Tema 3 Cuestiones avanzadas

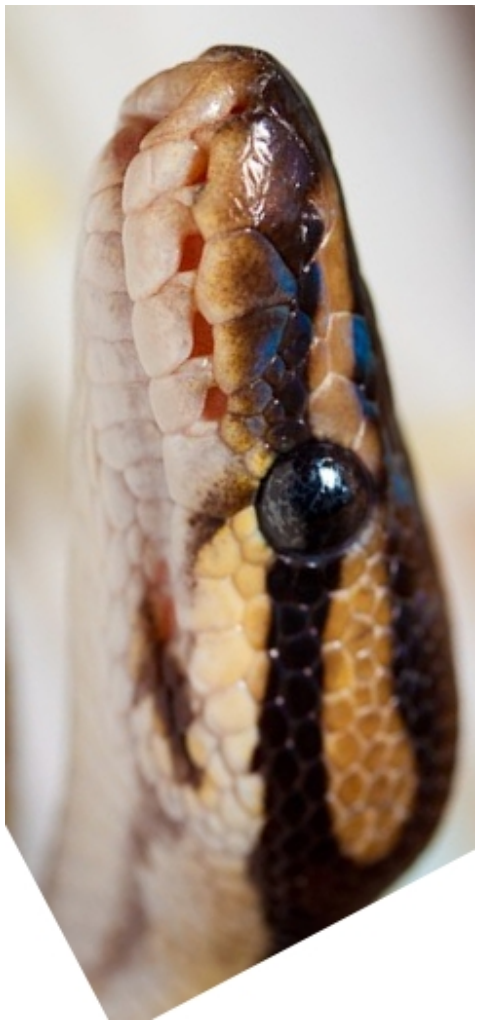
Curso de Python Avanzado

Juan Pedro Bolívar Puente

Instituto de Astrofísica de Andalucía

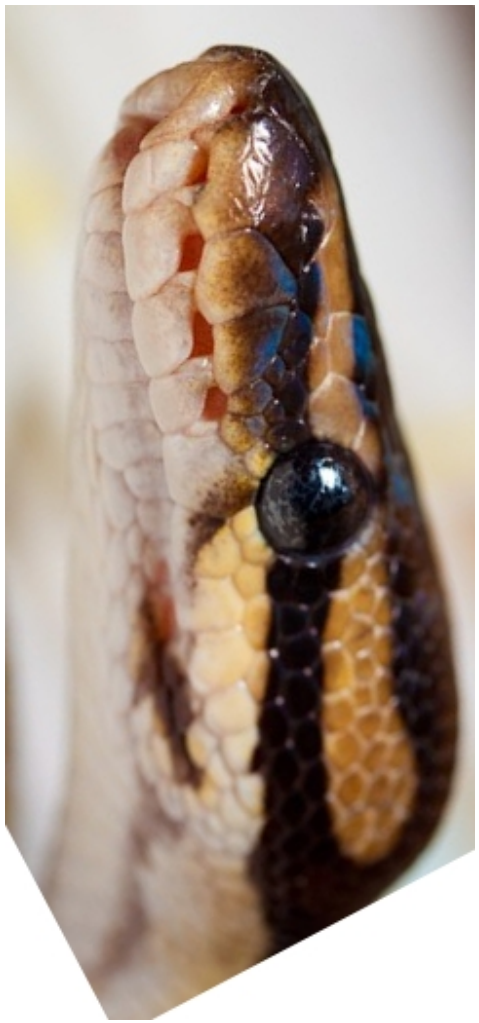
Mayo de 2011

Índice



1. Iteradores
2. Generadores
3. Decoradores
4. Gestión de recursos

Índice



1 Iteradores
2 Generadores
3 Decoradores
4 Gestión de recursos

Iteradores

Iterador =

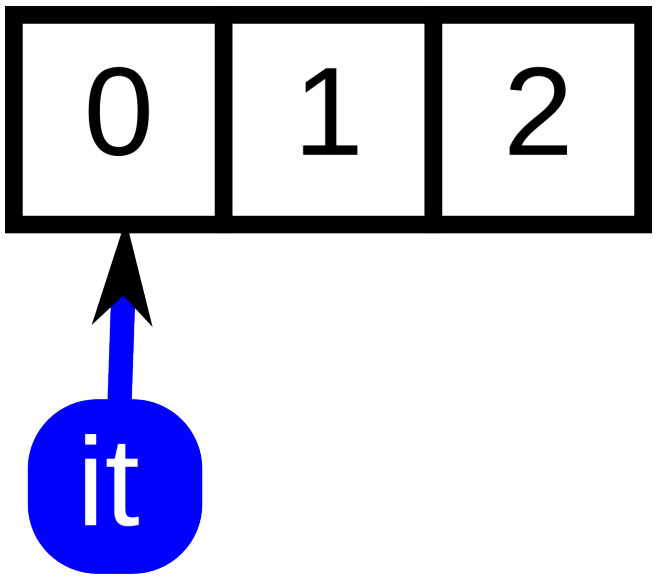
Apuntador a un elemento de una secuencia

Es objeto con:

`__iter__ (self)`
: Devuelve a sí mismo.

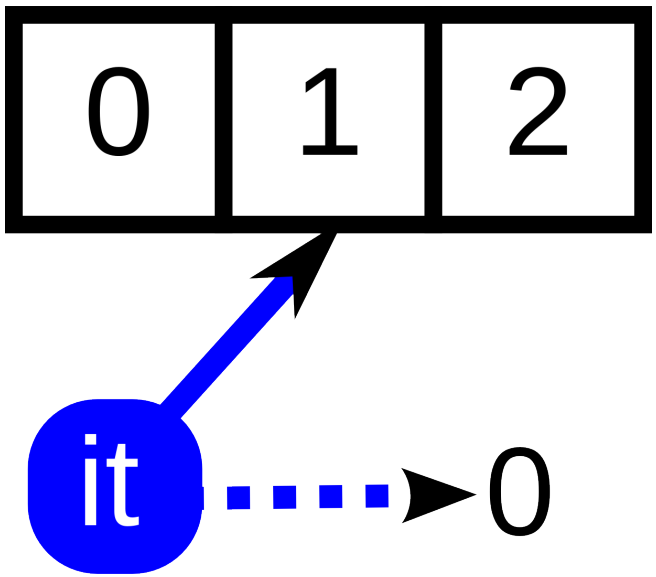
`next (self)`
: Devuelve el elemento actual de la sequencia y avanza el estado en un elemento. Si ha terminado lanza `StopIteration`

Ejemplo de iteración



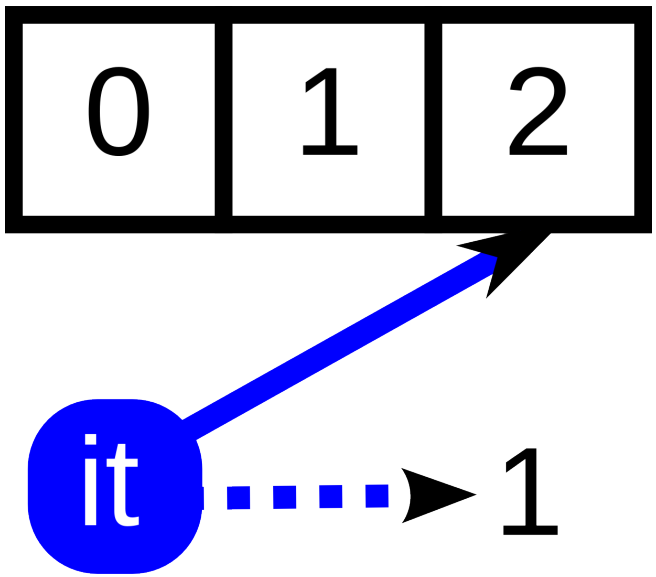

```
r = range (3)
it = iter (r)
it.next()   # 0
it.next()   # 1
it.next()   # 2
it.next()   # -
```

Ejemplo de iteración



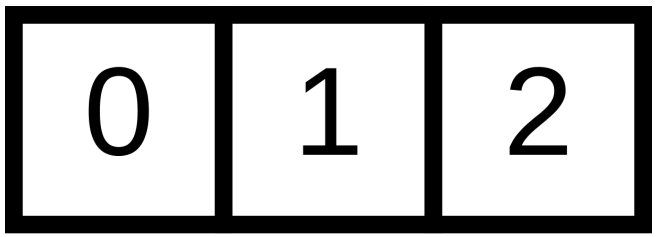

```
r = range (3)
it = iter (r)}
it.next() # 0
it.next()   # 1
it.next()   # 2
it.next()   # -
```

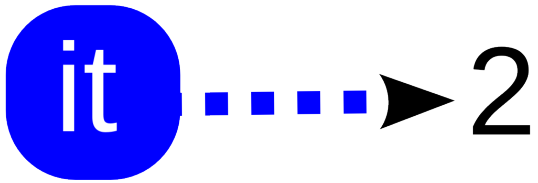
Ejemplo de iteración




```
r = range (3)
it = iter (r)
it.next()   # 0
it.next() # 1
it.next()   # 2
it.next()   # -
```

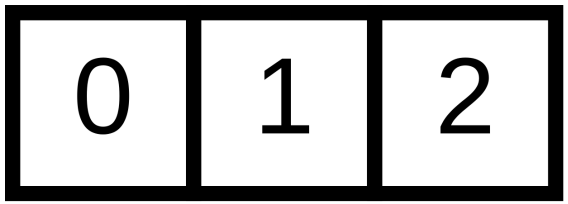
Ejemplo de iteración






```
r = range (3)
it = iter (r)
it.next()   # 0
it.next()   # 1
it.next() # 2
it.next()   # -
```

Ejemplo de iteración




```
r = range (3)
it = iter (r)
it.next()   # 0
it.next()   # 1
it.next()   # 2
it.next() # -
```

Equivalencia while — for

```
it = iter (range (10))
try:
    while True:
        print it.next ()
except StopIteration:
    pass
```

```
for x in range (10):
    print x
```

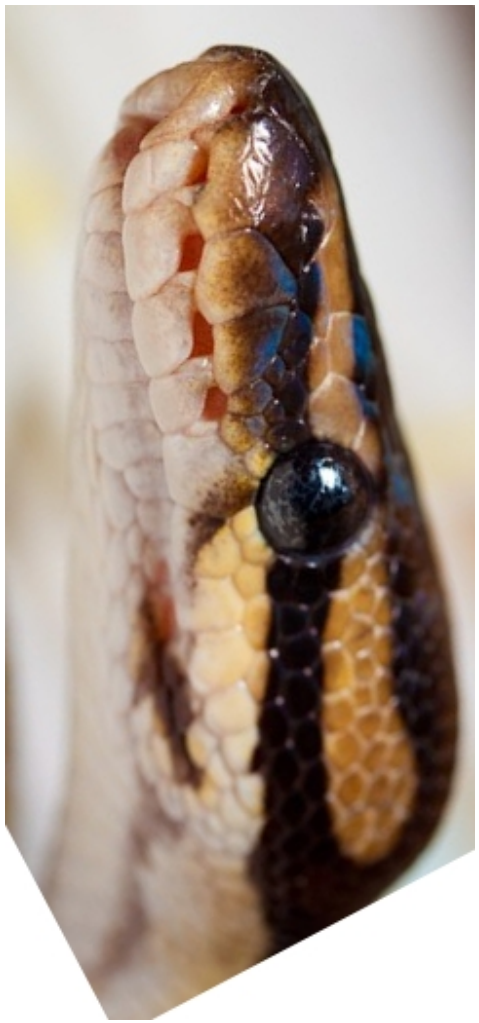
Ejemplo de iterador

Un iterador vacío

```
class IteradorVacio (object):
    def __iter__ (self):
        return self
    def next (self):
        raise StopIteration

for x in IteradorVacio ():
    print "No␣me␣ejecuto!"
```

Índice



1 Iteradores
2 Generadores
3 Decoradores
4 Gestión de recursos

Generadores

Generador = Iterador virtual

I.e. cada elemento no está almacenado, se calcula

range vs xrange

Probar con %timeit en Ipython

```
print xrange (5).__class__
for x in xrange (5):
    print x

print range (5).__class__
for x in range (5):
    print x
```

La biblioteca `itertools`

- Secuencias infinitas

  `repeat, cycle, count, ...`
- Manipulación de listas eficientes

  `izip, chain, islice, product, permutations, combinations, ...`
- Programación funcional eficiente

  `imap, ifilter, starmap, takewhile, dropwhile, tee, ...`

Ejemplos itertools

```
from itertools import *
it = chain ('Hola ', 'Mundo!')
print list (it)
it = chain ('Hola ', 'Mundo!')
it = ifilter (str.isupper, it)
print list (it)
a, b = tee (xrange (5), 2)
print list (a), list (b)
it = cycle ([1, 2])
it = islice (it, 10)
print list (it)
```

La sentencia `yield`

yield convierte
función $\longrightarrow$ generador

1. Aparentemente es como un `return`
2. Guarda el estado para la próxima iteración

Requiere Python $\geq$ 2.5

Inspirado en las continuations de Lisp

Ejemplo de yield

Generador de factoriales

```
def factoriales ():
    x = 1
    n = 2
    while True:
        yield x
        x *= n
        n += 1
```

Ejemplo de yield

Ejecución del generador factoriales

```
it = factoriales ()
print it.__class__
print it.next ()
print it.next ()

for x in islice (factoriales (), 10):
    print x
```

Generadores por comprensión

Lista por comprensión = Lista transformando otra secuencia

Generador expresión =

Generador transformando otra secuencia

Sintáxis, Python $\geq$ 2,4

```
(... for ... in ... [if ...])
```

¡Muchísimas veces no necesitamos listas!

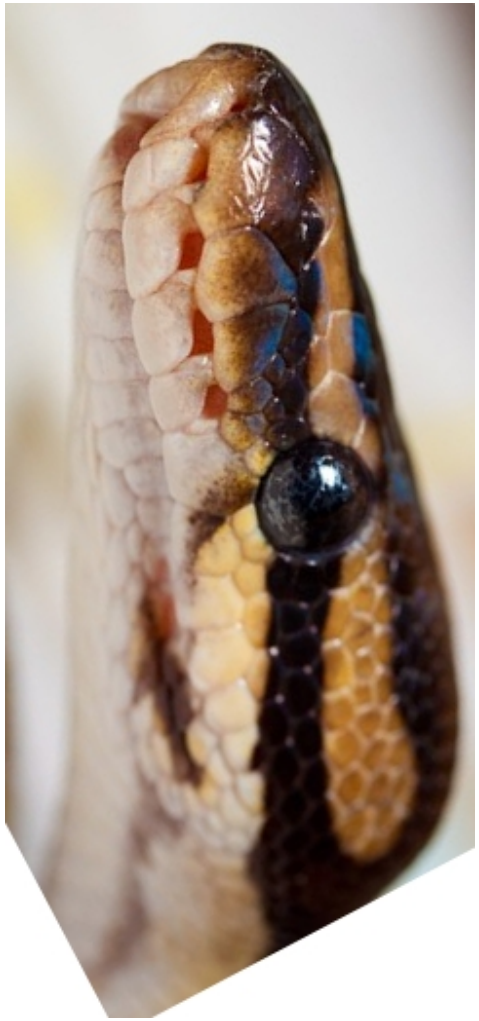
Muy útiles cuando reducimos la secuencia ...

```
sum ([x*x for x in range(10)])
sum (x*x for x in range(10))

max (len(line) for line in file)

dot = sum(x*y for x,y in izip(xv, yv))
```

Índice



1. Iteradores
2. Generadores
3. Decoradores
4. Gestión de recursos

Decoradores

Patrón común ...

```
def funcion (): ...
funcion = decorador (funcion)
```

La función `decorador`

- Devuelve una nueva función
- Que internamente llama a la función antigua

Decoradores

¡Se convirtió en sintáxis!

```
@decorador
def funcion (): ...
```

La función `decorador`

- Devuelve una nueva función
- Que internamente llama a la función antigua

Requiere Python $\geq$ 2.4

Ejemplo decorador

Traza de ejecución

```
def trace (fun):
    def wrapper (*a, **k):
        print "**>>:", fun.__name__, a, k
        res = fun (*a, **k)
        print "**<<:", fun.__name__
        return res
    return wrapper

@trace
def mi_funcion ():
    print "Hola␣Mundo!"
```

Decoradores con parámetros

Tras la @ podemos poner
cualquier expresión

Todo problema en computación puede ser resuelto añadiendo un nivel de indirección

David Wheeler

Decoradores con parámetros

Tras la @ podemos poner cualquier expresión

- Una función parametrizada devuelve el decorador
- El decorador devuelve la función

Ejemplo decorador con parámetros

```
import sys
def trace (out = sys.stdout):
    def decorator (fun):
        @wraps (fun)
        def wrapper (*a, **k):
            print >> out, "**>>", \
                  fun.__name__, a, k
            res = fun (*a, **k)
            print >> out, "**<<", \
                  fun.__name__
            return res
        return wrapper
    return decorator
```

Ejemplo decorador con parámetros

¡Recordad poner los paréntesis siempre!

```
@trace (sys.stderr)
def mi_funcion ():
    print "Hola mundo!"

@trace ()
def mi_funcion ():
    print "Hola mundo!"
```

Conservando los metadatos

Problema

Se pierden los metadatos de la función original

¿Metadatos?

- Nombre: `__name__`
- Documentación: `__doc__`
- Signatura: `inspect.getargspec ()`

El decorador `wraps (origfunc)`

El decorador `wraps (origfun)`
copia los metadatos de `origfun`

```
def trace (fun):
    @wraps (fun)
    def wrapper (*a, **k):
        print "**>>", fun.__name__, a, k
        res = fun (*a, **k)
        print "**<<", fun.__name__
        return res
    return wrapper
```

El decorador `wraps (origfunc)`

El decorador `wraps (origfun)`
copia los metadatos de `origfun`

¡Ahora sí!

```
@trace
def mi_funcion ():
    print "Hola Mundo!"

help (mi_funcion)
```

Decoradores comunes

@property

Métodos que se comportan como atributos.

@staticmethod

Método que se invoca sin instancia

Python **no** es Java, considera una función libre

@classmethod

Método que se invoca con la clase

Decoradores comunes

¡Esta clase tiene de tó'!

```
class Clasecilla (object):
    @property
    def propiedad (self):
        print "Propiedad"

    @staticmethod
    def estatico ():
        print "Metodo␣estatico"

    @classmethod
    def declase (cls):
        print "Metodo␣de␣clase:␣", cls
```

Decoradores comunes

Ejemplos de llamadas

```
Clasecilla.estatico ()
Clasecilla.declase ()

obj = Clasecilla ()
obj.propiedad
obj.estatico ()
obj.declase ()
```

Decoradores de clase

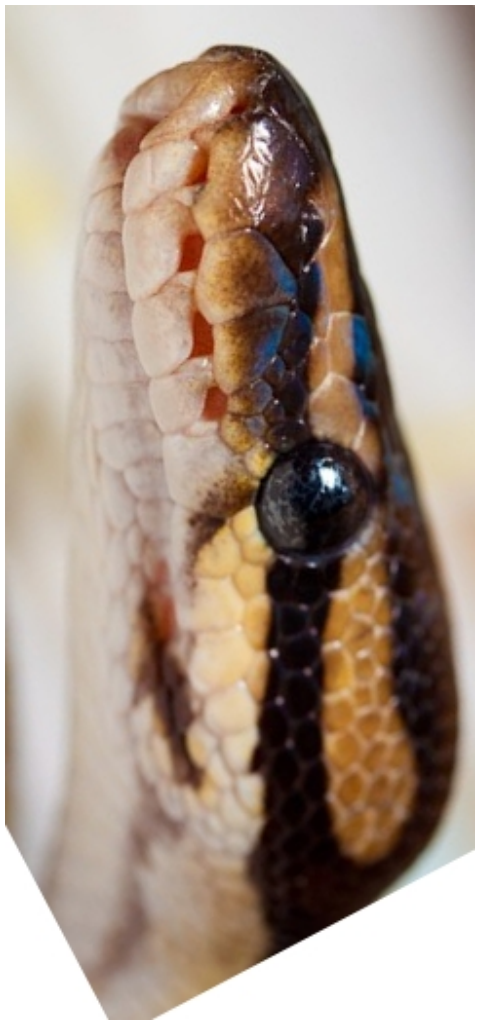
```
@decorador
class Clase (...): ...
```

Las clases también pueden decorarse

- Se recomienda devolver la clase original modificada
- A veces es mejor usar una metaclase

Requiere Python $\geq$ 2.6

Índice



1 Iteradores
2 Generadores
3 Decoradores
4 Gestión de recursos

Gestión de recursos

Todo recurso es finito

Y toda entidad finita es un recurso

Patrón típico

1. $r \leftarrow obtener_{recurso}()$
2. $usar(r)$
3. $liberar_{recurso}(r)$

Ejemplo en C

```
void* r = malloc (n);
usar (r);
free (r);
```

Gestión de recursos

Todo recurso es finito

Y toda entidad finita es un recurso

Patrón típico

1. $r \leftarrow obtener_{recurso}()$
2. $usar(r)$
3. $liberar_{recurso}(r)$

¡No sólo memoria!

- Ficheros
- Bases de datos
- *Mutexes*
- *Sockets*

Gestión de memoria

cpython usa conteo de referencias + recolector de basura

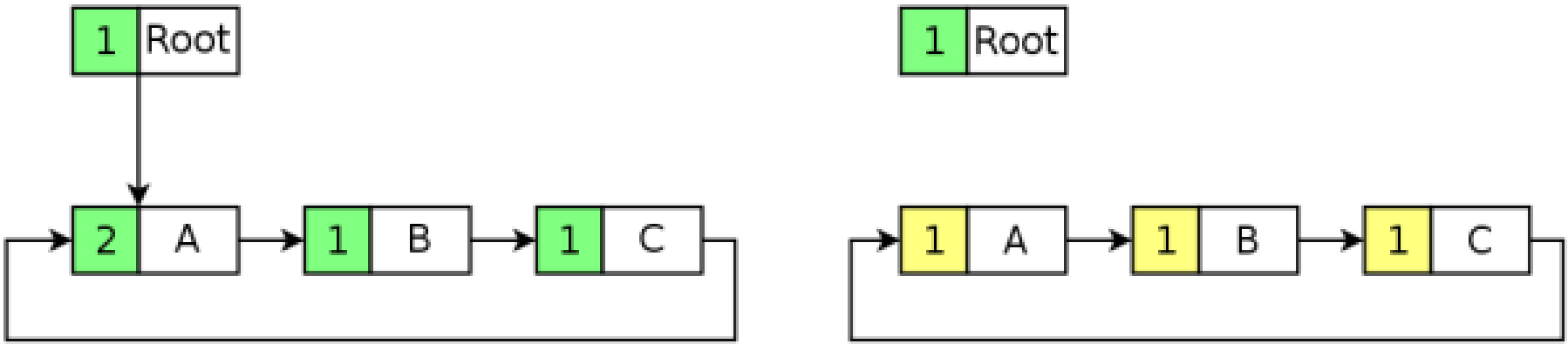


Figura: Ciclos en conteo de referencias requieren recolector

El recolector de basura es opcional, `gc.disable()`

Gestión de memoria

cpython usa conteo de referencias + recolector de basura

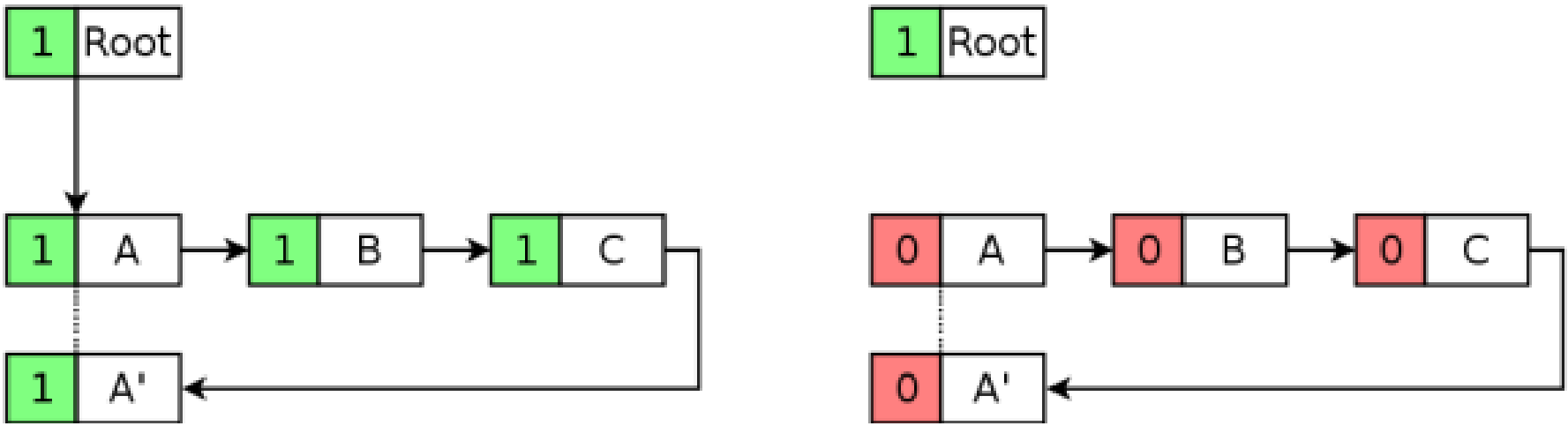


Figura: Los ciclos pueden romperse con referencias débiles

Las `weakref` no incrementan la cuenta

El módulo weakref

```
import weakref
class Foo (object): pass

root = Foo ()
child1 = weakref.proxy (root)
child2 = weakref.ref (root)

child1.attr = child1
print child2 ().attr
del root
print child2 ()
try: print child1.attr
except ReferenceError, err: print err
```

Otros recursos con el gestor de memoria

El método especial __**del**__ (*destructor*)

- Se invoca al liberar la memoria de un objeto
- Idealmente, bastaría para gestionar recursos
  Resource Adquisition Is Initialization, estilo C++

En la práctica ...

- Las trazas de pila guardan referencias
- Implementaciones no usan conteo de referencias
  PyPy, Jython, Iron Python

$\Rightarrow$ Sólo útil si el recurso es abundante

¿Por qué no basta la gestión explícita?

Busca el error aquí ...

```
def copyfile (infile, outfile):
    writer = open (outfile, 'w')
    reader = open (infile)

    writer.write (reader.read ())

    reader.close ()
    writer.close ()
```

¿Por qué no basta la gestión explícita?

¡Cuidadín con las excepciones!

```
def copyfile (infile, outfile):
    writer = open (outfile, 'w')
    reader = open (infile)

    writer.write (reader.read ())

    reader.close ()
    writer.close ()
```

Ejecutad con `infile` erróneo y ved con `lsof` que `outfile` sigue abierto

La sentencia finally

`finally` se ejecuta **siempre**

```
def copyfile (infile, outfile):
    writer = open (outfile, 'w')
    try:
        reader = open (infile)
        writer.write (reader.read ())
        reader.close ()
    finally:
        writer.close ()
```

La sentencia `with`

`with` asocia la *vida de un recurso*
a un bloque sintáctico

```
def copyfile (infile, outfile):
    with open (outfile, 'w') as w:
        with open (infile) as r:
            w.write (r.read ())
```

Requiere Python $\geq$ 2.5

La sentencia `with`

`with` asocia la *vida de un recurso*
a un bloque sintáctico

```
def copyfile (infile, outfile):
    with open (outfile, 'w') as w, \
         open (infile) as r:
        w.write (r.read ())
```

Requiere Python $\geq$ 2.7

Gestores de contexto

Gestor de contexto = Recurso usable con `with`

Objeto con métodos especiales

`__enter__ (self)`
: Se ejecuta al entrar en el bloque
: Devuelve el valor que se asocia con `as`

`__exit__ (self, exc_type, exc_val, exc_tb)`
: Se ejecuta al salir en el bloque
: Devuelve `True` para silenciar la excepción

Gestores de contexto

El decorador `contextlib.contextmanager` convierte generador → gestor de contexto

```
@contextmanager
def tag (name):
    print "<%s>" % name
    yield
    print "</%s>" % name

with tag ('body'):
    print "Contenido"
```

Recursos adicionales

¿Preguntas?

Muchas gracias por su atención.