Effective Python Programming

Effective Programming

- Get the job done
- Better, more maintainable code
- Use the language's strengths
- Python is not:
  - C++
  - Java
  - Perl
  - ...

Laziness

- In development, laziness can be good
- Do things right the first time
- Don't re-invent every wheel
- Death to NIH

Effective != Efficient

- Effective does not necessarily mean efficient
- Optimise for development time
- Then worry about performance
- We'll touch on efficient code, later

Target

- Python 2.4
- CPython

Python Fundamentals

- Learning by tinkering
- Everything's an object
- Namespaces
- EAFP
- Duck Typing

Short Attention Span Theatre

- Key things to remember
- Even if you sleep through the rest, you'll still get something

S.A.S. Theatre

- Rule #1: Dictionaries
- Rule #2: See Rule #1

Programming with the hood up

- Introspection
- Experimentation

Interactive Python

```
bonanza% python
Python 2.4.1 (#2, Mar 30 2005, 21:51:10)
[GCC 3.3.5 (Debian 1:3.3.5-8ubuntu2)] on linux2
Type "help", "copyright", "credits" or "license" ...
>>>
```

help and dir

- help(obj) – formats docstrings
- dir(obj) – shows attributes

help

```
>>> import os
>>> help(os.popen)
Help on built-in function popen in module posix:

popen(...)
    popen(command [, mode='r' [, bufsize]]) -> pipe

    Open a pipe to/from a command returning a file
object.
```

help...

```
>>> help(8)
Help on int object:

class int(object)
 |  int(x[, base]) -> integer
 |
 | Convert a string or number to an integer, if
 | possible.  A floating point argument will be
 | truncated towards zero (this does not include a
 | string representation of a floating point
 | number!)  When converting a string, use
 | the optional base.  It is an error to supply a
```

dir

```
>>> import popen2
>>> dir(popen2)
['MAXFD', 'Popen3', 'Popen4', '__all__',
'__builtins__', '__doc__', '__file__', '__name__',
'_active', '_cleanup', '_test', 'os', 'popen2',
'popen3', 'popen4', 'sys']
```

Everything is an object

- ints, strings, files
- functions, modules, classes
- Variables are just labels

Objects vs Names

- Variables are references
- Just a name referring to an object
- Stored in a namespace

  (defaults to the local namespace)

Namespaces...

- Namespaces are dictionaries!

```
>>> import sys
>>> def foo(a=1, b=2):
...     c = "hello"
...     print sys._getframe().f_locals
...
>>>
>>> foo(a=4)
{'a': 4, 'c': 'hello', 'b': 2}
```

Namespace lookup – Classic

- locals
- module globals
- built-ins

Assignment

- Assignment goes to local namespace
- Unless 'global' statement

global

- global only for assignment
- not needed for getting a value
- globals are slower!

Namespaces – nested scopes

- statically nested scopes

  (nested function definitions)
- useful for lambda:

```
def createUpdater(self):
    return lambda foo, bar: self.update(foo,bar)
```

- nested function calls

  example later of this

EAFP

- Easier to Ask Forgiveness than Permission
- Very basic Python concept

Permission...

- Permission:

```
if hasattr(obj, 'value'):
    value = obj.value
else:
    value = None
```

- Forgiveness:

```
try:
    read = obj.value
except AttributeError:
    value = None
```

EAFP

- Exceptions are expensive
- Checks can also be expensive
- Case-by-case – how often is it expected to fail?

Python Typing

- Weak vs Strong
- Static vs Dynamic
- C++/Java: strong static typing
- Python: strong dynamic typing

Duck-Typing

- Walks like a duck
- ... quacks like a duck
- ... it's a duck

Duck-Typing

- File-like objects
- Might only need 'read()'

Duck-Typing – File objects

```
def getData(obj):
    data = obj.read()
    print data
f = open('file.txt')
getData(f)
```

- Actually, that data file was gzipped:

```
import gzip
f = gzip.GzipFile('file.txt.gz')
getData(f)
```

More Duck-Typing

- The mapping interface (dictionary)
- Start with a dictionary
- Slot in a different implementation
- e.g. network, database, ...

Interfaces

- zope.interface
- PyProtocols
- Assert that an object implements an interface
- Documentation
- Adaptation
- Future Python

Structured Programming

Control Flow

- Iterators
- Generators
- for/else
- try/finally
- try/except/else
- switch statement

S.A.S. Theatre!

- enumerate

```
for n in range(len(sequence)):
    element = sequence[n]
```

- instead:

```
for n, element in enumerate(sequence):
```

- enumerate returns an iterator

```
>>> print enumerate([])
<enumerate object at 0xb7df418c>
```

Basic control flow

- while
- for
- try/except

Iterators

- Returns the next item each time
- No need to have all items in memory
- More flexible

Files are iterators

- Returns a line

```
>>> for line in open('/etc/resolv.conf'):
...     print "got line '%s'"%(line.strip())
...
got line 'nameserver 210.15.254.240'
got line 'nameserver 210.15.254.241'
got line 'nameserver 203.10.110.101'
got line 'nameserver 203.17.103.1'
```

Creating iterators

- iter() built-in
- turns a sequence into an iterator
- classes can have an __iter__ method that returns an iterator

More flexible for loops

- Call .next() to get the next item

```
iterobj = iter(sequence)
for item in iterobj:
    if item == 'extended':
        item = item + iterobj.next()
```

Token streams

```
tokens=['text:hello','style:bold','text:world',
        'text:goodbye','style:italic','text:world']
tokens = iter(tokens)
for tok in tokens:
    what,value = tok.split(':',1)
    if what == 'text':
        handlePlainToken(value)
    elif what == 'style':
        what, value = tok.next().split(':', 1)
        if style == 'bold':
            handleBoldToken(value)
        elif style == 'italic':
            handleItalicToken(value)
```

Push Iterator

- Sometimes you want to check the next item, but not always consume it
- Push it back onto the iterator!
- Like stdio's getc()/ungetc()

Push Iterator

```
class PushIterator:
    def __init__(self, iter):
        self.iter = iter
        self.pushed = []
    def push(self, value):
        self.pushed.append(value)
    def next(self):
        if self.pushed:
            return self.pushed.pop()
        else:
            return self.iter.next()
    def __iter__(self):
        return self
```

Peek Iterator

- Sibling to PushIterator
- Peek at the next result (without consuming it)

itertools

- high performance iterator manipulation
- functional programming

Generators

- “Good artists copy, great artists steal”
  - Picasso
- Stolen from Icon
- functions containing 'yield' are a generator
- When called, returns a generator object
- ... which is an iterator

Generators

- 'yield' passes a result to the caller
- execution is suspended
- when next() is called again, resumes where it left off

Generators

```
>>> def squares(start=1):
...     while True:
...         yield start * start
...         start += 1
...
>>> sq = squares(4)
>>> sq
<generator object at 0xb7df440c>
>>> sq.next()
16
>>> sq.next()
25
```

Generators

- finish when they fall off the end
- or `'return'`
- Generators can't 'return' a value
- Generators can be called multiple times

Multiple Generator Calls

```
>>> s1 = squares(5)
>>> s2 = squares(15)
>>> print s1.next(), s2.next(), s1.next(), s2.next()
25 225 36 256
```

Generator Example

- DB-API
- cursor.fetchone() - get one row
  - Inefficient!
- cursor.fetchall() - get all rows
  - Could consume a lot of memory
- cursor.fetchmany(N) – get N rows
  - Slightly fiddly

Possible Solutions

```
for row in cursor.fetchall():
    processResult(row)

row = cursor.fetchone()
while row:
    processResult(row)
    row = cursor.fetchone()

while True:
    rows = cursor.fetchmany(100)
    if not rows:
        break
    for row in rows:
        processResult(row)
```

Generator Version

```
def ResultIter(cursor, arraysize=1000):
    while True:
        results = cursor.fetchmany(arraysize)
        if not results:
            break
        for result in results:
            yield result
```

- Using this:

```
for res in ResultIter(cursor):
    processRow(res)
```

for/else

- for statements can have an else: clause
- executed when the for loop exhausts it's loop (no 'break', return or exception)

for/else example

```
for element in sequence:
    if elementMatched(element):
        correct = element
        break
else:
    print "no elements matched"
    correct = None
```

try/finally

- finally: clause is always executed
- resource cleanup

```
lock = acquireLock()
try:
    val = doSomeStuff()
    if val is None:
       raise ValueError('got None')
    elif val < 0:
       return
finally:
    lock.release()
```

try/finally

- Great for preventing those nightmare bugs
- “This should never happen”
- Chances are it will
- Program Defensively

  The Universe Hates You.
- Woefully underused

try/except/else

- try/except can have an else: statement
- executed when no exception occurred

try/except/else

- Put only the important bits in the try: block

```
try:
    import gtk
except:
    MyWindow = None
else:
    MyWindow = gtk.Window()
```

minimising code in a try: block

- This code has a problem:

```
try:
    data = obj.read()
except AttributeError:
    data = ''
```

- This masks any AttributeErrors in the read() method
- Source of hard-to-find bugs

switch statement

- python has no switch/case
- if/elif/elif/elif/else
- use a dictionary

Dispatch via dict

```
if indata == 'FOZZIE':
    showFozzie()
elif indata == 'KERMIT':
    showKermit()
    ...
else:
    showUnknownMuppet()
```

- becomes:

```
callDict = { 'FOZZIE': showFozzie,
             'KERMIT': showKermit,
             ... }
func = callDict.get(indata, showUnknownMuppet)
func()
```

Object Oriented Programming

Using Classes Pythonically

- New-style vs Old-style
- More Ducks!
- isinstance
- inheritance, mixins
- access control
- Simplifying your APIs

S.A.S. Theatre

- __del__ is not your friend

__del__ often considered harmful

- C++/Java-ism
- __del__ breaks garbage collector
- non-deterministic
- doesn't always get called usefully when Python is exiting
- use a weakref in some cases

New-style vs Old-style

- Python has two types of class
- Old-style classes are the original ones
- New-style (in 2.2) fix issues with these
  - Difference between C types and Python classes
  - Can inherit from built-in types (e.g. dict)
  - Some shiny new features

New-style vs Old-style

- New style derive from 'object'
- New style classes in 2.2
- Fix for implementation issues in original classes
- Many new features
  - properties
  - descriptors
  - __new__
  - .....

Use New Style Classes

- Most new code should use them
- Will become the default in Python 3.0

More Ducks!

- Objects supporting the mapping interface:
  - dictionaries
  - *dbm database files
  - shelves
  - db_row instances

Sometimes you care

- There are times when you care what things are passed
- Check for the methods you need

```
if hasattr(obj, 'read'):
    obj.read()
else:
    raise ValueError
```

- Or use an Interface

Interfaces

- Documentation
- Assertions

```
def getAudioFromSource(obj):
    if not IAudio(obj):
        raise ValueError('expected audio, got %r'%
                                                obj)
    return audio.read()
```

- Adaptation

  Automatically adapt an IFoo to an IBar

Interface Example

```
class IAudio(Interface):
    '''Lowlevel interface to audio source/sink.'''
    def close():
        '''Close the underlying audio device'''
    def reopen():
        '''Reopen a closed audio device'''
    def isOpen():
        '''Return True if underlying audio open'''
    def read():
        '''Return a packet of audio from device.'''
    def write(data):
        '''Write audio to device.'''
```

Using an Interface

```
class ALSAAudio(object):
    implements(IAudio)
    def reopen(self):
        ....
    def close(self):
        ....
    def isOpen(self):
        ....

alsa = ALSAAudio()
IAudio.implementedBy(alsa)
IAudio(alsa)
```

Interfaces are dynamic

- Interfaces can be changed at runtime
- Assert that a 3rd party object implements an interface
- Register an adapter from one Interface to another

isinstance

- Checking obj.__class__ is evil
- Breaks subclassing
- type(foo) is ancient, and clunky
- Use isinstance:

```
if isinstance(num, basestring):
    num = int(num)
if not isinstance(thing, (int, float)):
    raise ValueError('expected a number')
```

inheritance, mixins

- Multiple inheritance is useful
- Mixin classes – assemble components into a class
- Select from multiple implementations

Inheritance

- “Flat is better than nested”
- Excessive inheritance trees are painful
- Use inheritance when you need it

Design for subclassing

- Use self.__class__ instead of hardcoded class names

```
class Item:
    def __init__(self, value):
        self.value = value
    def getNextItem(self):
        newItem = self.__class__()
        newItem.value = self.value + 1
```

Base class methods

- Call base class methods from subclassed methods!
- Protects you if parent class changes
- Particularly important for `__init__`

```
class Monkey(Simian):
    def __init__(self, bananas=2, *args, **kwargs):
        self.bananas = bananas
        Simian.__init__(self, *args, **kwargs)
```

Don't Assume You Know Best

- You don't know how people will need to use your code
- Think about how someone could extend your classes
- Design classes accordingly
- Document the methods
- Factor out the useful bits

access control

- Another C++/Java-ism

  friend, public, private, ...
- Python: “Everyone's a consenting adult”
- Convention: leading underscore signals “implementation detail”
- Better yet: document the API of the class

__private names

- leading double underscore mangles names
- In theory useful to stop people stepping on implementation details
- In practice, annoying

  Personal goal for 2.5 is to remove all use from the stdlib

Simplifying your APIs

- Damien Conway's “Sufficiently Advanced Magic”
- But not quite that advanced...

Container object

- Get the objects in a container
- First attempt:

```
for child in container.getChildren():
    doSomethingWithObject(child)
```

- Unnecessary API
- Python already has a good way to spell this:

```
for child in container:
    doSomethingWithObject(child)
```

Container example

- Using the iterator protocol

```
class Container(object):
    def __init__(self, *children):
        self.children = children
    def __iter__(self):
        return iter(self.children)

cont = Container(1,2,3,4,5)
for c in cont:
    print c
```

__special__ methods

- Used to implement operators
- Examples:

```
__str__ - string representation
__setitem__ -
     obj[key] = val ==> obj.__setitem__(key,val)
__add__ - add something to this object
__getattr__ - get an attribute of this object
__eq__ - object is being compared to another
```

special methods, examples

- A + B

  First tries A.__add__(B)

  Then B.__radd__(A)

  (We're ignoring __coerce__)

- A == B

  In order: A.__eq__(B), B.__eq__(A), A.__cmp__(B), and then B.__cmp__(A)

Make the API Intuitive

- Every new method name is something that has to be
  - remembered
  - documented
- Finite amount of brain space
- Use intuitive operators
- But don't get *too* clever

C++-style cout

- A bit *too* magical to be recommended:

```
class CppFile:
    def __init__(self, fp):
        self.fp = fp
    def __lshift__(self, someobj):
        self.fp.write(str(someobj))
        return self

import sys
cout = CppFile(sys.stdout)
cout << "hello” << “world\n"
```

Demonstrating special methods

```
class chatty:
    def __init__(self, name):
        self.name = name
    def __getattr__(self, what):
        print self.name, "asked for", what
        raise AttributeError(what)
```

demonstration...

```
>>> A = chatty('A')
>>> B = chatty('B')
>>> A + B
A asked for __coerce__
A asked for __add__
B asked for __coerce__
B asked for __radd__
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: unsupported operand type(s) for +:
'instance' and 'instance'
>>> print A
A asked for __str__
A asked for __repr__
<__main__.chatty instance at 0xb7df47cc>
```

__getattr__ warning

- Special-case __special__ names
- Pain, otherwise
- Python uses lots of special methods
- e.g. __repr__ to print

```
def __getattr__(self, name):
    if name.startswith('__') and name.endswith('__'):
        raise AttributeError(name)
    return self.remoteObject.lookupName(name)
```

Get/Set methods

- instead of:

```
print someobject.getValue()
someobject.setValue(5)
```

- use:

```
print someobject.value
someobject.value = 5
```

Get/Set methods

- Sometimes, attributes need to be computed
- Use a property!

```
class Magic:
    def _getMagicNum(self):
        return findTheMagicNumber()
    def _setMagicNum(self, value)
        setTheMagicNumber(value)
    magicNumber = property(_getMagicNum,
                           _setMagicNum)
```

Properties

- Property takes 1-4 arguments:

```
attribute = property(getmeth, setmeth, delmeth,
                     doc='')
```

- set method and del method optional
- doc sets a docstring – use it

```
monkeyCount = property(getMonkeys,
        doc='The count of monkeys in this barrel')
```

- Only use property when needed
  - Attributes are a lot faster

Descriptors

- property returns an object called a descriptor
- extremely powerful, but you'll have to read up on them yourself
- descriptors are how 'self' gets inserted as the first argument of a method
- classmethod and staticmethod are also descriptors

staticmethod

- Very limited use
- Nearly always better to use a function

classmethod

- Alternate constructors

```
class TimeStamp(object):
    def __init__(self, h, m, s):
        self.h, self.m, self.s = h, m, s

    @classmethod
    def fromString(cls, string):
        h,m,s = [int(x) for x in string.split(':')]
        return cls(h, m, s)
```

- Alternate constructors make nicer API
- Not many other uses

@classmethod !?

- Decorators
- Replaces this:

```
def alternateCtor(cls, argumentList):
    ...
alternateCtor = classmethod(alternateCtor)
```

- With this:

```
@classmethod
def alternateCtor(cls, argumentList):
    ...
```

Decorators

- Decorators are passed the function object, return a new function object
- Remember to copy func_name and __doc__ if you create a new function object!

Laziness

- Python makes things easy
- But only if you let it
- Anytime you feel like you're doing too much, take a step back

S.A.S. Theatre

- dict.setdefault(key, defaultval)
- sets a value (if not already set), returns value

setdefault

- Turns this:

```
if key in dictobj:
    dictobj[key].append(val)
else:
    dictobj[key] = [val]
```

- Into this:

```
dictobj.setdefault(key, []).append(val)
```

Sequence Unpacking

- Most things Just Work:

```
a, b, c = threetuple
```

- Swap two values:

```
a, b = b, a
```

- Get and clear a value:

```
val, self.val = self.val, None
```

Stamping License Plates

- Any time you find yourself writing the same/similar code repeatedly, you're working too hard
- Use a template function (a closure)

Boring repetitive code

```
class Monkey:
  def displayNameAsHTML(self):
      cssClass = self.cssClass
      html='<div class="%s">%s</div>'%(cssClass,
                                       self.name)
      return html
  def displayBananasAsHTML(self):
      cssClass = self.cssClass
      html='<div class="%s">%s</div>'%(cssClass,
                                       self.bananas)
      return html
```

Templated Monkey

```
def _displayer(what):
    def template(self):
        val = getattr(self, what)
        cssClass = self.cssClass
        return '<div class="%s">%s</div>'%(cssClass,
                                           val)
        return html
    return template

class Monkey:
    displayNameAsHTML = _displayer('name')
    displayBananasAsHTML = _displayer('bananas')
```

Making Templates saner

```
def _displayer(what):
    def template(self, what=what):
        ...
    template.func_name = 'display%sAsHtml'%(
                                what.capitalize())
    template.__doc__ = 'Returns %s as HTML'%what
    return template
```

Universal Newline Mode

- Windows, Mac, Unix have different line endings
- Annoying
- Open files with 'rU', and it just works

```
fp = open('somefile.txt', 'rU')
```

codecs.open

- codecs.open provides a file-like object in a particular encoding
- useful for reading and writing
- more duck typing

Plan for Unicode

- Much easier if you start out allowing unicode
- Even if not, it's not too hard to retrofit

basestring

- Parent class for str and unicode
- Useful in isinstance() checks

```
>>> isinstance('moose', basestring)
True
>>> isinstance(u'møøse', basestring)
True
```

The Instant Guide to i18n

- i18n your strings:

```
from gettext import gettext as _
_('Enter your name')
```

- Decode strings on input:

```
bytes = "naïve"
unistr = bytes.decode('iso8859-1')
```

- Encode unicode on output:

```
print unistr.encode('iso8859-1', 'replace')
```

Batteries Included

- Python ships with a large std library
- Don't re-invent the wheel

When Things Go Pear-Shaped

- Debugging
- Testing

S.A.S. Theatre

- The single most useful Python trick I know:

```
import pdb; pdb.set_trace()
```

- When executed, drops into the Python debugger

unittest

- Python standard unittest module
- Port of Junit
- Makes writing tests not-too-horrible
- API is still a bit cumbersome

doctest

- More Pythonic
- Perfect for the lazy programmer
- Cut-n-paste from an interactive session
- Doubles as documentation and examples

Running Tests

- A good test runner makes it more likely people will actually run the tests
- test.py (in Zope3)

Dealing with tracebacks

- Sometimes, the default traceback isn't what you want
- `traceback` module

```
try:
        ....
except:
        e, v, tb = sys.exc_info()
        traceback.print_tb(t)
```

cgitb

- Debugging CGI scripts can be horrible

```
import cgitb; cgitb.enable()
```

- Displays nicely formatted tracebacks
  - context
  - variables
  - arguments
- Can log in text or html, to the browser or files

Making Life Hard For Yourself

- Things to avoid
- Recognising refactoring targets
- Learning from (bad) examples

S.A.S. Theatre

- Bare except: clauses will nearly always bite you later
- Silently eating exceptions is bad
- except: should either log the exception, or re-raise it
- Just 'raise' will re-raise the current exception

Consistent Coding Style

- Pick a style, and stick to it
  - PEP 008 has one, or choose your own
- Use tools where necessary
  - emacs python mode
  - reindent.py
- Pick a consistent indent style!

from module import *

- Don't
- Figuring out where a name comes from should not be painful

Circular Imports

- Circular imports lead to subtle bugs:
- Module A:

```
import moduleB
```

- Module B:

```
import moduleA
```

- Defer imports until you need them to fix this

more on exceptions

- Raising a new exception from an except: clause often makes debugging harder
- The new traceback will point at the except: block, not the original exception
- A bare 'raise' will re-raise the current exception

Refactoring Targets

- Learn to spot potential problems

Long Argument Lists

- Once a function gets past a couple of arguments, either:
  - refactor it
  - or use keyword arguments

```
def someHideousFunction(firstname, surname, addr1,
    addr2, zipcode, city, state):
    ....

someHideousFunction(firstname="Anthony",
                    surname="Baxter", ....
```

Format Strings

- Similar problem with format strings using % operator
- Use the dict form:

```
addrInfo = dict(firstname='Anthony',
                surname='Baxter', ...)
label = """%(firstname)s %(surname)s
           %(addr1)s
           %(addr2)s
           %(city)s, %(state)s
           %(zipcode)s""" % addrInfo
```

string.Template

- Even simpler

```
>>> from string import Template
>>> s = Template('$cheese is from ${country}')
>>> print s.substitute(cheese='Gouda', country='the
Netherlands')
Gouda is from the Netherlands
>>> print s.substitute(cheese='Gouda')
Traceback (most recent call last):
[...]
KeyError: 'country'
>>> print s.safe_substitute(cheese='Gouda')
Gouda is from ${country}
```

$ == Perl!?!

- The $ sign is *only* accepted in the strings passed to string.Template
- Lots of other languages use $ - it's the obvious choice

key in sequence

- Any time you use 'in', check the RHS
- If a sequence, how big will it get?
- O(N)
- Use a dict, or a set

Too many globals

- Overuse of `'global'` usually indicates poor code
- Refactor into a class, storing the global values as attributes

Learning by (bad) example

- Lots of Python code to learn from
- Some of it is old
- People pick up bad habits from this

map/filter/reduce

- map and filter using a lambda should be replaced with listcomps or genexprs
- reduce: just don't use
  - sum() replaced 90% of use cases
  - other use cases usually leads to headscratching

string module

- string methods are better

```
>>> import string
>>> name = 'anthony'
>>> print string.upper(name)
ANTHONY
>>> print name.upper()
ANTHONY
```

- Remember: strings are immutable, methods return a copy

backslash line continuations

- Almost never needed
- Not needed if there's open braces
- So wrap in ( )
- Works for import, too! (in 2.4)

```
from package.subpkg.module import (Aname, Bname,
                                   Cname, Dname)
```

has_key()

- key in dict
- dict.get(key, default)

Circular references

- Not so much of a problem now (garbage collector)
- Remember: `__del__` stops GC!

Regular Expressions

- Should not be the first hammer in your toolbox
- Sometimes, a real parser is better
- If you must use them...

re.compile()

- re.compile() returns a compiled expression
- much, much faster than re-compiling each time
- nicer API, too

named RE groups

- Use named RE groups rather than numbered ones

```
r = re.compile('(?P<name>[^ ]+) (?P<surname>[^ ]+)')
match = r.match('Anthony Baxter')
match.group('surname')
```

Defensive RE programming

- Check the string matches what you expect, first
- Debugging complex RE failures is a world of hurt

Efficient Python Programming

- Making Python Go Fast

S.A.S. Theatre

- function calls are slow

Making your Python code fast

- dicts, list.sort() are highly tuned
- globals are slower than locals
- list.pop(0), list.insert(0, value) are slow
  - reverse your list, or use collections.deque
- move fixed code out of the critical path

profile/hotshot

- Figure out where the slow bits are
- Get the code right first (with unit tests!) and then optimise

numeric/numarray

- Insanely optimised array operations
- If dealing with numbers, use them

Pyrex

- Python dialect that's compiled to C
- Start with straight Python code, add declarations to use C types
- Only optimise the hot spots
- Much easier than trying to write straight C code extensions

`__slots__`

- Optimisation trick
- Python objects store attributes in a dictionary
- `__slots__` is a fixed list
- reduces memory consumption when you have many small objects

`__slots__`

- Don't use `__slots__` as some sort of type-checking hack
- `__slots__` and subclassing == hurt

Tying it all back together

- Java DOM Node
- Good example of a bad Python API

Node.getChildNodes()

- Returns the child nodes of the current Node.
- Either:

  Node.childNodes

- or

  Make Node an iterator that iterates through it's children

getValue()/setValue()

- Instead, just use node.value
- Or use a property if value is computed, or needs to be checked when set

isSameNode()

- Implement a __eq__ method

```
thisNode == otherNode
```

- Note that 'is' might not be right

  if you can have two Node instances pointing at the same part of the DOM

compareDocumentPosition()

- Compares Node's document position to another Node.
- Instead, implement a .position attribute that can be compared:

```
thisNode.position < otherNode.position
```

- Or even Node.__lt__, Node.__gt__

  Could be a bit too clever/magic

parent/child reference cycle

- Remember, a parent child reference cycle isn't a problem

  (So long as Node doesn't implement a `__del__` method!)
- Still, don't create them unnecessarily

  References keep objects alive

Wrapping Up

- Write *Python* code in Python
- Might involve un-learning some habits from other languages
- More information in the notes for these slides...

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.
- Readability counts.
- Special cases aren't special enough to break the rules.
- Although practicality beats purity.
- Errors should never pass silently.
- Unless explicitly silenced.
- In the face of ambiguity, refuse the temptation to guess.
- There should be one – and preferably only one – obvious way to do it.
- Although that way may not be obvious at first unless you're Dutch.
- Now is better than never.
- Although never is often better than *right* now.
- If the implementation is hard to explain, it's a bad idea.
- If the implementation is easy to explain, it may be a good idea.
- Namespaces are one honking great idea -- let's do more of those!