Exceptions & error handling in Python 2 and Python 3

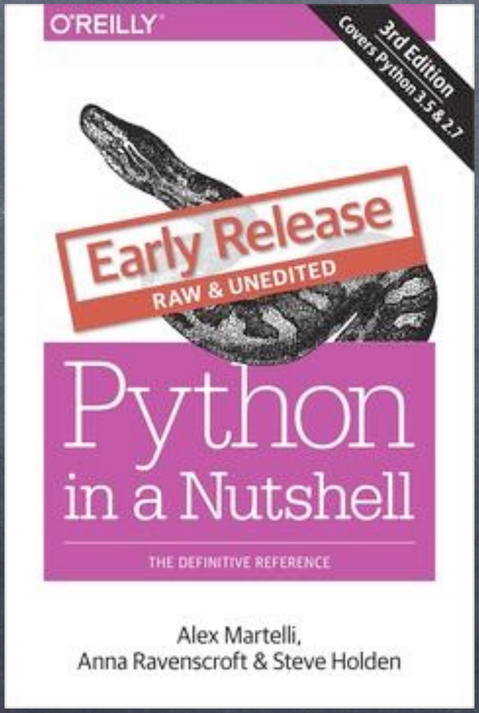
http://www.aleax.it/pycon16.pdf

Python in a Nutshell 3rd ed

This talk cover parts of Chapter 5 of the Early Release e-book version

50% off: http://www.oreilly.com/go/python45 disc.code TS2016; 40% on paper book pre-order

DRM-free e-book epub, mobi, PDF... copy it to all of your devices!



Send us feedback while we can still do major changes!

Exceptions: not always errors

```
>>> 1/0
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  ZeroDivisionError: division by zero

>>> next(iter([]))
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  StopIteration
>>>
```

Throughout, I use exc to mean exception

The `try` statement

`try`: ...block...

[`except` (type, type...) [`as` x]:] ...

(0+ `except` clauses, narrowest first)

[`else`:] ...

(optional: execute without exception guard iff no exc in block; must have 1+ `except`)

[`finally`:] ...

(optional: execute unconditionally at end; no `break`, `return` [`continue` forbidden])

The `raise` statement

`raise exception_object`

- must be an instance of `BaseException`
- (in v2, could be a subclass -- avoid that!)

`raise`

- must be in an `except` clause (or a function called, directly or not, from one)
- re-raises the exception being handled

When to raise and why

```
def cross_product(seq1, seq2):
  if not seq1 or not seq2:
    raise ValueError('empty seq arg')
  return [(x1, x2) for x1 in seq1
                   for x2 in seq2]
```

Note: no duplicate checks of errors that Python itself checks anyway, e.g `seq1` or `seq2` not being iterable (that will presumably give a `TypeError`, which is probably fine).

Exceptions wrapping (v3)

- v3 only (upgrade to v3, already!-)
- traceback is held by the exception object
  - `exc.with_traceback(tb)` gives a copy of exc with a different traceback
- last exc caught is `__context__` of new one
- `raise new_one from x` sets `__cause__` to x, which is `None` or exception instance

```
>>> def inverse(x):
...   try: return 1/x
...   except ZeroDivisionError as err:
...     raise ValueError() from err
>>> inverse(0)
Traceback (most recent call last):
  File "<stdin>", line 2, in inverse
ZeroDivisionError: division by zero
The above exception was the direct cause of the
following exception:
Traceback (most recent call last):
  File "<stdin>", line 4, in inverse
ValueError
>>> try: print('inverse is', inverse(0))
... except ValueError: print('no inverse there')
no inverse there
```

exc __context__ in v3

```
try: 1/0
except ZeroDivisionError:
    1+'x'

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero

During handling of the above exception,
another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
TypeError: unsupported operand type(s) for +:
'int' and 'str'
```

v2 would hide 1st exc; v3 shows both!

the `with` statement

`with x [as y]:` ...block...

...is roughly the same as...:

```
y = x.__enter__()
_ok = True
try: ...block...
except:
  _ok = False
  if not x.__exit__(*sys.exc_info()):
    raise
finally:
  if _ok: x.__exit__(None, None, None)
```

Making a context manager

```
@contextlib.contextmanager
def a_context_manager(args_if_any):
  # __init__ code if any
  try:
    # __enter__ code
    yield some_result
  except (handled, types) as exc:
    # __exit__ w/exception
    # maybe `raise` again if needed
  finally:
    # unconditional __exit__ code
```

Exceptions in generators

- caller may use `generator.throw(exc)`
  - like raise exc at generator's yield
  - in v2 may pass type, value, traceback
- typically used as...:

```
try: result = yield previous
except GeneratorExit: clean-up
```

- caller may use `generator.close()`
  - which is just like: `generator.throw(GeneratorExit())`

exception propagation

- exceptions "bubble up" through the stack of call frames (callee to caller to ...)
- until caught in an except clause
  - or `__exit__` in a context manager
- (as `finally` clauses or context managers are "bubbled through", they execute)
- if never caught, all the way to the top
  - where `sys.excepthook` may act
  - ...but essentially just reporting/logging
  - lastly, atexit-registered functions

exceptions hierarchy (v2)

BaseException
  Exception
    StandardError
      ...many types/subtypes...
      EnvironmentError
        IOError, OSError
    StopIteration
    Warning
  GeneratorExit
  KeyboardInterrupt
  SystemExit

exceptions hierarchy (v3)

BaseException
  Exception
    ...many types/subtypes...
    OSError (AKA: IOError, EnvironmentError)
      ...subtypes, e.g FileNotFoundError...
    StopIteration
   Warning
  GeneratorExit
  KeyboardInterrupt
  SystemExit

OSError subclasses (v3)

```
def read_or_def(path, default):
  try:
    with open(path) as f: return f.read()
  except IOError as e:
    if e.errno == errno.ENOENT:
      return default
    raise
def read_or_def(path, default):
  try:
    with open(path) as f:return f.read()
  except FileNotFoundError:
    return default
```

custom exceptions

- best practice: have your module or package define a custom exc class:
  - class Error(Exception): "docstring"
  - this lets client code easily catch errors specific to your module or package
- also use multiple inheritance to define your module's versions of standard exceptions:
  - class MyTE(Error, TypeError): "doc"
  - this also lets client code easily catch (e.g) TypeErrors wherever they come from

Strategies: LBYL, EAFP

- Look Before You Leap:
  - check that all preconditions are met
  - if not, raise exc (or otherwise give error)
  - if met, perform op (no exc expected)
- Easier to Ask Forgiveness than Permission
  - just try performing the operation
    - Python catches any prereq violations
    - and raises exc on your behalf
    - optionally catch to transform/enrich

LBYL problems

- duplicates work Python performs anyway to check preconditions
- obscures code clarity due to structure:
  - check, raise if it fails
    - ...(repeat N times)...
  - actual useful work (only at the end)
- some checks might erroneously be omitted
  - resulting in unexpected exceptions
- things (e.g filesystem) may change at any time (inc. between checks and operation!)

LBYL vs EAFP

```
def read_or_default(path, default):
  # spot the many problems...:
  if os.path.exists(path):
    with open(path) as f: return f.read()
  else:
    return default

def read_or_default(path, default):
  try:
    with open(path) as f: return f.read()
  except FileNotFoundError:
    return default
```

how to EAFP right

```
def trycall(obj, attr, default, *a, **k):
  try: return getattr(obj, attr)(*a, **k)
  except AttributeError: return default

def trycall(obj, attr, default, *a, **k):
  try: method = getattr(obj, attr)
  except AttributeError: return default
  else: return method(*a, **k)
```

Keep it narrow: DON'T guard too many operations within a `try` clause!

Errors in large programs

- consider all possible causes of errors...:
  - bugs in your code, or libraries you use
    - cover those with unit tests
  - mismatches between libraries' prereqs and your understanding of them
    - cover those with integration tests
  - invalid inputs to your code
    - great use for try/except!
    - remember: let Python do most checks!
  - invalid environment/config: ditto

Case-by-case handling

- the info you (the coder) need (to fix bugs etc) is NOT the same the user needs (to remedy invalid inputs/environment/etc)
- think of the user: everything else follows
  - never show the user a traceback
    - they can't do anything with it!
    - archive it, send to yourself (w/perm!), ...
  - design user error messages with care
    - focus on what they can/should do NOW!
- if feasible, restart the program (maybe with snapshot/restore; ideally with a "watchdog")

logging

- Python stdlib's logging package can be very rich and complex if used to the fullest
  - worth your effort! logs is how you debug (and optimize, etc) esp. server programs
  - ensure your logs are machine-parsable, design log-parsing scripts carefully
  - when in doubt, default to logging all info
    - program state, environment, inputs, ...
  - don't log just errors: logging.info can give you precious "base-case" comparisons too

avoid assert

- although `assert` seems an attractive way to check inputs and environment...
- ...it's NOT: it's an "attractive nuisance"!
- becomes no-op when you run optimized
  - but inputs &c can be wrong even then!
- often duplicates checks Python performs
  - it's usually a sub-case of LBYL...
- use ONLY for sanity checks on internal state (and as "executable docs") while you're developing and debugging your program!

Q & A

http://www.aleax.it/pycon16.pdf

http://www.oreilly.com/go/python45

? discount code: TS2016 !