

# Better, faster, smarter

Python: yesterday, today... and tomorrow



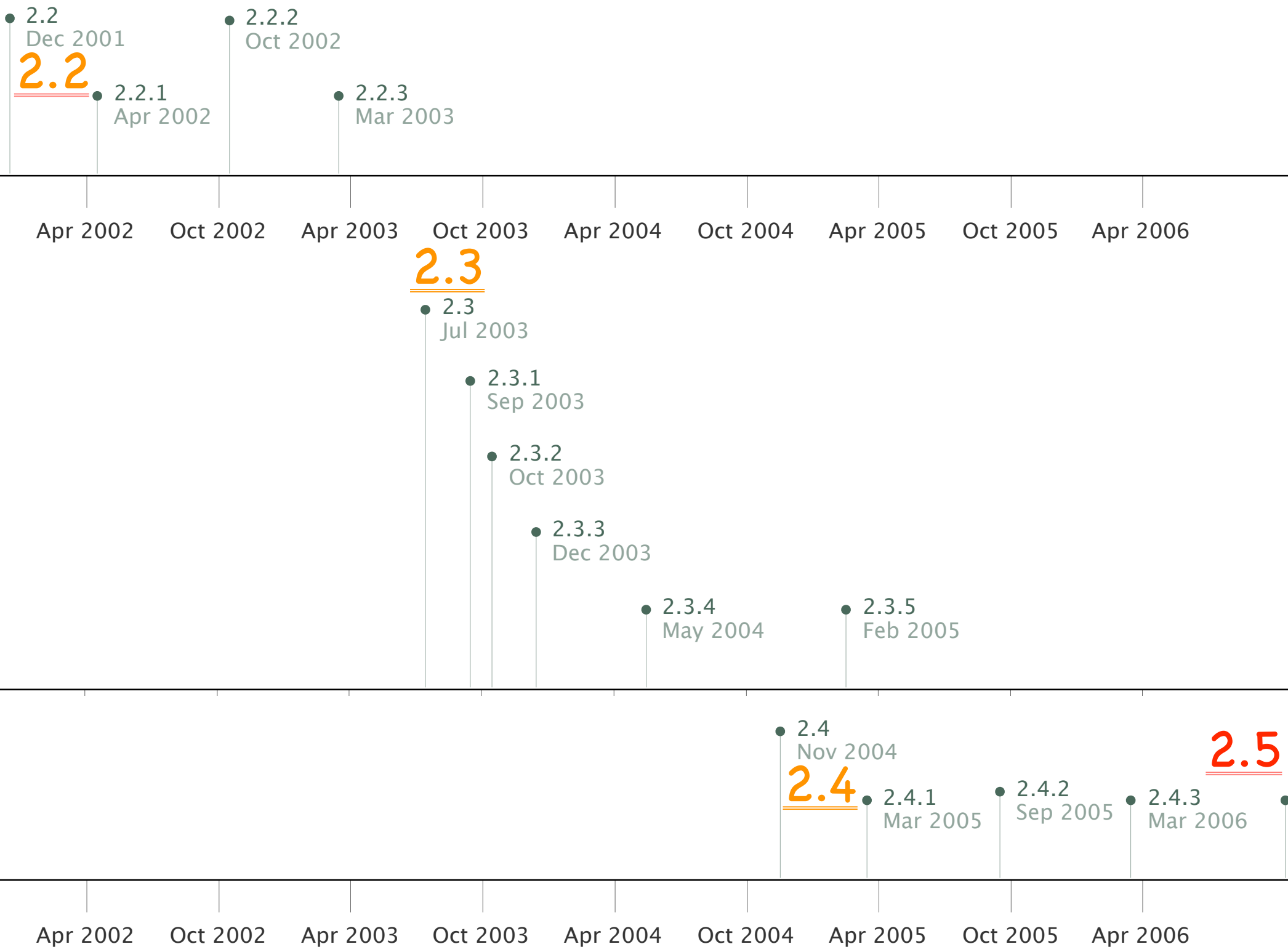
©2006 Alex Martelli [aleax@google.com](mailto:aleax@google.com)

# Outline of this talk

- a short reflection on Python evolution
  - 2.2 → 2.3 → 2.4 → ...
  - ... → highlights of Python 2.5
    - the "with" statement (RAII)
    - other language changes
    - additions to the standard library
    - optimizations
- Q & A
  - Qs are also welcome during the talk!-)

# 1 lang, many versions

- Jython (pending 2.2/2.3 release)
- IronPython (1.0, 8/06, Microsoft ~ CPython 2.4)
- pypy (0.9, 6/06 ~ CPython 2.4, but "beta")
- CPython (Classic Python) timeline
  - 2.2: 12/01 (...2.2.3: 5/03) major new stuff
  - 2.3: 7/03 (...2.3.5: 2/05) ~30% faster
  - 2.4: 11/04 (...2.4.4: 9/06) ~5% faster yet
  - 2.5: 9/06 (...?) ~10% (?) faster yet



# The 2.2 "revolution"

- 2.2 was a "backwards-compatible" revolution
  - new-style object model, descriptors, custom metaclasses...
  - iterators and generators
  - nested scopes
  - int/long merge, new division, bool (2.2.3)
  - standard library: XML/RPC (clients and servers), IPv6 support, email, UCS-4, ...
- nothing THAT big since, plus, new rule:
  - 2.N.\* has NO extra features wrt 2.N

## 2.2 highlights

```
class Newy(object): ...  
__metaclass__ = ...
```

```
def funmaker(...):  
    def madefun(...): ...  
    return madefun
```

```
def gen1(item):  
    yield item
```

```
for item in iter(f, sentinel): ...
```

## 2.3: stable evolution

- no changes to the language “proper”
- many, MANY optimizations/tweaks/fixes
  - import-from-ZIP, ever-more-amazing sort, Karatsuba multiplication, pymalloc, interned strs gc'able ...
- builtins: sum, enumerate, extended slices, enhancements to str, dict, list, file, ...
- stdlib, many new modules: bz2, csv, datetime, heapq, itertools, logging, optparse, platform, sets, tarfile, textwrap, timeit
  - & many enhancements: socket timeouts, ...

## 2.3 highlights

```
sys.path.append('some.zip')
```

```
sum([x**2 for x in xs if x%2])
```

```
for i, x in enumerate(xs): ...
```

```
print 'ciao'[::-1]
```

```
for line in open(fn, 'U'): ...
```

...and MANY new stdlib modules...!



## 2.4: mostly evolution

- “small” new language features:
  - genexps, decorators
  - many “peephole-level” optimizations
- builtins: sorted, reversed; enhancements to sort, str; set becomes built-in
- stdlib, new modules: collections, cookielib, decimal, subprocess
  - string.Template, faster bisect & heapq, operator itemgetter & attrgetter, os.urandom, threading.local, ...

## 2.4 language changes

```
sum(x**2 for x in xs if x%2)
```

```
like sum([x**2 for x in xs if x%2])
```

(without actually building the list!)

```
class Foo(object):
```

```
    @classmethod
```

```
    def bar(cls): return cls.__name__
```

```
print Foo().bar(), Foo.bar()
```

```
emits: Foo Foo
```

## 2.4 new built-ins

```
for item in sorted(sequence): ...  
(does not alter sequence in-place!)
```

```
for item in reversed(sequence): ...  
(does not alter sequence in-place; like...  
for item in sequence[::-1]: ...  
...but, more readable!-)
```

set and frozenset become built-in types

## 2.4 built-ins additions

```
print 'foo'.center(7, '+')
```

```
emits: ++foo++
```

```
print 'foo+bar+baz'.rsplit('+',1)[-1]
```

```
emits: baz
```

```
print sorted('abc d ef g'.split(), key=len)
```

```
emits: ['d', 'g', 'ef', 'abc']
```

## 2.4 new stdlib modules

`collections.deque`

double-ended queue -- methods: `append`, `appendleft`, `extend`, `extendleft`, `pop`, `popleft`, `rotate`

`decimal.Decimal`

specified-precision decimal floating point number, IEEE-754 compliant

`subprocess.Popen`

spawn and control a sub-process

## 2.4 stdlib additions

`list2d.sort(key=operator.itemgetter(1))`

`os.urandom(n)` -> n crypto-strong byte str

`threading.local()` -> TLS (attrs bundle)

`heapq.nlargest(n, sequence)`

also `.nsmallest` (whole module much faster)

## 2.5: evolution... plus!

- several language changes:
  - full support for “RAII”-style programming
    - new “with” statement, new contextlib module, generator enhancements...
  - absolute/relative imports, unified “try/except/finally” statement, “if/else” operator, exceptions are new-style
- new builtins: any/all, dict.\_\_missing\_\_
- new stdlib modules: ctypes, xml.etree, functools, hashlib, sqlite3, wsgiref, ...

# 2.5: many optimizations

- sets/frozensets recoded from scratch
- many speed-ups to string operations
- substantial speed-ups in struct
- new-style exceptions are faster
- and many minor optimization tweaks
  - smaller and phantom frames in calls
  - re uses Python allocator
  - some constant-folding at compile time
  - fewer system calls during importing
  - ...



# Resource Allocation Is Initialization

# in 2.4 and earlier, Java-like...:

```
resource = ...allocate it...
```

```
try:
```

```
    ...use the resource...
```

```
finally:
```

```
    ...free the resource...
```

# in 2.5, much “slimmer”...:

```
with ...allocate it... as resource:
```

```
    ...use the resource...
```

with automatic “freeing” at block exit!



# Many "with"-ready types

```
with open(filename) as f:  
    ...work with file f...  
# auto f.close() on block exit
```

```
someLock = threading.Lock()
```

```
with someLock:  
    # auto someLock.acquire() on block entry  
    ...work guarded by someLock...  
# auto someLock.release() on block exit
```

# The "with" statement

```
from __future__ import with_statement  
with <expr> as var: <with-block>
```

# makes and uses a \*context manager\*

```
_context = <expr>
```

```
var = _context.__enter__()
```

```
try: <with-block>
```

```
except: _context.__exit__(*sys.exc_info())
```

```
else: _context.__exit__(None, None, None)
```

Better than C++: can distinguish exception  
exits from normal ones!

# Your own context mgrs

- roll-your-own: write a wrapper class
  - usually `__init__` for initialization
  - `__enter__(self)` returns useful "var"
  - `__exit__(self, ext, exv, tbv)` performs the needed termination operations (exit is "normal" iff args are all None)
- extremely general
- slightly clunky/boilerplatey

# “with” for transactions

```
class Transaction(object):  
    def __init__(self, c): self.c = c  
    def __enter__(self): return self.c.cursor()  
    def __exit__(self, ext, exv, tbv):  
        if ext is None: self.c.commit()  
        else: self.c.rollback()
```

```
with Transaction(connection) as cursor:  
    cursor.execute(...)  
    ...and more processing as needed...
```

# Your own context mgrs

- contextlib module can help in many ways
- decorator contextlib.contextmanager lets you write a context mgr as a generator
  - yield the desired result of `__enter__`
  - within a try/finally or try/except/else
  - re-raise exception in try/except case
- function contextlib.nested “nests” context managers without needing special syntax
- function contextlib.closing(x) just returns x and calls `x.close()` at block exit

# Transaction w/contextlib

```
import contextlib
```

```
@contextlib.contextmanager
```

```
def Transaction(c):
```

```
    cursor = c.cursor()
```

```
    try: yield cursor
```

```
    except:
```

```
        c.rollback()
```

```
        raise
```

```
    else:
```

```
        c.commit()
```

# Other uses of contextlib

```
# syntax-free “nesting”  
# e.g., a locked transaction:  
with contextlib.nested(theLock,  
    Transaction(c)) as (locked, cursor): ...  
# auto commit or rollback, AND auto  
# theLock.release, on block exit  
  
# when all you need is closing, e.g:  
with contextlib.closing(  
    urllib.urlopen(...)) as f:  
    ...work with pseudo-file object f...  
# auto f.close() on block exit
```



# Generator enhancements

- yield can be inside a try-clause
- yield is now an expression
  - `x = g.send(value)` gives yield's value
  - `x = g.next()` is like `x = g.send(None)`
  - preferred syntax: `value = (yield result)`
- `g.throw(type [,value [,traceback]])`
  - `g.close()` is like `g.throw(GeneratorExit)`
- automatic `g.close()` when `g` is garbage-collected
  - this is what ensures try/finally works!

# Absolute/relative imports

- `from __future__ import absolute_import`
- means: `import X` finds `X` in `sys.path`
- you can `import .X` to find `X` in the current package
- also `import ..X` to find `X` in the package containing the current one, etc
- important “future” simplification of imports

# try/except/finally

try: <body>

except <spec>: <handler>

else: <ebody> # else-clause is optional

finally: <finalizer>

becomes equivalent to:

try:

try: <body>

except <spec>: <handler>

else: <ebody>

finally: <finalizer>

# if/else ternary operator

result = (whentrue if cond else whenfalse)

becomes equivalent to:

if cond:

    result = whentrue

else:

    result = whenfalse

- the parentheses are technically optional (!)
- meant to help with lambda & the like
- somewhat-controversial syntax...:-)

# Exceptions are new-style

BaseException

KeyboardInterrupt

Exception

GeneratorExit

StandardError

ArithmeticError

EnvironmentError

LookupError

# other "true" error classes

StopIteration

SystemExit

Warning

SystemExit

# any and all

```
def any(seq):  
    for item in seq:  
        if item: return True  
    return False
```

```
def all(seq):  
    for item in seq:  
        if not item: return False  
    return True
```

note: RIGHT behavior for empty sequence!

# dict.\_\_missing\_\_

- hook method called by `__setitem__` if the key is missing (==not in the dict)

- default implementation in dict itself:

```
def __missing__(self, key):  
    raise KeyError(key)
```

- meant to be overridden by subclasses

- `collections.defaultdict` subclasses dict:

```
def __missing__(self, key):  
    return self.default_factory()
```

- `default_factory` optionally set at `__init__` (default None == raise)

# ctypes

- load any shared library / DLL with `ctypes.CDLL(<complete name of library>)`
- call any function as a method of the CDLL
- automatically converts to int and char\*
- other conversions explicit with `c_int`, `c_float`, `create_string_buffer`, ...
- also accesses Python's C API
- essentially: a general-purpose Python FFI !
- dangerous: any programmer mistake or oversight can easily crash Python!



# Element-Tree

- new package `xml.etree` with modules `ElementTree`, `ElementPath`, `ElementInclude`
- highly Pythonic in-memory representation of XML document as tree, much slimmer (and faster!) than the DOM
- each XML element is a bit like a list of its children merged with a dict of its attrs
- scalable to large documents with included C accelerators and `.iterparse` incremental parsing (a bit like `pulldom`, but simpler, and keeps subtrees by default)

# functools

- `functools.partial` for “partial application” (AKA “currying”)
- `functools.update_wrapper` for proper setting of metadata for functions that wrap other functions
- `functools.wraps`: decorator equivalent of `functools.update_wrapper`

# hashlib

- replaces md5 and sha modules (which become wrappers to it!)
- adds SHA-224, SHA-256, SHA-384, SHA-512
- optionally uses OpenSSL as accelerator (but can be pure-Python if necessary!)

# sqlite3

- wrapper for the SQLite embedded DB
- DB-API-2 compliant interface
  - except that SQLite is “typeless” (!)
  - some extensions: optional timeout on connect, isolation level, detect\_type and type converters, executemany on iterators, executescript method, ...
- great way to “get started” on small app, can later migrate to PostgreSQL or other relational DB (MySQL, Oracle, whatever)

# wsgiref

- Web Server Gateway Interface
- standard “middleware” interface between HTTP servers and Python web frameworks
  - goal: any framework, any server
  - non-goal: direct use by web applications!
  - already widely supported by frameworks
  - <http://www.wsgi.org/wsgi> for more!
- stdlib now includes a “reference implementation” of WSGI (wsgiref)
  - includes basic HTTP server for debugging WSGI applications and interfaces