Writing Good Error Messages

Paul Keating
keating@acm.org
europython.2018@boargules.com
Roadmap

Introduction

Part 1: Good error messages are useful error messages

Part 2: A little framework for useful messages
Introduction

Paul Keating
on the internet: BoarGules

- Started with Python 1.5.2
- Has programmed in Python for a living since 1999
- First came to EuroPython at Charleroi, Belgium in 2003
- Supports an application with embedded languages
  - Python – with 2 huge APIs
  - An SQL dialect
  - A spreadsheet-like formula-oriented language
  - Other domain-specific mini-languages
- … that can all call one another
Useful for Who?

Be clear about who you expect to be reading your message:

● Interactive application: reader is probably an end-user
● Batch program: reader is a developer or does application support
● API: reader is a programmer (but maybe not very experienced)

Sometimes there are two audiences:

● The programmer who is calling into your library module
● The end-user of that programmer’s application
Is it understandable?
Is it understandable?

A stack trace is

- Indispensable to a programmer
- Of some value to a superuser
- Gobbledygook to everyone else
- Not useful to anyone without access to the source code
Is it understandable?

```python
phone_numbers = {
    "Paul": "+31641890432",
    "Emergency": "112",
    "Voicemail": "+316240641890432"
}

num = phone_numbers(customer)
```

Traceback (most recent call last):
  File "Tutorial101", line 26, in lookup_number
    num = phone_numbers(customer)
TypeError: 'dict' object is not callable
Is it understandable?

ResultSet object has no attribute 'prefix'.
Is it understandable?

ResultSet object has no attribute 'prefix'. You're probably treating a list of items like a single item. Did you call `find_all()` when you meant to call `find()`?
class ResultSet(list):
    """A ResultSet is just a list that keeps track of the
    SoupStrainer that created it."""
    def __init__(self, source, result=()):
        super(ResultSet, self).__init__(result)
        self.source = source
    def __getattr__(self, key):
        raise AttributeError("ResultSet object has no attribute '%s'. You're probably treating a list of items like a single item. Did you call find_all() when you meant to call find()?" % key)
Is it explicit?

try:
    FSettlementProcess.CreateSettlementsFromTrade(trade,
        defaultProcessMessage, nettingRuleQueryCache)
except:
    logger.Log("Something went wrong with trade {0}"
        .format(trade.Oid()))
Is it explicit? – traceback is your friend

try:
    FSettlementProcess.CreateSettlementsFromTrade(trade,
        defaultProcessMessage, nettingRuleQueryCache)
except:
    logger.Log("Something went wrong with trade {0}".format(trade.Oid()))
except RuntimeError:
    logger.Log("Unexpected failure while processing Trade {0}".format(trade.Oid()))
logger.Log(traceback.format_exc())
Is it unambiguous?

```python
if payment.original() and payment.original().type != payment.type:
    if not payment.type in LimitedPaymentType:
        raise ValidationError('Users with profile component "Add ' 'Pmts to Simulated" can only use limited fee types.')

...(many lines of code...)

if not (payment.original() and not payment.type in LimitedPaymentType):
    raise ValidationError('Users with profile component "Add ' 'Pmts to Simulated" can only use limited fee types.')
```
Is it unambiguous?

```python
if payment.original() and payment.original().type != payment.type:
    if not payment.type in LimitedPaymentType:
        raise ValidationError('Users with profile component "Add ' 'Pmts to Simulated" can only use limited fee types.

(...many lines of code...)

if not (payment.original() and not payment.type in LimitedPaymentType):
    raise ValidationError('Users with profile component "Add ' 'Pmts to Simulated" can only use limited fee types..')
```
Does it point in the right direction?

def validate_settlement(settle, action):
    if settle.record_type == 'Settlement':
        import FValidationSettlement
        FValidationSettlement.settlement_validations (settle, action)

    try :    validate_settlement(e, op)
except:   raise AttributeError("Error occurred in " "call to validate_settlement")
def validate_settlement(settle, action):
    if settle.record_type == 'Settlement':
        import FValidationSettlement
        FValidationSettlement.settlement_validations(settle, action)

    try:   validate_settlement(e, op)
    except: raise AttributeError("Error occurred in call to validate_settlement")

    validate_settlement(e, op)
Does it work?

```python
try:
    curve_name = get_default_spread_curve(ins)
    (...many lines of code...)
    ins.Commit()
except Exception as e:
    print("Cannot commit Instrument {0} on {1}\n{2}".format(ins.Name(), curve_name, e))
```
Does it work?

```python
try:
    curve_name = get_default_spread_curve(ins)
    (...many lines of code...)
    ins.Commit()
except Exception as e:
    print("Cannot commit Instrument \{0\} on \{1\}\n\{2\}".format(ins.Name(), curve_name, e))
```

But suppose the exception is in here somewhere...
Does it work? *Exception chains*

Traceback (most recent call last):
File "FAutoLink", line 303, in get_default_spread_curve
    u = ins.Underlying().Name()
AttributeError: 'NoneType' object has no attribute 'Name'

During handling of the above exception, another exception occurred:
Traceback (most recent call last):
File "FAutoLink", line 545, in link_instrument
    print("Cannot commit Instrument {0} on {1}\n{2}".format(ins.Name(), curve_name, e))
UnboundLocalError: local variable 'curve_name' referenced before assignment
Does it work? No exception chaining in Python 2

Traceback (most recent call last):
File "FAutoLink", line 545, in link_instrument
    print("Cannot commit Instrument {0} on {1}\n{2}".format(ins.Name(), curve_name, e))
UnboundLocalError: local variable 'curve_name' referenced before assignment
Does it work? Force a zero-divide to test the message

try:
    temp = 2 / 0
    curve_name = get_default_spread_curve(ins)
    (...many lines of code...)
    ins.Commit()
except Exception as e:
    print("Cannot commit Instrument {0} on {1}\n{2}".format(ins.Name(), curve_name, e))
Part 2:
A little framework for useful error messages

- The situation
- The requirement
- The solution
A little framework for useful error messages

The situation

- The software environment
- The people who write the error messages
- The validation rules
The software environment

Python is embedded in an application

Application calls your validation function before every database save

Your validation function gets the object about to be saved. It can

- Silently return (save succeeds)
- Change the object, then return (save succeeds)
- Raise an exception (application rejects save)
  - Even if you didn’t intentionally raise the exception
The software environment

Application rejects a save like this:
The people who write the error messages

May be professional coders, but may also be

- Back office superusers
- Risk managers
- Accountants
The validation rules

Complex corner-case validation is written by subject experts, not professional coders

End-users often report the message as a bug:

- “It won’t let me save this trade”
- “I ought to be allowed to do that/did the same thing yesterday”
- “Fix the error message”

Developers may also not understand the reason for the validation failure
A little framework for useful error messages

The requirement

- Simple cut’n’paste coding
- Must be possible to identify the rule (even if there are duplicate messages)
- Unintended exceptions must not bring the system to a halt
A little framework for useful error messages

The solution

- One simple class
- Distinguish between intentional exceptions and unintentional exceptions
Solution

Validation error class

class ValidationError(RuntimeError):
    def __init__(self, problem):
        RuntimeError.__init__(self,
            " 
{0} 
[{1}]".format(problem, _line()))

def _line():
    info = inspect.getframeinfo(
        inspect.currentframe().f_back.f_back)
    return "{0}:{2}:{1}".format(*info)
The software environment

Raising a ValidationError causes a pop-up that the end-user sees:
Solution

Validation callback

def validate_entity(entity, operation):
    try:
        my_validation_function(entity, operation)
    except ValidationError:
        raise
    except Exception:
        print("Untrapped exception in validation: please report "
              "to support team and include the traceback below")
        traceback.print_exc()
Solution

Validation callback

def validate_entity(entity, operation):
    try:
        my_validation_function(entity, operation)
    except ValidationError:
        raise
    except Exception:
        print("Untrapped exception in validation: please report "
              "to support team and include the traceback below")
        traceback.print_exc()
When your error message may have two audiences

Deliver different levels of message via different channels
- End-user message in pop-ups
- Stack traces in a log

Define your own exceptions (don’t just raise `RuntimeError`)

Raise different kinds of exception for
- Errors on the application programmer’s part
- Things you expect the application programmer can anticipate and translate
- Things neither of you can do anything about
To sum up ...

An error message is a call to action.

What do you expect the reader to do with your error message?

- Is it understandable?
- Is it explicit?
- Is it unambiguous?
- Does it point in the right direction?
Questions