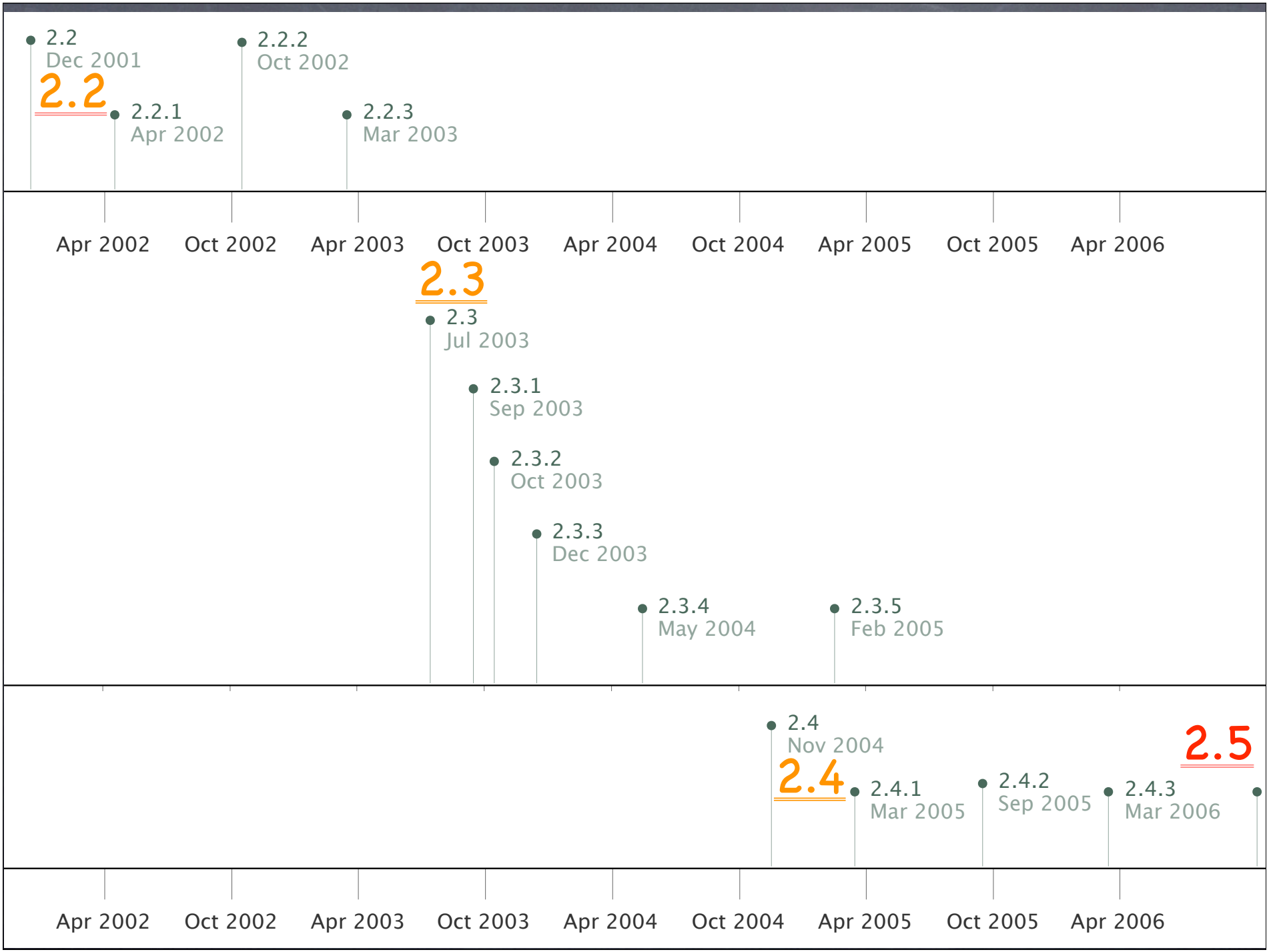
Better, faster, smarter

Python: yesterday, today... and tomorrow

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: 8/06 (...?) ~10% (?) faster yet


2.2
Dec 2001
2.2
2.2.1
Apr 2002
2.2.2
Oct 2002
2.2.3
Mar 2003
Apr 2002
Oct 2002
Apr 2003
Oct 2003
Apr 2004
Oct 2004
Apr 2005
Oct 2005
Apr 2006
2.3
2.3
Jul 2003
2.3.1
Sep 2003
2.3.2
Oct 2003
2.3.3
Dec 2003
2.3.4
May 2004
2.3.5
Feb 2005
2.4
Nov 2004
2.4
2.4.1
Mar 2005
2.4.2
Sep 2005
2.4.3
Mar 2006
2.5
Apr 2002
Oct 2002
Apr 2003
Oct 2003
Apr 2004
Oct 2004
Apr 2005
Oct 2005
Apr 2006

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
  - standard library: XML/RPC (clients and servers), IPv6 support, email, UCS-4, ...
- nothing THAT big since
- 2.N.*: NO extra features wrt 2.N

Google

2.2 highlights

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

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

def gen1(item):
  yield item

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


Google

2.3: stable evolution

- no changes to the language "proper"
- many, MANY optimizations (and small 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, ...

Google

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...!

Google

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 'abc d ef g'.split().sort(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 (and much faster)

2.5: evolution... plus!

- several language changes:
  - full support for "RAII"-style programming
    - a new "with" statement, new contextlib module, generator enhancements...
  - absolute and relative imports, unified "try/except/finally" statement, new "if/else" ternary operator, exceptions are new-style classes
- new builtins: any/all, dict.__missing__
- MANY new stdlib modules: functools, ctypes, xml.etree, hashlib, sqlite3, wsgiref...

Google

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!

Google

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
- somewhat 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...
```


Google

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()
```


Google

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() == 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
- rather-controversial syntax...:-)

Exceptions are new-style

```
object
  Exception
    GeneratorExit
    StandardError
      # all “real error” classes go here,
      # directly or under 3 abstract ones:
      ArithmeticError
      EnvironmentError
      LookupError
    StopIteration
    SystemExit
    Warning
```

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 behavior for empty seq argument...

dict.__missing__

- hook method called by __getitem__ for a missing key
- 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 == like dict)

functools

- partial: partial-application of functions
- wraps, update_wrapper: copies name, doc, &c from wrapped to wrapping function

```
t=functools.partial(pow,2)
print t(3), t(5), t(7) # 8 32 128

def f(): pass
@functools.wraps(f)
def g(): pass
print g # <function f at 0x67e70>
```

ctypes

- general-purpose Python FFI
- load any shared library / DLL with ctypes.CDLL(<complete name of library>)
- call any function as a method of the CDLL
- automatically converts to/from int, char*, wchar_t* (to/from: int/long, str, Unicode)
  - can convert explicitly with c_int, c_float, create_string_buffer, ...
- dangerous: any programer mistake or oversight can easily crash Python!

ctypes “hello world”

```
import ctypes
libc = ctypes.CDLL("libc.so.6")
# or, “libc.dylib” in Darwin/MacOSX, &c
# or, ctypes.cdll.msvcrt in Windows
result = libc.printf("hello world\n")
# emits: hello world
print result            # emits: 12
```

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)

etree example

```
from xml.etree import ElementTree as E
t=E.fromstring('''<x y="z">foo<a>bar</a>
                  baz<b>bat</b>boo</x>''')
for x in t.getiterator():
  print x.tag, x.text, x.tail.strip()

# emits:
x foo None
a bar baz
b bat boo
```

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!)

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
- stdlib now includes a "reference implementation" of WSGI (wsgiref)
  - includes basic HTTP server for debugging WSGI applications and interfaces

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)