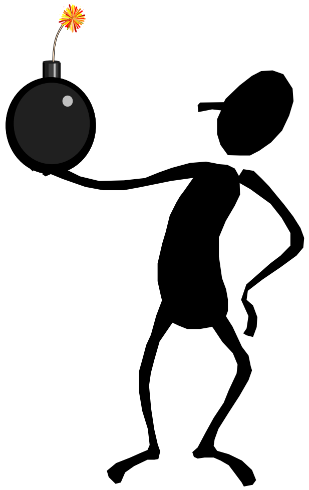
python™ 2 programming

Error Handling and Exceptions

Error Handling and Exceptions

- Contents

- Summary



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 changes 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\n")
    exit(1)
```

```
print >> sys.stderr,"Invalid types compared"
```

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 catagory
- Can be controlled through the -Wd command-line option
  - This makes warnings visible that are usually ignored
  - From 2.7 DeprecationWarnings are not displayed unless turned on using -Wd or a warnings filter

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','.*')

from struct import *
pack('i', 1.111)

print "Ending..."
```

Equivalent to -Wd option
RegExp filter (all warnings)

This raises a DeprecationWarning

Will not be executed

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

Exception syntax

- **Unhandled exceptions terminate the program**
- **Trapping an exception:**

```
try:
    code body
except (exception_tuple),var:
    exception handler
else:
    statements if no exception
finally:
    final statements
```

Optional, not executed if an exception occurs

Optional, always* executed

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 IOError:
    errmsg = "Could not open foo"
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!

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:

```
try:
    f = open("foo")
except IOError, err:
    sys.stderr.write("Could not open "+ err.filename +
                     ": " + err.strerror + "\n")
    sys.stderr.write(",".join(("Exception arguments",
                               str(err.errno),
                               err.strerror,
                               err.filename)))
```

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

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 myfunc():
    try:
        f = open("foo")
    finally:
        print >> sys.stderr,"Finally block"

try:
    myfunc()
except EnvironmentError:
    print >> sys.stderr,\
             "An Environment error occurred"
```

stderr

```
Finally block
An Environment error occurred
```

Order of execution

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

```
def myfunc():
    try:
        f = open("foo")
    except IOError, err:
        print err
    else:
        print "Everything is OK"
    finally:
        print >> sys.stderr, "Finally[illegible]lock"

try:
    myfunc()
except EnvironmentError :
    print >> sys.stderr, "An env erro[illegible]

print "We are all done"
```

1

2 If an exception was raised

2 If an exception was not raised

3

4 If an exception was trapped

at end If all exceptions were handled

The Python exception hierarchy

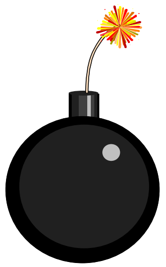
```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StandardError
      |    +-- BufferError
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- EnvironmentError
      |    +-- IOError
      |    +-- OSError
      |         +-- WindowsError (Windows)
      |         +-- VMSError (VMS)
      +-- EOFError
      +-- ImportError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
```

```
      +-- NameError
      |    +-- UnboundLocalError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
```

A common mistake

- Don't trap Exception
  - Can mask logic errors in your code
  - Trap a class lower in the exception tree
    - Generally we have a good idea of the expected errors
- It is also possible not to specify an exception
  - This traps every exception class below BaseException
  - Is even worse than trapping Exception!

```
try:
    f = open("foo")
except :
    print "Something happened"
```



- Don't do this at home!

The raise statement

- **Throw a standard Exception object, with data**

```
def myfunc(*arguments):
    if not all(arguments):
        raise ValueError("False argument in myfunc")

try:
    myfunc('Tom','',42)
except ValueError, err:
    print >> sys.stderr,"Oops:",err
```

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

- **If no Exception is specified in `raise`:**
  - **Repeat the current active Exception**
  - **If no current Exception, raise TypeError**

Raising our own Exceptions

- **Define our own Exception class**

```
class MyError(Exception):
    pass

def myfunc(*arguments):
    if not all(arguments):
        raise MyError,"False argument in myfunc"

try:
    myfunc('Tom','',42)
except MyError, err:
    print "Oops:",err
```

An empty class derived from Exception

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

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]
```

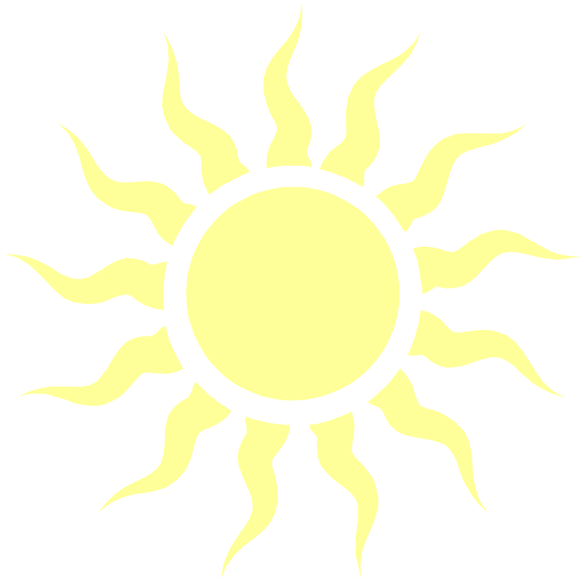
```
def myfunc(*arguments):
    assert all(arguments), "False argument in myfunc"
    ...
myfunc('Tom','',42)
```

```
...
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

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,

print var
```

```
<closed file 'gash.txt', mode 'r' at ...>
```