PYTHON NOTES

with internals

Yang Bo

RDBJ
Admaster

2015.5.25

Outline of the talk

- Python Object
- Built-in Types
- Memory Management
- Reference Count and GC
- ENV: Python in this slides means CPython2.7.6

Python Object

```
#define PyObject_HEAD                   \
    _PyObject_HEAD_EXTRA                \
    Py_ssize_t ob_refcnt;               \
    struct _typeobject *ob_type;

typedef struct _object {
    PyObject_HEAD
} PyObject;

#define PyObject_VAR_HEAD               \
    PyObject_HEAD                       \
    Py_ssize_t ob_size; /* Number of items in variable part */

typedef struct {
    PyObject_VAR_HEAD
} PyVarObject;
```

object.h

Python Object

- Foundations of Everything(interally)
- Not change size once created
- PyObject / PyVarObject
- Reference Count
- Type Info: _teypeobject
- Size Info
- Layout

Python Object

- inheritance
- polymorphism using PyObject
- use PyObject * not PyXXObject * when communicating

```
long PyObject_Hash(PyObject *v) {
    PyTypeObject *tp = v->ob_type;
    if (tp->tp_hash != NULL)
        return (*tp->tp_hash)(v);
    ......
}
```

object.c

Python Object

```
#define _Py_Dealloc(op) (                                       \
    _Py_INC_TPFREES(op) _Py_COUNT_ALLOCS_COMMA                  \
    (*Py_TYPE(op)->tp_dealloc)((PyObject *)(op)))
#endif /* !Py_TRACE_REFS */

#define Py_INCREF(op) (                                         \
    _Py_INC_REFTOTAL  _Py_REF_DEBUG_COMMA                       \
    ((PyObject*)(op))->ob_refcnt++)

#define Py_DECREF(op)                                           \
    do {                                                        \
        if (_Py_DEC_REFTOTAL  _Py_REF_DEBUG_COMMA               \
        --((PyObject*)(op))->ob_refcnt != 0)                    \
            _Py_CHECK_REFCNT(op)                                \
        else                                                    \
        _Py_Dealloc((PyObject *)(op));                          \
    } while (0)
```

object.h

Python Object/object and type

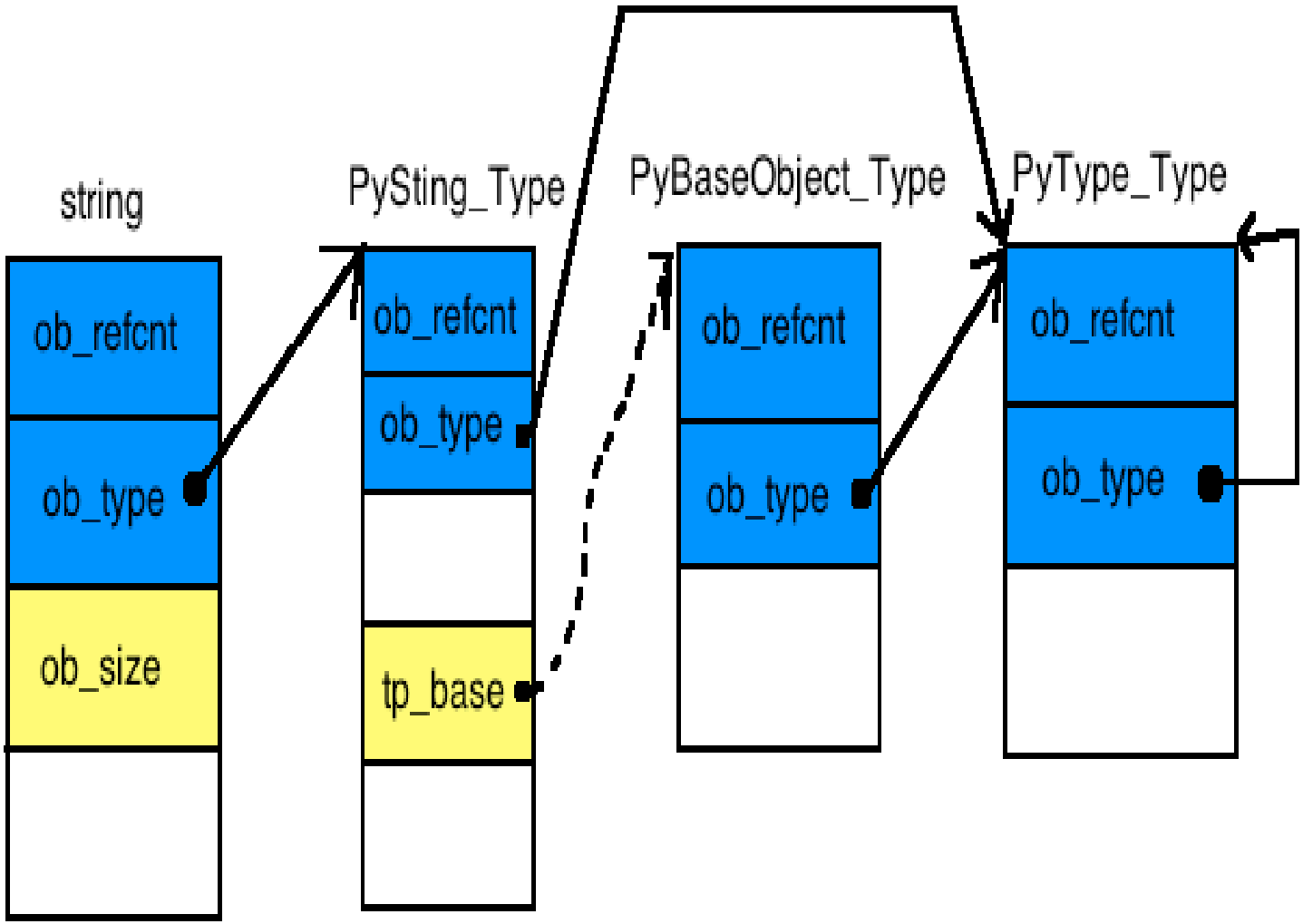
<type 'object'> and <type 'type'>

- 'object' in python is PyBaseObject_Type indeed, not PyObject
- 'object' is base class of everything, including 'type'
- type of all types is 'type'
- type of 'object' is 'type'

```
PyAPI_DATA(PyTypeObject) PyType_Type; /* built-in 'type' */
PyAPI_DATA(PyTypeObject) PyBaseObject_Type; /* built-in 'object
    ' */
```

object.h

Python Object/example



Built-in Types/Integer

```
typedef struct {
    PyObject_HEAD
    long ob_ival;
} PyIntObject;
```

intobject.h

Built-in Types/Integer

- cache small: [-5, 257)
- cache block: PyIntBlock, each 82
- block_list and free_list
- gc: int_dealloc
- memory usage: largest quantity of integers at one exact time, not how many integers totally had in history

Built-in Types/Integer

```
In [6]: %timeit float("1233")
The slowest run took 13.29 times longer than the fastest. This
    could mean that an intermediate result is being cached
1000000 loops, best of 3: 233 ns per loop

In [7]: %timeit int("1233")
The slowest run took 5.72 times longer than the fastest. This
    could mean that an intermediate result is being cached
1000000 loops, best of 3: 708 ns per loop
```

Why? Let's observe int_new(objects/intobjects.c) and float_new(objects/floatobjects.c).

Built-in Types/String

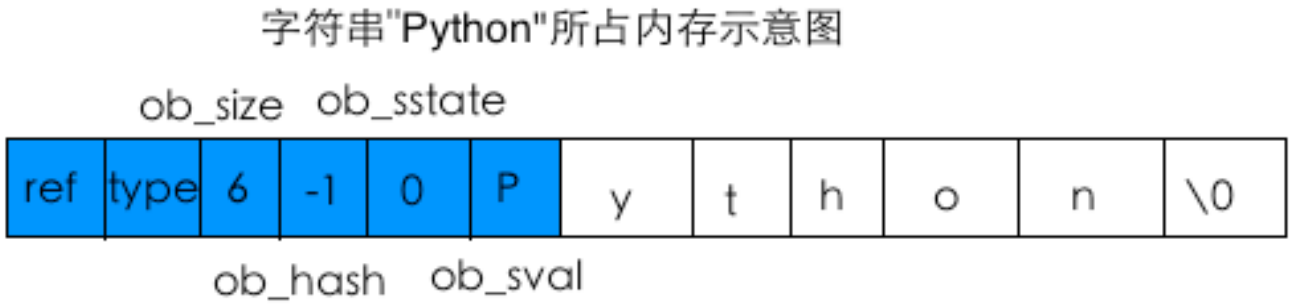
```
typedef struct {
    PyObject_VAR_HEAD
    long ob_shash; // init as -1
    int ob_sstate; // whether interned
    char ob_sval[1]; //trick , 'cos this is not only for gnu c
} PyStringObject;
```

stringobject.h

Built-in Types/String

- not changable.
  - the good: used as key in dict
  - the bad: efficiency in join
- you can have \0 in your string, for size is determined by ob_size and tp_itemsize
- PyString_FromString PyString_FromStringAndSize
- intern

Built-in Types/String



Built-in Types/String

intern

- a cache mechanism, working with hash can speed up python by 20%
- happens at compile time, not run time
- by default, only cache strings consist of numbers, alphabets and underscores, but built-in intern() can be used to cache any string.
- internaly, use a dict <PyObject *, PyObject *> named 'interned' to store, and the two refs in interned don't count.
- nullstring and characters array use intern
- names like "__name__", "__doc__" are interned
- see implementations in codeobject.c stringobject.c

Built-in Types/String

efficiency of "+="/"+" and "join"

- string_join allocate memory once for all the items
- += and + allocate memory everytime it executes, but optimized for short string, which resize the lvalue. see BINARY_ADD, INPLACE_ADD, string_concatenate in Python/ceval.c
- older version of concatenate is string_concat in Object/stringobject.c which malloc a new memory

PEP8: do not rely on CPython's efficient implementation of in-place string concatenation for statements in the form a += b or a = a + b . This optimization is fragile even in CPython (it only works for some types) and isn't present at all in implementations that don't use refcounting.

Built-in Types/String

stringlib fastsearch

- based on a mix between boyer-moore and horspool
- 'in' operation

Built-in Types/List

```
typedef struct {
    PyObject_VAR_HEAD
    /* Vector of pointers to list elements.  list[0] is ob_item
       [0], etc. */
    PyObject **ob_item;
    /* ob_item contains space for 'allocated' elements.  The
       number * currently in use is ob_size.
     * Invariants:
     *     0 <= ob_size <= allocated
     *     len(list) == ob_size
     *     ob_item == NULL implies ob_size == allocated == 0
     * list.sort() temporarily sets allocated to -1 to detect
       mutations.
     * Items must normally not be NULL, except during
       construction when * the list is not yet visible
       outside the function that builds it.  */
    Py_ssize_t allocated;
} PyListObject;
```

listobject.h

Built-in Types/List

- ob_size and allocated is like size and capacity of vector in c++
- only one method to init: PyList_New
- cache mechanism: free_list, which hold 80 objects
- list_dealloc to return PyListObject to free_list
- memory management in list_resize:
  - newsize $<$ allocated && newsize $>$ allocated/2, no need to realloc
  - (0, 4, 8, 16, 25, 35, 46, 58, 72, 88)
  - otherwise shrink or enlarge

Built-in Types/List

misc notes

- insert when $>$ len(list) or $<$ -len(list)
- xrange / range
- tim sort
- consider array
- consider tuple
  - allocate once
  - cache 2000, each $<$20 elements
  - read-only, better fit for concurrency

Built-in Types/Dict

```
typedef struct _dictobject PyDictObject;
struct _dictobject {
    PyObject_HEAD
    Py_ssize_t ma_fill;  /* # Active + # Dummy */
    Py_ssize_t ma_used;  /* # Active */
    Py_ssize_t ma_mask;
    PyDictEntry *ma_table;
    PyDictEntry *(*ma_lookup)(PyDictObject *mp, PyObject *key,
        long hash);
    PyDictEntry ma_smalltable[PyDict_MINSIZE];
};
```

dictobject.h

Built-in Types/Dict

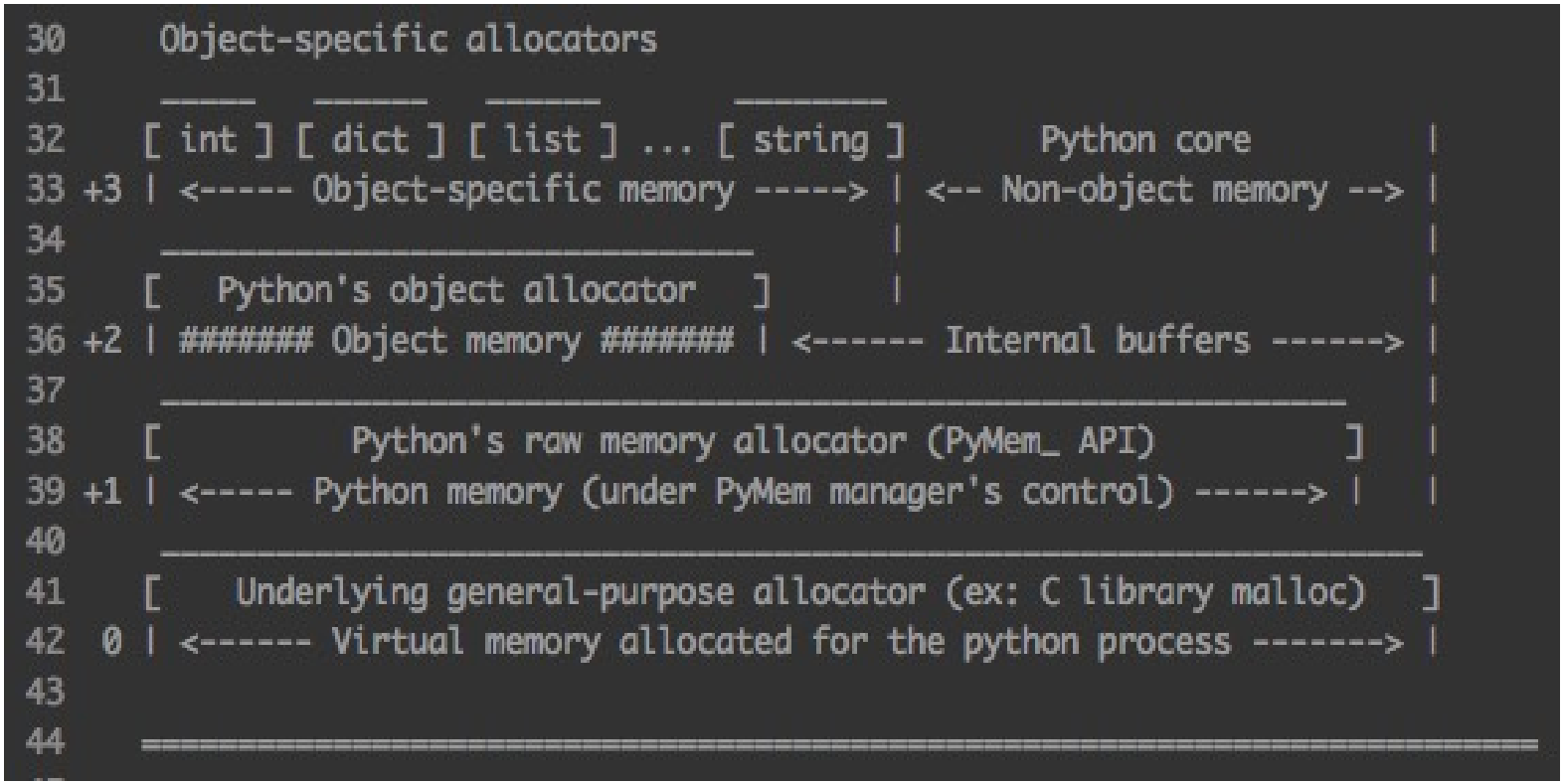
```
typedef struct {
    /* Cached hash code of me_key.  Note that hash codes are C
       longs.
     * We have to use Py_ssize_t instead because dict_popitem()
         abuses
     * me_hash to hold a search finger.
     */
    Py_ssize_t me_hash;
    PyObject *me_key;
    PyObject *me_value;
} PyDictEntry;
```

dictobject.h

Built-in Types/Dict

- hash table with open addressing
- unused/active/dummy
- ma_smalltable and ma_table
- lookdict and lookdict_string
- PyDict_New
- cache mechanism: free_list, which hold 80 objects
- memory management in dictresize:
  - if fill $>$ 2/3 size, adjust size.
  - size * 2 or size * 4

Memory Allocators



Memory Allocators

- Layer 0 is OS's allocators
- Layer 1 is just a simple wrapper of Layer 0, in forms of both macro and funtion
- Layer 2 is the core mechanism–pymalloc. PyObject_{Malloc,Realloc,Free}
- Layer 3 is type specific memory cache

pymalloc

- block: works for <256B, when >256B, use Layer 1. block must be 8B alignment.
- pool: same as mem page size(4KB), manage blocks of same size
- arena: has 64 pools.
- usedpools: array of head pointers of USED pool

Mark-Sweep Garbage Collection

- break unreachable reference cycles
- doesn't explicity deallocate any objects, just breaks cycles
- automaticlly run by interpreter every once in a while. sys.getcheckinterval()
- auto gc object without ___del___
- stop the world to gc
- expensive, cost is linear to number of references of vector in program.

Generational GC

- most objects live either for a very short time or for a very long time
- divide object into 3 generations(0, 1, 2)
- each new object is generation 0.
- n-th gen. gc analyses objects of gen. 0-n
- any object that survived n-th gen. gc is promoted to n+1-th gen.
- (700, 10, 10) 700 is alloc - dealloc, 10s are times of prior gen gc.

Memory Leak

- leak in modules written in c/c++
- integers and floats
- gc won't collect any objects in cycles where at least one object has ___del___
- hidding references

Memory Leak

Hidding References

- closures, functools.partial, etc
- sys.exc_traceback keeps last exception handled in this stack frame including the whole stack state when exception occurs.
- sys.last_traceback keeps unhandled excption including whole stack state

What to do

- use context manager("with") instead of __del__
- use weakref
- if you really need __del__, be careful of ref cycle
- tools: pdb/objgraph/guppy/heapy/memory_profiler/line_profiler/gc.collect()

Q & A