CS 3101-1 - Programming Languages: Python

Lecture 5: Exceptions / Standard Library

Daniel Bauer (bauer@cs.columbia.edu)

October 08 2014

Contents

Exceptions

Standard Library

Exceptions

Standard Library

Programs are Error Prone

1. Syntax errors.
2. Incorrect program behavior (wrong result).
3. Errors at runtime
   - Name errors (undefined variables).
   - Type errors (operation not supported by type).
   - Numeric Errors (division by 0).
   - IO errors (file not found, cannot write to file...).
   - ...

Programs are Error Prone

1. Syntax errors. ⇐ Detected by interpreter
2. Incorrect program behavior (wrong result).
3. Errors at runtime
   - Name errors (undefined variables).
   - Type errors (operation not supported by type).
   - Numeric Errors (division by 0).
   - IO errors (file not found, cannot write to file...).
   - ...

Programs are Error Prone

1. Syntax errors. ⇐ Detected by interpreter
2. Incorrect program behavior (wrong result). ⇐ Testing (later)
3. Errors at runtime
   - Name errors (undefined variables).
   - Type errors (operation not supported by type).
   - Numeric Errors (division by 0).
   - IO errors (file not found, cannot write to file...).
   - ...

Programs are Error Prone

1. Syntax errors. ⇐ Detected by interpreter
2. Incorrect program behavior (wrong result). ⇐ Testing (later)
3. Errors at runtime ⇐ Exception Handling
   - Name errors (undefined variables).
   - Type errors (operation not supported by type).
   - Numeric Errors (division by 0).
   - IO errors (file not found, cannot write to file...).
   - ...

Exceptions

- Exception = “Message” object that indicates an error or anomalous condition.
- When an error is detected, Python *raises* an exception.
- Exception is propagated through the call hierarchy.
- Exception can be *handled* by the program.
- If the exception is not handled and arrives at the top-level:
  - Program terminates.
  - Error message and traceback report is printed.

Example Error and Traceback

- Traceback contains the path the exception took through the call hierarchy
- Includes module names, function names, line numbers.

error_test.py

```
def foo(a):
    x = bar(a)
    print('done.')
    return(x)

def bar(b):
    return b[0] / b[1]
```

```
>>> foo(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "error_test.py", line 2, in foo
    x = bar(a)
  File "error_test.py", line 7, in bar
    return b[0] / b[1]
TypeError: 'int' object is not subscriptable
```

try ... except statements (1)

- If an error occurs in the block indented below try:
  - Execution is interrupted at the point of error.
  - Optional except block is executed if exception has the right type (exception captured in local variable).
  - Execution is resumed below the try ... except block.

```
def foo(a):
   try:
     x = bar(a)
   except TypeError, ex:
     print('caught error.')
     print(ex)
     x = None
   print('done.')
   return(x)

def bar(b):
  return b[0] / b[1]
```

```
>>> foo([4,2])
done.
2
>>> foo(42)
caught error.
'int' object is not
    subscriptable
done.
```

try ... except statements (2)

- Can use multiple except blocks for different types:

```python
try:
   x = bar(a)
except TypeError: # Binding the exception
                  # object is optional
   print('caught Type error.')
except ZeroDivisionError, ex:
   print('caught div0 error.')
```

- Can use tuple of exception types.

```python
try:
   x = bar(a)
except (TypeError, ZeroDivisionError):
   print('caught either a type or a div0 error.')
```

- No exception type: catch all exceptions (use sparingly!).

```python
try:
   x = bar(a)
except:
   print('caught some exception.')
```

`try ... except ... else`

- Optional `else` block is run only if `try` block terminates normally.
- Avoids unintentionally handling exceptions that occur in the `else` block.

```
try:
    x = bar(a)
except ZeroDivisionError:
    print('caught a div0 error from bar.')
else:
    try:
        y = 72 / x # Can cause a different
                   # div0 error!
    except ZeroDivisionError:
        print('caught another div0 error.')
```

try ... except ... finally

- finally block is executed no matter what!
  - When the try block terminates normally.
  - When an exception is caught.
  - Even if break, return, continue is called or another exception is raised.

```python
def foo(x):
    try:
        y = x[0]
        return y
    except IndexError:
        return 0
    finally:
        print("Done.")
```

```
>>> foo([])
Done
0
>>> foo([42])
Done.
42
>>> foo(42)
Done.
...
TypeError: 'int' object is
    not subscriptable
```

Raising Exceptions

- Exceptions can be raised if internal errors occur.
- Exceptions can be initiated explicitly with `raise`.
- First expression: Exception class
- Second expression: passed to exception class `__init__`.

```
x = raw_input()
if x == "yes":
    foo()
elif x == "no":
    bar()
else:
    raise ValueError, \
          "Expected either 'yes' or 'no'."
```

Passing on Exceptions

- Can pass on Exceptions through the call hierarchy after partially handling them.

```
def foo(x):
    try:
        y = x[0]
        return y
    except IndexError:
        print("Foo: index 0 did not exist.")
        print("Let someone else deal with it.")
        raise # Re-raise exception
```

Exceptions in the Iterator Protocol (review)

- `__iter__(self)` method that returns itself.
- `next(self)` method that returns the next element.
  - if no element is left, calling `next(self)` raises a `StopIteration` exception.
- Python 3: `__next__(self)`

```
class ReverseIterLst(list):
   def __iter__(self):
      self.index = len(self)
      return self
   def next(self):
      if self.index == 0:
         raise StopIteration
      else:
         self.index -= 1
         return \
          self[self.index]
```

```
>>> l = \
 ReverseIterLst([1,2,3])
>>> for x in l:
...    print x
...
3
2
1
```

Built-in and Custom Exceptions

- List of built-in exceptions:
  `http://docs.python.org/library/exceptions.html`
- Can write our own exceptions (exceptions are classes):
  - Can subclass any of the defined Exceptions (try to be as specific as possible).

```
class EmptyListException(IndexException):
    """ An Exception that indicates that we found an
        empty list.
    """

def foo(x):
    try:
        y = x[0]
        return y
    except EmptyListException, ex:
        sys.stderr.write(
        "Argument list cannot be empty.\n")
        return None
```

Using Exceptions Properly

- Write exception handlers only if you know how to handle the exception (i.e. it's easy to back-off or the exception is normal behavior).
- Except specific exception classes, rather than Exception or StandardError (can mask unexpected errors).
- Raise informative exceptions rather then just terminating the program.
- Some recommend to use exceptions for control flow (I don't!):
  - Easier to Ask for Forgiveness than for Permission (EAFP).

```
x = {'a':1, 'b':2, 'c':1}
y = {}
for a in x:
    try:
        y[x[a]].append(a)
    except KeyError:
        y[x[a]] = [a]
```

with Statement (1)

- Need to handle resources (e.g. files): acquire, use, free.
- Consider the problem of closing a file after using it, no matter what:

```
# Acquire resource
f = open(’trash.txt’, ’w’)
try: # Do something that can fail
    f.write(’Spam.\n’)
finally: # Clean-up and free resource
    f.close()
```

- Can instead use:

```
with open(’spam.txt,’w’) as f:
    f.write(’Spam.\n’)
```

- file object provides functionality to set up and do clean-up.

with Statement (2)

with works with any object implementing the *context manager* protocol.

- __enter__(self) is invoked when with statement is entered.
- __exit__(self) replaces finally block and is executed after the with block terminates (normally or with an exception).

This

```
with expression as name:
     statement
```

Translates roughly to

```
_tmp = expression
name = _tmp.__enter__()
try:
    statement
finally:
    temporary.__exit__()
```

Exceptions

Standard Library

Standard Library

- So far: structure of the programming language itself.
- Python comes with a ‘batteries included’ philosophy.
  - A lot of built-in functionality.
  - Large standard library of modules.
- Will only cover some important / representative modules.
- See docs for more: `http://docs.python.org/library/index.html`

Some Important Modules (1)

General Purpose:

- `sys` - **Access runtime environment.**
- `collections` — **More Container Datatypes**
- `itertools` - Fancy iterators.
- `functools` - Functional programming tools.
- `math` - Mathematical functions.
- `pickle` - Save data to files.
- `subprocess` - Spawn child processes.

Strings:

- `re` - **Regular Expressions.**
- `codecs` - Encoding/Decoding strings.

File I/O:

- `os` - interact with the operating system.
- `os.path` - **pathname operations / browse fs.**
- `gzip, bz2, zipfile, tarfile` - compressed archives.
- `csv` - read/write comma separated value file.
- `xml` - XML utils

Some Important Modules (2)

Internet / Networking:

- `socket` - low-level networking.
- `urllib` - Open resources by URL.
- `cgi` — CGI scripts.
- `smtplib` / `email` - send e-mail.
- `json` - use JSON encoded data.

GUI:

- `TKinter` - built-in GUI

Debugging / Profiling:

- `logger` - built-in logging
- `pdb` - Python debugger
- `trace` - Trace statement execution.
- `hotshot` - Profiler

sys

System (i.e. interpreter)-specific parameters and functions.

sys Module - IO Stream File Objects

- sys.stdin is the terminal input.
- sys.stdout is the terminal output.
- sys.stderr is the error stream.
  - By default stderr is printed to terminal as well.
  - In UNIX/Linux/Mac: can ‘pipe’ different streams to files

```
$ python error_test.py  >stdout.out 2>stderr.log
$
```

sys Module - path

- sys.path - a list of strings that determine where python searches for modules.
  - Environment variable PYTHONPATH is appended to default path.

```
>$ export PYTHONPATH="$PYTHONPATH:/Users/daniel/
    project/"
>$ python
Python 2.7.2 (default, Jan 21 2012, 18:42:05)
[GCC 4.2.1] on darwin
Type "help", "copyright", "credits" or "license" for
    more information.
>>> import sys
>>> sys.path
['', '/Library/Python/2.7/site-packages',
 '/Users/daniel/project/']
```

sys Module - Terminating Interpreter

- `sys.exit([arg])` - terminate the interpreter immediately.
- `arg` is the error code:
  - 0 - successful termination.
  - 1 - termination with error.

```
>>> import sys
>>> sys.exit(0)
daniel:$ _
```

sys Module - Command Line Arguments

- sys.argv is a list containing command line arguments.
  - sys.argv[0] is the name of the script
  - all other elements are arguments passed to the script.

test_args.py

```
import sys
print sys.argv
```

```
daniel:$ python test_args.py
['test_args.py', 'foo', 'bar']
```

Collections

High-Performance Container Datatypes

collections Module - defaultdict

- A dictionary class that automatically supplies default values for missing keys.
- Is initialized with a *factory* object, that creates the default values.
  - Can be a function or a class object (calling a class instantiates it).
  - Can be a basic type (list, set, dict, int initializes to 0).

```
>>> from collections import defaultdict
>>> x = defaultdict(list)
>>> x['yellow'].append('banana')
>>> x
defaultdict(<type 'list'>, {'yellow': ['banana']})
>>> x['pink']
[]
```

collections Module - Counter

- Easy interface to count hashable objects in collections (often strings).
- Once created, they are dictionaries mapping each object to its count.
- Support method most_common([n])
- Can be updated with other counters or dictionaries.

```
>>> from collections import Counter
>>> x = Counter(’banana’)
>>> x
Counter({’a’: 3, ’n’: 2, ’b’: 1})
>>> x.most_common(2)
[(’a’, 3), (’n’, 2)]
>>> x.update({’b’:1})
>>> x[’b’]
2
```

os

Interacting with the operating system.

os.path

Filename Manipulation

os.path Module - manipulate pathnames

- `os.path.abspath(path)` - Returns the absolute pathname for a relative path.

```
>>> os.path.abspath("test")
'/Users/daniel/cs3101/lecture-3/test'
```

- `os.path.join(path1, path2, ... )` - Concatenates pathnames (system specific).

```
>>> os.path.join("Users","daniel","cs3101")
'Users/daniel/cs3101'
```

- `os.path.basename(path)` - Returns the basename of a path (” for dirs).

```
>>> os.path.basename("/Users/daniel/test.txt")
'test.txt'
```

- `os.path.isfile(path)` - returns `True` if the path points to a file.
- `os.path.isdir(path)` - returns `True` if the path points to a directory.

os Module - list content of a directory

- os is a powerful module for interacting with the operating system.
- For homework: os.listdir(path) lists files in a directory.

```
>>> os.listdir("/Users/daniel/listtest/")
['test1.txt', 'test2.txt']
```

Pickle

Object serialization / Data persistence

Pickle Module - Object serialization

- Provides a convenient way to store Python objects in files and reload them.
- Alows saving/reloading program data or transferring them over a network.
- Can pickle almost everything:
  - All standard data types
  - User defined functions, classes and instances.
  - Works on complete object hierarchies.
  - Classes and functions need to be defined when un-pickling.

```
with open(’pickled_foo.pickle’,’w’) as f:
    pickle.dump(foo, f)
```

```
with open(’pickled_foo.pickle’,’r’) as f:
    foo = pickle.load(f)
```

Pickle Module - Protocols and cPickle

- Normally pickle uses a plaintext ASCII protocol.
- Newer protocols available.
  - 0 - ASCII protocol
  - 1 - old binary format (backward compatible)
  - 2 - new protocol ($\geq$ python 2.3, more efficient)
- More efficient reimplementation of Pickle in C.
  - Always use this for large object hierarchies (up to 1000x faster).

```
import cPickle
with open('pickled_foo.pickle','wb') as f:
    cPickle.dump(foo, f, protocol = 2)
```