Error Handling

- Objectives
  - Catch and handle errors
  - Learn about Python’s exception hierarchy
  - Use tracebacks to quickly locate errors
  - Define custom errors and exceptions
  - Raise built-in and custom errors

Error handling background

- Errors are communicated via ***exceptions***
  - For code you write
  - For built-in errors
    - syntax errors
    - file IO errors

Exception hierarchy

```
BaseException
    SystemExit
    KeyboardInterrupt
    Exception
          StandardError
             ArithmeticError
             ImportError
             LookupError
             MemoryError
             NameError
                 UnboundLocalError
             ReferenceError
             RuntimeError
                 NotImplementedError
             SyntaxError
                 IndentationError
                     TabError
             ValueError
```

System errors are BaseExceptions

Errors you handle (or define!) derive from Exception

Unfortunately this includes some code level errors.

Note: All meaningful exceptions end in ‘Error’

Common exceptions

Exception Type	Purpose or situation when encountered
`Exception`	All built-in, non-system-exiting exceptions are derived from this class
`StandardError`	The base class for all built-in exceptions
`ArithmeticError`	Various arithmetic errors
`LookupError`	A key or index used on a mapping or sequence is invalid: `IndexError, KeyError`
`EnvironmentError`	Exceptions that can occur outside the Python system: `IOError`, `OSError`
`AttributeError`	An attribute reference or assignment fails (e.g. `u.name` is read only)
`KeyboardInterrupt`	The user hits the interrupt key (normally Control-C)
`MemoryError`	When an operation runs out of memory
`NotImplementedError`	In user defined base classes, abstract methods should raise this exception

Unhandled errors

- Tracebacks are history of the call that lead to the exception
  - They are display in ‘reverse’ order (oldest → newest)

```
# user 11 doesn’t exist
find_user(11)
```

When there is an error, execution stops and (without error handling) a **traceback** is displayed (AKA stacktrace)

Original caller

Source of first error

```
Traceback (most recent call last):
  File "D:/exceptions.py", line 24, in <module>
    find_user(11)
  File "D:/exceptions.py", line 16, in find_user
    sketchyMethod(userId)
  File "D:/exceptions.py", line 9, in sketchyMethod
    raise IndexError("The index 11 was not found")
IndexError: The index 11 was not found

Process finished with exit code 1
```

Catching exceptions [bare]

Code which may result in an error

```
try:
    function_which_may_cause_error()
    another_risky_function()
except:
    print("Sorry, that didn’t work out so well.")
```

Something failed, but we don’t know what or have any details.

Catching and handling exceptions [with object]

Code which may result in an error

```
try:
    function_which_may_cause_error()
    another_risky_function()
except Exception as e:
    print("Error: " + str(e))
```

Catching an exception object gives some indication what happened.

Catching and handling exceptions [by type]

Code which may result in an error

```
try:
    u = find_user(11)
    u.registered = true
    save_user(u)
except UserNotFoundError as e:
    print("The user with ID {0} doesn't exist".
          format(e.user_id))
except Exception as e:
    print("Error: " + str(e))
```

Error conditions can be segregated by error type with multiple except blocks

Types **must** be listed from **most specific to most general**

Catching and handling exceptions [with finally]

Code which
may result in
an error

```
fout = create_file_stream()
try:
    u = find_user(11)
    u.registered = true
    save_user(u)
    fout.write("User updated")
except Exception as e:
    print("Error: " + str(e))
 finally:
    fout.close()
```

Exception block is optimal (do you
want to handle the error here?)

finally block will always run

Raising errors

Use **raise** keyword to 'throw' the error.

```
def find_user(userId):
    if userId <= 0:
        raise TypeError("User ID cannot be negative")

    user = repository.find_user(userId)

    if not user:
         raise UserNotFoundError(userId)

    # work with user...
```

Custom exceptions

- Creating your own exceptions is as easy as creating a class.

Should end in **Error**

Must derive from **`Exception`** (not `BaseException`)

```
class UserError(Exception):

    def __init__(self, user_id, msg=""):
        self.user_id = user_id
        self.message = msg

        baseMsg = "userId = {0}, message = {1}".format(
                user_id, msg)

        super().__init__(baseMsg)
```

Pass the message, other data, along to the base Exception

Capture custom fields

```
if not user:
   raise UserError(userId)
```

Deterministic cleanup [other classes]

with block ensures cleanup (effectively try / finally)

```
def cleanup_method():
    with create_file(r"d:\temp\test.txt") as fout:
        fout.write("This is a test")
        print("wrote file...")
```

declare variable for guarded type

`fout.close()` is called right here.

Summary

- Use try / except blocks to handle errors
- Python has a good, but imperfect exception hierarchy
- Tracebacks contain most error info needed to debug
- Custom exceptions should derive from Exception
- Raise exceptions using the raise keyword