Python高级编程（一）

Happy Day #4
2011.7

CPython 2.6

Topics

- Object Model / Descriptor / Meta Pragramming
- Decorator
- Generator
- Performance Tuning (if we have enough time)

Object Model

What is Object?

- Identity
- State
- Behavior

- 数据（属性）和方法的集合
- Python里，方法就是callable的属性

2 Object Models

- Old style classes
- New style classes

后面只说new style classes

Creat a class

```
class C1(object):
    a = 'a'

    def f1(self):
        return 'f1'
```

Creat a class

```
class C1(object):
    a = 'a'

    def f1(self):
        return 'f1'
```

类定义是可执行语句

Creat a class

```
class C1(object):
    a = 'a'

    def f1(self):
        return 'f1'
```

类定义是可执行语句

类本身也是对象

```
>>> isinstance(C1, object):
True
>>> type(C1)
<type 'type'>
```

Class Attributes

Class Attributes

```
>>> C1.__name__
 'C1'
```

Class Attributes

```
>>> C1.__name__
 'C1'

>>> C1.__module__
'__main__'
```

Class Attributes

```
>>> C1.__name__
 'C1'

>>> C1.__module__
'__main__'

 >>> C1.__doc__
```

Class Attributes

```
>>> C1.__name__
 'C1'

>>> C1.__module__
'__main__'

>>> C1.__doc__

>>> C1.__dict__
<dictproxy object at 0x1005647c0>
```

Class Attributes

```
>>> C1.__name__
 'C1'

>>> C1.__module__
'__main__'

>>> C1.__doc__


>>> C1.__dict__
<dictproxy object at 0x1005647c0>

>>> C1.__dict__['a']
'a'
```

Class Attributes

```
>>> C1.__name__
 'C1'

>>> C1.__module__
'__main__'

>>> C1.__doc__

>>> C1.__dict__
<dictproxy object at 0x1005647c0>

>>> C1.__dict__['a']
'a'

>>> C1.a
'a'
```

unbound method

unbound method

```
>>> C1.__dict__['f1']
<function f1 at 0x10055fcf8>
```

unbound method

```
>>> C1.__dict__['f1']
<function f1 at 0x10055fcf8>


>>> C1.f1
<unbound method C1.f1>
```

unbound method

```
>>> C1.__dict__['f1']
<function f1 at 0x10055fcf8>


>>> C1.f1
<unbound method C1.f1>


>>> C1.f2 = lambda self: 'f2'
```

unbound method

```
>>> C1.__dict__['f1']
<function f1 at 0x10055fcf8>


>>> C1.f1
<unbound method C1.f1>


>>> C1.f2 = lambda self: 'f2'


>>> C1.__dict__['f2']
<function <lambda> at 0x10055fc08>
```

unbound method

```
>>> C1.__dict__['f1']
<function f1 at 0x10055fcf8>


>>> C1.f1
<unbound method C1.f1>


>>> C1.f2 = lambda self: 'f2'


>>> C1.__dict__['f2']
<function <lambda> at 0x10055fc08>


>>> C1.f2
<unbound method C1.<lambda>>
```

unbound method (2)

unbound method (2)

```
>>> C1.f1.im_class
<class '__main__.C1'>
```

unbound method (2)

```
>>> C1.f1.im_class
<class '__main__.C1'>
>>> C1.f1.im_func
<function f1 at 0x10055fc80>
```

unbound method (2)

```
>>> C1.f1.im_class
<class '__main__.C1'>
>>> C1.f1.im_func
<function f1 at 0x10055fc80>
>>> C1.f1.im_func is C1.__dict__['f1']
True
```

unbound method (2)

```
>>> C1.f1.im_class
<class '__main__.C1'>
>>> C1.f1.im_func
<function f1 at 0x10055fc80>
>>> C1.f1.im_func is C1.__dict__['f1']
True
>>> c1 = C1()
>>> class C2(object): pass
...
>>> c2 = C2()
>>> C1.__dict__['f1'](c1)
'f1'
>>> C1.f1(c1)
'f1'
```

unbound method (2)

```
>>> C1.f1.im_class
<class '__main__.C1'>
>>> C1.f1.im_func
<function f1 at 0x10055fc80>
>>> C1.f1.im_func is C1.__dict__['f1']
True
>>> c1 = C1()
>>> class C2(object): pass
...
>>> c2 = C2()
>>> C1.__dict__['f1'](c1)
'f1'
>>> C1.f1(c1)
'f1'
>>> C1.__dict__['f1'](c2)
'f1'
>>> C1.f1(c2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unbound method f1() must be called with C1 instance as first
argument (got C2 instance instead)
```

继承

```
>>> class C3(C1): pass
...
>>> C3.a
'a'

>>> C3.__dict__['a']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'a'
```

继承

```
>>> class C3(C1): pass
...
>>> C3.a
'a'

>>> C3.__dict__['a']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'a'

>>> C3.__bases__
(<class '__main__.C1'>,)
>>> C3.__bases__[0].a
'a'
```

多重继承

多重继承

```
>>> class C4(C2, C1): pass
...
>>> C4.a
'a'
```

多重继承

```
>>> class C4(C2, C1): pass
...
>>> C4.a
'a'
>>> C4.__bases__
(<class '__main__.C2'>, <class '__main__.C1'>)
```

多重继承

```
>>> class C4(C2, C1): pass
...
>>> C4.a
'a'
>>> C4.__bases__
(<class '__main__.C2'>, <class '__main__.C1'>)
>>> C4.__bases__[0].a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'C2' has no attribute 'a'
```

多重继承

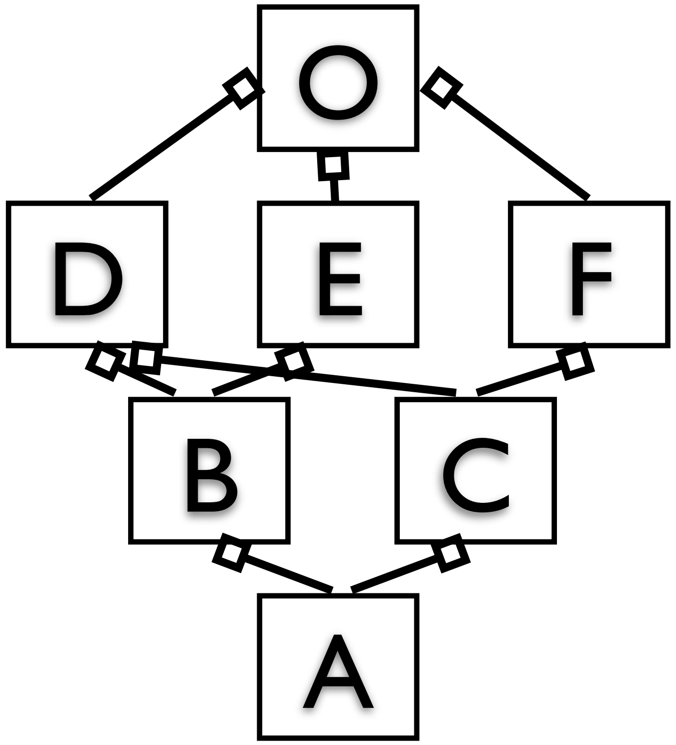
```
>>> class C4(C2, C1): pass
...
>>> C4.a
'a'
>>> C4.__bases__
(<class '__main__.C2'>, <class '__main__.C1'>)
>>> C4.__bases__[0].a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'C2' has no attribute 'a'
>>> C4.__bases__[1].a
'a'
```

Method Resolution Order

- Old-Style Classes: depth-first
- New-Style Classes: C3

Method Resolution Order

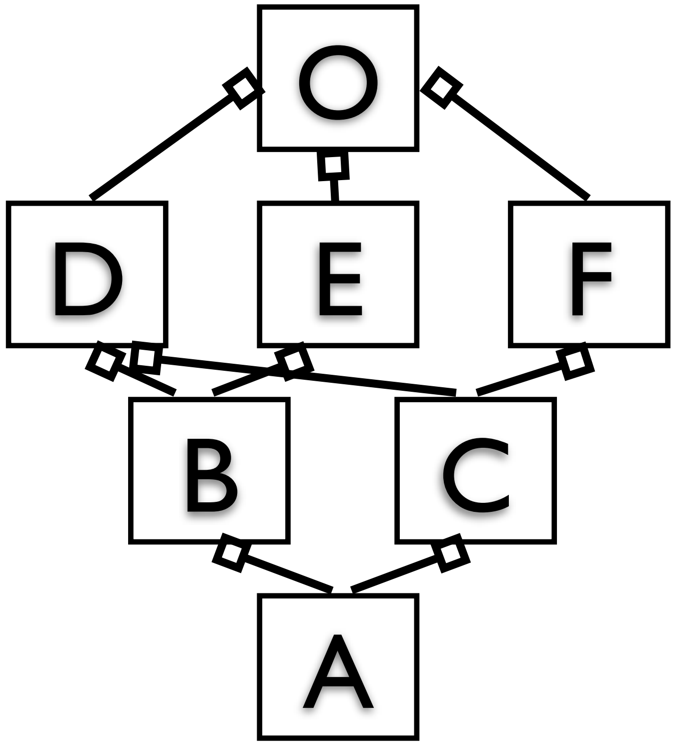
```
>>> O = object
>>> class F(O): pass
>>> class E(O): pass
>>> class D(O): pass
>>> class C(D,F): pass
>>> class B(D,E): pass
>>> class A(B,C): pass
```



Method Resolution Order

```
>>> O = object
>>> class F(O): pass
>>> class E(O): pass
>>> class D(O): pass
>>> class C(D,F): pass
>>> class B(D,E): pass
>>> class A(B,C): pass
```

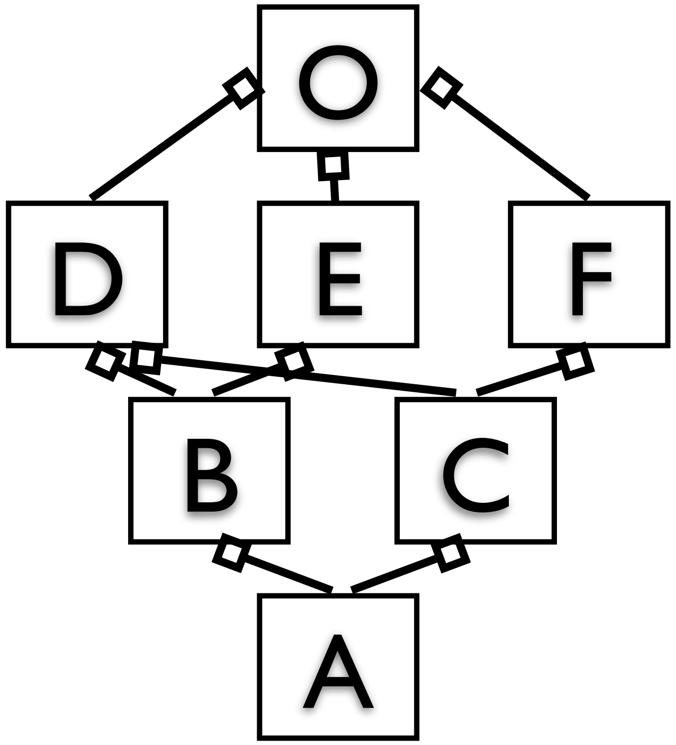
```
L[O] = O
```



Method Resolution Order

```
>>> O = object
>>> class F(O): pass
>>> class E(O): pass
>>> class D(O): pass
>>> class C(D,F): pass
>>> class B(D,E): pass
>>> class A(B,C): pass
```

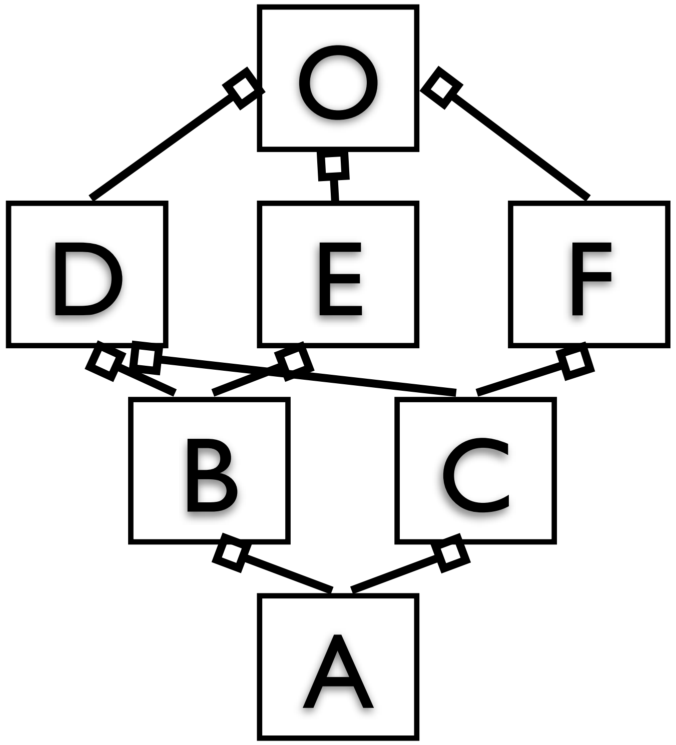
```
L[O] = O
L[F] = F O
```



Method Resolution Order

```
>>> O = object
>>> class F(O): pass
>>> class E(O): pass
>>> class D(O): pass
>>> class C(D,F): pass
>>> class B(D,E): pass
>>> class A(B,C): pass
```

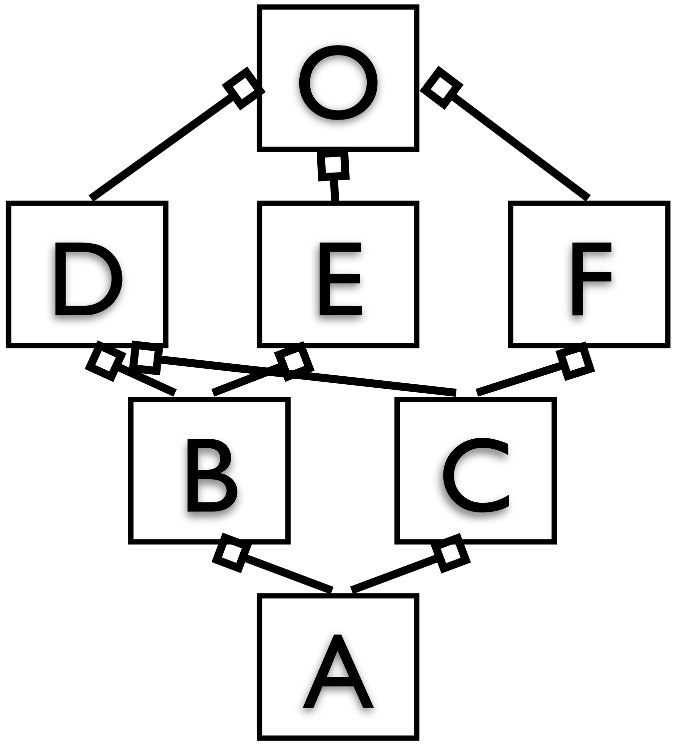
```
L[O] = O
L[F] = F O
L[E] = E O
```



Method Resolution Order

```
>>> O = object
>>> class F(O): pass
>>> class E(O): pass
>>> class D(O): pass
>>> class C(D,F): pass
>>> class B(D,E): pass
>>> class A(B,C): pass
```

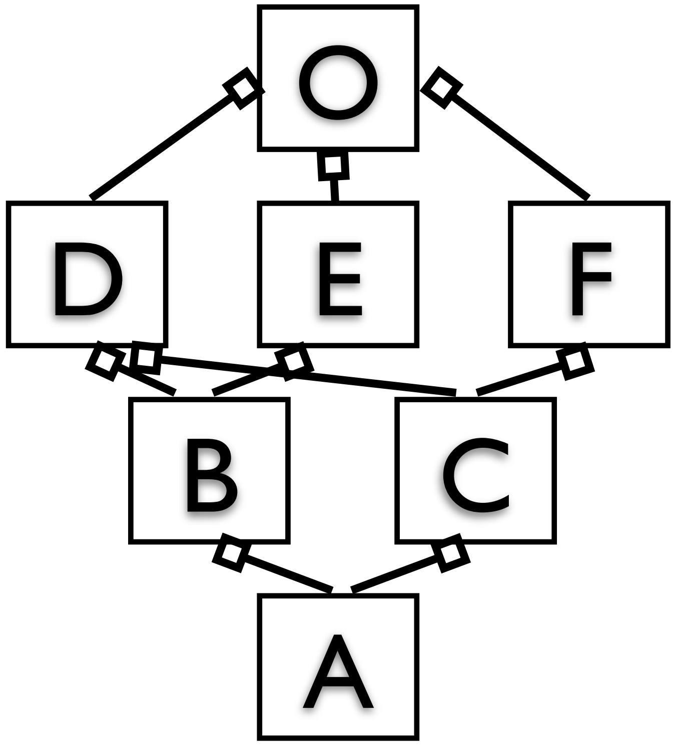
```
L[O] = O
L[F] = F O
L[E] = E O
L[D] = D O
```



Method Resolution Order

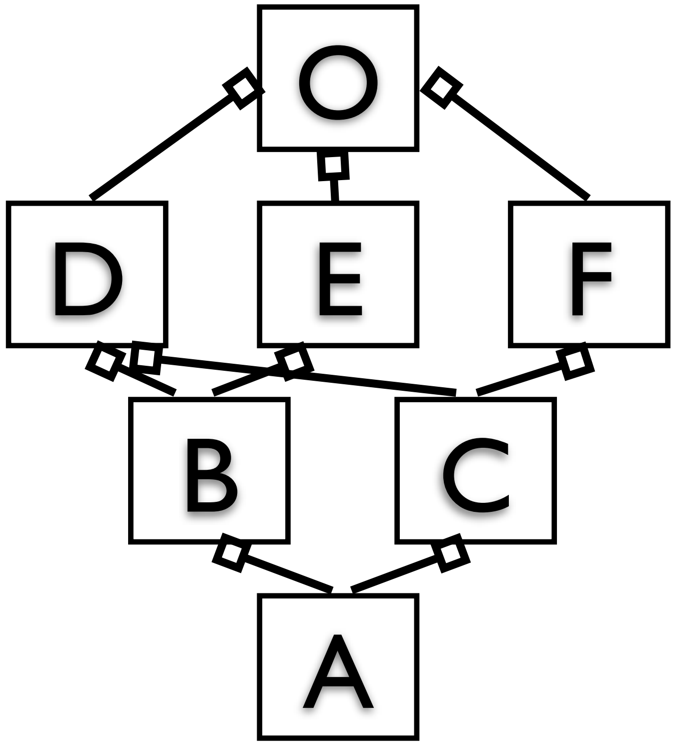
```
>>> O = object
>>> class F(O): pass
>>> class E(O): pass
>>> class D(O): pass
>>> class C(D,F): pass
>>> class B(D,E): pass
>>> class A(B,C): pass
```

```
L[O] = O
L[F] = F O
L[E] = E O
L[D] = D O
L[C] = C merge(DO,FO,DF)
     = C D merge(O,FO,F)
     = C D F merge(O,O)
     = C D F O
```



Method Resolution Order

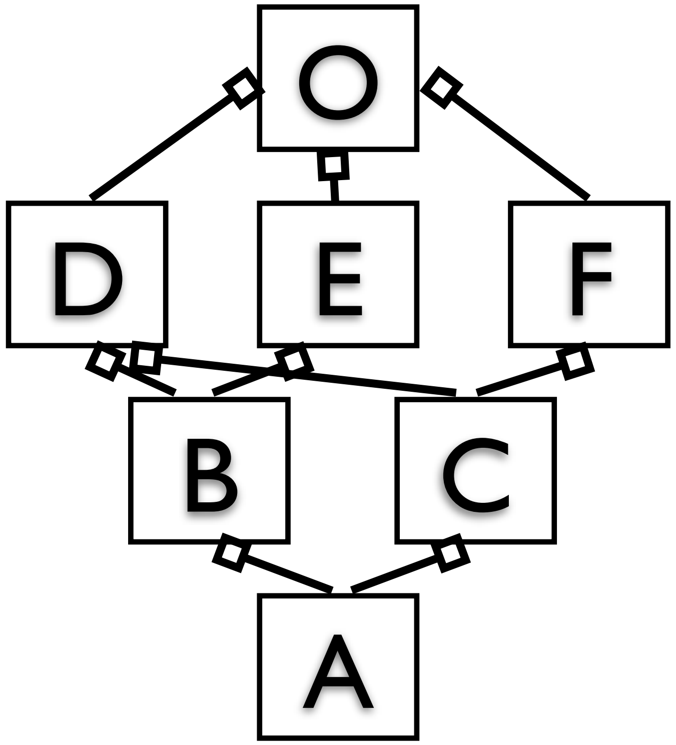
```
>>> O = object
>>> class F(O): pass
>>> class E(O): pass
>>> class D(O): pass
>>> class C(D,F): pass
>>> class B(D,E): pass
>>> class A(B,C): pass
```



```
L[O] = O
L[F] = F O
L[E] = E O
L[D] = D O
L[C] = C merge(DO,FO,DF)
     = C D merge(O,FO,F)
     = C D F merge(O,O)
     = C D F O
L[B] = B merge(DO,EO,DE)
     = B D merge(O,EO,E)
     = B D E merge(O,O)
     = B D E O
```

Method Resolution Order

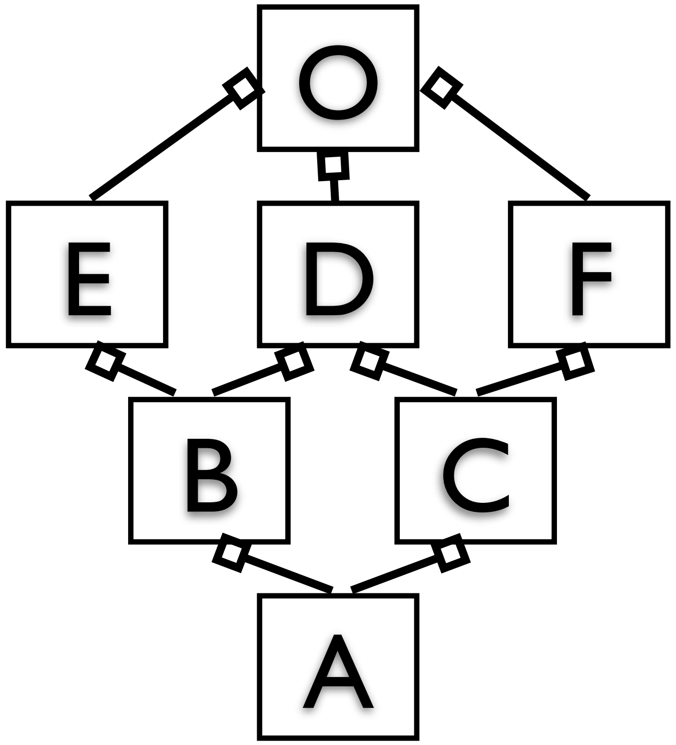
```
>>> O = object
>>> class F(O): pass
>>> class E(O): pass
>>> class D(O): pass
>>> class C(D,F): pass
>>> class B(D,E): pass
>>> class A(B,C): pass
```



```
L[O] = O
L[F] = F O
L[E] = E O
L[D] = D O
L[C] = C merge(DO,FO,DF)
     = C D merge(O,FO,F)
     = C D F merge(O,O)
     = C D F O
L[B] = B merge(DO,EO,DE)
     = B D merge(O,EO,E)
     = B D E merge(O,O)
     = B D E O
L[A] = A merge(BDEO,CDFO,BC)
     = A B merge(DEO,CDFO,C)
     = A B C merge(DEO,DFO)
     = A B C D merge(EO,FO)
     = A B C D E merge(O,FO)
     = A B C D E F merge(O,O)
     = A B C D E F O
```

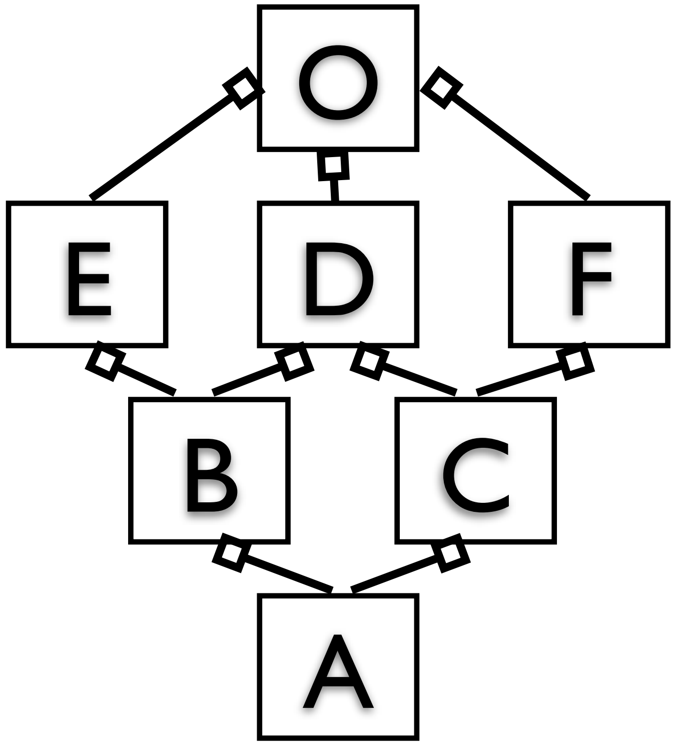
Exercise

```
>>> O = object
>>> class F(O): pass
>>> class E(O): pass
>>> class D(O): pass
>>> class C(D,F): pass
>>> class B(E,D): pass
>>> class A(B,C): pass
```



Exercise

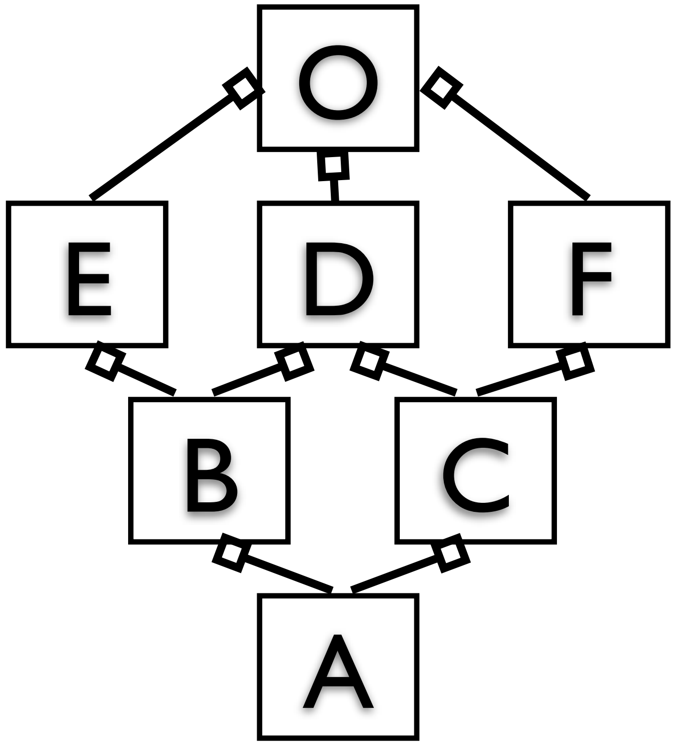
```
>>> O = object
>>> class F(O): pass
>>> class E(O): pass
>>> class D(O): pass
>>> class C(D,F): pass
>>> class B(E,D): pass
>>> class A(B,C): pass
```



```
L[O] = O
L[F] = F O
L[E] = E O
L[D] = D O
L[C] = C merge(DO,FO,DF)
     = C D merge(O,FO,F)
     = C D F merge(O,O)
     = C D F O
```

Exercise

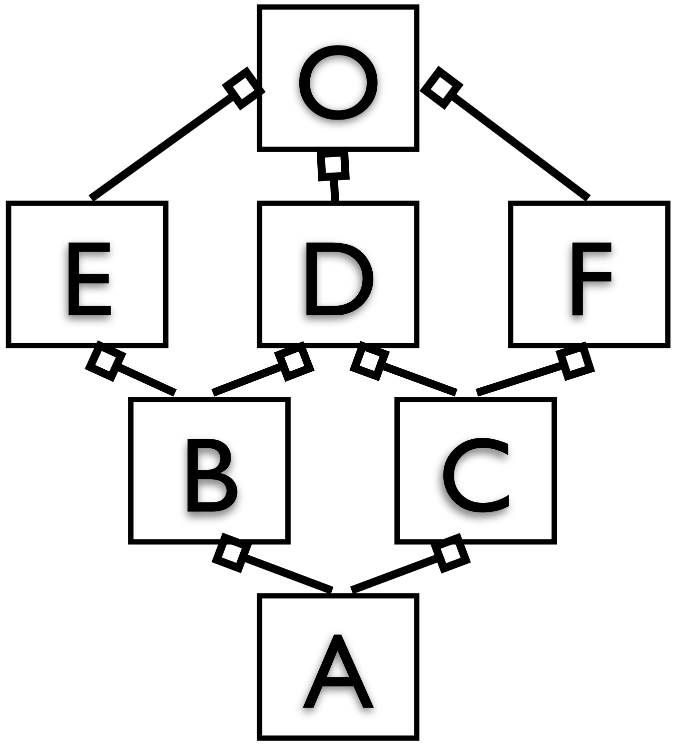
```
>>> O = object
>>> class F(O): pass
>>> class E(O): pass
>>> class D(O): pass
>>> class C(D,F): pass
>>> class B(E,D): pass
>>> class A(B,C): pass
```



```
L[O] = O
L[F] = F O
L[E] = E O
L[D] = D O
L[C] = C merge(DO,FO,DF)
     = C D merge(O,FO,F)
     = C D F merge(O,O)
     = C D F O
L[B] = B merge(EO,DO,ED)
     = B E merge(O,DO,D)
     = B E D merge(O,O)
     = B E D O
```

Exercise

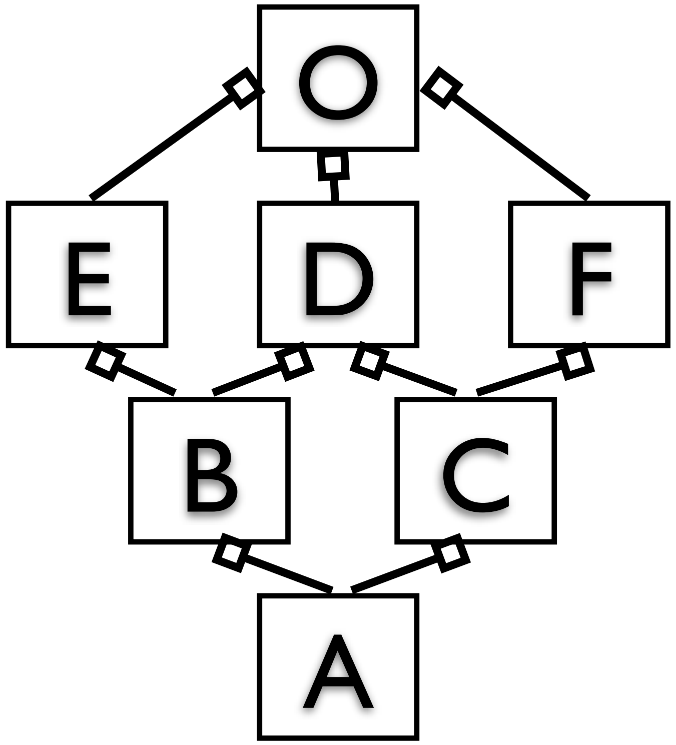
```
>>> O = object
>>> class F(O): pass
>>> class E(O): pass
>>> class D(O): pass
>>> class C(D,F): pass
>>> class B(E,D): pass
>>> class A(B,C): pass
```



```
L[O] = O
L[F] = F O
L[E] = E O
L[D] = D O
L[C] = C merge(DO,FO,DF)
     = C D merge(O,FO,F)
     = C D F merge(O,O)
     = C D F O
L[B] = B merge(EO,DO,ED)
     = B E merge(O,DO,D)
     = B E D merge(O,O)
     = B E D O
L[A] = A merge(BEDO,CDFO,BC)
     = A B merge(EDO,CDFO,C)
     = A B E merge(DO,CDFO,C)
     = A B E C merge(DO,DFO)
     = A B E C D merge(O,FO)
     = A B E C D F merge(O,O)
     = A B E C D F O
```

Exercise

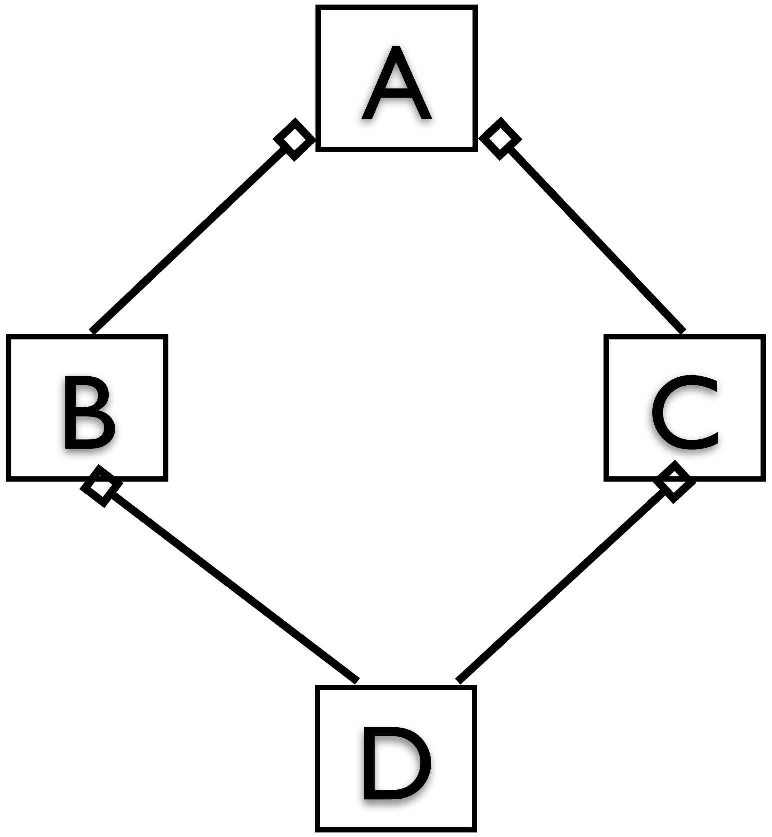
```
>>> O = object
>>> class F(O): pass
>>> class E(O): pass
>>> class D(O): pass
>>> class C(D,F): pass
>>> class B(E,D): pass
>>> class A(B,C): pass
```



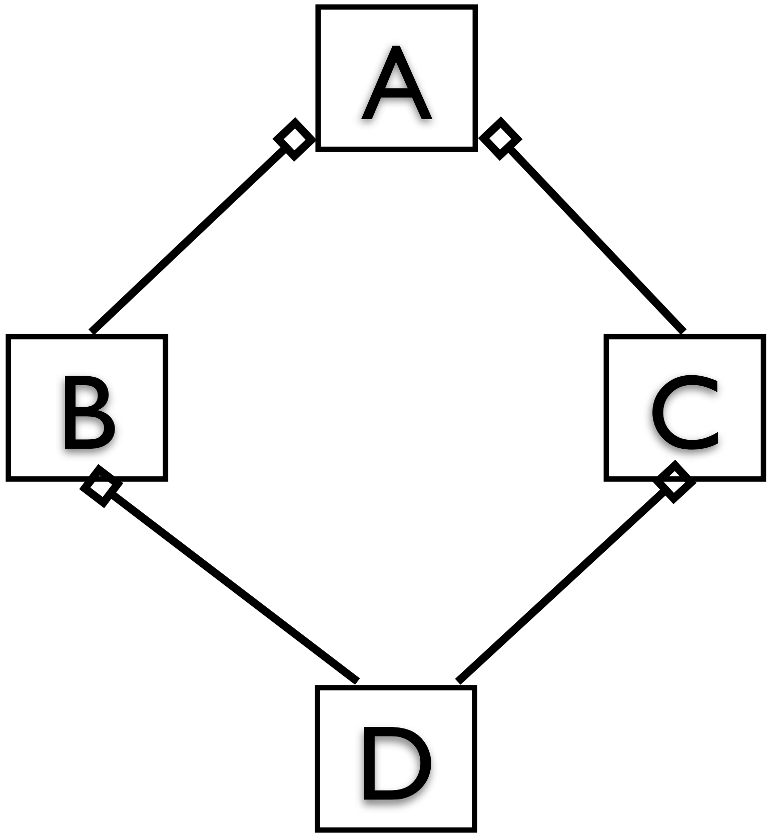
```
L[O] = O
L[F] = F O
L[E] = E O
L[D] = D O
L[C] = C merge(DO,FO,DF)
     = C D merge(O,FO,F)
     = C D F merge(O,O)
     = C D F O
L[B] = B merge(EO,DO,ED)
     = B E merge(O,DO,D)
     = B E D merge(O,O)
     = B E D O
L[A] = A merge(BEDO,CDFO,BC)
     = A B merge(EDO,CDFO,C)
     = A B E merge(DO,CDFO,C)
     = A B E C merge(DO,DFO)
     = A B E C D merge(O,FO)
     = A B E C D F merge(O,O)
     = A B E C D F O
```

```
A.mro()
```

Diamond Inheritance

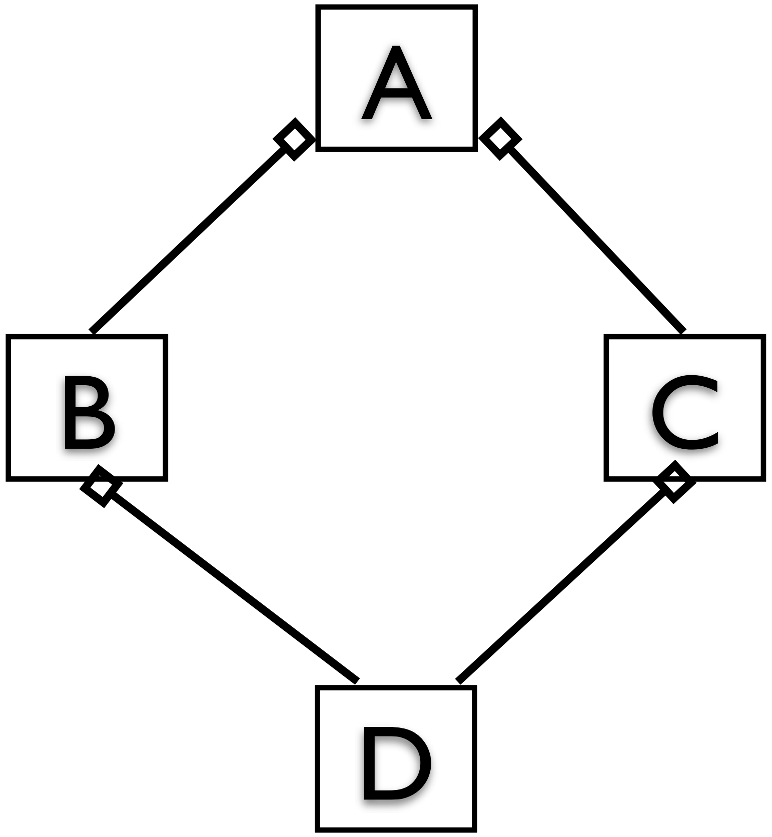


Diamond Inheritance



```
old style:
L[C] = C A
L[D] = D B A C A
```

Diamond Inheritance



```
old style:
L[C] = C A
L[D] = D B A C A


C3:
L[C] = C A
L[D] = D B C A
```

Super Call

- 如何调用父类的方法?
- Traditional Super Call
  - 直接调用父类的方法，将子类对象传入
  - Parent.method(self)
- Cooperative Super Call
  - 用super函数

Traditional Super Call

```
class A(object):
    def f(self):
        print "in A"
class B(A):
    def f(self):
        print "in B"
        A.f(self)
class C(A):
    def f(self):
        print "in C"
        A.f(self)
class D(B,C):
    def f(self):
        print "in D"
        B.f(self)
        C.f(self)
```

Traditional Super Call

```
class A(object):
    def f(self):
        print "in A"
class B(A):
    def f(self):
        print "in B"
        A.f(self)
class C(A):
    def f(self):
        print "in C"
        A.f(self)
class D(B,C):
    def f(self):
        print "in D"
        B.f(self)
        C.f(self)
```

```
>>> d = D()
>>> d.f()
in D
in B
in A
in C
in A
```

Cooperative Super Call

```
class A(object):
    def f(self):
        print "in A"
class B(A):
    def f(self):
        print "in B"
        super(B, self).f()
class C(A):
    def f(self):
        print "in C"
        super(C, self).f()
class D(B, C):
    def f(self):
        print "in D"
        super(D, self).f()
```

Cooperative Super Call

```
class A(object):
    def f(self):
        print "in A"
class B(A):
    def f(self):
        print "in B"
        super(B, self).f()
class C(A):
    def f(self):
        print "in C"
        super(C, self).f()
class D(B, C):
    def f(self):
        print "in D"
        super(D, self).f()
```

```
>>> d = D()
>>> d.f()
in D
in B
in C
in A
```

Super Considered Harmful?

Super Considered Harmful?

- 在某个基类方法里未调用super(...).m()会导致MRO中其后的基类方法都得不到调用，不仅仅是该类的基类。

Super Considered Harmful?

- 在某个基类方法里未调用super(...).m()会导致MRO中其后的基类方法都得不到调用，不仅仅是该类的基类。
- 难以明确知道super(...).m()会调用哪个类的方法，可能会传错参数。

Super Considered Harmful?

- 在某个基类方法里未调用super(...).m()会导致MRO中其后的基类方法都得不到调用，不仅仅是该类的基类。
- 难以明确知道super(...).m()会调用哪个类的方法，可能会传错参数。
- http://fuhm.net/super-harmful/ has more info

实例

实例

```
>>> c1 = C1()  # class is callable
```

实例

```
>>> c1 = C1()  # class is callable
>>> c1.b = 'b'
>>> c1.__dict__['b']
'b'
```

实例

```
>>> c1 = C1()  # class is callable
>>> c1.b = 'b'
>>> c1.__dict__['b']
'b'
>>> c1.a
'a'
>>> c1.__dict__['a']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'a'
```

实例

```
>>> c1 = C1()  # class is callable
>>> c1.b = 'b'
>>> c1.__dict__['b']
'b'
>>> c1.a
'a'
>>> c1.__dict__['a']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'a'
>>> c1.__class__
<class '__main__.C1'>
>>> c1.__class__.a
'a'
```

bound method

bound method

```
>>> c1.f1
<bound method C1.f1 of <__main__.C1 object at 0x10055da50>>
```

bound method

```
>>> c1.f1
<bound method C1.f1 of <__main__.C1 object at 0x10055da50>>
>>> 'f1' in c1.__dict__
False
```

bound method

```
>>> c1.f1
<bound method C1.f1 of <__main__.C1 object at 0x10055da50>>
>>> 'f1' in c1.__dict__
False
>>> c1.__class__.f1
<unbound method C1.f1>
```

bound method

```
>>> c1.f1
<bound method C1.f1 of <__main__.C1 object at 0x10055da50>>
>>> 'f1' in c1.__dict__
False
>>> c1.__class__.f1
<unbound method C1.f1>
>>> isinstance(c1, c1.__class__.f1.im_class)
True
>>> c1.f1.im_class
<class '__main__.C1'>
```

bound method

```
>>> c1.f1
<bound method C1.f1 of <__main__.C1 object at 0x10055da50>>
>>> 'f1' in c1.__dict__
False
>>> c1.__class__.f1
<unbound method C1.f1>
>>> isinstance(c1, c1.__class__.f1.im_class)
True
>>> c1.f1.im_class
<class '__main__.C1'>
>>> c1.f1(c1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f1() takes exactly 1 argument (2 given)
```

bound method

```
>>> c1.f1
<bound method C1.f1 of <__main__.C1 object at 0x10055da50>>
>>> 'f1' in c1.__dict__
False
>>> c1.__class__.f1
<unbound method C1.f1>
>>> isinstance(c1, c1.__class__.f1.im_class)
True
>>> c1.f1.im_class
<class '__main__.C1'>
>>> c1.f1(c1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f1() takes exactly 1 argument (2 given)
>>> c1.f1.im_self
<__main__.C1 object at 0x10055da50>
```

bound method

```
>>> c1.f1
<bound method C1.f1 of <__main__.C1 object at 0x10055da50>>
>>> 'f1' in c1.__dict__
False
>>> c1.__class__.f1
<unbound method C1.f1>
>>> isinstance(c1, c1.__class__.f1.im_class)
True
>>> c1.f1.im_class
<class '__main__.C1'>
>>> c1.f1(c1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f1() takes exactly 1 argument (2 given)
>>> c1.f1.im_self
<__main__.C1 object at 0x10055da50>
>>> c1.f1()
'f1'
```

bound method

```
>>> c1.f1
<bound method C1.f1 of <__main__.C1 object at 0x10055da50>>
>>> 'f1' in c1.__dict__
False
>>> c1.__class__.f1
<unbound method C1.f1>
>>> isinstance(c1, c1.__class__.f1.im_class)
True
>>> c1.f1.im_class
<class '__main__.C1'>
>>> c1.f1(c1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f1() takes exactly 1 argument (2 given)
>>> c1.f1.im_self
<__main__.C1 object at 0x10055da50>
>>> c1.f1()
'f1'
>>> c1.f1.im_func(c1.f1.im_self)
'f1'
```

Add method to an instance?

Add method to an instance?

```
>>> C1.f2 = lambda self: 'f2'
>>> c1.f2()
'f2'
```

Add method to an instance?

```
>>> C1.f2 = lambda self: 'f2'
>>> c1.f2()
'f2'
>>> c1.f3 = lambda self: 'f3'
>>> c1.f3()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: <lambda>() takes exactly 1 argument (0 given)
```

Add method to an instance?

```
>>> C1.f2 = lambda self: 'f2'
>>> c1.f2()
'f2'
>>> c1.f3 = lambda self: 'f3'
>>> c1.f3()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: <lambda>() takes exactly 1 argument (0 given)
>>> c1.f3
<function <lambda> at 0x10063daa0>
```

Add method to an instance? (2)

Add method to an instance? (2)

```
>>> type(C1.f1)
<type 'instancemethod'>
```

Add method to an instance? (2)

```
>>> type(C1.f1)
<type 'instancemethod'>
>>> import types
>>> type(C1.f1) is types.MethodType
True
```

Add method to an instance? (2)

```
>>> type(C1.f1)
<type 'instancemethod'>
>>> import types
>>> type(C1.f1) is types.MethodType
True
>>> m3 = types.MethodType(f3, c1, C1)
```

Add method to an instance? (2)

```
>>> type(C1.f1)
<type 'instancemethod'>
>>> import types
>>> type(C1.f1) is types.MethodType
True
>>> m3 = types.MethodType(f3, c1, C1)
>>> m3
<bound method C1.f3 of <__main__.C1 object at 0x10055da50>>
```

Add method to an instance? (2)

```
>>> type(C1.f1)
<type 'instancemethod'>
>>> import types
>>> type(C1.f1) is types.MethodType
True
>>> m3 = types.MethodType(f3, c1, C1)
>>> m3
<bound method C1.f3 of <__main__.C1 object at 0x10055da50>>
>>> c1.f3 = m3
```

Add method to an instance? (2)

```
>>> type(C1.f1)
<type 'instancemethod'>
>>> import types
>>> type(C1.f1) is types.MethodType
True
>>> m3 = types.MethodType(f3, c1, C1)
>>> m3
<bound method C1.f3 of <__main__.C1 object at 0x10055da50>>
>>> c1.f3 = m3
>>> c1.f3()
'f3'
```

Add method to an instance? (2)

```
>>> type(C1.f1)
<type 'instancemethod'>
>>> import types
>>> type(C1.f1) is types.MethodType
True
>>> m3 = types.MethodType(f3, c1, C1)
>>> m3
<bound method C1.f3 of <__main__.C1 object at 0x10055da50>>
>>> c1.f3 = m3
>>> c1.f3()
'f3'
>>> c1.__dict__['f4']
<bound method C1.f3 of <__main__.C1 object at 0x10055da50>>
```

Add method to an instance? (2)

```
>>> type(C1.f1)
<type 'instancemethod'>
>>> import types
>>> type(C1.f1) is types.MethodType
True
>>> m3 = types.MethodType(f3, c1, C1)
>>> m3
<bound method C1.f3 of <__main__.C1 object at 0x10055da50>>
>>> c1.f3 = m3
>>> c1.f3()
'f3'
>>> c1.__dict__['f4']
<bound method C1.f3 of <__main__.C1 object at 0x10055da50>>
>>> c2.f4 = m3
>>> c2.f4.im_self
<__main__.C1 object at 0x10055da50>
```

Customizing attribute access

```
object.__getattr__(self, name)
    -> value | AttributeError
    (只在attribute lookup没有找到时才触发）

object.__setattr__(self, name, value)

object.__delattr__(self, name)
```

Example

```
class Delegator(object):
    def __init__(self, delegatee):
        self.__dict__['o'] = delegatee

    def __getattr__(self, name):
        return getattr(self.o, name)

    def __setattr__(self, name, value):
        setattr(self.o, name, value)

    def __delattr__(self, name):
        delattr(self.o, name)
```

Example

```
class Delegator(object):
    def __init__(self, delegatee):
        self.__dict__['o'] = delegatee

    def __getattr__(self, name):
        return getattr(self.o, name)

    def __setattr__(self, name, value):
        setattr(self.o, name, value)

    def __delattr__(self, name):
        delattr(self.o, name)
```

```
>>> d = Delegator(c1)
>>> c1.a = 2
>>> d.a
2
```

Example

```
class Delegator(object):
    def __init__(self, delegatee):
        self.__dict__['o'] = delegatee

    def __getattr__(self, name):
        return getattr(self.o, name)

    def __setattr__(self, name, value):
        setattr(self.o, name, value)

    def __delattr__(self, name):
        delattr(self.o, name)
```

```
>>> d = Delegator(c1)
>>> c1.a = 2
>>> d.a
2
>>> c1.o = 3
>>> d.o
<__main__.C1 object at 0x10055da50>
```

Private Attribute

```
class Delegator(object):
    def __init__(self, delegatee):
        self.__o = delegatee

    def __getattr__(self, name):
        return getattr(self.__o, name)
```

Private Attribute

```
class Delegator(object):
    def __init__(self, delegatee):
        self.__o = delegatee

    def __getattr__(self, name):
        return getattr(self.__o, name)
```

```
>>> d = Delegator(c1)
>>> c1.__o = 2
>>> d.__o
2
```

Private Attribute

```
class Delegator(object):
    def __init__(self, delegatee):
        self.__o = delegatee

    def __getattr__(self, name):
        return getattr(self.__o, name)
```

```
>>> d = Delegator(c1)
>>> c1.__o = 2
>>> d.__o
2
>>> c._Delegator__o = 3
>>> d._Delegator__o
<__main__.C1 object at 0x10055da50>
```

更猛一点的

```
object.__getattribute__(self, name)
    -> value | AttributeError
```

- called unconditionally
- if AttributeError raised, __getattr__() is checked
- To avoid inifinite recursion, use BaseClass.__getattribute__(self, name) to access any attributes it needs

Example

```
class Delegator(object):
    def __init__(self, delegatee):
        super(Delegator, self).__setattr__('o', delegatee)

    def __getattribute__(self, name):
        o = super(Delegator, self).__getattribute__('o')
        return getattr(o, name)

    def __setattr__(self, name, value):
        o = super(Delegator, self).__getattribute__('o')
        setattr(o, name, value)

    def __delattr__(self, name):
        o = super(Delegator, self).__getattribute__('o')
        delattr(o, name)
```

Example

```
class Delegator(object):
    def __init__(self, delegatee):
        super(Delegator, self).__setattr__('o', delegatee)

    def __getattribute__(self, name):
        o = super(Delegator, self).__getattribute__('o')
        return getattr(o, name)

    def __setattr__(self, name, value):
        o = super(Delegator, self).__getattribute__('o')
        setattr(o, name, value)

    def __delattr__(self, name):
        o = super(Delegator, self).__getattribute__('o')
        delattr(o, name)
```

```
>>> d = Delegator(c1)
>>> c1.o = 2
>>> d.o
2
```

Customize a Specific Attribute Access

- Descriptor

Descriptor

Descriptor

- A new style object containing the following methods:

```
__get__(self, instance, owner_class)
__set__(self, instance, value)
__delete__(self, instance)
```

Descriptor

- A new style object containing the following methods:

```
__get__(self, instance, owner_class)
__set__(self, instance, value)
__delete__(self, instance)
```

- This object appears in the class dictionary of another new-style class (owner class)

Descriptor

- A new style object containing the following methods:

```
__get__(self, instance, owner_class)
__set__(self, instance, value)
__delete__(self, instance)
```

- This object appears in the class dictionary of another new-style class (owner class)
- The customized method will be called when the owner class (or its instances) accesses the object as attribute

Example

```
>>> class Descriptor(object):
...     def __get__(self, instance, owner):
...         return 'd', instance, owner
...
>>> class Owner(object):
...     descriptor = Descriptor()
...
>>> Owner.descriptor
('d', None, <class '__main__.Owner'>)
>>> owner = Owner()
>>> owner.descriptor
('d', <__main__.Owner object at 0x10055d990>, <class
'__main__.Owner'>)
```

Real World Example

```
# luzong/mixin/props.py

class PropsItem(object):
    def __init__(self, name, default=None, output_filter=None):
        ...

    def __get__(self, obj, objtype):
        r = obj.get_props_item(self.name)
        if r is None:
            return self.default
        else:
            return self.output_filter(r)
        else:
            return r

    def __set__(self, obj, value):
        obj.set_props_item(self.name, value)


# luzong/widgets/poll.py

class PollWidget(CommentPermissionMixin, Widget):
    choice_text_length = PropsItem('choice_text_length', '50')
```

豆瓣 douban

Data/Non-data Descriptor

Data/Non-data Descriptor

- Data Descriptor:

```
  has __set__() or __delete__()
  not overridable by instance
  e.g. property
```

Data/Non-data Descriptor

- Data Descriptor:
  has __set__() or __delete__()
  not overridable by instance
  e.g. property

- Non-Data Descriptor:
  has __get__() only
  overridable by instance
  e.g. function, staticmethod, classmethod

Exercise

```
Implement property()

class C(object):
    def get_x(self):
        return 1
    x = MyProperty(get_x)

c = C()
c.x  => 1
```

Answer

```
class MyProperty(object):
    def __init__(self, fget=None, fset=None, fdel=None, doc=None):
        self.fget, self.fset, self.fdel, self.__doc__ = \
                 fget, fset, fdel, doc

    def __get__(self, obj, objtype):
        if obj is None:
            return self
        if self.fget is None:
            raise AttributeError
        return self.fget(obj)

    def __set__(self, obj, value):
        if self.fset is None:
            raise AttributeError
        self.fset(obj, value)

    def __delete__(self, obj):
        if self.fdel is None:
            raise AttributeError
        self.fdel(obj)
```

Builtin Non-Data Descriptors

Descriptor	Called from an Object	Called from a Class
function	f(obj, *args)	f(*args)
staticmethod	f(*args)	f(*args)
classmethod	f(type(obj), *args)	f(klass, *args)

Builtin Non-Data Descriptors

Descriptor	Called from an Object	Called from a Class
function	f(obj, *args)	f(*args)
staticmethod	f(*args)	f(*args)
classmethod	f(type(obj), *args)	f(klass, *args)

```
class C(object):
    @staticmethod
    def f(x):
        return x

    @classmethod
    def g(cls, x):
        return x*2
```

Builtin Non-Data Descriptors

Descriptor	Called from an Object	Called from a Class
function	f(obj, *args)	f(*args)
staticmethod	f(*args)	f(*args)
classmethod	f(type(obj), *args)	f(klass, *args)

```
class C(object):
    @staticmethod
    def f(x):
        return x

    @classmethod
    def g(cls, x):
        return x*2
```

```
>>> C.f(1)
1
>>> C().f(1)
1
>>> C.g(1)
2
>>> C().g(1)
2
```

属性访问小结

属性访问小结

1. __getattribute__

属性访问小结

1. __getattribute__
2. data descriptor

属性访问小结

1. __getattribute__
2. data descriptor
3. __dict__

属性访问小结

1. __getattribute__
2. data descriptor
3. __dict__
4. for C in o.__bases__ + type(o).__mro__:

   check C.__dict__ and apply descriptor

属性访问小结

1. __getattribute__
2. data descriptor
3. __dict__
4. for C in o.__bases__ + type(o).__mro__:

   check C.__dict__ and apply descriptor
5. __getattr__

属性访问小结

1. __getattribute__
2. data descriptor
3. __dict__
4. for C in o.__bases__ + type(o).__mro__:

    check C.__dict__ and apply descriptor

5. __getattr__
6. raise AttributeError

Create a Instance

Create a Instance

- __new__
  - 创建实例

Create a Instance

- __new__
  - 创建实例
- __init__
  - 初始化实例

__new__

__new__

```
x = C(*args, **kwargs)
```

__new__

```
x = C(*args, **kwargs)
```

等价于

```
x = C.__new__(C, *args, **kwargs)
if isinstance(x, C)
    type(x).__init__(x, *args, **kwargs)
```

__new__

```
x = C(*args, **kwargs)
等价于
x = C.__new__(C, *args, **kwargs)
if isinstance(x, C)
    type(x).__init__(x, *args, **kwargs)
```

```
>>> class C6(object):
...   def __new__(cls, *a, **kw):
...     return 1
...
>>> C6()
1
```

Singleton Pattern

```
class Singleton(object):
    _singletons = {}

    def __new__(cls, *args, **kwargs):
        if cls not in cls._singletons:
            cls._singletons[cls] = \
                super(Singleton, cls).__new__(
                           cls, *args, **kwargs)
        return cls._singletons[cls]
```

Meta Class

```
classes : objects = ? : classes

type(1) is int

type(int) is type is types.TypeType

class C1(object): pass
type(C1) is type is types.TypeType

class C2: pass
type(C2) is types.ClassType
```

Let's make some confusion

Let's make some confusion

```
>>> type(object) is type
```

Let's make some confusion

```
>>> type(object) is type
True
```

Let's make some confusion

```
>>> type(object) is type
True
>>> type(type) is type
```

Let's make some confusion

```
>>> type(object) is type
True
>>> type(type) is type
True
```

Let's make some confusion

```
>>> type(object) is type
True
>>> type(type) is type
True
>>> isinstance(type, object)
```

Let's make some confusion

```
>>> type(object) is type
True
>>> type(type) is type
True
>>> isinstance(type, object)
True
```

Let's make some confusion

```
>>> type(object) is type
True
>>> type(type) is type
True
>>> isinstance(type, object)
True
>>> isinstance(object, type)
```

Let's make some confusion

```
>>> type(object) is type
True
>>> type(type) is type
True
>>> isinstance(type, object)
True
>>> isinstance(object, type)
True
```

Let's make some confusion

```
>>> type(object) is type
True
>>> type(type) is type
True
>>> isinstance(type, object)
True
>>> isinstance(object, type)
True
>>> isinstance(type, type)
```

Let's make some confusion

```
>>> type(object) is type
True
>>> type(type) is type
True
>>> isinstance(type, object)
True
>>> isinstance(object, type)
True
>>> isinstance(type, type)
True
```

Let's make some confusion

```
>>> type(object) is type
True
>>> type(type) is type
True
>>> isinstance(type, object)
True
>>> isinstance(object, type)
True
>>> isinstance(type, type)
True
>>> isinstance(object, object)
```

Let's make some confusion

```
>>> type(object) is type
True
>>> type(type) is type
True
>>> isinstance(type, object)
True
>>> isinstance(object, type)
True
>>> isinstance(type, type)
True
>>> isinstance(object, object)
True
```

Let's make some confusion

```
>>> type(object) is type
True
>>> type(type) is type
True
>>> isinstance(type, object)
True
>>> isinstance(object, type)
True
>>> isinstance(type, type)
True
>>> isinstance(object, object)
True
>>> issubclass(type, object)
```

Let's make some confusion

```
>>> type(object) is type
True
>>> type(type) is type
True
>>> isinstance(type, object)
True
>>> isinstance(object, type)
True
>>> isinstance(type, type)
True
>>> isinstance(object, object)
True
>>> issubclass(type, object)
True
```

Let's make some confusion

```
>>> type(object) is type
True
>>> type(type) is type
True
>>> isinstance(type, object)
True
>>> isinstance(object, type)
True
>>> isinstance(type, type)
True
>>> isinstance(object, object)
True
>>> issubclass(type, object)
True
>>> issubclass(object, type)
```

Let's make some confusion

```
>>> type(object) is type
True
>>> type(type) is type
True
>>> isinstance(type, object)
True
>>> isinstance(object, type)
True
>>> isinstance(type, type)
True
>>> isinstance(object, object)
True
>>> issubclass(type, object)
True
>>> issubclass(object, type)
False
```

Create a class

Create a class

```
type(name, bases, attribs):
     name: string, C.__name__
    bases: tuple, C.__bases__
  attribs: dict, C.__dict__
```

Create a class

```
type(name, bases, attribs):
     name: string, C.__name__
    bases: tuple, C.__bases__
  attribs: dict, C.__dict__

class C(object): pass
```

Create a class

```
type(name, bases, attribs):
     name: string, C.__name__
    bases: tuple, C.__bases__
  attribs: dict, C.__dict__

class C(object): pass
==>
C = type('C', (object,), {})
```

Inside class statement

Inside class statement

```
>>> class C(object):
...     a = 1
...     def __init__(self, b):
...         self.b = b
```

Inside class statement

```
>>> class C(object):
...     a = 1
...     def __init__(self, b):
...         self.b = b
==
>>> d = {}
>>> exec """
... a = 1
... def __init__(self, b):
...     self.b = b
... """ in globals(), d
>>> C = type('C', (object,), d)
```

Customized Metaclass

Customized Metaclass

```
>>> class Ugly(object): pass
```

Customized Metaclass

```
>>> class Ugly(object): pass
>>> Ugly
<class '__main__.Ugly'>
```

Customized Metaclass

```
>>> class Ugly(object): pass
>>> Ugly
<class '__main__.Ugly'>
>>> class MetaPretty(type):
...     def __repr__(cls):
...         return "I'm the class %s" % cls.__name__
...
```

Customized Metaclass

```
>>> class Ugly(object): pass
>>> Ugly
<class '__main__.Ugly'>
>>> class MetaPretty(type):
...     def __repr__(cls):
...         return "I'm the class %s" % cls.__name__
...
>>> class Pretty(object):
...     __metaclass__ = MetaPretty
...
```

Customized Metaclass

```
>>> class Ugly(object): pass
>>> Ugly
<class '__main__.Ugly'>
>>> class MetaPretty(type):
...     def __repr__(cls):
...         return "I'm the class %s" % cls.__name__
...
>>> class Pretty(object):
...     __metaclass__ = MetaPretty
...
>>> Pretty
I'm the class Pretty
```

Customized Metaclass

```
>>> class Ugly(object): pass
>>> Ugly
<class '__main__.Ugly'>
>>> class MetaPretty(type):
...     def __repr__(cls):
...         return "I'm the class %s" % cls.__name__
...
>>> class Pretty(object):
...     __metaclass__ = MetaPretty
...
>>> Pretty
I'm the class Pretty
>>> type(Pretty)
<class '__main__.MetaPretty'>
```

Real-world Example

```
# luzong/base_page.py
class MetaObserver(type):
    kind_map = {}
    name_map = {}
    def __init__(mcs, name, bases, attrs):
        if attrs.get('kind'):
            mcs.add_to_map(attrs['kind'])
        elif 'kinds' in attrs:
            for kind in attrs['kinds']:
                mcs.add_to_map(kind)
    def add_to_map(mcs, kind):
        mcs.kind_map[kind] = mcs
        if mcs.kind_name:
            mcs.name_map[mcs.kind_name] = mcs
        else:
            for kind_name in mcs.kind_names:
                 mcs.name_map[kind_name] = mcs
```

```
class MetaPage(object):
    __metaclass__ = MetaObserver
    kind = None
    kinds = []

# luzong/vote/__init__.py
class Vote(Commentable, MetaPage):
    kind = K_VOTE
    kind_name = 'vote'
```

Better Singleton

```
class Singleton(object):
    _singletons = {}
    class __metaclass__(type):
        def __call__(cls, *args, **kwargs):
            S = Singleton
            if cls not in S._singletons:
                super_ = super(S.__metaclass__, cls)
                S._singletons[cls] = super_.__call__(*args, **kwargs)
            return S._singletons[cls]
```

Enforcing Naming Rules

```
class MetaEnsureAttribNames(type):
    def __new__(mcl, name, bases, attrs):
        if any(a for a in attrs if not a.islower()):
            raise TypeError("invalid attribute")
        return super(MetaEnsureAttribNames,
                mcl).__new__(mcl, name, bases, attrs)
```

Combine Two Metaclasses

Combine Two Metaclasses

```
>>> class M1(type): pass
...
```

Combine Two Metaclasses

```
>>> class M1(type): pass
...
>>> class M2(type): pass
...
```

Combine Two Metaclasses

```
>>> class M1(type): pass
...
>>> class M2(type): pass
...
>>> class B1(object): __metaclass__ = M1
...
```

Combine Two Metaclasses

```
>>> class M1(type): pass
...
>>> class M2(type): pass
...
>>> class B1(object): __metaclass__ = M1
...
>>> class B2(object): __metaclass__ = M2
...
```

Combine Two Metaclasses

```
>>> class M1(type): pass
...
>>> class M2(type): pass
...
>>> class B1(object): __metaclass__ = M1
...
>>> class B2(object): __metaclass__ = M2
...
>>> class C(B1, B2): pass
...
```

Combine Two Metaclasses

```
>>> class M1(type): pass
...
>>> class M2(type): pass
...
>>> class B1(object): __metaclass__ = M1
...
>>> class B2(object): __metaclass__ = M2
...
>>> class C(B1, B2): pass
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Error when calling the metaclass bases
    metaclass conflict: the metaclass of a derived class must be a (non-
strict) subclass of the metaclasses of all its bases
```

Combine Two Metaclasses

```
>>> class M1(type): pass
...
>>> class M2(type): pass
...
>>> class B1(object): __metaclass__ = M1
...
>>> class B2(object): __metaclass__ = M2
...
>>> class C(B1, B2): pass
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Error when calling the metaclass bases
    metaclass conflict: the metaclass of a derived class must be a (non-
strict) subclass of the metaclasses of all its bases
>>> class M3(M1, M2): pass
...
```

Combine Two Metaclasses

```
>>> class M1(type): pass
...
>>> class M2(type): pass
...
>>> class B1(object): __metaclass__ = M1
...
>>> class B2(object): __metaclass__ = M2
...
>>> class C(B1, B2): pass
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Error when calling the metaclass bases
    metaclass conflict: the metaclass of a derived class must be a (non-
strict) subclass of the metaclasses of all its bases
>>> class M3(M1, M2): pass
...
>>> class C(B1, B2): __metaclass__ = M3
...
```

Dangerous!

[Metaclasses] are deeper magic than 99% of users should ever worry about. If you wonder whether you need them, you don't (the people who actually need them know with certainty that they need them, and don't need an explanation about why).

- Tim Peters @ c.l.python

Decorator

Decorator is just a syntax sugar

Decorator is just a syntax sugar

```
class C(object):
    def f():
        return 1
    f = staticmethod(f)
```

Decorator is just a syntax sugar

```
class C(object):
    def f():
        return 1
    f = staticmethod(f)

=>

class C(object):
    @staticmethod
    def f():
        return 1
```

Decorator is often to decorating functions

```
>>> def mydeco(func):
...     def _(*args, **kwargs):
...         print "before"
...         retval = func(*args, **kwargs)
...         print "after"
...         return retval
...     return _
...
>>> @mydeco
... def f():
...     print "in f()"
...     return 1
...
>>> f()
before
in f()
after
1
```

...but sometimes not

```
>>> class C(object):
...     @property
...     def a(self):
...         return 1
...
>>> C().a
1
```

...but sometimes not

```
>>> class C(object):
...     @property
...     def a(self):
...         return 1
...
>>> C().a
1
```

```
>>> @apply
... def pi():
...     import math
...     return math.pi
...
>>> pi
3.141592653589793
```

Decorator with Parameters

```
def mydeco(s):
    def deco(func):
        def _(*args, **kwargs):
            print "this function is decorated by %s" % s
            return func(*args, **kwargs)
        return _
    return deco

@mydeco("decorator")
def f():
    return 1
```

callable object as decorator

```
class MyDeco(object):
    def __init__(self, s):
        self.s = s

    def __call__(self, func):
        def _(*args, **kwargs):
            print "this function is decorated by %s" % self.s
            return func(*args, **kwargs)
        return _

@MyDeco("decorator")
def f():
    return 1
```

multiple decorator

```
@deco1
@deco2
def f():
    pass

f = deco1(deco2(f))
```

Some useful decorators

- @require_login
- @check_permission("edit")
- @cache("user:{uid}")
- @in_transaction
- @async
- @validate_form

functools.wraps

```
>>> from functools import wraps
>>> def my_decorator(f):
...     @wraps(f)
...     def wrapper(*args, **kwargs):
...         print "Calling decorated function"
...         return f(*args, **kwargs)
...     return wrapper
...
>>> @my_decorator
... def example():
...     """Docstring"""
...     print "Called example function"
...
>>> example.__name__
'example'
>>> example.__doc__
'Docstring'
```

Class Decorator

```
class A(object):
    pass
f = foo(bar(A))

@foo
@bar
class A(object):
    pass
```

can do something like in metaclass' __init__

Class Decorator Example

```
def register(kind, kind_name):
    def deco(cls):
        register.registry[kind] = cls
        register.registry[kind_name] = cls
        return cls
    return deco
register.registry = {}

@register(kind=K_VOTE, kind_name='vote')
class Vote(Commentable):
    pass
```

Exercise

Implement @property

```
class C(object):
    @property
    def x(self):
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x
```

Answer

```
class MyProperty(object):
    def __init__(self, fget=None, fset=None, fdel=None,
doc=None):
        ...

    def setter(self, func):
        self.fset = func

    def deleter(self, func):
        self.fdel = func

    def __get__(self, objtype, obj):
        ...
    ....
```

Generator

Iteration

```
>>> for x in [1, 4, 5, 10]:
...     print x,
...
1 4 5 10
```

Iterables

Iterables

- list/tuple : an item per iteration

Iterables

- list/tuple : an item per iteration
- dict: a key per iteration

Iterables

- list/tuple : an item per iteration
- dict: a key per iteration
- string: a character per iteration

Iterables

- list/tuple : an item per iteration
- dict: a key per iteration
- string: a character per iteration
- file: a line per iteration

Iteration Protocol

Iteration Protocol

```
for x in iterable:
    ...
```

Iteration Protocol

```
for x in iterable:
    ...

it = obj.__iter__()
while True:
    try:
        x = it.next()
    except StopIteration:
        break
    ....
```

User-defined Iterables

```
class XRange(object):
    def __init__(self, start=0, stop=10):
        self.count = start
    def __iter__(self):
        return self
    def next(self):
        if self.count >= stop:
            raise StopIteration
        c = self.count
        self.count += 1
        return c

>>> for x in XRange(1, 5):
...    print x,
1 2 3 4
```

Generators

```
def XRange(start=0, stop=10):
    count = start
    while count < stop:
        yield count
        count += 1

>>> for i in Xrange(1, 5):
...     print i,
1 2 3 4
```

Iteration over Generators

Iteration over Generators

```
def XRange(start=0, stop=10):
    print "Start executing XRange()"
    while start < stop:
        yield start
        start += 1
```

Iteration over Generators

```
def XRange(start=0, stop=10):
    print "Start executing XRange()"
    while start < stop:
        yield start
        start += 1

>>> g = XRange(1, 4)
```

Iteration over Generators

```
def XRange(start=0, stop=10):
    print "Start executing XRange()"
    while start < stop:
        yield start
        start += 1

>>> g = XRange(1, 4)
>>> g
<generator object XRange at 0x1005594b0>
```

Iteration over Generators

```
def XRange(start=0, stop=10):
    print "Start executing XRange()"
    while start < stop:
        yield start
        start += 1

>>> g = XRange(1, 4)
>>> g
<generator object XRange at 0x1005594b0>
>>> g.next()
Start executing XRange()
1    (function suspended)
```

Iteration over Generators

```
def XRange(start=0, stop=10):
    print "Start executing XRange()"
    while start < stop:
        yield start
        start += 1
```

```
>>> g = XRange(1, 4)
>>> g
<generator object XRange at 0x1005594b0>
>>> g.next()
Start executing XRange()
1   (function suspended)
>>> g.next()
2   (function resumed)
```

Iteration over Generators

```
def XRange(start=0, stop=10):
    print "Start executing XRange()"
    while start < stop:
        yield start
        start += 1

>>> g = XRange(1, 4)
>>> g
<generator object XRange at 0x1005594b0>
>>> g.next()
Start executing XRange()
1   (function suspended)
>>> g.next()
2   (function resumed)
>>> g.next()
3
```

Iteration over Generators

```
def XRange(start=0, stop=10):
    print "Start executing XRange()"
    while start < stop:
        yield start
        start += 1

>>> g = XRange(1, 4)
>>> g
<generator object XRange at 0x1005594b0>
>>> g.next()
Start executing XRange()
1   (function suspended)
>>> g.next()
2   (function resumed)
>>> g.next()
3
>>> g.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Infinite Generator

```
def XRange(start=0, stop=None):
    while stop is None or start < stop:
        yield start
        start += 1

>>> for i in XRange(1):
...     print i
1
2
3
4
.
.
.
```

Generator Expression

```
lst = [1, 2, 3, 4]

# list comprehension
[x**2 for x in lst if x>2]
=> [9, 16]

# generator expression
(x**2 for x in lst if x>2)
=> a generator which yields 9, and then 16
```

Example: Calculate the Data Transferred

```
wwwlog = open("access-log")
total = 0
for line in wwwlog:
    bytestr = line.split()[-1]
    if bytestr != '-':
        total += int(bytestr)

print "Total bytes:", total
```

Generator as Pipe

```
wwwlog = open("access-log")
bytecolumn = (line.split()[-1] for line in wwwlog)
bytes = (int(x) for x in bytecolumn if x != '-')

print "Total bytes:", sum(bytes)
```

some useful functions

some useful functions

- enumerate(iterable[, start]) -> (start, p1), (start+1, p2), ...

some useful functions

- enumerate(iterable[, start]) -> (start, p1), (start+1, p2), ...
- xrange([start], stop[, step])

some useful functions

- enumerate(iterable[, start]) -> (start, p1), (start+1, p2), ...
- xrange([start], stop[, step])
- iter(collection) -> collection.__iter__()

some useful functions

- enumerate(iterable[, start]) -> (start, p1), (start+1, p2), ...
- xrange([start], stop[, step])
- iter(collection) -> collection.__iter__()
- iter(callable, sentinel) -> call callable until it returns sentinel

some useful functions

- enumerate(iterable[, start]) -> (start, p1), (start+1, p2), ...
- xrange([start], stop[, step])
- iter(collection) -> collection.__iter__()
- iter(callable, sentinel) -> call callable until it returns sentinel
- dict.iteritems() -> (k1, v1), (k2, v2), ...

... and itertools module

... and itertools module

```
count([n]) --> n, n+1, n+2, ...
```

... and itertools module

```
count([n]) --> n, n+1, n+2, ...
cycle(p) --> p0, p1, ... plast, p0, p1, ...
```

... and itertools module

```
count([n]) --> n, n+1, n+2, ...
cycle(p) --> p0, p1, ... plast, p0, p1, ...
repeat(elem [,n]) --> elem, elem, elem, ... endlessly or up to n times
```

... and itertools module

```
count([n]) --> n, n+1, n+2, ...
cycle(p) --> p0, p1, ... plast, p0, p1, ...
repeat(elem [,n]) --> elem, elem, elem, ... endlessly or up to n times
izip(p, q, ...) --> (p[0], q[0]), (p[1], q[1]), ...
```

... and itertools module

```
count([n]) --> n, n+1, n+2, ...
cycle(p) --> p0, p1, ... plast, p0, p1, ...
repeat(elem [,n]) --> elem, elem, elem, ... endlessly or up to n times
izip(p, q, ...) --> (p[0], q[0]), (p[1], q[1]), ...
izip_longest(p, q, ...) --> (p[0], q[0]), (p[1], q[1]), ...
```

... and itertools module

```
count([n]) --> n, n+1, n+2, ...
cycle(p) --> p0, p1, ... plast, p0, p1, ...
repeat(elem [,n]) --> elem, elem, elem, ... endlessly or up to n times
izip(p, q, ...) --> (p[0], q[0]), (p[1], q[1]), ...
izip_longest(p, q, ...) --> (p[0], q[0]), (p[1], q[1]), ...
ifilter(pred, seq) --> elements of seq where pred(elem) is True
```

... and itertools module

```
count([n]) --> n, n+1, n+2, ...
cycle(p) --> p0, p1, ... plast, p0, p1, ...
repeat(elem [,n]) --> elem, elem, elem, ... endlessly or up to n times
izip(p, q, ...) --> (p[0], q[0]), (p[1], q[1]), ...
izip_longest(p, q, ...) --> (p[0], q[0]), (p[1], q[1]), ...
ifilter(pred, seq) --> elements of seq where pred(elem) is True
ifilterfalse(pred, seq) --> elements of seq where pred(elem) is False
```

... and itertools module

```
count([n]) --> n, n+1, n+2, ...
cycle(p) --> p0, p1, ... plast, p0, p1, ...
repeat(elem [,n]) --> elem, elem, elem, ... endlessly or up to n times
izip(p, q, ...) --> (p[0], q[0]), (p[1], q[1]), ...
izip_longest(p, q, ...) --> (p[0], q[0]), (p[1], q[1]), ...
ifilter(pred, seq) --> elements of seq where pred(elem) is True
ifilterfalse(pred, seq) --> elements of seq where pred(elem) is False
islice(seq, [start,] stop [, step]) --> elements from seq[start:stop:step]
```

... and itertools module

```
count([n]) --> n, n+1, n+2, ...
cycle(p) --> p0, p1, ... plast, p0, p1, ...
repeat(elem [,n]) --> elem, elem, elem, ... endlessly or up to n times
izip(p, q, ...) --> (p[0], q[0]), (p[1], q[1]), ...
izip_longest(p, q, ...) --> (p[0], q[0]), (p[1], q[1]), ...
ifilter(pred, seq) --> elements of seq where pred(elem) is True
ifilterfalse(pred, seq) --> elements of seq where pred(elem) is False
islice(seq, [start,] stop [, step]) --> elements from seq[start:stop:step]
imap(fun, p, q, ...) --> fun(p0, q0), fun(p1, q1), ...
```

... and itertools module

```
count([n]) --> n, n+1, n+2, ...
cycle(p) --> p0, p1, ... plast, p0, p1, ...
repeat(elem [,n]) --> elem, elem, elem, ... endlessly or up to n times
izip(p, q, ...) --> (p[0], q[0]), (p[1], q[1]), ...
izip_longest(p, q, ...) --> (p[0], q[0]), (p[1], q[1]), ...
ifilter(pred, seq) --> elements of seq where pred(elem) is True
ifilterfalse(pred, seq) --> elements of seq where pred(elem) is False
islice(seq, [start,] stop [, step]) --> elements from seq[start:stop:step]
imap(fun, p, q, ...) --> fun(p0, q0), fun(p1, q1), ...
starmap(fun, seq) --> fun(*seq[0]), fun(*seq[1]), ...
```

... and itertools module

```
count([n]) --> n, n+1, n+2, ...
cycle(p) --> p0, p1, ... plast, p0, p1, ...
repeat(elem [,n]) --> elem, elem, elem, ... endlessly or up to n times
izip(p, q, ...) --> (p[0], q[0]), (p[1], q[1]), ...
izip_longest(p, q, ...) --> (p[0], q[0]), (p[1], q[1]), ...
ifilter(pred, seq) --> elements of seq where pred(elem) is True
ifilterfalse(pred, seq) --> elements of seq where pred(elem) is False
islice(seq, [start,] stop [, step]) --> elements from seq[start:stop:step]
imap(fun, p, q, ...) --> fun(p0, q0), fun(p1, q1), ...
starmap(fun, seq) --> fun(*seq[0]), fun(*seq[1]), ...
tee(it, n=2) --> (it1, it2 , ... itn) splits one iterator into n
```

... and itertools module

```
count([n]) --> n, n+1, n+2, ...
cycle(p) --> p0, p1, ... plast, p0, p1, ...
repeat(elem [,n]) --> elem, elem, elem, ... endlessly or up to n times
izip(p, q, ...) --> (p[0], q[0]), (p[1], q[1]), ...
izip_longest(p, q, ...) --> (p[0], q[0]), (p[1], q[1]), ...
ifilter(pred, seq) --> elements of seq where pred(elem) is True
ifilterfalse(pred, seq) --> elements of seq where pred(elem) is False
islice(seq, [start,] stop [, step]) --> elements from seq[start:stop:step]
imap(fun, p, q, ...) --> fun(p0, q0), fun(p1, q1), ...
starmap(fun, seq) --> fun(*seq[0]), fun(*seq[1]), ...
tee(it, n=2) --> (it1, it2 , ... itn) splits one iterator into n
chain(p, q, ...) --> p0, p1, ... plast, q0, q1, ...
```

... and itertools module

```
count([n]) --> n, n+1, n+2, ...
cycle(p) --> p0, p1, ... plast, p0, p1, ...
repeat(elem [,n]) --> elem, elem, elem, ... endlessly or up to n times
izip(p, q, ...) --> (p[0], q[0]), (p[1], q[1]), ...
izip_longest(p, q, ...) --> (p[0], q[0]), (p[1], q[1]), ...
ifilter(pred, seq) --> elements of seq where pred(elem) is True
ifilterfalse(pred, seq) --> elements of seq where pred(elem) is False
islice(seq, [start,] stop [, step]) --> elements from seq[start:stop:step]
imap(fun, p, q, ...) --> fun(p0, q0), fun(p1, q1), ...
starmap(fun, seq) --> fun(*seq[0]), fun(*seq[1]), ...
tee(it, n=2) --> (it1, it2 , ... itn) splits one iterator into n
chain(p, q, ...) --> p0, p1, ... plast, q0, q1, ...
takewhile(pred, seq) --> seq[0], seq[1], until pred fails
```

... and itertools module

```
count([n]) --> n, n+1, n+2, ...
cycle(p) --> p0, p1, ... plast, p0, p1, ...
repeat(elem [,n]) --> elem, elem, elem, ... endlessly or up to n times
izip(p, q, ...) --> (p[0], q[0]), (p[1], q[1]), ...
izip_longest(p, q, ...) --> (p[0], q[0]), (p[1], q[1]), ...
ifilter(pred, seq) --> elements of seq where pred(elem) is True
ifilterfalse(pred, seq) --> elements of seq where pred(elem) is False
islice(seq, [start,] stop [, step]) --> elements from seq[start:stop:step]
imap(fun, p, q, ...) --> fun(p0, q0), fun(p1, q1), ...
starmap(fun, seq) --> fun(*seq[0]), fun(*seq[1]), ...
tee(it, n=2) --> (it1, it2 , ... itn) splits one iterator into n
chain(p, q, ...) --> p0, p1, ... plast, q0, q1, ...
takewhile(pred, seq) --> seq[0], seq[1], until pred fails
dropwhile(pred, seq) --> seq[n], seq[n+1], starting when pred fails
```

... and itertools module

```
count([n]) --> n, n+1, n+2, ...
cycle(p) --> p0, p1, ... plast, p0, p1, ...
repeat(elem [,n]) --> elem, elem, elem, ... endlessly or up to n times
izip(p, q, ...) --> (p[0], q[0]), (p[1], q[1]), ...
izip_longest(p, q, ...) --> (p[0], q[0]), (p[1], q[1]), ...
ifilter(pred, seq) --> elements of seq where pred(elem) is True
ifilterfalse(pred, seq) --> elements of seq where pred(elem) is False
islice(seq, [start,] stop [, step]) --> elements from seq[start:stop:step]
imap(fun, p, q, ...) --> fun(p0, q0), fun(p1, q1), ...
starmap(fun, seq) --> fun(*seq[0]), fun(*seq[1]), ...
tee(it, n=2) --> (it1, it2 , ... itn) splits one iterator into n
chain(p, q, ...) --> p0, p1, ... plast, q0, q1, ...
takewhile(pred, seq) --> seq[0], seq[1], until pred fails
dropwhile(pred, seq) --> seq[n], seq[n+1], starting when pred fails
groupby(iterable[, keyfunc]) --> sub-iterators grouped by value of keyfunc(v)
```

Coroutine

```
def grep(pattern):
    print "Looking for %s" % pattern
    while True:
        line = (yield)
        if pattern in line:
            print line

>>> g = grep("python")
>>> g.next()
Looking for python
>>> g.send("blah blah blah")
>>> g.send("python generators rock!")
python generators rock!
```

Closing a Coroutine

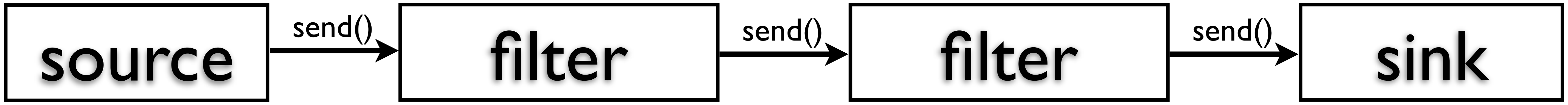
```
def grep(pattern):
    try:
        while True:
            line = (yield)
            if pattern in line:
                print line
    except GeneratorExit:
        print "Goodbye"

>>> g = grep("python")
>>> g.next()
>>> g.send("python generator rocks!")
python generator rocks!
>>> g.close()
Goodbye
```

send() returns what yielded out

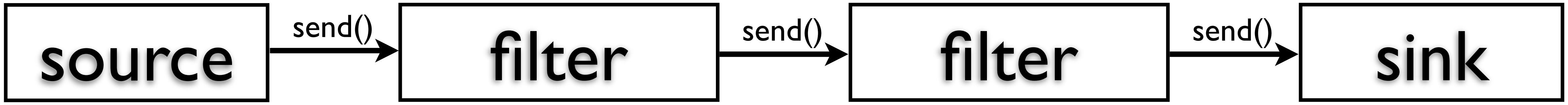
```
def grep(pattern):
    r = None
    while True:
        line = (yield r)
        r = line if pattern in line else None

>>> g = grep("python")
>>> g.next()
>>> g.send("blah blah")
>>> g.send("python generator rocks!")
'python generator rocks!'
```

Coroutine Pipelines

Coroutine Pipelines



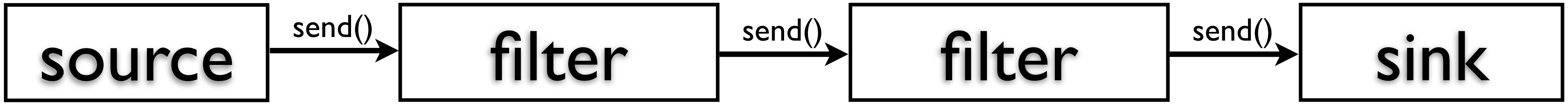
Coroutine Pipelines



source send() filter send() filter send() sink

```
def source(target):
    target.next()
    target.send(data1)
    target.send(data2)
    ...
    target.close()
```

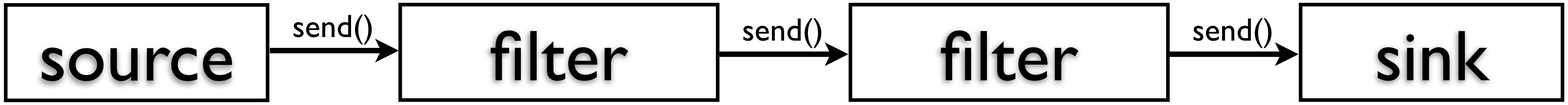
Coroutine Pipelines



```
def source(target):
    target.next()
    target.send(data1)
    target.send(data2)
    ...
    target.close()
```

```
def filter(target):
    target.next()
    try:
        while True:
            data = (yield)
            data = process(data)
            target.send(data)
    except GeneratorExit:
        target.close()
```

Coroutine Pipelines



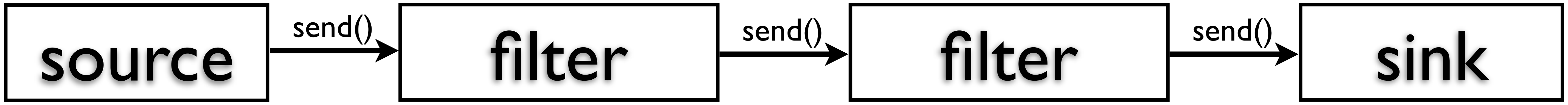
source send() filter send() filter send() sink

```
def source(target):
    target.next()
    target.send(data1)
    target.send(data2)
    ...
    target.close()
```

```
def filter(target):
    target.next()
    try:
        while True:
            data = (yield)
            data = process(data)
            target.send(data)
    except GeneratorExit:
        target.close()
```

```
def sink():
    while True:
        data = (yield)
        data = process(data)
```

Coroutine Pipelines



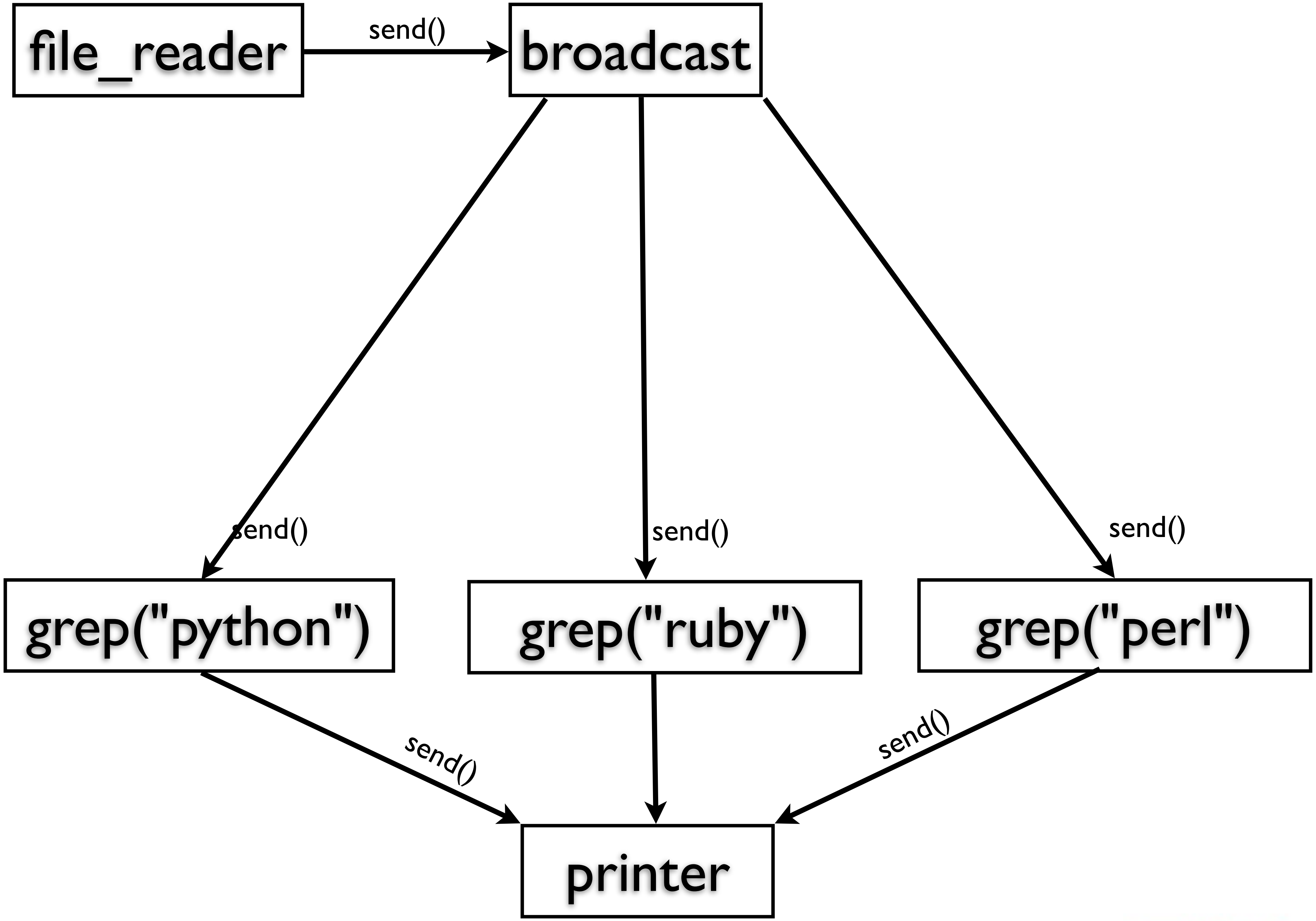
```
def source(target):
    target.next()
    target.send(data1)
    target.send(data2)
    ...
    target.close()
```

```
def filter(target):
    target.next()
    try:
        while True:
            data = (yield)
            data = process(data)
            target.send(data)
    except GeneratorExit:
        target.close()
```

```
def sink():
    while True:
        data = (yield)
        data = process(data)
```

```
source(filter1(filter2(sink())))
```

Example



file_reader

broadcast

grep("python")

grep("ruby")

grep("perl")

printer

Concurrent Coroutine

```
def co_sendto(f):
    try:
        while True:
            item = (yield)
            pickle.dump(item, f)
            f.flush()
    except StopIteration:
        f.close()

def co_recvfrom(f, target):
    try:
        while True:
            item = pickle.load(f)
            target.send(item)
    except EOFError:
        target.close()
```

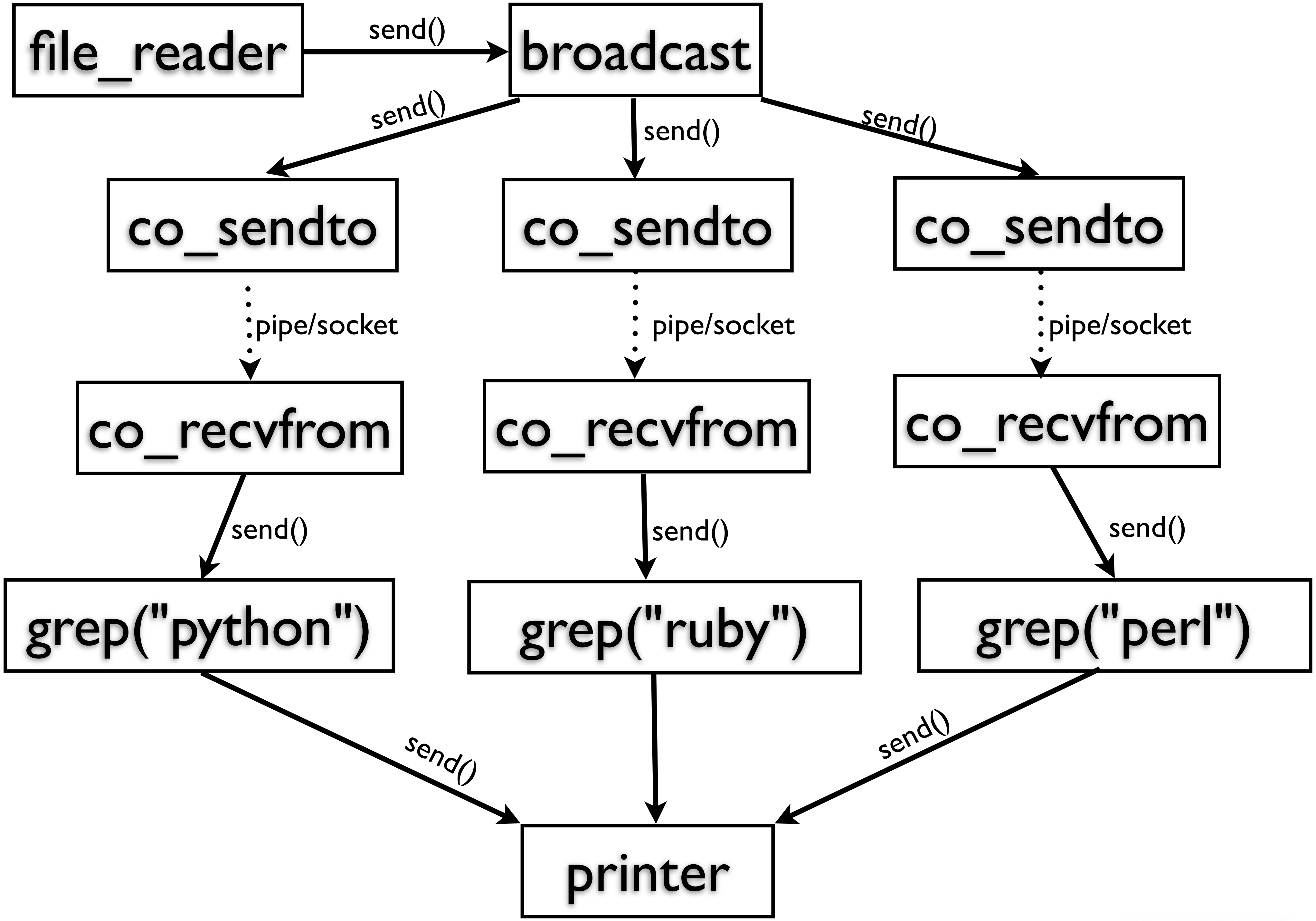
Concurrent Coroutine

```
def co_sendto(f):
    try:
        while True:
            item = (yield)
            pickle.dump(item, f)
            f.flush()
    except StopIteration:
        f.close()

def co_recvfrom(f, target):
    try:
        while True:
            item = pickle.load(f)
            target.send(item)
    except EOFError:
        target.close()
```

f could be:

- pipe
  - subprocess.Popen().stdin / stdout
- socket
  - socket.makefile('r' / 'w')

Example


file_reader
send()
broadcast
send()
send()
send()
co_sendto
co_sendto
co_sendto
pipe/socket
pipe/socket
pipe/socket
co_recvfrom
co_recvfrom
co_recvfrom
send()
send()
send()
grep("python")
grep("ruby")
grep("perl")
send()
send()
printer

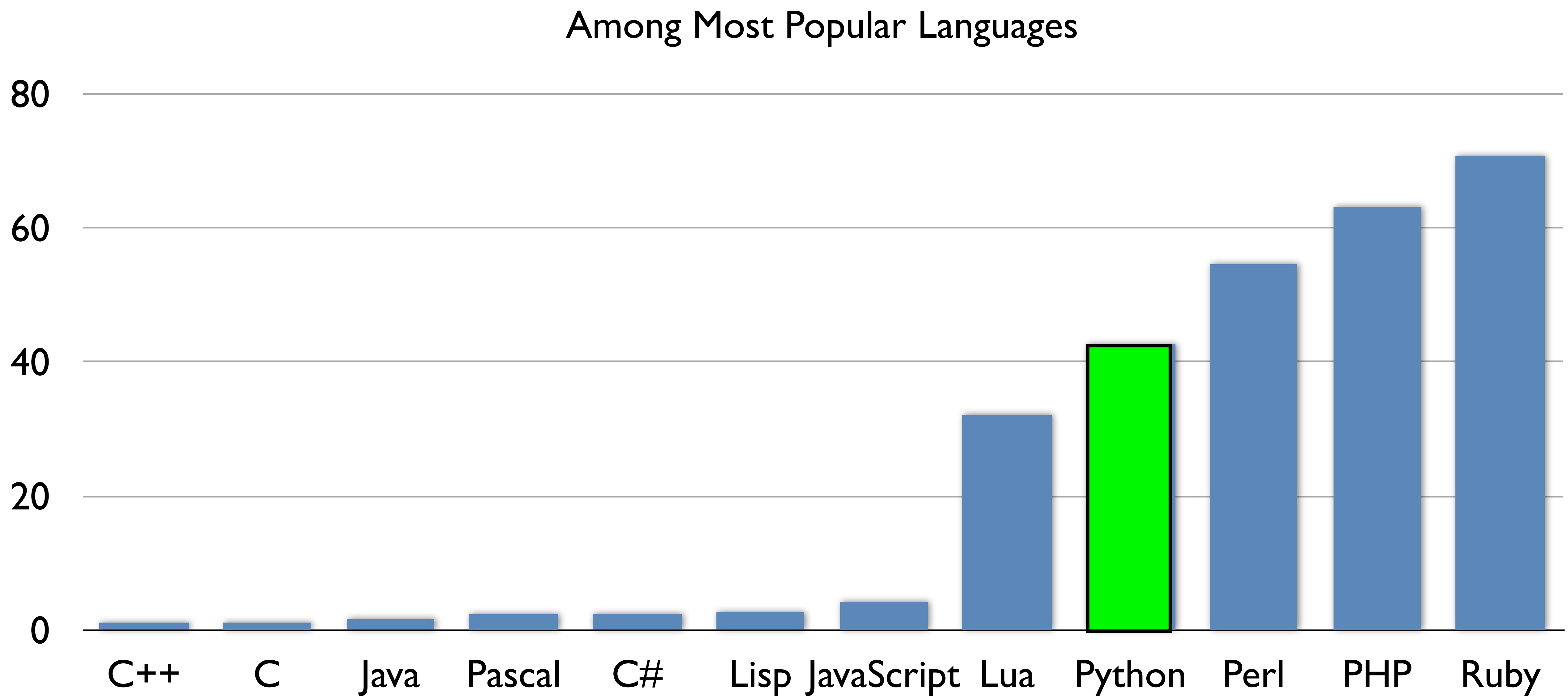
Coroutine Libraries / Frameworks

- Kamaelia
- cogen
- greenlet
- concurrence
- eventlet
- gevent

Coroutine (IMO) is much better than event/callback based model

Performance Tuning

Python's Speed



6 Steps to Gain Speed

1) Find performance bottlenecks

2) Use better algorithms / architecture

3) Use faster tools

4) Write optimized code

5) Hire optimizers

6) Write your own extension modules

Find Performance Bottlenecks

- Profile, no guess
  - profile
    - a pure Python module
  - cProfile
    - written in C, new in Python 2.5
    - same interface with profile, but lower overhead
  - hotshot
    - written in C, new in Python 2.2
    - not maintained and might be removed

豆瓣 douban

cProfile Usage

- cProfile.run('foo()')
- cProfile.run('foo()', 'profile.result')
- python -m cProfile -o profile.result myscript.py
- p = pstats.Stats('profile.result')
- p.sort_stats('cumulative').print_stats()
    - sort by 'cumulative' to find what algorithms are taking time
    - sort by 'time' to find what functions are taking time
- RunSnakeRun for GUI guys
- RTFM, please
- for IPython, type %prun?

Line Profile

- line_profile and kernprof

```
@profile
def slow_function(a, b, c):
    ...

$ kernprof.py -l -v script_to_profile.py
...
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           @profile
     2                                           def f():
     3         1            3      3.0      0.2      s = 0
     4      1001          934      0.9     48.6      for i in xrange(1000):
     5      1000          984      1.0     51.2          s += i
     6         1            1      1.0      0.1      return s
```

Use Better Algorithms / Architecture

- How to calculate sum([1, 2, ..., 100])?

How To Know Which is Better?

- timeit!
- python -m timeit -s "setup" "statement"
- e.g. which is faster, "d.has_key(k)" or "k in d"?

```
$ python -m timeit -s "d=dict(zip(range(1000), range(1000)))"
"d.has_key(500)"
1000000 loops, best of 3: 0.223 usec per loop

$ python -m timeit -s "d=dict(zip(range(1000), range(1000)))"
"500 in d"
10000000 loops, best of 3: 0.115 usec per loop
```

Use Bettern Algorithms / Architecture

- How to calculate sum([1, 2, ..., 100])?

```
s = 0
for i in range(101):
    s += i
```

8.3usec

```
s = sum(range(101))
```

2.8usec

```
s = sum(xrange(101))
```

2.03usec

```
s = (1 + 100) * 100 / 2
```

0.109usec

Use Better Algorithms / Architecture

- membership testing:
    - set & dict: O(1)
    - tuple & list: O(n)
- string concatenation:
    - ''.join(seq): O(n)
    - '+' or '+=': O(n**2)
- return iterator instead of a large list
- cache

Use Better Algorithm / Architecture

- multi-threading
  - threading
- multi-processing
  - fork
  - subprocess
  - multiprocessing
- async
  - asyncore
  - twisted
  - eventlet/greenlet

Use Better Algorithm / Architecture

- XML-RPC / Json-RPC / Thrift / Protocol Buffer
- Pyro
- Parrallel Python
- PyOpenCL / PyCUDA

Use Faster Tools

- use iterator form
    - range() -> xrange()
    - map() -> itertools.imap()
    - list comprehension -> generator expression
    - dict.items() -> dict.iteritems()
    - for i in range(len(seq)): ->
        - for item in seq:
        - for i, item in enumerate(seq):

Use Faster Tools

- use builtin types
  - list, tuple, set, dict
  - array, collections.deque, heapq

```
lst = []
for i in xrange(10000):
    lst.insert(0, i)

lst = collections.deque()
for i in xrange(10000):
    lst.appendleft(i)
```

25317% faster

Use Faster Tools

```
sorted(lst, reverse=True)[:10]
```

```
heapq.nlargest(10, lst)
```

613% faster

Use Faster Tools

- SAX is faster and memory efficient than DOM
- use C version of modules
  - profile -> cProfile
  - StringIO -> cStringIO
  - pickle -> cPickle
  - elementTree -> cElementTree / lxml
- select has lower overhead than poll (and epoll at low number of connections)
- numpy is essential for high volume numeric work

Write Optimized Code

- use key= instead of cmp= when sorting

```
lst = open('/Users/hongqn/projects/shire/luzong/group.py').read().split()

lst.sort(cmp=lambda x, y: cmp(x.lower(), y.lower()))


lst.sort(key=str.lower)
```

377% faster

Write Optimized Code

- local variables are faster than global variables

```
def f():
    for i in xrange(10000):
        r = abs(i)

def f():
    _abs = abs
    for i in xrange(10000):
        r = _abs(i)
```

28% faster

- you can eliminate dots, too

Write Optimized Code

- inline function inside time-critical loops

```
def f(x):
    return x + 1
for i in xrange(10000):
    r = f(i)
```

```
for i in xrange(10000):
    r = i + 1
```

187% faster

Write Optimized Code

- do not import modules in loops

```
for i in xrange(10000):
    import string
    r = string.lower('Python')
```

```
import string
for i in xrange(10000):
    r = string.lower('Python')
```

178% faster

Write Optimized Code

- list comprehensions are faster than for-loops

```
lst = []
for i in xrange(10000):
    lst.append(i)
```

```
lst = [i for i in xrange(10000)]
```

213% faster

Write Optimized Code

- use "while 1" for time-critical loops (readability lost!)

```
a = 0
while True:
    a += 1
    if a > 10000:
        break
```

```
a = 0
while 1:
    a += 1
    if a > 10000:
        break
```

78% faster

Write Optimized Code

- "not not x" is faster than "bool(x)" (not recommended!)

```
bool([])
```

```
not not []
```

196% faster

Hire Optimizers

- sys.setcheckinterval()
  - Python checks for thread switch and signal handling periodly (default 100 python virtual instructions)
  - set it to a larger value for better performance in cost of responsiveness

Hire Optimizers

- gc.disable()
  - disable automatic garbage collection
- gc.set_threshold()
  - collect less frequently

Hire Optimizers

- Psyco
- PyPy
- Shed Skin
- numexpr

Write Your Own Extension Modules

- Python/C API
- ctypes
- SWIG
- Pyrex / Cython
- Boost.Python
- Weave

Gold Rule

Premature optimization is the root of all evil.

-- Donald Knuth