POWERING POTENTIAL

Exception Handling

Module 17

EXCEPTION HANDLING

Contents

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_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

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!

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

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
```

Order of execution

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

```
def my_func():
    try:
        f = open("foo")
    except FileNotFoundError as err:
        print(err)
    else:
        print("Everything is OK")
    finally:
        print("Finally block", file=     stderr)

try:
    my_func()
except OSError:
    print("An OS error occurred", fil

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 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`.

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
```

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
```

py3

- **If no exception is specified:**
  - 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 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

SUMMARY

- **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**