is raised, an exception object is created and, like other classes, exception has attributes. New with Python 3 is the **`__traceback__`** attribute, which shows the stack trace.
Different types of exceptions are actually subclasses of the exception class, and different subclasses have their own attributes. For example, as the slide shows, subclass **`FileNotFoundError`** includes an attribute **`filename`**. This was introduced in Python 3.3, in previous versions `IOError` was used. The Python exception class hierarchy is shown on a later slide.
One way of understanding the syntax of the **`except`** statement is to think of it as being a special kind of function call. One parameter is passed - the exception object, and we specify which type we will accept. The name of that exception object within the "function" is the name given following **`as`**. Object attributes can then be interrogated using this name.
The syntax for the **`as`** clause changed at Python 3, it replaces a comma in Python 2.

The finally block

- **The `finally` block is (almost*) always executed**
- Even if an exception occurs
- * `os._exit()` inside the `try` block ignores the finally block
- **The `finally` block is executed *before* stack unwind**

```
def my_func():
    try:
        f = open("foo")
    finally:
        print("Finally block", file=sys.stderr)

try:
    my_func()
except OSError:
    print("An OS error occurred", file=sys.stderr)
```

```
Finally block
An OS error occurred
```

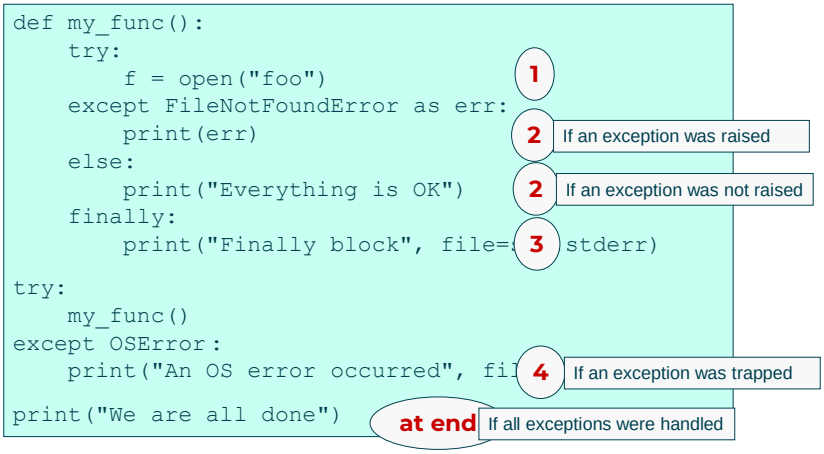
The idea of the **`finally`** block is that this can contain tidying or exit code which must always be executed. It is particularly useful as an alternative to a destructor, since, as we saw earlier, `__del__` might not be called.

It is also often used to release locks used, for example, in multithreading. If a lock is obtained in the **`try`** block, then we can guarantee that it will be unlocked if we put that code in a **`finally`** block. There is a danger with this, however. Just because we can release resources in the **`finally`** block, does not mean that the resource is necessarily consistent - the **`try`** block might have blown up! There is no automatic "rollback" functionality - you have to code that yourself. In the **`finally`** block code, do not assume *any* statements in the **`try`** block worked. Test handles and variables (usually for `None`) before trying to use or free them.

We cannot access the exception (if there is one) from within a **`finally`** block, and if the **`finally`** block itself raises an exception then the original one is lost.

Order of execution

- Either the except block or the else block is executed before the finally block



Python attempts to execute the code in the **`try`** block. During execution, if an exception occurs, an **`except`** block is searched for to handle that exception. If none is found, then the stack is unwound until one is found, or the default exception handler is executed and the program ends. If an exception handler is found then that is executed, followed by the **`finally`** block (if there is one).
If an exception is not raised, then the **`else`** block is executed (if there is one), then the **`finally`** block.

The Python 3 exception hierarchy (1)

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StopAsyncIteration
    +-- ArithmeticError
        +-- FloatingPointError
        +-- OverflowError
        +-- ZeroDivisionError
    +-- AssertionError
    +-- AttributeError
    +-- BufferError
    +-- EOFError
    +-- ImportError
        +-- ModuleNotFoundError
    +-- LookupError
        +-- IndexError
        +-- KeyError
    +-- MemoryError
    +-- NameError
        +-- UnboundLocalError
```

```
+-- Exception
    ....
    +-- OSError
        +-- BlockingIOError
        +-- ChildProcessError
        +-- ConnectionError
            +-- BrokenPipeError
            +-- ConnectionAbortedError
            +-- ConnectionRefusedError
            +-- ConnectionResetError
        +-- FileExistsError
        +-- FileNotFoundError
        +-- InterruptedError
        +-- IsADirectoryError
        +-- NotADirectoryError
        +-- PermissionError
        +-- ProcessLookupError
        +-- TimeoutError
```

From Python 3.3 several exceptions, including `EnvironmentError` and `IOError`, are aliases for `OSError`.

Although objects of any type may be thrown — not just those within the hierarchy shown above — it is sensible to use classes already defined in the standard exception hierarchy. If a new kind of exception is needed, a class for it can be derived from an existing class. For example, the database interface has its own exception hierarchy.

Be aware there could be changes to this hierarchy at every major release. At Python version 3.3, exceptions EnvironmentError, IOError, VMSError, and WindowsError are aliases of OSError, and are kept for backward compatibility.

It is a temptation to trap all errors, but that is a bad idea. There are those errors that we know are likely to occur and those we do not expect. Let the unexpected crash our program - at least the program then halts before it can do any damage.

Trapping all errors can lead to a false sense of security, and can mask real bugs that we did not expect.

The Python 3 exception hierarchy (2)

```
+-- Exception
....
    +-- ReferenceError
    +-- RuntimeError
        +-- NotImplementedError
        +-- RecursionError
    +-- SyntaxError
        +-- IndentationError
            +-- TabError
    +-- SystemError
    +-- TypeError
    +-- ValueError
        +-- UnicodeError
            +-- UnicodeDecodeError
            +-- UnicodeEncodeError
            +-- UnicodeTranslateError
```

```
+-- Exception
....
+-- Warning
        +-- DeprecationWarning
        +-- PendingDeprecationWarning
        +-- RuntimeWarning
        +-- SyntaxWarning
        +-- UserWarning
        +-- FutureWarning
        +-- ImportWarning
        +-- UnicodeWarning
        +-- BytesWarning
        +-- ResourceWarning
```

Note that, although they are technically part of the exception hierarchy, warnings cannot be trapped using the `try…except` mechanism, unless turned into an exception using a filter.

assert

- **Raise an exception based on a boolean statement**
  - AssertionError is raised if the boolean is False
  - May be associated with additional data

```
assert expression [, associated_data]
```

```
def my_func(*arguments):
    assert all(arguments), 'False argument in my_func'
    ...

my_func('Tom', '', 42)
```

```
AssertionError: False argument in myfunc
```

- **Not usually a good idea in production code**
  - Comment out **`assert`** statements for production
  - Or run with `-O` (oh), or set PYTHONOPTIMIZE to 0

→ Sets `__debug__` to false

The Python built-in **`assert`** has been inspired by the C/C++ macro of the same name, although it behaves differently. It is designed to provide sanity tests within code in a simpler form than having to type a full test. It is designed to be used during development, so usually you would not explicitly test for an assertion failure - crashing the program is probably what you need.
That is not so good for an end-user, and we would not expect to find assertions in production code. Most people would just comment out **`assert`** calls after testing, but there are other ways of ignoring them.
The `-O` command-line option (`O` for Optimise) will ignore all **`assert`** statements:

```
python -O scriptname
```

This sets the builtin variable **`__debug__`** to false (default is true). The **`__debug__`** variable cannot be assigned, so it cannot be altered in a conventional way.
The same effect can be made by setting the environment variable PYTHONOPTIMIZE to a value such as 0 (this is the optimization level). An empty PYTHONOPTIMIZE environment variable will raise `AssertionError` exceptions, any value will ignore them.
In the example, we are using the built-in **`all()`** to test that all the arguments to the function are True. This is a common assertion to make, but remember that zero is False yet might be a legitimate value.

The raise statement

- **Throw a standard exception object, with data**
- Syntax change at Python 3

```
def my_func(*arguments):
    if not all(arguments):
        raise ValueError('False argument in my_func')

try:
    my_func('Tom', '', 42)
except ValueError as err:
    print('Oops:', err, file=sys.stderr)
```

```
Oops: False argument in my_func
```

- **If no exception is specified:**
- Repeat the current active exception
- If no current exception, raise TypeError

py3

The **`raise`** statement syntax change at Python 3. We used to throw strings, but in Python 3, the system has been tided and we throw standard exceptions. We can get the same effect by passing a string argument to, for example, `ValueError`. We can write to the __cause__ attribute (not normally set) using, for example:

`raise` *some_exception* `from` *another_exception*

We can also create our own class and derive it from exception, or one of the other subclasses. This is discussed on the next slide.

Raising our own exceptions

- Define our own exception class

```
class MyError(Exception):
    pass

def my_func(*arguments):
    if not all(arguments):
        raise MyError('False argument in my_func')

try:
    my_func('Tom', '', 42)
except MyError as err:
    print('Oops:', err, file=sys.sdterr)
```

An empty class derived from exception

```
Oops: False argument in myfunc
```

- In Python 3 we no longer raise string exceptions

In early versions of Python, we passed a string to **`raise`**, and caught it using **`except`** (similar to C++). This has been corrected in Python 3 - we can only raise exceptions, not strings.
The syntax is still very simple, and it will not be unusual to see several customised exception classes in an application. The example shown is empty, and that will normally be the case.
The full syntax of the **`raise`** statement (note: it is not a built-in) includes the suffix `with traceback(`*traceback*`)`.
The traceback part would be used to carry forward data from a previous exception, for example:

```
except FileNotFoundError as err:
        raise MyError('something went wrong') from
err
```

It is also possible to create your own tracebacks using the **`with_traceback()`** exception method.

Getting tracebacks

- **sys.exc_info()**
  - Returns a tuple of type, value, and a *traceback* object
  - The traceback object incudes attributes such as the line number where the exception occurred, and the stack trace

```
try:
    open("some file name")
except FileNotFoundError as err:
    tipe, val, tb = sys.exc_info()
    print("Exception lineno:", tb.tb_lineno)
```

- **The traceback module**
  - Includes utilities to trace back through an exception cascade
  - `traceback.print_exc()` is short for `traceback.print_exception(*sys.exc_info())`

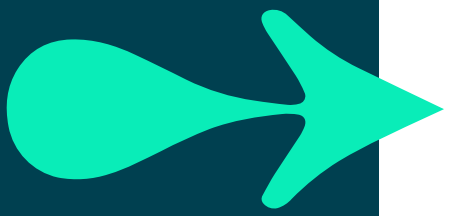
Exceptions can occur in heavily nested function calls, and even exceptions can be nested. Several modules in the standard library can help us.
sys has three variables, `sys.last_type`, `sys.last_value`, and `sys.last_traceback`, conveniently retrieved as a tuple by `sys.exc_info()`. The traceback object can be used, although it is generally not accessed directly, but via the **`traceback`** module. **`traceback`** has several useful methods that act on the current exception, but the most commonly used is `traceback.print_exc()`, which prints the exception stack-trace in the same way as the interpreter. For example:

```
class Exc1(Exception):
    pass
def func1():
    try:
        open("some file name")
    except OSError as err:
        raise Exc1(err)
import sys, import traceback
try:
    func1()
except:
    traceback.print_exc()
```

The exception information is shown on the console terminal, which might not be desirable, so `traceback.format_exc()` returns a

string in the same format which can used for your own error logging.

SUMMARY



- **At its simplest level, write error messages to stderr**
- **Most modern languages support exception handling**
- It is particularly suited to object orientation
- **Exceptions are built-in to Python**
- Many built-ins raise exceptions
- **Exceptions are not necessarily an error**
- **Handle it!**
- Trap code with try:
- Handle with except:
- Also support else: and finally:
- **We can also raise our own exceptions**
- **Use assert for boolean tests, but not for production code**

Context managers - with

- **Context managers execute entry and exit code**
- Special methods `__enter__` and `__exit__`
- `__exit__` may handle exceptions, or close resources
- Used with `with`

```
with context_object as variable:
    code
```

- File objects are context objects

→ Means we do not need finally blocks

```
with open('gash.txt', 'r') as var:
    for line in var:
        print(line, end='')

print(var)
```

```
<_io.TextIOWrapper name='gash.txt' encoding='cp1252'>
```

Context managers were introduced in Python 2.5. Their advantage is that we can initialise and destroy objects within its context. A destructor might not get called immediately, but the `__exit__` method will be called on exit from the context, even in the event of an exception.