Persistencia en Python

Jesús Cea Avión
https://www.jcea.es/
jcea@jcea.es
25 de febrero de 2015

Persistencia en Python
Sobre mí

Jesús Cea Avión – I+D, Freelance

Twitter: @jcea

Email: jcea@jcea.es

Web: https://www.jcea.es/ - https://blog.jcea.es/

Persistencia en Python
¿Por qué?

- Subir el nivel de abstracción:
  - Overlays → memoria virtual.
  - Almacenamiento → sistemas de ficheros.
  - Gestión de datos → bases de datos ACID.

- Reducir la “inadaptación de la impedancia”:
  - Gestión explícita del almacenamiento → transparencia.
  - Programas “eternos”.

Persistencia en Python
Premisa fundamental

Trabajar como si todos los datos estuviesen en memoria.

Nuestro programa no debe preocupas del almacenamiento, serialización o ajustarnos a ninguna estructura.

Persistencia en Python
Propuesta simple 1

Trabajar con todo en memoria y volcar a disco un “pickle” de vez en cuando

- Simple de implementar pero no escala.
- No hay concurrencia entre procesos.

Persistencia en Python
Propuesta simple 2
Trabajar con todo en memoria y volcar a disco un “log” de cambios tras cada transacción

- Simple de implementar, el tráfico de escritura es proporcional al volumen de cambios.
- Recargar la aplicación puede ser lento: volcado completo esporádico.
- Patrón de diseño simple: “command”: se puede hacer “undo”, se puede ver la historia.
- No hay concurrencia entre procesos.

Persistencia en Python
Propuesta simple 3 (I)

Propuesta 2 pero definiendo una clase base “persistente”

- Queremos que la detección de cambios en los objetos sea automática:

```
def __setattr__(self, name, value):
    if name[:3] != '_p_' :  # Explicar el _p_
        self._p_note_change()
    super().__setattr__(name, value)
```

- Habitualmente programado en C.

Persistencia en Python
Propuesta simple 3 (II)

- Definimos tipos básicos persistentes: diccionarios, listas, tuplas, “sets”.
- Podemos definir clases propias persistentes simplemente heredando de “persistente”.
  - La herencia múltiple en Python nos ayuda.
- Estos objetos detectan automáticamente cuándo son modificados. Toman nota y vuelcan cambios a disco al finalizar la transacción.
  - ¡Se puede hacer “rollback” de cambios!.

Persistencia en Python
Propuesta simple 3 (III)

```
[Llega petición]
try :
    [Haz la operación que sea]
   (no te preocupes del almacenamiento)
    persistencia.commit()
except Exception :
    persistencia.rollback()
    raise
```

Persistencia en Python
Propuesta simple 3 (IV)

- El tiempo de grabación es bajo, pero el tiempo de carga en RAM es alto debido al “replay”.
- Limitados por la RAM disponible (podemos tirar de “swap” si el “Working Set” cabe).
- No hay concurrencia entre procesos.

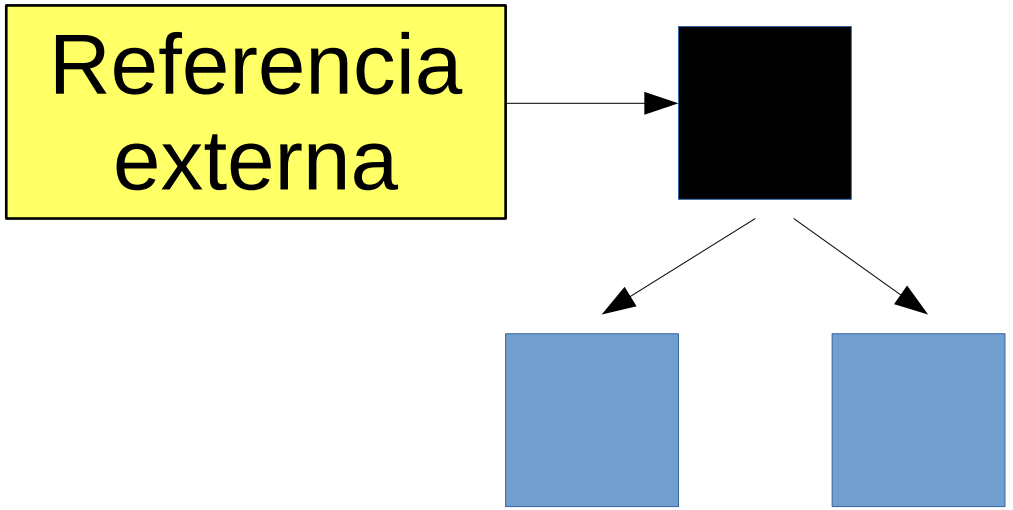
Persistencia en Python
Propuesta simple 4 (I)

- Supongamos que cada instancia persistente nueva que se cree recibe un identificador único.
- Supongamos que cuando se vuelca un objeto persistente, volcamos su “pickle” y todo su grafo de objetos conectados hasta tropezar con otros objetos persistentes.
  - Se vuelcan referencias a esos otros objetos persistentes
- Supongamos que el resultado se almacena en disco indexado por identificador único.

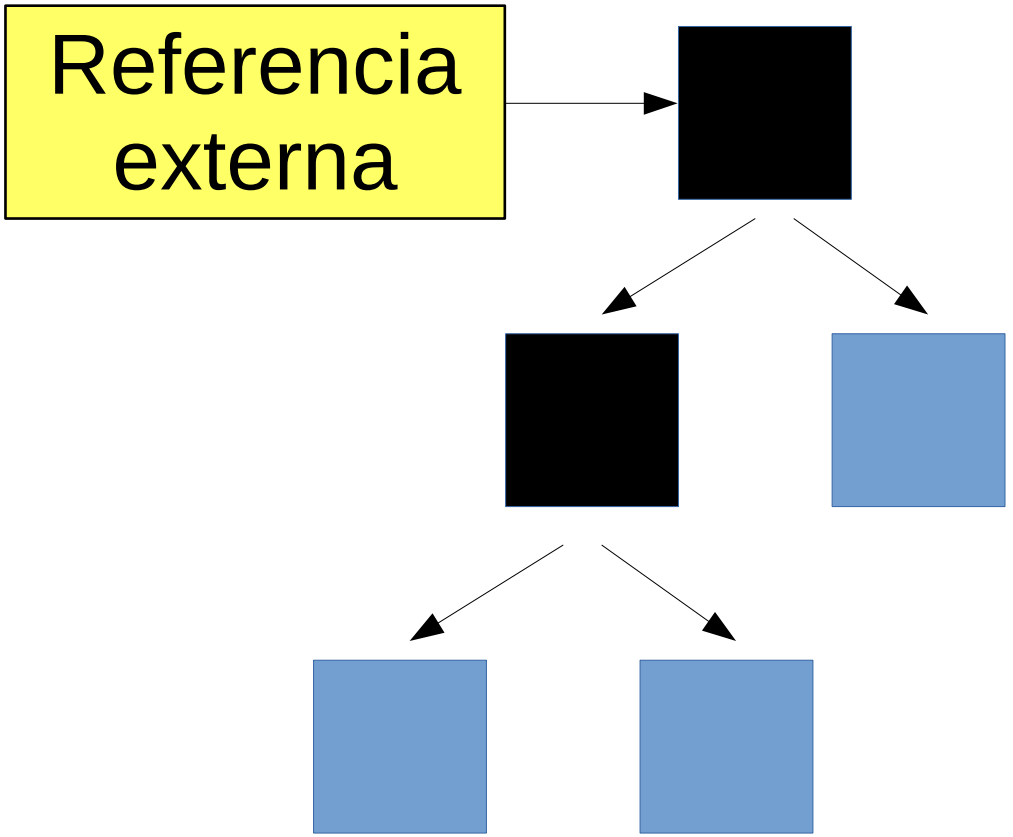
Persistencia en Python
Propuesta simple 4 (II)

- Uso de “pickle.persistent_id()” y “pickle.persistent_load()”. Nos permite:
  a) Cortar el “pickle” en la frontera con otros objetos persistentes.
  b) Al cargar un objeto, poder recrear su frontera con objetos “ghost”.
- Los objetos “ghost” se reconstruyen desde el disco cuando hay cualquier acceso a los mismos, de lectura o de escritura.

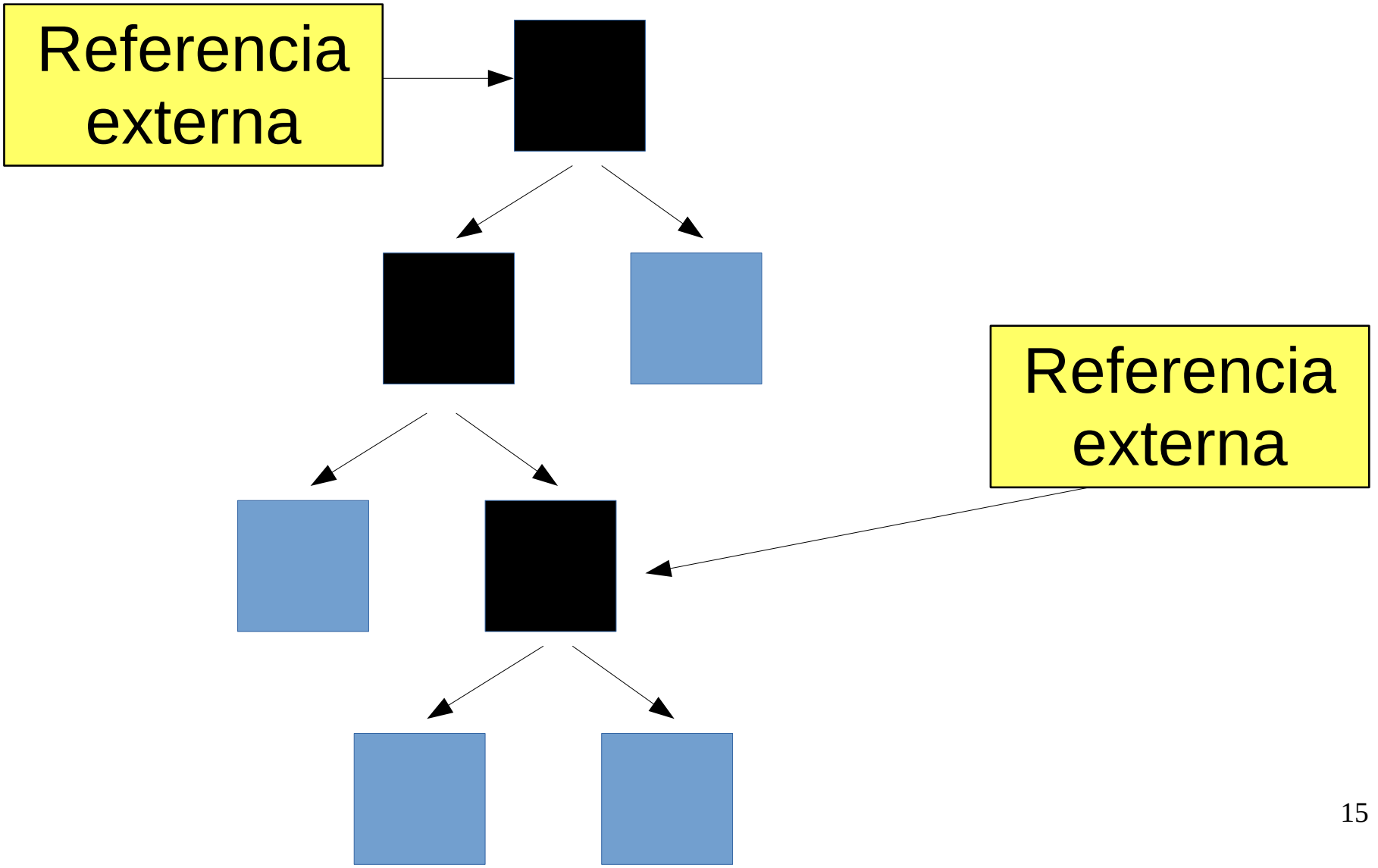
Persistencia en Python
Propuesta simple 4 (III)



Persistencia en Python
Propuesta simple 4 (IV)



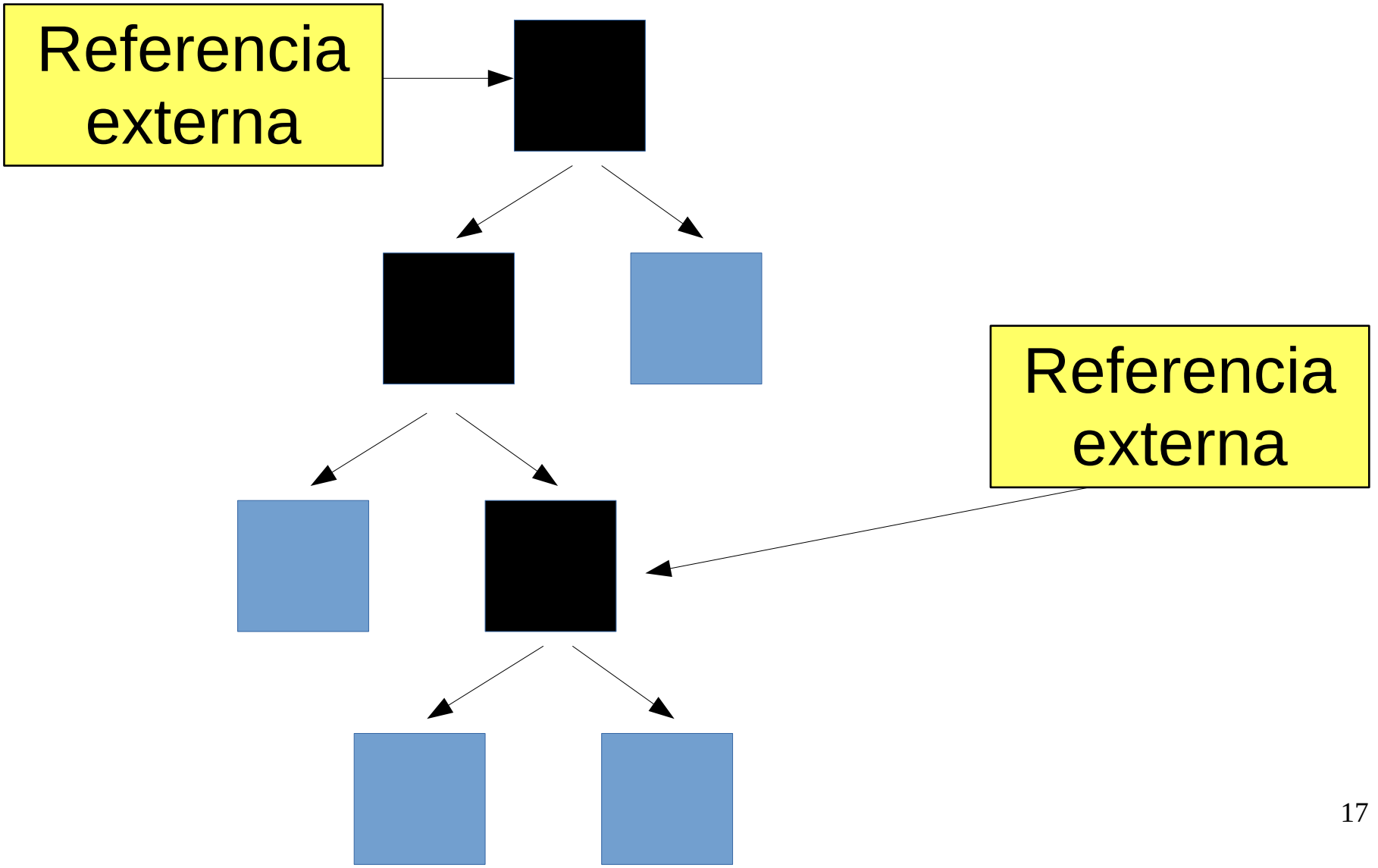
Persistencia en Python
Propuesta simple 4 (V)



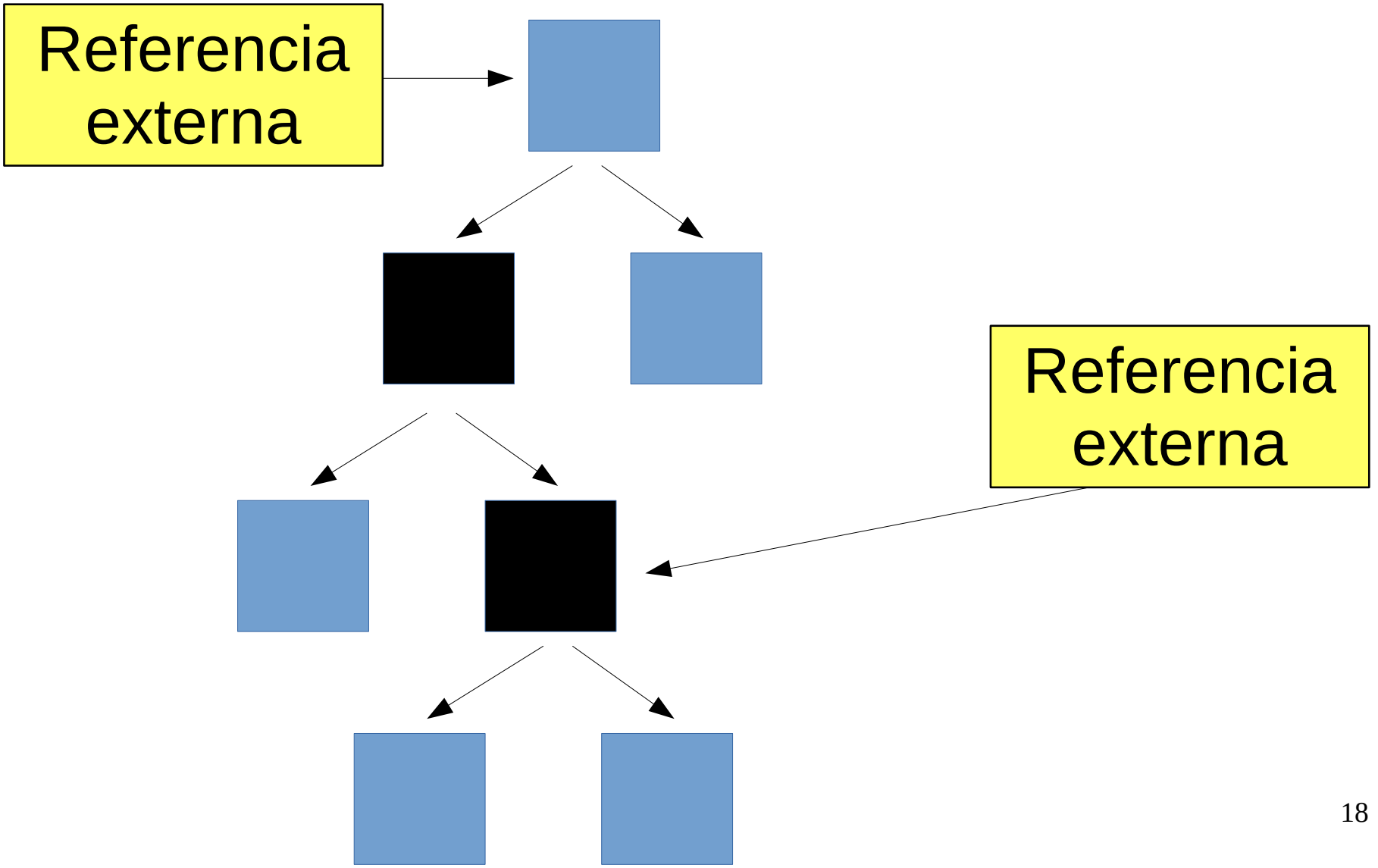
Persistencia en Python
Propuesta simple 4 (VI)

- No hace falta cargar todos los objetos en RAM al arrancar. De hecho no cargamos nada.
- El grafo se va completando en memoria a medida que vamos referenciando objetos.
- Podemos ir eliminando de memoria los objetos menos usados pasándolos primero a “ghost”.
- Si un objeto no tiene referencias, se elimina automáticamente por el proceso normal.

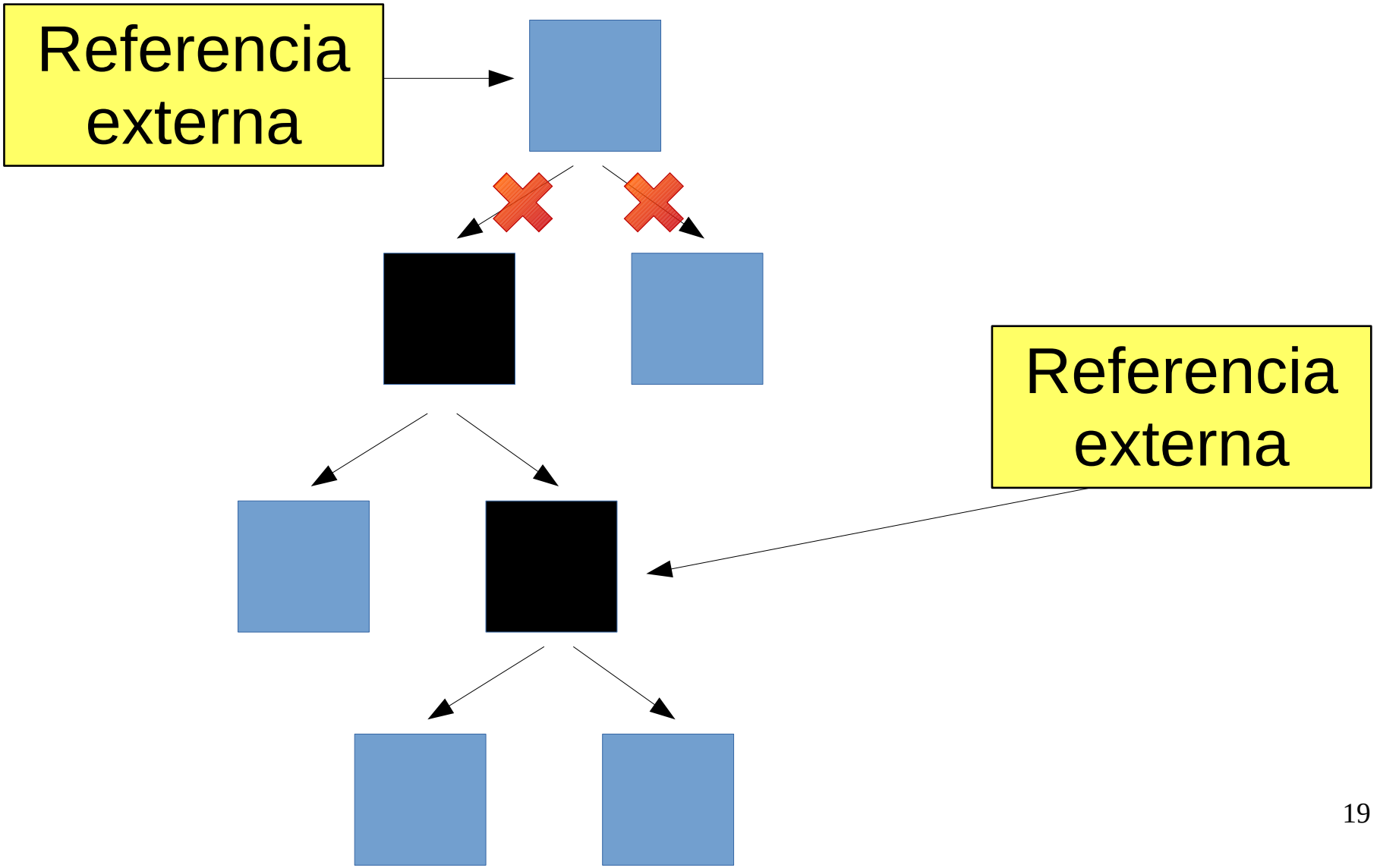
Persistencia en Python
Propuesta simple 4 (VII)



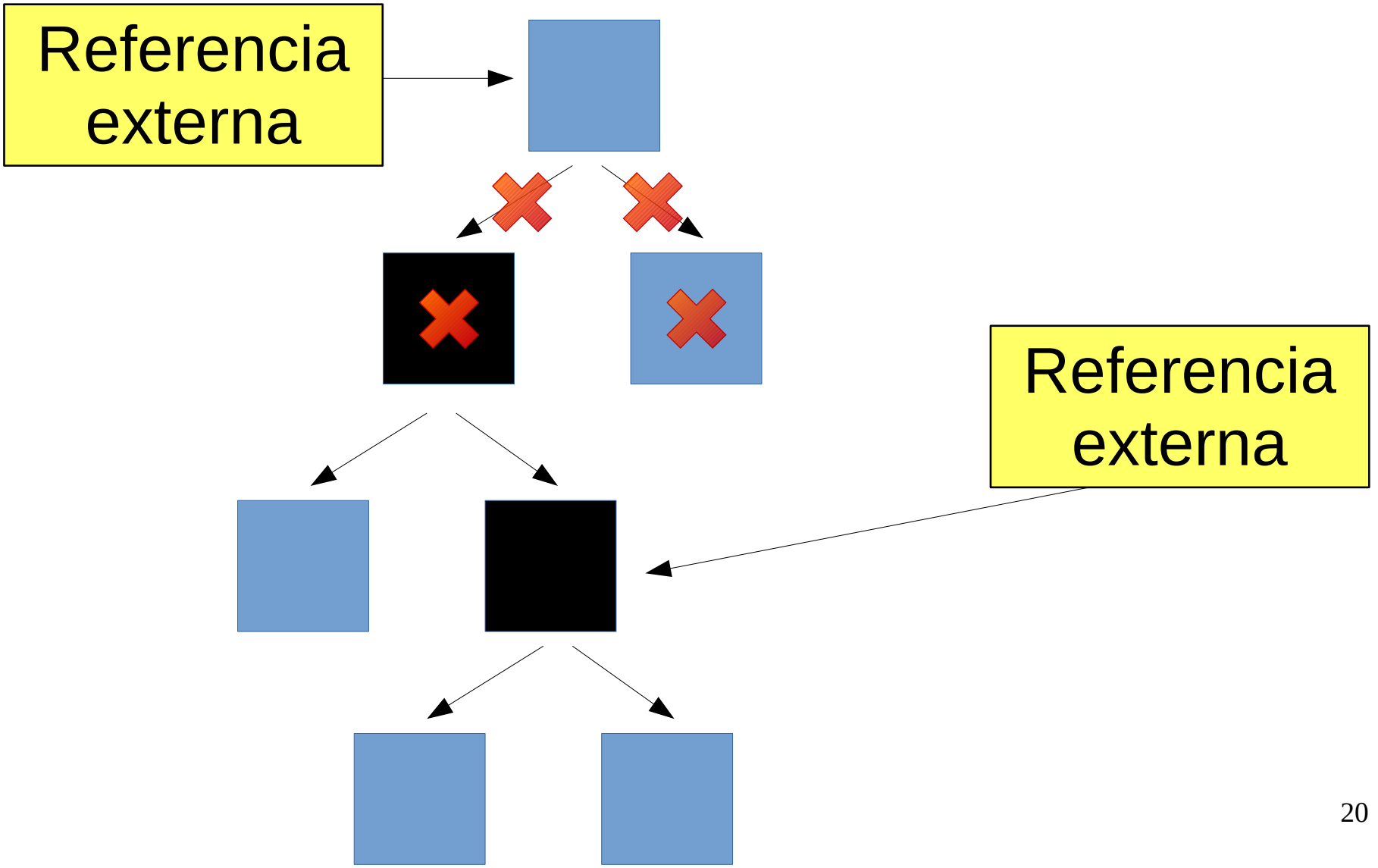
Persistencia en Python
Propuesta simple 4 (VIII)



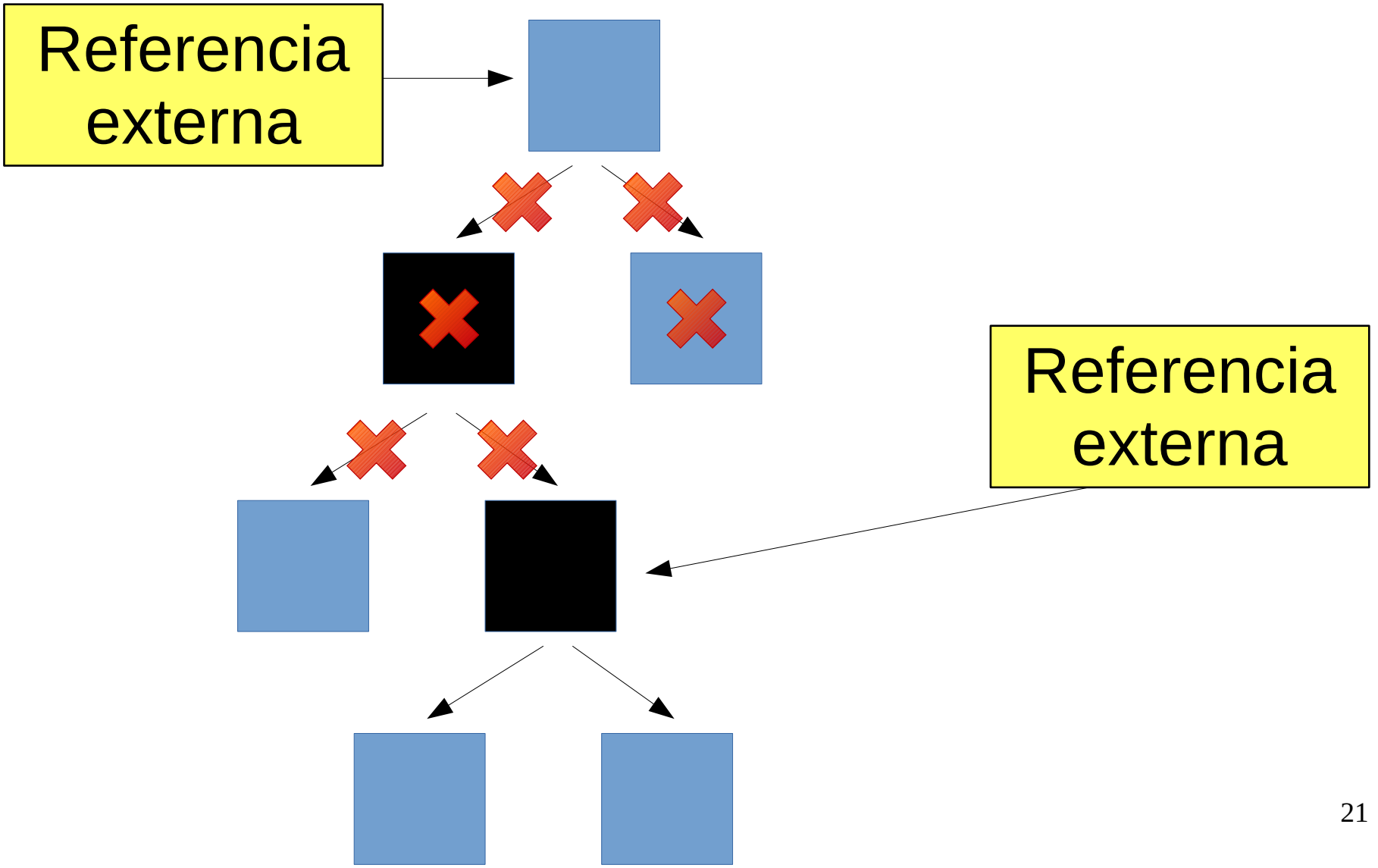
Persistencia en Python
Propuesta simple 4 (IX)



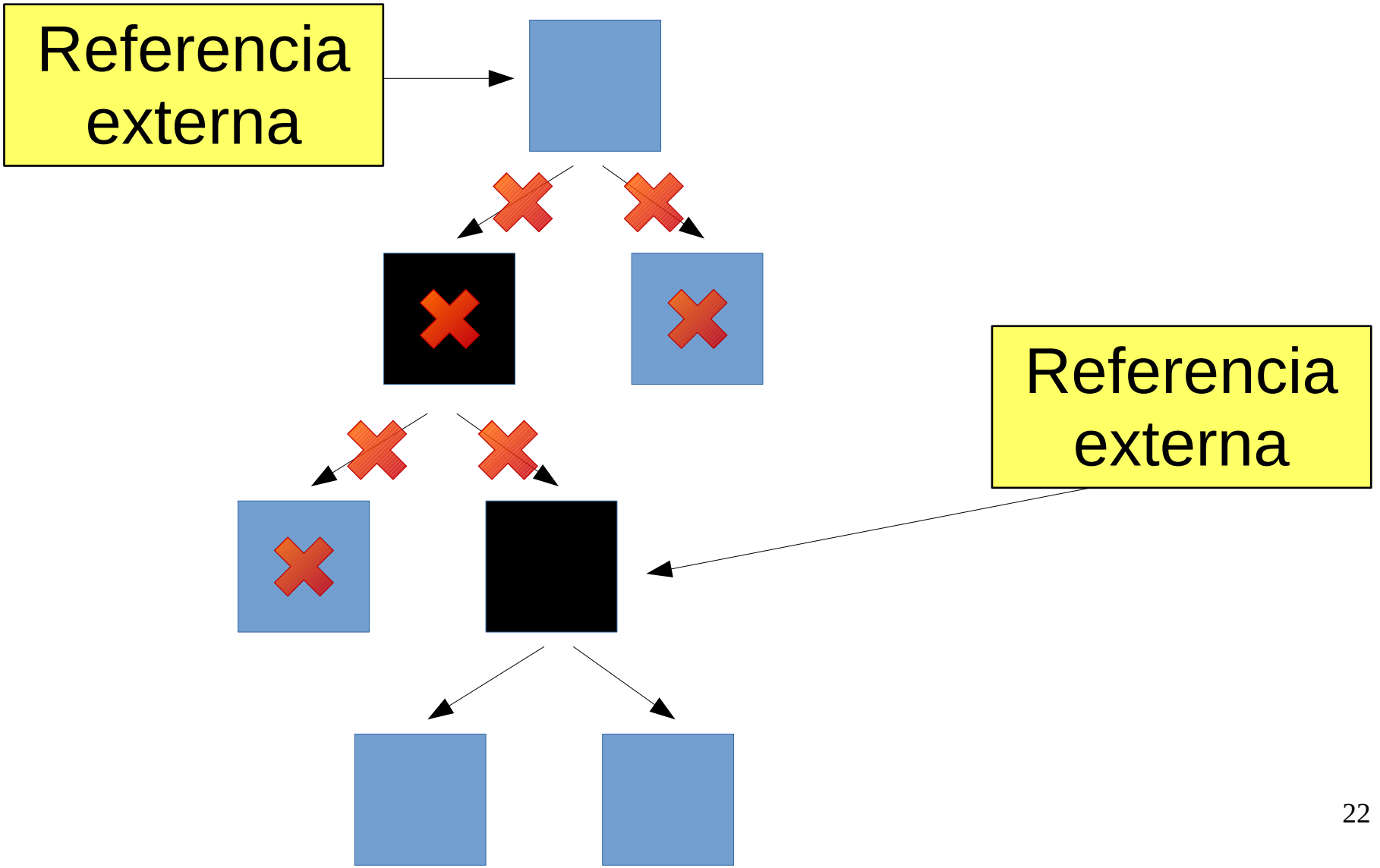
Persistencia en Python
Propuesta simple 4 (X)



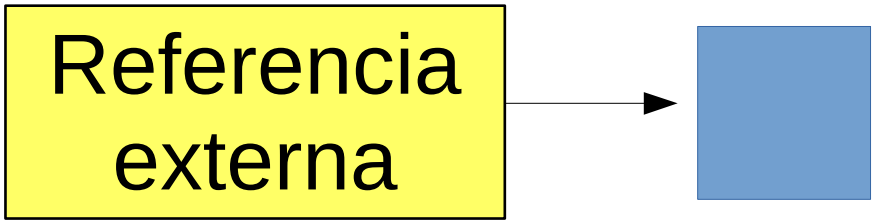
Persistencia en Python
Propuesta simple 4 (XI)

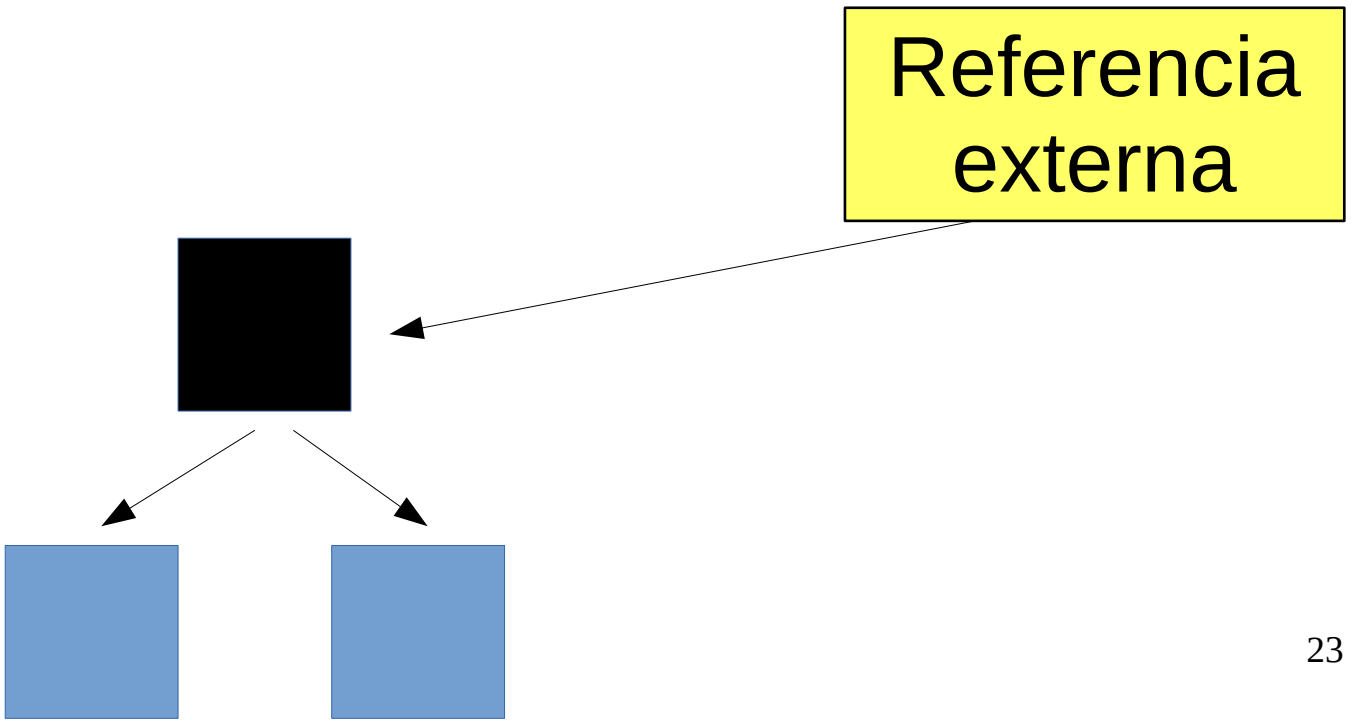


Persistencia en Python
Propuesta simple 4 (XII)



Persistencia en Python
Propuesta simple 4 (XIII)





Persistencia en Python
Propuesta simple 5 (I)

¿Y la concurrencia entre procesos?

Invalidaciones a través de concurrencia optimista

Persistencia en Python
Propuesta simple 5 (II)

- A la hora del “commit”, se comprueba qué objetos han sido modificados desde que empezamos la transacción.
- Los objetos modificados en el almacenamiento se convierten en “ghost”.
- Si alguno de los objetos modificados ha intervenido en la transacción actual (tanto en lectura como en escritura), la transacción se aborta, se invalidan los objetos (“ghost”) y se vuelve a repetir.
- En casos patológicos con mucha contención: bloqueos, reservas o subdivisión.
  - Contador distribuído.

Persistencia en Python
Propuesta simple 5 (III)

```
[Llega petición]
while True :
  persistencia.sync() # usualmente rollback()
  try :
      [Haz la operación que sea]
      (no te preocupes del almacenamiento)
      persistencia.commit()
  except Exception :
      persistencia.rollback()
      raise
  except ConflictException :
      Continue  # Repetimos el bucle
  break
```

Persistencia en Python
Propuesta simple 5 (IV)
BTree persistente

- Para escalar a grandes cantidades de datos se define un nuevo tipo persistente: BTree.
- Un diccionario persistente con 10 millones de entradas comparado con un BTree con 10 millones de entradas.
- El uso es idéntico al diccionario persistente, pero mucho más eficiente si no necesitamos acceder a todos los elementos a la vez o solo queremos modificar unos pocos.
- Algunos métodos adicionales.

Persistencia en Python
Ejemplo (I)

```
[root@babylon5 /]# durus -c --address=/var/correo_electronico/socket_unix
Durus /var/correo_electronico/socket_unix (rwxr-xr-x correo correo)
    connection -> the Connection
    root       -> the root instance
>>> root
<PersistentDict 0>
>>> root.items()
[('claves', <BTree 1>), ('buzones', <BTree 2>), ('vacation_queue', <BTree
3>), ('version', '2011072000'), ('pop3_before_smtp', <BTree 4>),
('to_delete', <PersistentList 5129632>), ('boletines', <BTree 5>),
('expiration', <BTree 1135872>)]
>>> root["buzones"]["jcea@jcea.es"]
<PersistentDict 159>
>>> root["buzones"]["jcea@jcea.es"].items()
[('TLS', False), ('pop3_locked_until', 0), ('num_mensajes_unread', 2),
('total_accumulated_mensajes', 3915976), ('size_unread', 22671),
('mensajes', <BTree 160>), ('last_pop3', 1424885041.678121), ('SSL',
True), ('pop3_must_delete', <BTree 161>), ('num_mensajes', 2),
('metamensajes', <BTree 1186993>), ('size', 22671)]
>>> root["buzones"]["jcea@jcea.es"]["mensajes"]
<BTree 160>
```

Persistencia en Python
Ejemplo (II)

```
>>> root["buzones"]["jcea@jcea.es"]["mensajes"].get_min_item()
(3915990, <PersistentDict 14323941>)
>>> root["buzones"]["jcea@jcea.es"]["mensajes"].get_min_item()[1]
<PersistentDict 14323941>
>>> root["buzones"]["jcea@jcea.es"]["mensajes"].get_min_item()[1].items()
[('uidl', 'ee6cc84dca07f1ebf8b015c6fd271fd3'), ('time',
1424885298.271245), ('mensaje', <BTree 14323942>), ('size', 17178)]
>>> root["buzones"]["jcea@jcea.es"]["mensajes"].get_min_item()[1]
["mensaje"].keys()
[0, 1853, 6139, 10456, 14698]
```

Persistencia en Python
Ejemplo (III)

UN PROCESO REALIZA MODIFICACIONES EN LOS OBJETOS DE FORMA CONCURRENTE

```
>>> root["buzones"]["jcea@jcea.es"]["mensajes"].get_min_item()
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "build/bdist.solaris-2.10-i86pc/egg/durus/btree.py", line 432, in get_min_item
    assert self, 'empty BTree has no min item'
  File "build/bdist.solaris-2.10-i86pc/egg/durus/btree.py", line 331, in __nonzero__
    return bool(self.root.items)
  File "build/bdist.solaris-2.10-i86pc/egg/durus/persistent.py", line 174, in
_p_load_state
    self._p_connection.load_state(self)
  File "build/bdist.solaris-2.10-i86pc/egg/durus/connection.py", line 185, in
load_state
    pickle = self.get_stored_pickle(oid)
  File "build/bdist.solaris-2.10-i86pc/egg/durus/connection.py", line 117, in
get_stored_pickle
    self._handle_invalidations(invalid_oids, read_oid=oid)
  File "build/bdist.solaris-2.10-i86pc/egg/durus/connection.py", line 308, in
_handle_invalidations
    raise ReadConflictError([read_oid])
ReadConflictError: oids=[14320155]
```

Persistencia en Python
Ejemplo (IV)

```
[root@babylon5 /]# durus -c
[...]
    connection -> the Connection
    root       -> the root instance

>>> root["prueba"]="Python Madrid!"
>>> connection.commit()
```

```
[root@babylon5 /]# durus -c
[...]
    connection -> the Connection
    root       -> the root instance

>>> print root["prueba"]
Python Madrid!
>>> del root["prueba"]
>>> connection.commit()
```

```
>>> print root["prueba"]
Python Madrid!
>>> root["prueba"] = "hola ke ase!"
>>> connection.commit()
Traceback (most recent call last):
  File "<console>", line 1, in [….]
WriteConflictError(conflicts)
WriteConflictError: oids=[0]
>>> connection.abort()
>>> root.keys()
['claves', 'buzones',
'vacation_queue', 'version',
'pop3_before_smtp', 'to_delete',
'boletines', 'expiration']
```

Persistencia en Python
Sistemas de persistencia

ZODB

http://www.zodb.org/en/latest/

Durus

https://www.mems-exchange.org/software/DurusWorks/

Persistencia en Python
Conclusiones

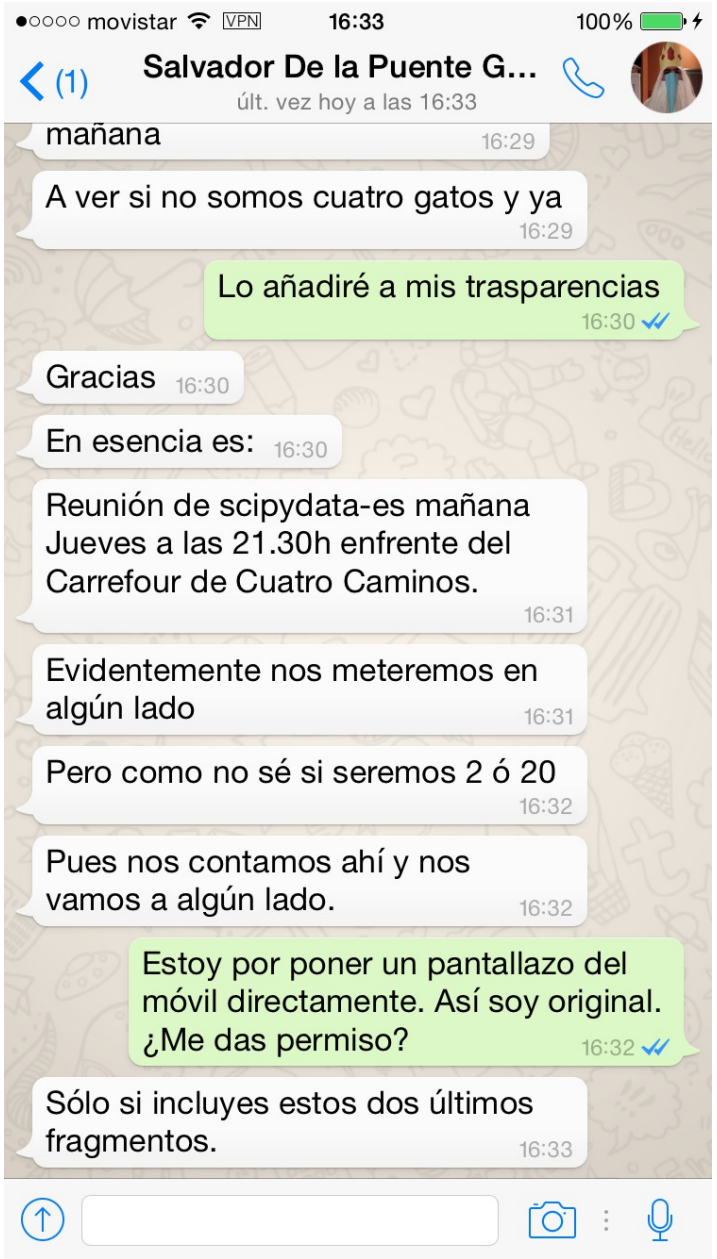
- La persistencia te abstrae del almacenamiento. Tu programa no se preocupa de ello.
- No mezclas paradigmas diferentes: Python con SQL por medio, Python con ORM.
- Sólo necesitas un sistema de almacenamiento clave → valor transaccional (***fichero gestionado***, Lightning DB, Berkeley DB, SQL sin aprovechar SQL).
  - ***El fichero gestionado*** es básicamente un log de cambios con un índice. Ideal para pocas escrituras.

Persistencia en Python
Sección anuncios (I)

Hoy se ha publicado Python 3.4.3

(Ya tardáis en actualizar)

Persistencia en Python
Sección anuncios (II)



Reunión de SCIPYDATA-ES **mañana**

Persistencia en Python
¿Preguntas?

- Migración de objetos de Python 2 a Python 3.
- Actualización del software.
- Versionado de objetos cuando evoluciona su código.
- Explicar más sobre los objetos “ghost”.
- Compatibilidad con otros lenguajes.
- Conocimiento interno detallado.
- Mantenimiento de las librerías.
- objeto.atributo = objeto.atributo.
- Pitfalls: Know How Durus 3.6: https://www.jcea.es/artic/know_how-durus-3_6.htm