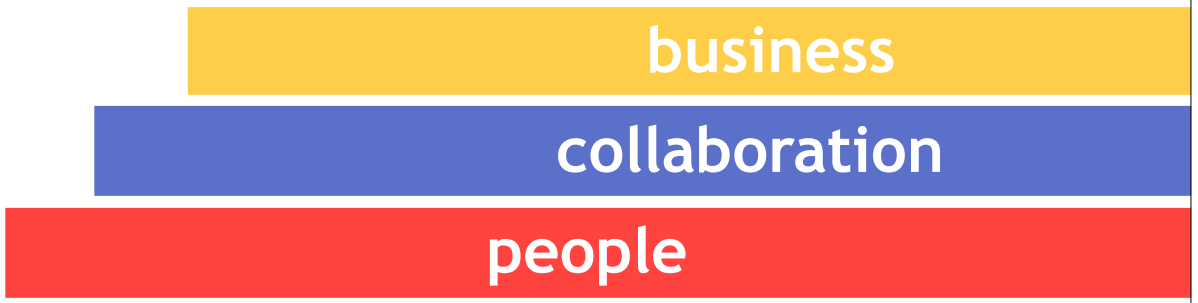


Re-learning Python

Alex Martelli

STRAKT

This talk & its audience

- you know, or used to know, s/thing about Python 1.5.2 (or other Python < 2.2)
- you're experienced programmers in some other language[s] (I'm covering in 1h about 2 days' worth of "normal" tutorials)
- you'd like to understand whether it's worth your while to re-learn Python today, and what are the highlights

Python's Progress

Ver.	released	new for (months)	
1.5.2	1999/04	17	stable
2.0	2000/09	07	s/what stable
2.1	2001/04	08	s/what stable
2.2	2001/12	19	stable
2.3	2003/07	(15?)	**very** stable
(2.4	2004/10?)		

Five years' worth of goodies

- Unicode and codecs
- list comprehensions
- iterators and generators
- new classes, metaclasses, descriptors
- nested scopes
- ..."and a cast of thousands"...
- + lots of library additions/enhancements

Python Versions' Stability

- revolutions 1.5.2→2.0→2.1→2.2
- stability 2.2→2.3
  - only very minor language changes
  - overall, ~3 years' worth of stable experience
  - focus on implementation speed, size, solidity
  - (library does keep growing & getting better)
- ...net result...:

Never been readier for prime time!

Unicode and codecs

- unicode strings vs plain byte strings
  - **methods** make them polymorphic
- encode/decode for string transforms (including to/from Unicode/plain)

```
print 'ciao'.decode('rot13')
```

pnvb

List comprehensions

```
L = [ x*x for x in LL if x>0 ]
```

like, in set theory, $\{x^*x \mid x \in L,\ x>0\}$ -- is just like...:

```
L = []
for x in LL:
    if x>0:
        L.append(x*x)
```

pluses: 1 thought → 1 compact idiom

it's an *expression*...

Iterators: in 1.5.2 ...

To allow looping with a `for`, one had to sort-of-emulate a *sequence*...:

```
class iterable:
  def __getitem__(self, i):
    if self.done():
       raise IndexError
    return self.next()
```

+ safeguards vs random-access, restart...

STRAKT

Iterators: since 2.2 ...

A class is *iterable* if it has a special method `__iter__` returning an **iterator** object:

```
class iterable:
  def __iter__(self):
    return my_iter(self)
```

Each instance of the iterator class keeps track of one iteration's state, returns `self` from `__iter__`, has a method `next`

Iterators: an iterator class

```
class myiter:
  def __init__(self, ...):...
  def __iter__(self):return self
  def next(self):
    [[...advance one step...]]
    if [[it's finished]]:
      raise StopIteration
    return [[the next value]]
```

Iterators: the for statement

`for x in itrbl:` *body*

is now defined to be fully equivalent to:

```
_tmp = iter(itrbl)
while True:
    try: x = _tmp.next()
    except StopIteration: break
    body
```

Iterator example

```
class enumerate:
  def __init__(self, seq):
    self.i = 0; self.seqit = iter(seq)
  def __iter__(self): return self
  def next(self):
    result = self.i, self.seqit.next()
    self.i += 1
    return result
```

Using enumerate

Rather than...:

```
for i in range(len(L)):
    if L[i] > 23:
        L[i] = L[i] - 12
```

we can code:

```
for i, item in enumerate(L):
    if item > 23:
        L[i] = item - 12
```

Simple generators

- functions containing new keyword `yield`
- on call, build and return an iterator **x**
- at each call to **x**`.next()`, function body resumes executing until next time a `yield` or `return` execute
- upon `yield`, **x**`.next()`'s result is `yield`'s argument (ready to resume...)
- upon `return`, raises `StopIteration`

Generator example

```
def enumerate(seq):
    i = 0
    for item in seq:
        yield i, item
        i += 1
```

(Note that enumerate is actually a *built-in* in today's Python).

Nested scopes: in 1.5.2 ...

- just 3 scopes: local, global, built-ins
- we had to use the *fake-default trick*...:

```
def make_adder(addend):
    def f(augend, addend=addend):
        return augend+addend
    return f
```

One problem: f *could* erroneously be called with 2 arguments

Nested scopes: since 2.1 ...

```
def make_adder(addend):
  def adder(augend):
    return augend+addend
  return adder
```

Access to variables from enclosing scope is automatic (*read-only*! specifically: no re-binding of names [mutation of objects is no problem, scoping is about **names**]).

A new object-model

- 1.5.2's object model had some issues...:
  - 4 separate "kinds" of objects
    - types, classes, instances of classes, instances of types
    - no simple ways to mix / interchange them
  - "black magic" function → method transformation
  - metaclasses: mind-blowing complexity
  - simplistic multiple-inheritance name resolution
- they need to stay a while, for backwards compatibility -- *classic classes*
- but side by side, a new OM emerges

The class statement today

`class X [`*`bases`*`] : [`*`body`*`]`

- execute body to build dict d,
- find metaclass M and call (instantiate) it:

```
X = M('X', bases, d)
```

- bind the resulting object to the name

expected (not enforced): `type(X) is M`

→classes are instances of metaclasses

"Find metaclass", how?

- `__metaclass__` in class body
- inherited from leftmost base
- `__metaclass__` in globals
- last-ditch default: `types.ClassType`
  - NB: *classic* classes are still the last-ditch default
- all built-in types have metaclass `type`
- new built-in `object`: just about *only* that

Making a new-style class

- most usual way:

```
class X(object): ...
```

  - (also OK: `class X(list)`, &c)
- gives several new optional features wrt classic classes
- *one* compatibility issue to watch out for:
- implicit special-method lookup is on the *class*, **not** on the *instance*

Lookup of special methods

```
class sic: pass
def f(): return 'foo'
x=sic(); x.__str__=f; print x

class nu(object): pass
y=nu(); y.__str__ = f; print y
```

- lookup always on class is more regular and predictable (e.g. `__call__`)

Descriptors

- a class/type now holds *descriptor* objects
- each descriptor has `__get__` (may have `__set__` iff it's a *data descriptor)*
  - "data descriptor" → has priority on instance dict
- `x.y` → `type(x).y.__get__(x)`
- `x.y=z` → `type(x).y.__set__(x,z)`
- optionally also `__delete__` & `__doc__`

Properties

```
class rect(object):
  def __init__(self,x,y):
    self.x=x; self.y=y
  def getArea(self):
    return self.x*self.y
  def setAtea(self, area):
    self.y = float(area)/self.x
  area=property(getArea,setArea)
```

STRAKT

why properties matter a **lot**

- without properties, one might code many accessor methods `getThis`, `setThat`...
  - "just in case" some attribute access should need to trigger some code execution in some future version
- accessors end up being 90+% boilerplate
  - "boilerplate code" is a very, **very** bad thing
- with properties, always support natural, plain `x.this`, `x.that=23` syntax
  - can refactor attribute→property if ever needed

Functions are now descriptors

```
>>> def f(x, y): return x+y
>>> plus23 = f.__get__(23)
>>> print plus23(100)
123
>>>
```

- so, the function → method transformation has no "magic" any more (follows from general rules)

staticmethod, classmethod

```
class nu(object):
  def f(): return 'hey'
  f = staticmethod(f)
  def g(cls): return 'ho%s'%cls
  g = classmethod(g)
class sb(nu): pass
print nu.f(), nu.g(), nu().f()
print sb.f(), sb.g(), sb().g()
```

classmethod example

```
class dict:
  def _fks(cls, seq, val=None):
    x = cls()
    for k in seq: x[k]=val
    return x
  fromkeys = classmethod(_fks)
```

- actually part of builtin `dict` since 2.3
- an alternate ctor is a typical classmethod

Method __new__

- `type.__call__(cls, *a, **k)` now operates through a simple "template method" design pattern:

```
nu = cls.__new__(cls, *a, **k)
if isinstance(nu, cls):
    cls.__init__(nu, *a, **k)
return nu
```

- eases caching, singletons, ...

Subclassing (e.g.) str

```
class ust(str):
    def __new__(cls, val):
        return str.__new__(cls,
                           val.upper())
```

- can't do it in `__init__` -- that's too late, since strings are immutable
- `__new__` makes it easy

Other new special methods

- `__iadd__`, `__imul__`, ...: optional "in-place" methods to support `+=`, `*=`, ...
- `__unicode__`: like `__str__`
- `__floordiv__`, `__truediv__`: like `__div__` but for trunc/nontrunc div's
- `__getattribute__`,`__contains__`
- `__eq__`, `__lt_`, `__le__`, `...`

Name resolution order: classic

```
clas sic:
  def f(): return 'sic.f'
  def g(): return 'sic.g'
class d1(sic):
  def f(): return 'd1.f'
class d2(sic):
  def g(): return 'd2.g'
class leaf(d1, d2): pass
```

Name resolution order: new

```
clas nu(object):
  def f(): return 'nu.f'
  def g(): return 'nu.g'
class d1(nu):
  def f(): return 'd1.f'
class d2(nu):
  def g(): return 'd2.g'
class leaf(d1, d2): pass
```

Cooperative super-delegation

```
class base(object): pass
class d1(base):
  def __init__(self, **k):
    self.w = k.get('w')
    super(d1,self).__init__(**k)
```

- "steps upwards" to next class in self's `__mro__` (name-resolution order)

Custom metaclasses

- A rare need, but...:

```
class mymeta(type):
   def __new__(c,n,b,d):
      d.update(this_and_that)
      return type.__new__(c,n,b,d)
class funky:__metaclass__=mymeta
```

- subclass `type`, override `__new__` → quite typical custom metaclass traits

__slots__

- normally, any class instance has a dict to allow per-instance attributes
- for tiny instances in great numbers (e.g. points), that's a lot of memory
- `__slots__` → no per-instance dicts, all attribute names are listed right here
- saves memory -- no other real use

__slots__ example

```
class point(object):
    __slots__ = 'x', 'y'
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

- *subclass* point and the per-instance dict perks up again -- unless `__slots__` is defined at *every* level in the hierarchy

Augmented assignment

- `a += b` now means...:
- if `type(a)` has `__iadd__`,
  - `a = type(a).__iadd__(a, b)`
- otherwise,
  - `a = a + b`
- polymorphism between mutable and immutable types
- watch out for "the performance trap"!

"The" performance trap

```
s = ''
for subs in alotofsmallstrings:
   s += subs
```

- unmitigated disaster O(N²) performance
- here's the optimal O(N) alternative:

```
s = ''.join(alotofsmallstrings)
```

- `s = sum(alotofsmallstrings)` would be disaster too (hence forbidden)

...and a cast of thousands...

- GC enhancements, `weakref`
- `import/as`, new import hooks, zipimport
- `%r`, `zip`, `sum`, `int/long` w/base, `bool`
- function attributes
- dicts: `setdefault`, `pop`, `**k`, iteration
- enhanced slices, `list.index` start/stop
- string enhancements: `in`, `strip`
- file enhancements: `'U'`, iteration

Ex: lines-by-word file index

```
# build a map word->list of line #s
idx = {}
for n, line in enumerate(file(fn,'U')):
  for word in line.split():
    idx.setdefault(word, []).append(n)
# print in alphabetical order
words = idx.keys()
words.sort()
for word in words:
  print word, idx[word]
```

Other examples of new stuff

- `sys.path.append('modules.zip')`
- `import CGIHTTPServer as C`
- `for a, b in zip(L1, L2): ...`
- `if 'der' in 'Cinderella': ...`
- `for x in backwards[::-1]: ...`
- `print int('202221','3')`
- `print sum([n*n for n in Ns])`
- `dbg=options.pop('dbg',False)`

STRAKT

The Library (of Alexandria?-)

- Python's standard library has always been rich ("batteries included")
- grows richer and richer with time
- thus (inevitably) some parts slowly get deprecated to "make space" for new and mode general ones
- great 3rd party packages always competing to get in as best-of-breed

New packages

- bsddb, curses (were modules)
- compiler, hotshot, idlelib
- encoding
- logging
- email
- xml
- warnings
- distutils

Parsing, changing, writing XML

```
from xml.dom import minidom as M
doc = M.parse('foo_in.xml')
cmt = doc.createComment('hey!')
doc.appendChild(cmt)
print>>file('foo_out.xml','w'
   ), doc.toprettyxml(' '*4)
```

- SAX and pulldom also available (and preferable for big documents)

Other new modules

- doctest, unittest, inspect, pydoc
- optparse, atexit, mmap
- tarfile, bz2, zipfile, zipimport
- datetime, timeit
- heapq, textwrap, gettext
- itertools
- xmlrpc clients and servers

Strong support for unit testing

- doctest checks all examples in docstrings
- unittest follows in Kent Beck's tradition (see his "Test-driven development by example" book: 1/3 is Python)
- DocTestSuite allows piggybacking of unitest on top of doctest
- Python 2.3 comes with 60k lines of Python source worth of unit tests

pydoc and the help function

- leverage docstrings and introspection

```
>>> help(list)
Help on class list in module __builtin__:
class list(object)
 |  list() -> new list
 |  list(seq) -> new list initialized from seq's items
 |
 |  Methods defined here:
 |
 |  __add__(...)
 |      x.__add__(y) <==> x+y
 |
 |  __contains__(...)
 |      x.__contains__(y) <==> y in x
```

timeit measures performance

```
$ python timeit.py '{}'
100000 loops, best of 3: 1.24 usec per loop
$ python timeit.py '{}.get(23)'
100000 loops, best of 3: 3.27 usec per loop
$ python timeit.py '{}.setdefault(23)'
100000 loops, best of 3: 3.7 usec per loop
```

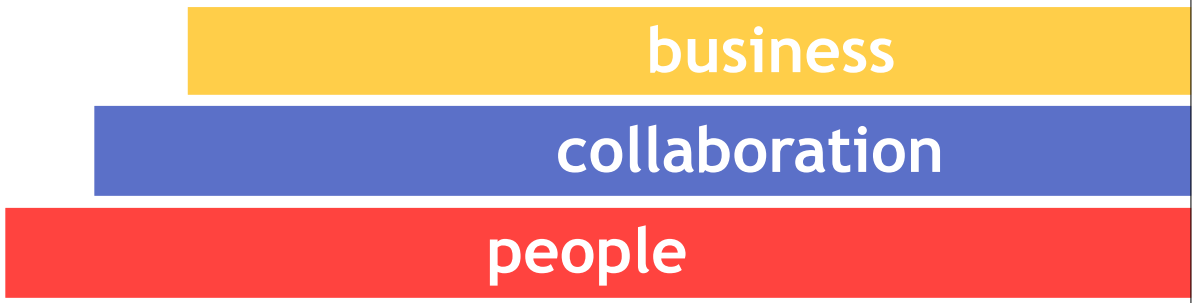
→create & recycle empty dict: 1.24 µs

→get method call: 2.03 µs more

→setdefault: other 0.43 µs on top

Enhancements to modules

- time.strptime: pure portable Python
- random.sample; Mersenne Twister
- socket: supports IPv6, SSL, timeout
- UserDict: DictMixin
- array: Unicode support
- pickle: new protocol
- shelve: new safe/writeback mode
- ...



AMK's "What's New in Python" summaries:

http://www.amk.ca/python/

GvR essay about the new object-model:

http://www.python.org/2.2/descrintro.html

AM, DA, ML, MLH, and many others, on all of this...:

Python in a Nutshell, Learning Python, The Python Cookbook, Practical Python

AM, JH on iterators, generators, metaclasses, ...:

http://www.strakt.com