DIEE - Università degli Studi di Cagliari

OOP and Scripting in Python

Part 3 – Advanced Features

Giuliano Armano – DIEE Univ. di Cagliari

Part 3 - Advanced Features

Python: Advanced Features

- Callables
- Iterators
- Functional programming
- Reflection and introspection

Callables

Part 3 – Advanced Features: Callables

Callables

- Types that support the function call operation are named “callable”
- List of “callable” types:

Functions	YES
Methods	YES
Types (e.g., tuples, lists, dictionaries)	YES
Class instances (supporting `__call__`)	YES

Callables (e.g., list-to-dict)

```
>>> q = [('x',1),('y',2),('z',3)]
>>> q
[('x', 1), ('y', 2), ('z', 3)]
>>> dict(q)
{'y': 2, 'x': 1, 'z': 3}
>>>
```

Callables: Function Objects

```
>>> class callable:
...   def __init__(self,function):
...     self.function = function
...   def __call__(self,*args):
...     return self.function(*args)
...

>>> def inc(x):
...   return x+1
...

>>> INC = callable(inc)
>>> INC(34)
35
```

Iterators

Part 3 – Advanced Features: Iterators

Iterators

- Iterators are standard tools for iterating over a sequence (string, tuple, list, dictionary)
- Iterators can be used also for iterating on instances
- In any case, when the iteration reached its end, a StopIteration exception is raised
- The module itertools contains useful iterators

Any for statement actually uses an iterator to perform iteration (and StopIteration forces a “break”)

Iterating over a Sequence (string)

```
>>> it = iter('abc')
>>> it.next()
a
>>> it.next()
b
>>> it.next()
c
>>> it.next()

Traceback (most recent call last):
File "<pyshell#493>", line 2, in -toplevel- print it.next()
StopIteration
>>>
```

Iterating over a Sequence (string)

How to avoid the StopIteration exception ...

```
>>> it = iter('abc')
>>> try:
...   while True:
...     print it.next()
... except StopIteration:
...   print 'End Iteration'
...

a
b
c
End Iteration
>>>
```

Iterating over a Sequence (list)

```
>>> it = iter([1,2,'a'])
>>> while True:
...     print it.next()
...

1
2
a

Traceback (most recent call last):
File "<pyshell#493>", line 2, in -toplevel- print it.next()
StopIteration
>>>
```

Iterators (on objects) - 1a

Using delegation to perform iteration

```
>>> class Counter:
...     def __init__(self):
...         self.cnt = -1
...     def __call__(self):
...         self.cnt += 1
...         return self.cnt
...
```

Any iterator built upon an instance of Counter actually delegates __call__ to perform the actual computation

Iterators (on objects) - 1b

```
>>> c = Counter()
>>> it = iter(c,5)
>>> it.next()           # same as: c.__call__()
0
>>> it.next()           # same as: c.__call__()
1
>>> c.__call__()
2
```

Note: In this case “it.next()” is equivalent to “c.__call__()”

Iterators (on objects) - 1c

Using delegation to perform iteration

```
>>> it = iter(Counter(),5)
>>> while True:
...     print it.next()
...

0
1
2
3
4

Traceback (most recent call last):
File "<pyshell#476>", line 2, in -toplevel- print it.next()
StopIteration
```

when it.next() returns 5 a StopIteration is raised

Iterators (on objects) - 1d

Using delegation to perform iteration

```
>>> it = iter(Counter(),5)
>>> for x in it:
...    print x
...

0
1
2
3
4
>>>
```

Iterators (on objects) - 2a

Any object can be made "iterable"

```
>>> class Counter:
...     def __init__(self, maxvalue):
...       self.maxvalue = maxvalue
...     def __iter__(self):
...       self.cnt = -1
...       return iter(self,self.maxvalue)
...     def __call__(self):
...       self.cnt += 1
...       return self.cnt
...
```

On creation, the iterator delegates __iter__ to return a valid "iterable" sequence

Sentinel!!!

Iterators (on objects) - 2b

```
>>> for x in Counter(5):
...    print x
...

0
1
2
3
4
>>>
```

Iterators (itertools)

```
from itertools import *
```

- Some itertools:

```
chain (*iterables)
count(n=0)
cycle(iterable)
imap(function, *iterables)

... etc. ...
```

Itertools (chain)

from itertools import *

```
>>> for x in chain([1,2,3],['a','b','c']):
...    print x
...

1
2
3
a
b
c
>>>
```

Itertools (count)

`from itertools import *`

```
>>> for x in count():
...     print x
...

0
1
2
3
4
5
... etc. ...
```

Itertools (count)

... equivalent to itertools.count

```
>>> def count(n=0):
...   while True:
...     yield n; n += 1
...

>>> for x in count():
...   print x
...

0
1
... etc. ...
```

Itertools (cycle)

```
from itertools import *
```

```
>>> for x in cycle([1,2,3]):
...    print x
...

1
2
3
1
2
3
1

... etc. ...
```

Itertools (imap)

from itertools import *

```
>>> it = imap(lambda x,y: x+y,[1,2,3],[4,5,6,7,8,9])
>>> it.next()
5
>>> it.next()
7
>>> it.next()
9
>>> it.next()

Traceback (most recent call last):
File "<pyshell#22>", line 1, in -toplevel- it.next()
StopIteration
```

Functional Programming

Part 3 – Advanced Features: Functional Programming

Functional Programming

Lambda (anonymous) functions	YES
Call function by name	YES
Function composition	YES
Sequence processing (map, filter, reduce)	YES

Lambda (Anonymous) Functions

```
>>> def inc(y=1):
...    return lambda x: x+y
...

>>> inc1 = inc()
>>> inc2 = inc(2)
>>> inc1(10)
11
>>> inc2(10)
12
>>>
```

Call Function by Name

```
>>> def add(*numbers):
...    res = 0
...    for x in numbers:
...      res += x
...    return res
...

>>> add(1,2,3,4)
10
>>> apply(add,[1,2,3,4])       # deprecated!
10
>>>
```

Function Composition

```
>>> def compose(f1,f2):
...     return lambda x: f1(f2(x))
...

>>> lsqrt = compose(log,sqrt)
>>> lsqrt(10)
1.15129254649702
>>> log(sqrt(10))
1.15129254649702
>>>
```

Sequence Processing: map

```
>>> def add10(x):
...    return x+10
...

>>> map(add10,[10,20,30,40])
[20, 30, 40, 50]
>>>
```

Sequence Processing: map

```
>>> a = ['x','y','z']
>>> b = [1,2,3]
>>> w = map(lambda x,y: (x,y),a,b)
>>> w
[('x', 1), ('y', 2), ('z', 3)]
>>> dict(w)
{'y': 2, 'x': 1, 'z': 3}
>>>
```

Sequence Processing: filter

```
>>> filter(lambda x: x < 35,[10,20,30,40])
[10, 20, 30]
```

Sequence Processing: reduce

```
>>> reduce(lambda x,y: x+y,[1,2,3,4])
10
>>> ((1+2)+3)+4
10
>>> def logsin(x,y):
...   return log(abs(x)) * sin(y)
...

>>> reduce(logsin,[10,20,30])
-0.73406113699093767
>>> logsin(logsin(10,20),30)
-0.73406113699093767
```

Reflection and Introspection

Part 3 – Advanced Features: Reflection and Introspection

Reflection vs. Meta-Programming

- Meta-programming is the art of developing methods and programs to read, manipulate, and/or write other programs
- When what is developed are programs that can deal with themselves, we talk about Reflective Programming (or Reflection)
- We take as our definition of reflection the loosest interpretation: reflection is evidenced in a program that is able to change it's structure or behavior at run-time

Reflection vs. Introspection

- Introspection is a programmatic facility built on top of reflection and a few supplemental specifications (e.g., Java beans)
- Introspection provides somewhat higher-level information about a class than does reflection, and the information provided can be customized by the class provider or packager independant of the class itself
- Introspection is especially designed to be useful in conjunction with visual application assembly tools (e.g., JavaBeans)

Reflection in Purely Reflective Systems

- In a purely reflective system, one would expect to find the following implementation abstraction:
  - A program interacts with the meta-objects only through the meta-object protocol (MOP)
  - Base-objects interact with meta-objects and can interact with each other only through meta-objects
  - Base-objects maintain the actual information based on structural and/or behavioral descriptions maintained by the meta-objects

Reflection in Python

`from new import *`

- In Python, we see an implementation where (i) the MOP, (ii) meta-objects, and (iii) base objects are combined into one entity, whose type is classobj
- All functions to add / modify / delete attributes and methods are encapsulated in the classobject.c source file
- The structure-changing behavior of the interpreter loosely corresponds to reflection because the implementation allows runtime changing of instance object's classes

Reflection in Python: Inspector

`from inspect import *`

- Functions:

```
getmembers(object,predicate=None)
...
```

- Predicates:

`isclass(object)`	`classobj`
`ismethod(object)`	`class/instancemethod`
`isfunction(object)`	`function`
`isbuiltin(object)`	`str, int, long, ...`
`...`	

Reflection in Python: Inspector

```
from inspect import *
```

```
>>> class Blob:
...    def __init__(self, x=0, y='pluto'):
...      self.x = x; self.y = y
...    def foo(self):
...      return self.x, self.y
...

>>> getmembers(Blob,ismethod)
[('__init__', <unbound method Blob.__init__>),
  ('foo', <unbound method Blob.foo>)]
>>> isclass(Blob)
True
```

Reflection in Python: Inspector

```
from inspect import *
```

```
>>> class Blob:
...    def __init__(self, x=0, y='pluto'):
...      self.x = x; self.y = y
...    def foo(self):
...      return self.x, self.y
...

>>> getmembers(Blob,ismethod)
[('__init__', <unbound method Blob.__init__>),
  ('foo', <unbound method Blob.foo>)]
>>> isclass(Blob)
True
```

Type …

- The built-in function **type** accepts an object and returns the type object that represents it
- The built-in function **isinstance** accepts an object and a type and returns a boolean

```
>>> class blob:
...    pass
...
>>> type(blob)
<type 'classobj'>
```

```
>>> a = blob()
>>> type(a)
<type 'instance'>
>>> isinstance(a,blob)
True
>>>
```

Typical Reflective Operations (read)

Getting the instance attributes of a class	NO
Getting the attribute values of an instance	YES
Getting the methods of a class or instance	YES

Getting the Attribute Values of an Instance

```
from inspect import *
```

```
>>> class A:
...   def __init__(self,x=0,y=0):
...     self.x = x; self.y = y
...

>>> a = A()
>>> a
<__main__.A instance at 0x00A9FFD0>
>>> a.__dict__
{'y': 0, 'x': 0}
>>>
```

Getting the Attribute Values of an Instance

from inspect import *

```
>>> class A:
...    def __init__(self,x=0,y='pluto'):
...      self.x = x; self.y = y
...

>>> a=A()
>>> for attr, value in a.__dict__.items():
...    print "ATTR = %s, VALUE = %s" % (attr, value)

ATTR = y, VALUE = pluto
ATTR = x, VALUE = 0
```

Getting the (Unbound) Methods of a Class

from inspect import *

```
>>> class A:
...    """Class A - documentation"""
...    def __init__(self,x=0,y='pluto'):
...      self.x = x; self.y = y
...    def foo(self):
...      return self.x, self.y
...

>>> getmembers(A,ismethod)
[('__init__', <unbound method A.__init__>), ('foo',
  <unbound method A.foo>)]
>>>
```

Getting the (Bound) Methods of an Instance

`from inspect import *`

```
>>> getmembers(A,ismethod)
[('__init__', <unbound method A.__init__>), ('foo',
  <unbound method A.foo>)]
>>> a = A()
>>> getmembers(a,ismethod)
[('__init__', <bound method A.__init__ of
  <__main__.A instance at 0x00AB6E68>>), ('foo',
  <bound method A.foo of <__main__.A instance at
  0x00AB6E68>>)]
>>>
```

Typical Reflective Operations (write)

Class Declaration	YES
Object instantiation	YES
Class Mutation	YES
Object Mutation	YES
Changing the Link Instance-to-Class	YES

Class declaration (using a “MacroOp”) - 1

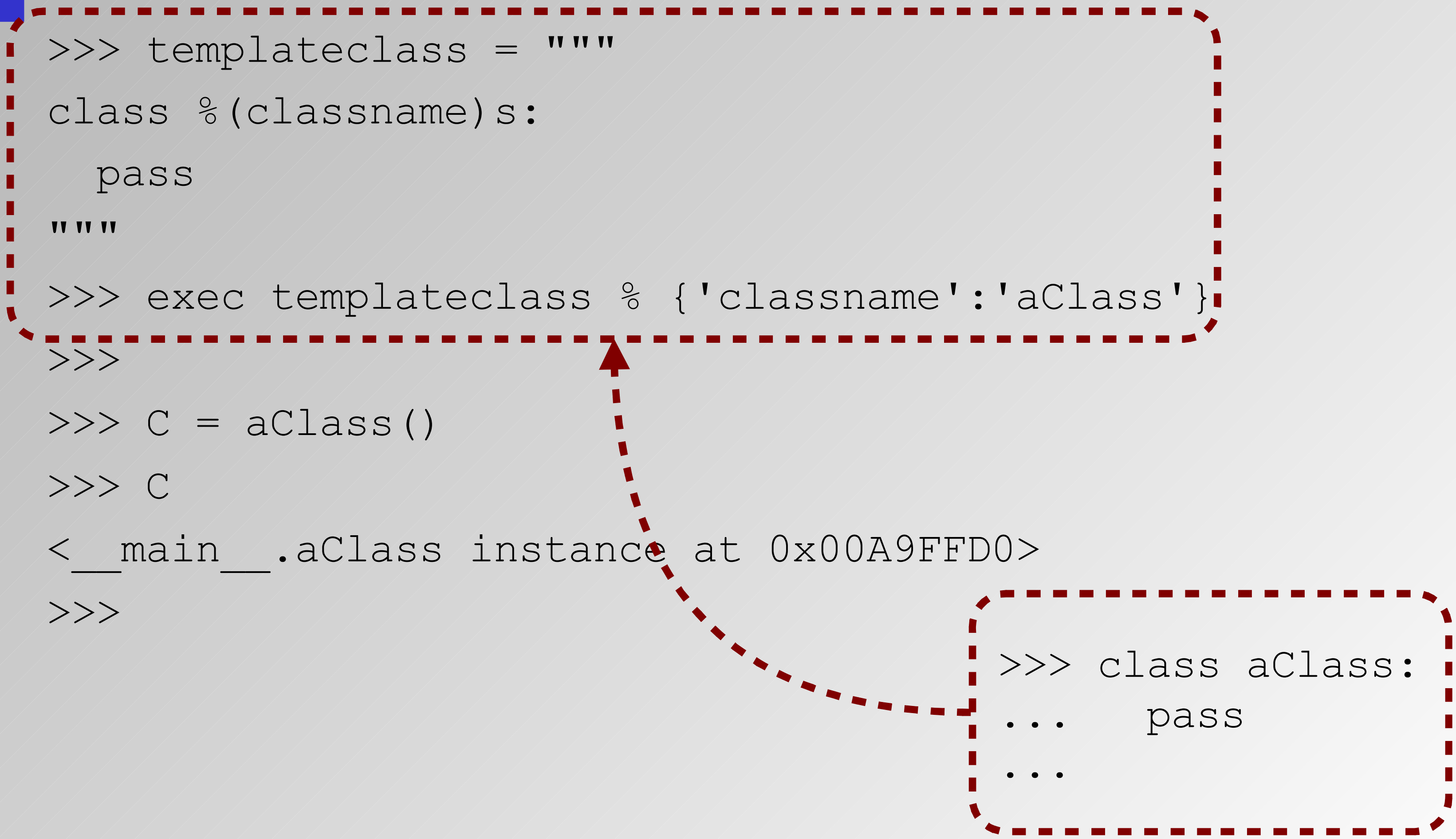
```
>>> C

Traceback (most recent call last):
  File "<pyshell#27>", line 1, in -toplevel- C
NameError: name 'C' is not defined

>>> A = "class aClass:\n\tpass\n"
>>> exec(A)
>>>
>>> C = aClass()
>>> C
<__main__.aClass instance at 0x00A9FFD0>
>>>
```

```
>>> class aClass:
...     pass
...
```

Class declaration (using a “MacroOp”) - 2



```
>>> templateclass = """
class %(classname)s:
  pass
"""
>>> exec templateclass % {'classname':'aClass'}
>>>
>>> C = aClass()
>>> C
<__main__.aClass instance at 0x00A9FFD0>
>>>
```

```
>>> class aClass:
...    pass
...
```

Class Declaration (using a “MacroOp“) - 3

```
>>> templatecode="""
class %(class_name)s:
    def __init__(self,%(slot1)s,%(slot2)s):
        self.%(slot1)s = %(slot1)s
        self.%(slot2)s = %(slot2)s
"""
>>> exec templatecode % { 'class_name' : 'Bip',
  'slot1' : 'x', 'slot2' : 'y' }
>>> Bip
<class __main__.Bip at 0x00AB7570>
>>> b = Bip(1,2)
>>> print b.x, b.y
1 2
```

Class Declaration (using “classobj”)

`from new import *`

```
>>> def Blob__init(self,x):
...     self.x = x
...

>>> Blob = classobj('Blob',(),{'__init__':Blob__init})
>>> Blob
<class __main__.Blob at 0x00AB77E0>
>>> dir(Blob)
['__doc__', '__init__', '__module__']
>>> b = Blob(10)
>>> b.x
10
```

Object Creation (1)

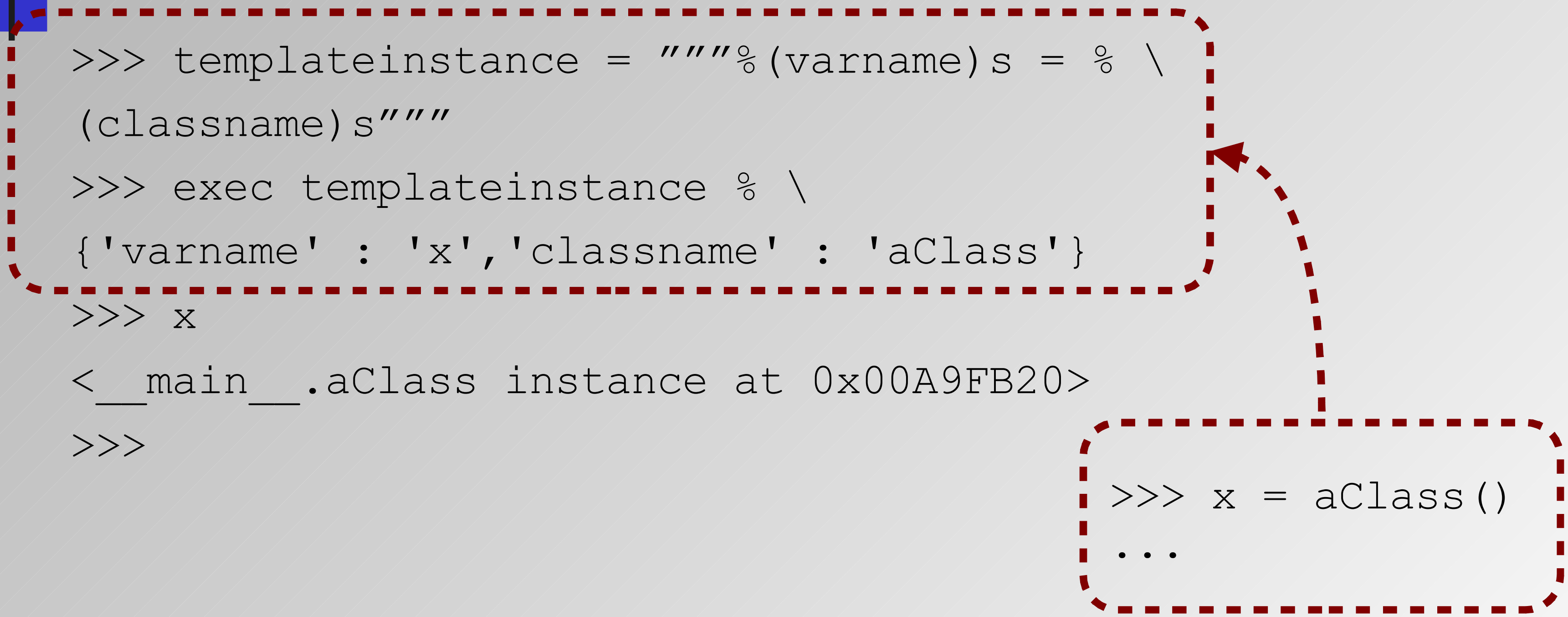
```
>>> x

Traceback (most recent call last):
File "<pyshell#153>", line 1, in -toplevel- x
NameError: name 'x' is not defined

>>> B = "x = aClass()\n"
>>> exec(B)
>>> x
<__main__.aClass instance at 0x00A9FB20>
>>>
```

```
>>> x = aClass()
...
```

Object Creation (2)



```
>>> templateinstance = ”””%(varname)s = % \
(classname)s”””
>>> exec templateinstance % \
{'varname' : 'x','classname' : 'aClass'}
>>> x
<__main__.aClass instance at 0x00A9FB20>
>>>
```

```
>>> x = aClass()
...
```

Object Creation (3)

```
>>> t = instance(Z,{'x' : 10, 'y':20})
>>> t
<__main__.Z instance at 0x00AB9468>
>>> t.x
10
>>> t.y
20
>>> t.__dict__
{'y': 20, 'x': 10}
>>>
```

Class Mutation

```
>>> class Point:
...   def __init__(self,x=0,y=0):
...     self.x, self.y = x,y
...

>>> from math import sqrt
>>> def distance(p1,p2):
...   return sqrt( (p1.x-p2.x)**2 + (p1.y-p2.y)**2 )
...

>>> w1, w2 = Point(1,1), Point(2,4)
>>> Point.distance = distance     # adding a method !!!
>>> print w1.distance(w2)
3.16227766017
```

Object Mutation

```
>>> class Point:
...   def __init__(self,x=0,y=0):
...     self.x, self.y = x,y
...

>>> w1 = Point()
>>> w1.z = 0
>>> print "z = ", w1.z
z = 0
```

Changing the Link Instance-to-Class

```
>>> class W:
...   def __init__(self):
...     self.x = 1
...   def foo(self):
...     print "W::x = ", self.x
...

>>> w = W()
>>> w.foo()
W::x =  0
```

Changing the Link Instance-to-Class

```
>>> class Z:
...   def __init__(self):
...     self.y = 1
...   def foo(self):
...     print "Z::x = ", 10 * self.x
...

>>> w.__class__ = Z
>>> w.foo()
Z::x =  10
>>> w.__class__ = W
>>>
```

Adding an Attribute to a Class

```
>>> class Z:
...   def __init__(self):
...     self.y = 1
...   def foo(self):
...     print "Z::x = ", 10 * self.x
...

>>> z = Z()
>>> z.__dict__
{'y': 1}
>>> Z.e = 10
```

Adding an Attribute to a Class

```
>>> Z.e
10
>>> z.e
10
>>> z.e = 11
>>> Z.e
10
```

```
>>> s = Z()
>>> s.e
10
>>> Z.e = 22
>>> s.e
22
>>>
```

Adding a Method to a Class

```
>>> s.foo()

Traceback (most recent call last):
  File "<pyshell#414>", line 1, in -toplevel-
    s.foo1()
AttributeError: Z instance has no attribute 'foo'

>>> def foo(self):
...     print "foo!"
...

>>> Z.foo = foo
>>> s.foo()
foo!
```

Adding a Method to an Instance

```
>>> class MethodWrapper:
...   def __init__(self,instance,method):
...     self.instance = instance
...     self.method = method
...   def __call__(self,*args):
...     return self.method(self.instance,*args)
...
```

... through a method wrapper

Adding a Method to an Instance

```
>>> def zot(self):
...    return 100
...

>>> s.zot = MethodWrapper(s,zot)
>>> s.zot()
100
>>>
```

... through a method wrapper

Adding a Method to an Instance

from new import *

```
>>> def zot(self):
...    return 100
...

>>> s.zot = instancemethod(s,zot)
>>> s.zot()
100
>>>
```

... using “instancemethod”