O'Reilly Open Source Convention 2001 San Diego, CA Python Track Keynote

Guido van Rossum
Zope Corporation
guido@zope.com
guido@python.org

Sorry I Couldn't Make It!

- My wife spent a night in the hospital
  - Complications with her pregnancy
- She and our baby are doing fine now
  - But this is scary stuff!
- Our due date is November 2nd
  - I'll disappear for 3-6 weeks around then...

Where to Start?

- So much to talk about!
  - New Python release(s!)
  - New Python license
  - New Python logo
  - New corporate name
  - Have you flamed me on c.l.py recently?

New Corporate Name

- On Monday, Digital Creations officially changed its name to **Zope Corporation**
- Also known as **Zope**
- Website: **Zope.com**
- Zope CVS opened up
- Little else changes

New Python Logo

Designed by Just van Rossum and Erik van Blokland
www.letterror.com

Many Logo Variations

ZOPE

Python Software Foundation

- Owns and releases Python software
- Established in Delaware
- Bylaws on line: www.python.org/psf/
- Applying for non-profit status

Next Python Conference

- February 4-7, 2002
  - Alexandria, VA (near Washington, DC)
- Four tracks:
  - Refereed Papers
  - Zope
  - Python Tools
  - Business-to-Business
- CFP: see www.python10.org

New Python License

- At last, Python is GPL-compatible again
- Which versions are GPL-compatible?
  - 1.5.2 and before, 2.0.1, 2.1.1, 2.2 and later
  - But not: 1.6, 1.6.1, 2.0, 2.1
- Why were those not GPL-compatible?
  - Mostly, choice of law clause in CNRI license
- Who cares?
  - FSF; Debian, other binary release builders

What Is GPL-compatibility?

- GPL-compatibility allows release of Python linked with GPL-licensed library
  - For example, GNU readline
- Python is **Open Source Compliant**
  - A much more liberal requirement
- Python is **not** released under the GPL!
  - No "viral" requirements in license

Recent Python Releases

- 2.0.1 - GPL-compatible bug fix release
  - June 2001
- 2.1.1 - GPL-compatible bug fix release
  - July 2001
- 2.2a1 - first alpha of new release
  - July 2001; 2.2 final planned for October

What's A Bug Fix Release

- Idea introduced by PEP 6; thanks Aahz!
- Fix bugs without **any** incompatibilities
- Full binary and byte code compatibility
- **No new features**
  - I.e. full **two-way** compatibility; code developed under 2.1.1 will run the same under 2.1 (unless it hits a bug in 2.1, obviously); even extensions!
- Thanks to the release managers!
  - Moshe Zadka (2.0.1), Thomas Wouters (2.1.1)

About Python 2.2

- What's new?
  - Nested scopes are the law
  - Iterators and Generators
  - Type/Class unification
  - Unicode, UCS-4
  - XML-RPC
  - IPv6
  - New division, phase 1

Nested Scopes: It's The Law!

- Introduced in 2.1 with *future statement:*
  - from __future__ import nested_scopes
- Future statement is unnecessary in 2.2
  - But still allowed
- Motivating example:

```
def new_adder(n):
    return lambda x: return x+n
```

Iterators: Cool Stuff

- Generalization of for loop machinery
- New standard protocols:
  - 'get iterator' protocol on any object:
    - it = iter(x) # returns iterator for x
  - 'next value' protocol on iterator objects:
    - it.next() # returns next value
    - # raises StopIteration when exhausted

Iterators: For Loops

- Equivalency between these two:
  - for el in sequence: print el
  - __it = iter(sequence)
    while 1:
        try:
            el = __it.next()
        except StopIteration:
            break
        print el

Iterators: Why

- No need to fake sequence protocol to support 'for' loop (a common pattern):
  - for key in dict: ...
  - for line in file: ...
  - for message in mailbox: ...
  - for node in tree: ...
- Lazy generation of sequence values
- Can do iterator algebra, e.g. zipiter()

Iterators: More

- Some non-sequences have iterators:
  - Dictionaries
    - for k in dict: print key, dict[k]
    - for k, v in dict.iteritems(): print k, v
    - Related feature: if k in dict: print k, dict[k]
  - Files
    - for line in open("/etc/passwd"): print line,
  - Class instances
    - Define your own __iter__() method

Generators: *Really* Cool Stuff

- An easy way to write iterators
  - Thanks to Neil Schemenauer & Tim Peters
- from __future__ import generators

```
def inorder(t):
    if t:
        for x in inorder(t.left): yield x
        yield t.label
        for x in inorder(t.right): yield x
```

Generator: Example

- def zipiter(a, b):
    while 1:
        yield a.next(), b.next()
- for x, y in zipiter(range(5), "abcde"):
    print (x, y),
- (0, 'a') (1, 'b') (2, 'c') (3, 'd') (4, 'e')

Generators: Why

- Often an algorithm that generates a particular sequence uses some local state expressed by a combination of variables and "program counter"
- For example: tokenizer, tree walker
- Generating the whole sequence at once is nice but can cost too much memory
- Saving all the state in instance variables makes the algorithm much less readable
- Using a callback is cumbersome for the consumer of the values

Generators: How

- Presence of yield signals the parser
  - Implementation "suspends" the frame
- Calling the generator function:
  - Creates the frame in suspended mode
  - Returns a special-purpose iterator
- Calling the iterator's next():
  - Resumes the frame until a yield is reached
  - Raises StopIteration when upon return
    - Or upon falling off the end

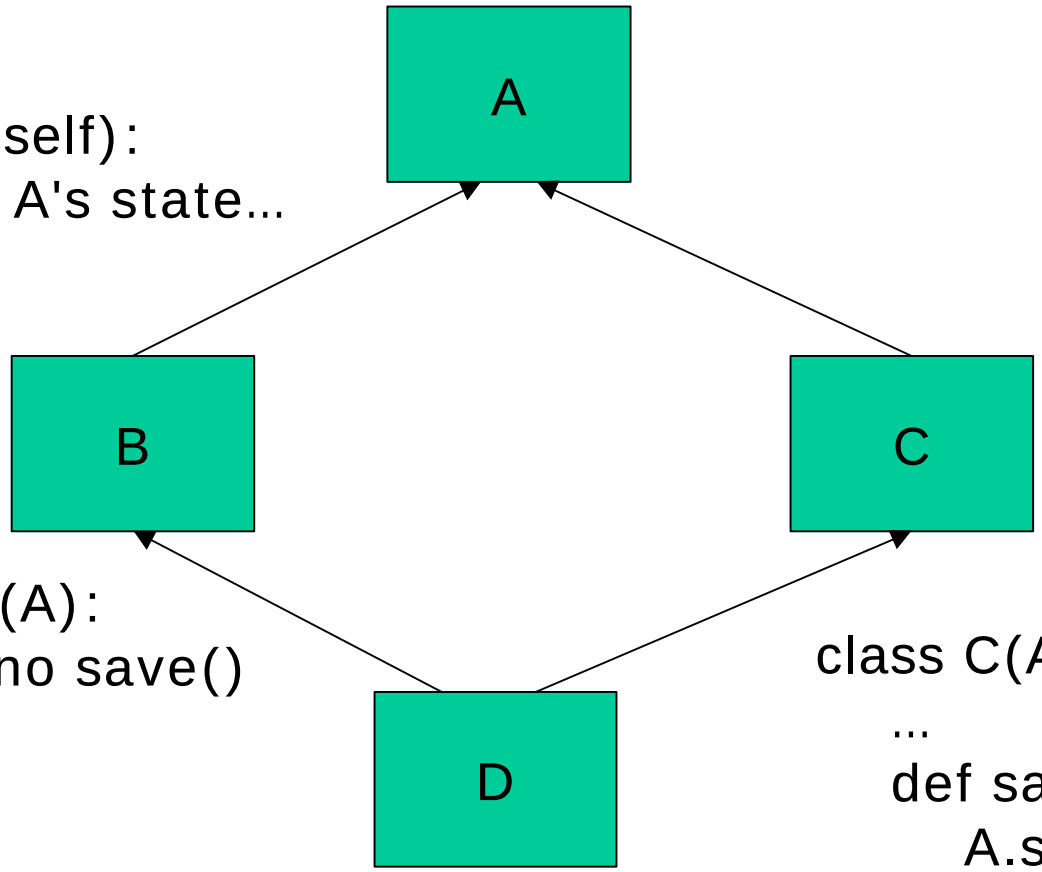
Type/Class Unification

- class mydict(dictionary):
  ```
      def __getitem__(self, key):
          try:
              return dictionary.__getitem__(self, key)
          except KeyError:
              return 0.0 # default value
  ```
- a = range(10)
- assert a.__class__ is type(a) is list
- list.append.__doc__
- list.append(a, 11)
- list.__getitem__(a, 4)

Method Resolution Order

- Order in which bases are searched for methods
- Trivial with single inheritance
- Relevant for multiple inheritance
- Classic Python rule: left-right depth-first
- Classic rules makes a lot of sense; natural extension of single inheritance rule for tree-shaped inheritance graph; but...
- Classic rule breaks down when there is a **common base class**

```
class A:
    ...
    def save(self):
        ...save A's state...
```




```
class B(A):
    ... # no save()
```

```
class C(A):
    ...
    def save(self):
        A.save(self)
        ...save C's state...
```

```
>>> x = D()
>>> x.save()
```

```
class D(B,C):
    ... # no save()
```

 # is C's state saved???

Proposed New MRO

- Informal requirements:
  - Same as classic rule when dependency graph is a tree (no common base classes)
  - Most derived method wins
- Algorithm embeds a topological sort in a total ordering
- In diamond example: [D, B, C, A]
- Metaclass can override MRO policy

Pros And Cons

- Pro: avoids the diamond surprise
- Pro: matches other languages' rules
- Con: harder to explain
  - But: same as classic unless shared base!

- IMO, classic rule makes common bases too hard to use
- Neither rule deals with **conflicts**

Common Base Class

- 'object' class: the ultimate base class
- Defines standard methods:
  - __repr__, __str__
  - __getattr__, __setattr__, __delattr__
  - __cmp__, __hash__, __lt__, __eq__, ...
- Override __getattr__ properly:

```
class C(object):
    def __getattr__(self, name):
        if name == 'x': return ...
        return object.__getattr__(self, name)
```

Conflicts

- What if both B and C define a save() method?
  - D has to implement a save() that somehow maintains the combined invariant
- Should the system detect such conflicts?
  - Probably, but maybe not by default, since current practice allows this and so flagging the conflicts as errors would break code. Maybe a warning could be issued, or maybe it could be a metaclass policy

Unicode, UCS-4

- ./configure --enable-unicode=ucs4
  - Not yet on Windows
- Compiles with Py_UNICODE capable of holding 21 bits
  - Trades space for capability to handle all 17 Unicode planes
- Alternative: UTF-16 and surrogates
  - Indexing characters would be too slow

XML-RPC

- XML-RPC: easy, interoperable RPC
- Code by Fredrik Lundh
- Client is a library module (xmlrpclib)
- Server frameworks in Demo/xmlrpc/
- See http://www.xmlrpc.com

IPv6

- ./configure --enable-ipv6
- Code by Jun-ichro "itojun" Hagino
- Integration by Martin von Loewis
- Modified socket module:
  - socket.getaddrinfo(host, port, ...)
  - socket.getnameinfo(sockaddr, flags)
  - socket.AF_INET6 address type, if enabled
- Supported by httplib

Have You Flamed Me On C.L.PY Recently?

- The problem: int and float are not really two types, more one-and-a-half...
- 1+x == 1.0+x, 2*x == 2.0*x, etc.
  - x can be int or float
  - Result has same mathematical value
- But **not**: 1/x == 1.0/x
  - Result depends on type of x

Why Is This A Problem?

- def velocity(distance, time):
    return distance/time
- velocity(50, 60) # returns 0
- Violated principle (only by division):
  - In an expression involving numerical values of different types, the mathematical value of the result (barring round-off errors) depends only on the mathematical values of the inputs, regardless of their types

In Other Words...

- There are really two different operators, "int division" and "float division", both of which make sense for numerical values
- In expressions yielding float results, int inputs are **usually** okay, except when division is used

Why Is This A Problem?

- Python has no type declarations
- In statically typed languages, velocity() would have explicit float arguments
- The work-around is really ugly:
  - x = float(x) # Broken if x is complex
  - x = x+0.0 # Broken if x is -0.0
  - x = x*1.0
    - Works, but what if x is a Numeric array...

Proposed Solution

- Eventually (in Python 3.0? :-)
  - Use / for float division, // for int division
- Transitional phase (at least two years):
  - Enable // immediately (in Python 2.2)
  - Use "from __future__ import division" to enable / as float division
  - Command line switch to override default
    - Will remain for a while after transition
  - Standard library will work either way

Variations

- Spell int division as x div y
  - New keyword creates additional problems
- Spell int division as div(x, y)
  - Hard to do a global substitute

How About Python <= 2.1?

- If you **have** to maintain code that must run correctly under Python 2.1 or older as well as under the eventual scheme, you can't use // or the future statement
- Use divmod(x,y)[0] for int division
- Use x*1.0/y for float division
- But it would be much easier to use the command line switch, if you can

Alternatives

- Status quo
  - Real problems in some application domains
- Use // for float division
  - The / operator remains ambivalent
- Directive for division semantics
  - Makes the original design bug a permanent wart in the language (violates TOOWTDI)
- Drop automatic coercion int to float :-)

Why Not Rational Numbers?

- Rationals would be a fine solution
- But there is discussion about the design
  - To normalize or not; performance
- Changing int/int to return a rational breaks just as much code, so would require the same transitional measures
- Switching int/int from float to rational later won't break much code
  - Mathematical values are the same

A Proper Numeric Tower

- Eventually, Python may have a proper numeric tower, like Scheme, where the different numeric types are just representational optimizations
- int < long < rational < float < complex
- Implies 1 and 1.0 should act the same
- Requires int/int to be float or rational
- Float division switch is enabler!

Decimal Floating Point

- An alternative to binary floating point
- Just as inexact
- Slower, because emulated in software
- But no surprises like this:

```
>>> 1.1
1.1000000000000001
>>>
```

Case Insensitivity

[ducks :-]

Some Of My Favorite Apps

- Jython
- wxPython
- PyChecker
- IDLE fork
- Pippy (Python on Palm)
- Python on iPAQ (Linux "familiar" 0.4)
- PySol