Dynamic aspects of Python

Martin Matusiak

Software Technology Colloquium Talk / 10.04.08

Python facts

- Created by Guido van Rossum (remains benevolent dictator)
- Dynamically typed language
  - No compilation step
  - Variables are not declared before use
- Mature language (1991-)
- Canonical implementation: CPython
- Runs on more platforms than Java
  - AIX, BeOS, iPod, OS/2, Palm OS, Playstation/PSP, Psion, QNX, Sharp Zaurus, Sparc, VMS, VxWorks, WindowsCE/PocketPC
  - also on the Java VM and the .NET VM

Today's program

- Python primer
- Objects and classes
- Metaprogramming

Waypoint

Python primer

Everything is an object

- No primitive datatypes

```
num = 4          # the number 4 is an object
num.<tab>
> num.__abs__
> num.__add__
> num.__class__
```

- Objects have
  - a class

```
obj.__class__
```

  - a namespace

```
obj.__dict__
```

  - methods

```
obj.__str__      # the object's string representation
```

  - members

```
obj.__doc__
```

Variables are labels

```
obj = object()   # instantiate object from 'object' class,
                 # bind to the label 'obj'

obj2 = obj       # bind a new name 'obj2' to existing object bound to
                 # 'obj'

obj == obj2      # object identity test
> True           # same object
```

Iterable datatypes

- Lists

```
a = [1, "name"]
```

- Tuples

```
b = (1, "name")
```

  - Same properties as lists, but immutable

- Dictionaries

```
c = {"name": "James", 42: "fourtytwo" }
```

- Common properties
  - Members of each structure can be arbitrary objects
  - All are iterable and support

```
for item in structure:      # iteration over members
    print item
if item in structure:       # membership test
    print item
```

  - Used _all over_ Python

Functions

- Define a function 'fun' taking two arguments

```
def fun(a, b):
    return a+b
```

- Call the function

```
fun(1, 2)
> 3
```

- Evaluate the function without calling it

```
fun
> <function fun at 0x2ad638bf3e60>
```

- Investigate the function object

```
fun.<tab>
> fun.__call__          # __call__ method is what makes it callable
fun.__call__(1, 2)      # same as fun(1, 2)
> 3
```

Functions

```
def fun(a, b):
    return a+b
```

- Significant whitespace
  - Makes Python code very comprehensible
  - Higher signal-to-noise :)
- Duck typing
  - ”if it walks like a duck and quacks like a duck, it must be a duck”

```
fun(1, "2")
> <type 'exceptions.TypeError'>: unsupported operand type(s) for +:
  'int' and 'str'
```

  - but both 1 and ”2” have an __add__ method:

```
num = 1; s = "2"
num.__add__(s)
> NotImplemented
s.__add__(num)
> <type 'exceptions.TypeError'>: cannot concatenate 'str' and 'int'
  objects
```

Namespaces

- Global namespace
  - Assignments outside a function or class update the global namespace

```
'new_symbol' in globals()      # globals() returns a dictionary of the
                               # global namespace
> False
new_symbol = 14
globals()['new_symbol']
> 14
```

  - The global namespace is global to the current module
- Builtin namespace (immutable)
  - Contains objects available at any time, eg. 'int'
  - Global namespace shadows the builtin namespace

```
int                  # 'int' is bound in the builtin namespace
> <type 'int'>
int = 1              # binding the name 'int' in global namespace
int
> 1
```

Namespaces

- Local namespace
  - Assignments inside blocks are made in the local namespace

```
def fun():
    print 'new_symbol' in locals()     # equivalent to globals()
    new_symbol = 14
    print locals()['new_symbol']

fun()
> False
> 14
```

  - Global symbols can also be reached from functions using 'global'

```
def fun():
    global new_symbol       # now refers to the outer scope
    new_symbol = 14

fun()
new_symbol
> 14
```

Modules

- Python code is stored in modules (.py files)
- Modules are executed top-down

```
import re               # import regular expression namespace from re.py
                        # under the name 're'
```

  - modules are objects...

```
re.__class__
> <type 'module'>
dir(re)                 # list the contents of an object
> re.compile
> re.findall
> re.match
```

  - Namespaces can also be merged on import

```
from re import *
```

  - Existing bindings for 'compile', 'findall' etc will be overwritten

Waypoint

Objects and Classes

Objects

- The very first class definition
  - Deriving from the 'object' class

```
class MyClass(object):

    # shared among all instances of this class
    class_member = "member of a class"

     def __init__(self):
         # unique for every instance
         self.instance_member = "member of an instance"

    def fun(self, s):
         return s+self.instance_member
```

Investigating instances

- Creating an instance of the class 'MyClass'

```
obj = MyClass()
```

  - ”every object belongs to a class...”

```
obj.__class__
> <class '__main__.MyClass'>
```

  - what about the members?

```
obj.instance_member
> 'member of an instance'
obj.class_member
> 'member of a class'
```

  - and the methods?

```
obj.fun("hi, ")
> 'hi, member of an instance'
```

Investigating instances

- ”every object has a namespace...”

```
obj.__dict__
> {'instance_member': 'member of an instance'}
```

  - The 'class_member' variable is not present, it belongs to the class
  - Likewise the methods of the object
- The namespace is a mapping from names to objects
  - It is mutable

```
obj.instance_member_new = 'new member of an instance'
obj.__dict__
> {'instance_member': 'member of an instance',
> 'instance_member_new': 'new member of an instance'}
```

  - Attribute access amounts to retrieving the attribute from the namespace
  - This identity holds:

```
obj.__dict__['instance_member_new'] == obj.instance_member_new
> True
```

Investigating classes

- Classes are objects, objects belong to a class

```
MyClass.__class__
> <type 'type'>
```

  - 'type' is the top most type in Python

- Classes have a tuple listing their base classes

```
MyClass.__bases__
> (<type 'object'>,)
```

- What about the namespace?

```
MyClass.__dict__.items()
> [('fun', <function fun at 0x2ad638bf7aa0>),
('class_member', 'member of a class'),
('__init__', <function __init__ at 0x2ad638bf7b18>)]
```

  - we have our 'missing' class member
  - and the methods of our class
  - as usual, this namespace too is writable

Functions as methods

- How a function becomes a method

```
class MyClass(object):
    def __init__(self):
        self.value = 1

    def foo(self, n):
        return n+self.value
```

  - 'foo' is now stored in the class namespace

```
MyClass.foo
> <unbound method MyClass.foo>
```

  - 'unbound' means it is not bound to any _instance_ of the class
  - it cannot be called directly from the class

```
MyClass.foo(1)
> unbound method foo() must be called with MyClass instance as first
  argument (got int instance instead)
```

Functions as methods

```
obj = MyClass()
```

- from the point of view of an instance, the method is bound

```
obj.foo
> <bound method MyClass.foo of <MyClass object at 0x2ac49af64d90>>
```

- and is callable

```
obj.foo(1)       # even though:  def foo(self, n):
> 2
```

- methods are called with one argument missing
- the argument 'self' is added behind the scenes
- the method can also be called from the class
  - the user must supply the instance on which it is called

```
MyClass.foo(obj, 1)
> 2
```

- or just as well:

```
MyClass.foo(MyClass(), 1)
> 2
```

Mutable namespaces

- Adding a method to a class

```
def fun(self, x):
    return x

MyClass.fun = fun       # bind function 'fun' to the name 'fun'
MyClass.__dict__.items()
> [('fun', <function fun at 0x2ac49af716e0>),]
MyClass().fun(1)        # call the (now) method 'fun'
> 1
```

- Adding a method to an instance only

```
def bar(x):
    return x

obj = MyClass()
obj.bar = bar
obj.__dict__.items()    # bar is bound only in the instance's namespace
> [('bar', <function bar at 0x2ac49af5c5f0>)]
```

Classes are callable

- Recall how a '__call__' method made a function callable

```
MyClass.<tab>
> MyClass.__call__
```

  - This class is callable
  - In fact, calling the class amounts to instantiating an instance of the class:

```
obj = MyClass()
```

  - Arguments to this call are passed onto the '__init__' method

```
obj = MyClass(”Jimmy”, 102)

class MyClass(object):
    def __init__(self, name, age)
        self.name = name
        self.age = age
```

Callable objects

- If all we need is the '__call__' method...
  - we can make our own callable instances!

```
class MyClass(object):
    def __call__(self, *args):
        return args

obj = MyClass()
```

  - A __call__ method is just like any other method
  - Except it is run 'directly' on the object, just like a function call

```
obj(2, "jimmy")
> (2, 'jimmy')
```

- Does this make 'obj' a function?
  - No, merely an object with the callable _property_ of a function

```
obj
> <MyClass object at 0x2ac49af4ff10>
```

Waypoint

Metaprogramming

Motivating decorators

- Suppose we want to add tracing information to a function:

```
def fun(s):
    print "Entering function 'fun', argument is: %s" % s
    sw = s.swapcase()
    print "Leaving function 'fun', result is: %s" % sw
    return sw
```

  - This is very invasive...

```
fun("aB")
> Entering function 'fun', argument is: aB
> Leaving function 'fun', result is: Ab
> 'Ab'
```

A tracing decorator

- A decorator replaces the function in the namespace
  - A new function object is bound to the old name
  - The new function wraps the original function

```
def trace(func):
    def f(*args):
        print "Entering function '%s', arguments are: %s" %
               (func.__name__, args)
        result = func(*args)
        print "Leaving function '%s', result is: %s" %
               (func.__name__, result)
        return result
    return f

@trace                          # invoking decorator
def fun(s):
    return s.swapcase()
#fun = trace(fun)               # result of invoking decorator
```

A synchronizing decorator

- Python has a Global Interpreter Lock
  - Can only be held by one thread at any time
- Decouple synchronization from the business logic:

```
lock = threading.Lock()

def synchronize(func):
    def f(*args):
        lock.acquire()
        result = func(*args)
        lock.release()
        return result
    return f

@synchronize
def withdraw(n):
    balance = account.balance - n
    account.balance = balance
```

A memoizing decorator

- Suppose we have a long running function
  - We wish to cache the result of the function on a given input value
  - This decorator has to be an object (why?)

```
class memoize(object):
    def __init__(self, func):
        self.func = func
        self.cache = {}

    def __call__(self, *args):
        try:
            return self.cache[args]
        except KeyError:
            self.cache[args] = self.func(*args)
            return self.cache[args]

@memoize
def fun(n):
    return math.log(n**n)
```

Introducing metaclasses

- Every object belongs to a class
  - including class objects
  - the class's class is called the metaclass

```
obj.          __class__.    __class__
# instance      class         metaclass
```

- The default metaclass is 'type'

```
obj.__class__.__class__
> <type 'type'>
```

- But then classes too can be instantiated?

How classes are instantiated

```
class MyClass(object):
    class_member = 1

    def foo(self, n):
        return n+self.class_member
```

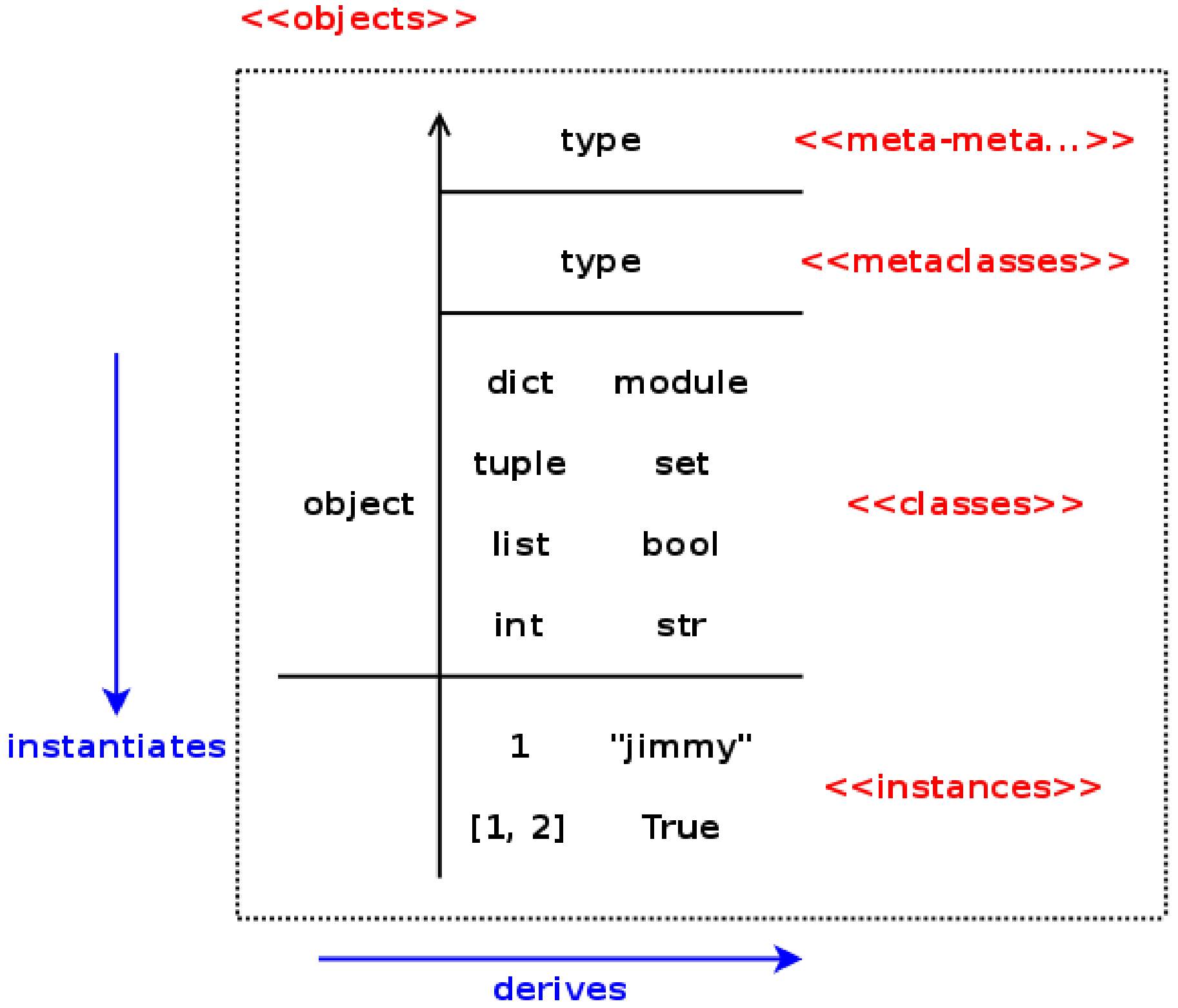
- The 'class' keyword is seen
- Assignments/declarations are collected in a namespace
- The class objected is created, bound to the class name

- Another way to accomplish the same thing:

```
def foo(self, n):
    return n+self.class_member

name = ”MyClass”
bases = ()                  # empty tuple defaults to 'object' as base class
namespace = {'foo': foo, 'class_member': 1}

MyClass = type(name, bases, namespace)
```


<<objects>>
type
<<meta-meta...>>
type
<<metaclasses>>
dict
module
tuple
set
object
<<classes>>
list
bool
int
str
instantiates
1
"jimmy"
<<instances>>
[1, 2]
True
derives

A printable metaclass

- Suppose we want to add a method to all classes
  - We craft a metaclass
  - it derives from 'type', not 'object'
  - we override the '__init__' method and set the method

```
class Printable(type):
    def __init__(cls, name, bases, namespace):
        super(Printable, cls).__init__(name, bases, namespace)

        # method to iterate over all objects in my namespace
        # and display them
        def printme(self):
            for (key, value) in self.__dict__.items():
                print "%s:  %s" % (key, value)

        cls.printme = printme     # set this name in the class namespace
```

A printable metaclass

- Using the new metaclass amounts to setting '__metaclass__'

```
class MyClass(object):
    __metaclass__ = Printable

    def __init__(self, value):
        self.value = value
```

  - Now instantiate an instance and call the method

```
MyClass(1).printme()
> value:  1
```

- The metaclass can be set for all classes in a module
  - by setting '__metaclass__' at the top of the module

A tracing metaclass

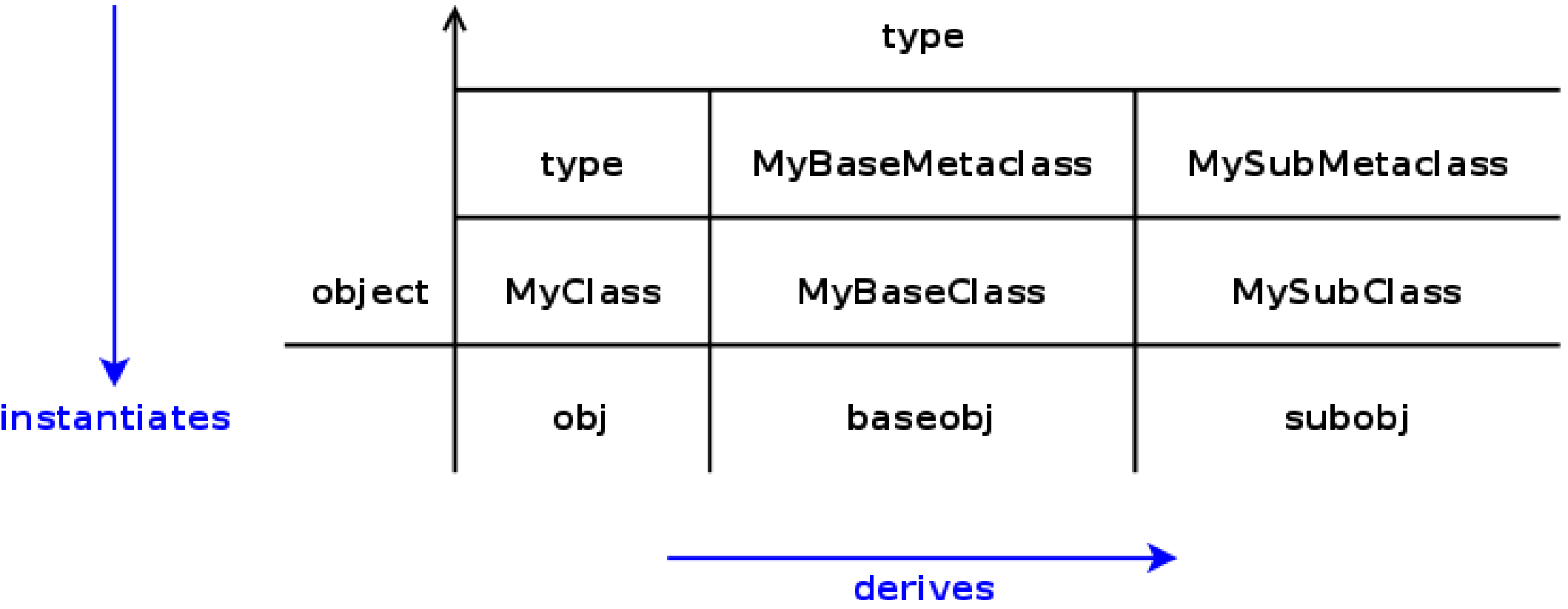
- Recall the 'trace' decorator
  - Suppose we want to decorate every method of a class with @trace...

```
class Tracable(type):
    def __init__(cls, name, bases, namespace):
        super(Tracable, cls).__init__(name, bases, namespace)
        for (key, value) in cls.__dict__.items():
            if isinstance(value, type(lambda x:x)):
                setattr(cls, key, trace(value))

class MyClass(object):
    __metaclass__ = Tracable

    def fun(self, value):
        return value

MyClass().fun(1)
> Entering function 'fun', arguments are: (<MyClass object at
  0x2ac49af996d0>, 1)
> Leaving function 'fun', result is: 1
> 1
```


type
type
MyBaseMetaclass
MySubMetaclass
object
MyClass
MyBaseClass
MySubClass
obj
baseobj
subobj
instantiates
derives

Metaclass composability

- A class can only have one metaclass
  - how can we compose them?
  - inheritance!

```
class Composed(Tracable, Printable):
    pass

class MyClass(object):
    __metaclass__ = Composed

    def __init__(self, value):
        self.value = 1

obj = MyClass(1)
> Entering function '__init__', arguments are: (<MyClass object at
  0x2ac49afa6350>, 1)
> Leaving function '__init__', result is: None
```

Metaclass composability

- What about the method 'printme' injected by Printable?

```
obj.printme()
> Entering function 'printme', arguments are: (<MyClass object at
  0x2ac49af99690>,)
> value:  1
> Leaving function 'printme', result is: None
```

- So in effect, 'printme' was injected before Tracable wrapped all methods with @trace
  - what determines this?
  - the order of derivation

```
class Composed(Printable, Tracable):
    pass

obj.printme()
> value:  1
```