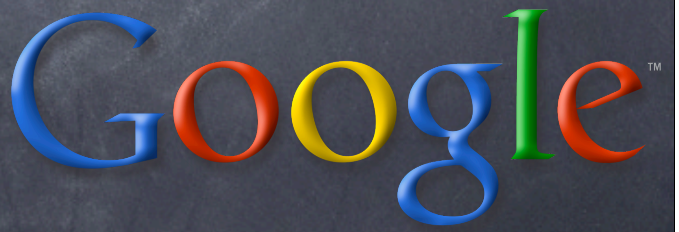
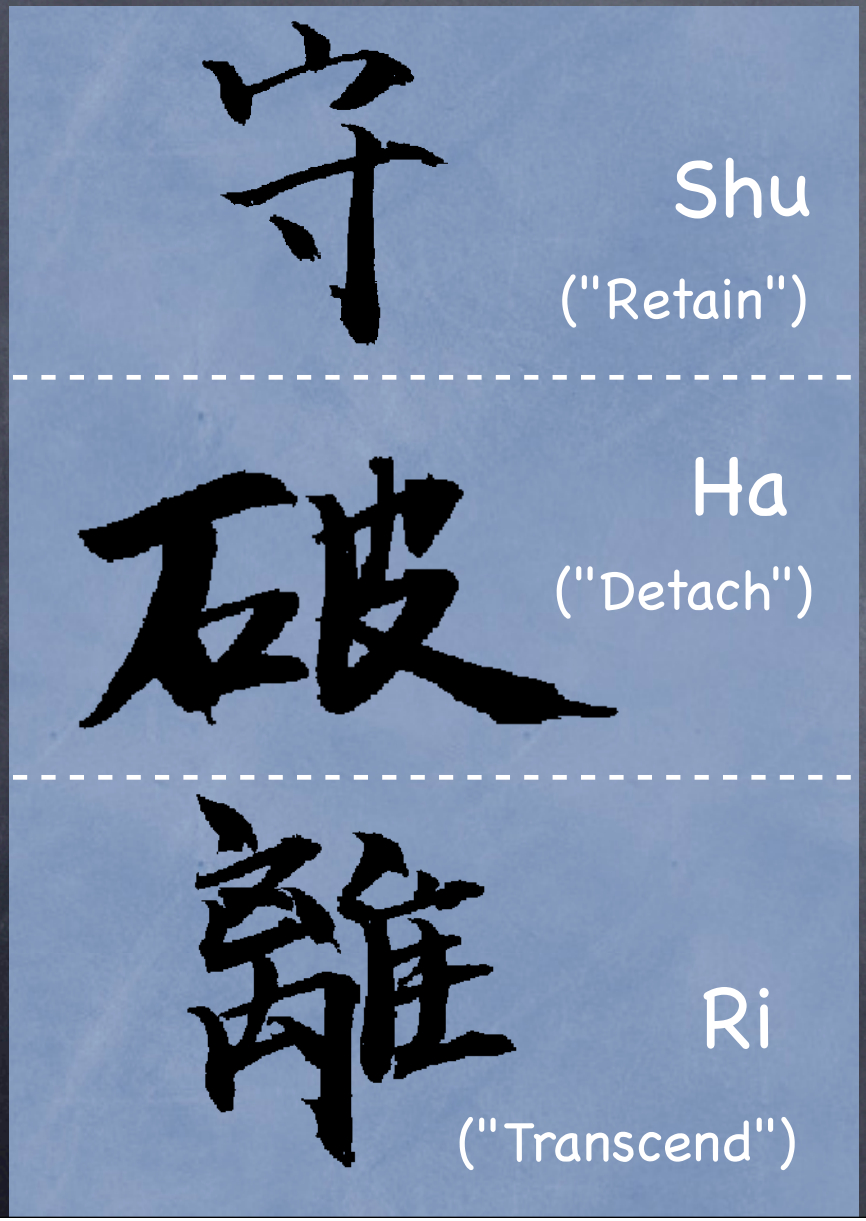
# Slightly Advanced Python:
## some well-documented internals

http://www.aleax.it/ut_pyadv.pdf




©2008 Google -- aleax@google.com

# Audience level for this talk






Google

# This talk's contents

- this talk addresses classic-Python 2.4
  - just a note or three about Python 3000...

- (7p) the class statement: metaclasses &c
- (7p) attribute lookup and descriptor objects
- (2p) introspection
- (2p) garbage collection
- (3p) stack frames and tracebacks
- (9p) bytecode inspection and generation

Google

# The class statement

```
class Name(bases):
  <body>
          -->
```

1. execute <body> "as if it was a function w/o arguments", creating a dict d (w/all the "locals" of that function... kinda)

2. find the class's metaclass M

3. Name = M(Name, bases, d)



Google

# class body & dict

```
def makeSandwich(wid='plenty'):
  class Philly(object):
    steak = 'you bet'
    cheese = 'swiss'
    if wid: onions = wid
    print locals()
makeSandwich()
makeSandwich(None)
```

```
$ python makemeasandwich.py
{'wid': 'plenty', 'cheese': 'swiss', '__module__':
'__main__', 'steak': 'you bet', 'onions': 'plenty'}
{'wid': None, 'cheese': 'swiss', '__module__':
'__main__', 'steak': 'you bet'}
```

Google

# A peek at class stmt (1)

```
import dis
def f():
  class x(object):
    y = 23
dis.dis(f)
  3           0 LOAD_CONST               1 ('x')
              3 LOAD_GLOBAL              0 (object)
              6 BUILD_TUPLE              1
              9 LOAD_CONST               2 (<code object x at 0x52120,
file "t2.py", line 3>)
             12 MAKE_FUNCTION            0
             15 CALL_FUNCTION            0
             18 BUILD_CLASS
             19 STORE_FAST               0 (x)
             22 LOAD_CONST               0 (None)
             25 RETURN_VALUE
```



Google

# A peek at class stmt (2)

```
print f.func_code.co_consts
dis.dis(f.func_code.co_consts[2])

(None, 'x', <code object x at 0x52120, file "t2.py", line
3>)
  3           0 LOAD_GLOBAL              0 (__name__)
              3 STORE_NAME               1 (__module__)

  4           6 LOAD_CONST               1 (23)
              9 STORE_NAME               2 (y)
             12 LOAD_LOCALS
             13 RETURN_VALUE
```



Google

# Finding the metaclass

1. is '__metaclass__' in d?
2. else, leafmost(type(b) for b in bases)
   if there ISN'T a leafmost one, error!
   (see 20.17, p.783, Cookbook 2nd ed)
3. else, is '__metaclass__' in globals()?
4. else, use types.ClassType (OLD style)
   default to the legacy mode for bw-comp
   we don't really cover legacy mode here
   it's old and crufty and NOT good...!



Google

# Instantiating the metaclass

- like any other instantiation... we CALL it!
  - M(...) does NOT use M.__call__!
  - rather, it uses type(M).__call_
    - key clarification in new-type classes!
  - type.__call_ does 2-step construction:

```
x = M.__new__(M, name, bases, d)
if isinstance(x, M):
    M.__init__(x, name, bases, d)
```



Google

# class statement limitations

- class body is executed BEFORE the metaclass M is known (so M has no control whatsoever there)
- class body execution returns a dict (so, no order-relevance is possible)
- Python 3000 relieves those limitations:
  - metaclass=M in class statement
  - M.__prepare__ can return any mapping
  - also, class decorators (removes >50% of the use cases for custom metaclasses;-)



Google

# Attribute lookup

- "getting": x.foo
  - also, identically: getattr(x, 'foo')
- "setting": x.foo = value
  - also, identically: setattr(x, 'foo', value)
- ...what about, say, x.foo += 1...?
  - how would YOU find out?-)



Google

# The x.attr += "mystery";-)

```
class X(object):
  def __getattr__(self, name):
    print 'get', name
    return 23
  def __setattr__(self, name, value):
    print 'set', name, value
x = X()
x.foo += 1

get foo
set foo 24
```



Google

# x.foo attribute "getting"

__getattribute__ does this: DON'T override

1. is 'foo' in type(x) [[or any of the type(x).__mro__ in order]]?
   1.1 if yes, is it a DESCRIPTOR?
       1.1.1 if yes, see later
       1.1.2 else, stash it temporarily and continue
2. is 'foo' in x.__dict__ (or __slots__)?
3. were we in 1.1.2 perchance?
4. try __getattr__ (if any)
5. raise AttributeError



Google

# Descriptors

- Objects whose type has __get__
  - if also __set__, "non-data descriptors"
    - AKA "non-overriding descriptors"
  - priority:
    - data descriptors
    - instance vars
    - non-data descriptors
    - class vars
    - "dunder getattr"



Google

# Descriptors

- for a type: B.foo becomes
  `B.__dict__['foo'].__get__(None, B)`
- for an instance: x.foo (B==type(x)) becomes
  `B.__dict__['foo'].__get__(x, B)`

- it's all about...:
  - object.__getattribute__ vs
  - type.__getattribute__ ...!



Google

# Descriptor e.g: functions

```
class Function(object):
  . . .
  def __get__(self, o, t=None):
    return types.MethodType(self, o, t)
```



Google

# Descriptor e.g: property

```
class property(object):
  def __init__(self, fget=None,
               fset=None, fdel=None, doc=None):
    self.fget = fget # etc etc
  def __get__(self, obj, objtype=None):
    if obj is None: return self
    if self.fget is None: raise ...
    return self.fget(obj)
  def __set__(self, obj, val):
    if self.fget is None: raise ...
    self.fset(obj, val)
```



Google

# Introspection

- putting it all together: module inspect
  - http://docs.python.org/lib/module-inspect.html
- DON'T bother w/dir, vars, __classes__, __bases__, ...: inspect structures it well!
  - especially: http://docs.python.org/lib/inspect-types.html doc's internal types
- NOT sys._getframe! http://docs.python.org/lib/inspect-stack.html works better...
- DEBUGGING yes, testing maybe, prod NO!-)



Google

# inspecting objects

- inspect.getmembers(obj, predicate=None)
  - predicates in inspect: is... (builtin, class, code, datadescriptor, frame, function, method, methoddescriptor, module, routine, traceback)
- inspect.getclasstree(classes), getmro(class)
- inspect.getargspec(f), getargvalues(frame)
- inspect.getsource(obj) & friends



Google

# Garbage Collection

- each Python implementation is fully empowered to follow a different strategy
  - Jython, IronPython: punt to VM
  - pypy: ... (...depends on back-end...!)
  - CPython: what we'll cover here
- main strategy: reference-counting
  - frees stuff ASAP... when feasible
  - "garbage loops" a problem
- secondary strategy: generational mark & sweep collection for loop - module gc



Google

# gc's wonders

- functions...:
  - enable, disable, isenabled, collect
  - get_debug, set_debug
  - get_threshold, set_threshold
    - for each of the 3 "generations"
  - get_objects, get_referrers, get_referents
- .garbage: all unreachable but uncollectable objects (due to __del__ methods!) or more if DEBUG_SAVEALL debug flag is set



Google

# Stack (frames & tb's)

- again, first see stdlib module 'inspect'!-)
- however, some understanding can't hurt...
  - for debugging: priceless!
  - for testing: hm, well, if you hafta
  - for production code: it's RIGHT OUT
    - please DON'T use "black magic" in prod
    - "production" functionality even on your desktop/laptop should eschew it too
    - there's really no need: most likely a white magic IS available (pls Q&A!-)



Google

# Traceback objects

- sys.exc_info()[2] is a traceback object: it offers just 4 attributes
  - tb_frame: the frame object at this level
  - tb_lasti: index in bytecode
  - tb_lineno: line number in Python source
  - tb_next: next-inner traceback object (called-by this level)
- tracebacks form a linked-list ``queue''
- stdlib's traceback module has MANY helper functions to format or print some or all of this info & optionally exception info

Google

# Frame objects

- from tracebacks, inspect's currentframe(), stack(), trace(), get[inner,outer]frames(), ...
- MANY attributes: f_exc_..., f_back (the calling frame), f_code, f_builtins, f_globals, f_locals, f_lasti, f_lineno, f_trace, ...
- "the stack" is a linked list of frame objects (linked through .f_back of each frame)
  - each call creates, activates, links 1 more frame object at top-of-stack
  - you often deal w/it in a traceback case (tracebacks link "the other way")



Google

# Bytecode

- CPython's implementation compiles to custom bytecodes for a specialized stack machine -- ALL execution is of bytecode
  - Jython uses JVM bytecode, IronPython uses MS CLR, pypy -- well, who knows!-)
- *looking* at bytecode may be a great way to help understand what's going on!
- *altering* bytecode, sounds like it'd be ever so cool, but... just PLAY with that, PLEASE, *DON'T* use it "productively"!-)



Google

# Bytecode & Python VM

- simple stack-oriented VM w/some pretty high-level (specialized) opcodes
- http://www.python.org/doc/2.4/lib/bytecodes.html
- 0-operand bytecodes:
  - stack manip: POP_TOP, ROT_TWO, ...
  - operators: UNARY_NOT, BINARY_ADD, INPLACE_DIVIDE, ...
  - indexing/slicing: BINARY_SUBSCR, SLORE_SUBSCR, DELETE_SUBSCR, SLICE+0, SLICE+1, STORE_SLICE+0, DELETE_SLICE+2...
  - PRINT_ITEM, BREAK_LOOP, BUILD_CLASS, YIELD_VALUE, RETURN_VALUE, ...



Google

# Bytecodes w/operands

- 1-operand bytecodes (operand: 2 bytes number)
  - EXTENDED_ARG: to make 4-bytes operands
  - LOAD_...: CONST, NAME, ATTR, GLOBAL, FAST, CLOSURE, DEREF
  - STORE_..., DELETE_...: similar (no CONST!-)
  - BUILD_...: TUPLE, LIST, MAP, SLICE
  - flow control: JUMP_..., SETUP_..., CALL_..., FOR_ITER, RAISE_VARARGS
  - SETUP_...: LOOP, EXCEPT, FINALLY
  - MAKE_FUNCTION, MAKE_CLOSURE



Google

# Looking at bytecode

- 'import dis' and use dis.dis(whatever)
  - module, class, method, function, code obj
- shows one bytecode per output line, with:
  - line # (for first bytecode in source line)
  - ">>" marker for targets of jumps
  - address
  - bytecode code-name
  - operand (numeric value)
  - operand (name or value in parentheses)



Google

# dis.dis example

```
def f(x):
  if x<23: return 'zap'
  else: return 'zop'
dis.dis(f)
  2           0 LOAD_FAST                0 (x)
              3 LOAD_CONST               1 (23)
              6 COMPARE_OP               0 (<)
              9 JUMP_IF_FALSE            8 (to 20)
             12 POP_TOP
             13 LOAD_CONST               2 ('zap')
             16 RETURN_VALUE
             17 JUMP_FORWARD             5 (to 25)
        >>   20 POP_TOP
  3          21 LOAD_CONST               3 ('zop')
             24 RETURN_VALUE
        >>   25 LOAD_CONST               0 (None)
             28 RETURN_VALUE
```



Google

# Understanding a bug

```
>>> -2**2
-4
>>> def f(x, y): return -x**y
...
>>> dis.dis(f)
  1           0 LOAD_FAST                0 (x)
              3 LOAD_FAST                1 (y)
              6 BINARY_POWER
              7 UNARY_NEGATIVE
              8 RETURN_VALUE
```



Google

# Byteplay

- http://byteplay.googlecode.com/svn/trunk/byteplay.py
- if you're pining for assembly...!-)
  - Code.from_code(co_obj) -> Code instance
  - code.to_code() -> code object
  - .code: list of 2-tuples (opcode, arg or None) (opcode can also be Label, SetLineno) with nice specialized __repr__
  - .args (list of names), .freevars (list of names), .name, .docstring, .firstlineno, ...



Google

# Byteplay example

```
import byteplay
def f(x): return x+x
c = byteplay.Code.from_code(f.func_code)
print c.code
 2       1 LOAD_FAST        x
         2 LOAD_FAST        x
         3 BINARY_ADD
         4 RETURN_VALUE
c.code[1] = (byteplay.LOAD_CONST, 23)
import new
g = new.function(c.to_code(), {}, 'g')
print g(100)
```



Google

# Bytecodehacks (NOT 2.*)

- http://bytecodehacks.sourceforge.net
- bytecodehacks.code_editor: mutable versions of code objs (EditableCode), function objs (Function), method objs (InstanceMethod) + CodeString (lets you jump to *labels*)
  - instantiate (from existing obj or scratch)
  - modify
  - make_... (code, function...) to get Py obj
- .ops: functions generating opcodes from 'good' args, e.g. LOAD_GLOBAL('aglobname')
- .rationalize: minimal peephole optimization

Google

# Q & A

http://www.aleax.it/ut_pyadv.pdf

Google