Playing with CPython (3.4) Objects Internals

JESUS ESPINO GARCIA, DEVELOPER
Introduction
What I want?
I want to play with python

100 + 2 == 103
I want to play with python

True == False
I want to play with python

truncatet((1, 2, 3)) == (1, 2)
Objects
Object

- Object == instance.
- C Structs with data.
- A block of reserved memory with data in it.
- Has a type (and only one) that defines its behavior.
- The objects type doesn’t change during the lifetime of the object (with exceptions).
● Every object has an ID (which is the address in memory)
● Every object has a reference counter, and when it reaches 0, the object memory is freed.
Basic structure

- `ob_refcnt`: reference counter.
- `ob_type`: pointer to the type object.
- `...`: Any extra data needed by the object.
The None Object
None structure

• Is the simplest object in python.
• Doesn’t need extra data.
• It’s a singleton object for all the CPython interpreter.
Examples

All my examples start with this code

```python
>>> import ctypes
>>> longsize = ctypes.sizeof(ctypes.c_long)
>>> intsize = ctypes.sizeof(ctypes.c_int)
>>> charsize = ctypes.sizeof(ctypes.c_char)
```
>>> ref_cnt = ctypes.c_long.from_address(id(None))
>>> ref_cnt.value = 0
Fatal Python error: deallocating None
Current thread 0x00007f2fb8d2a700:
File "<stdin>", line 1 in <module>
[2]
10960 abort (core dumped) python3
The int Object
int structure

- `ob_refcnt`
- `ob_type`
- `ob_size`
- `ob_digit[]`

- `ob_size`: stores the number of digits used.
- `ob_digit`: Is an array of integers.
- The value is $\sum \text{ob_digit}[\text{position}] \times (1024^3)^{\text{position}}$
int examples

7

1024^3

ob_refcnt
ob_type
ob_size
ob_digit[]
Accessing int

```python
>>> x = 100
c_long(1)
>>> ctypes.c_uint.from_address(id(x) + longsize * 3)
c_uint(100)
>>> x = 1024 * 1024 * 1024
c_long(2)
>>> ctypes.c_uint.from_address(id(x) + longsize * 3)
c_uint(0)
>>> ctypes.c_uint.from_address(id(x) + longsize * 3 + intsize)
c_uint(1)
```
Very bad things

```python
>>> x = 1000
>>> int_value = ctypes.c_uint.from_address(id(x) + longsize * 3)
>>> int_value.value = 1001
>>> x
1001
>>> 1000
1000
```
Very bad things

```python
>>> x = 100
>>> int_value = ctypes.c_uint.from_address(id(x) + longsize * 3)
>>> int_value.value = 101
>>> x
101
101
>>> 100
101
>>> 100 + 2
103
```
The bool Object
bool structure

- True
  - ob_refcnt
  - ob_type
  - ob_size: 1
  - ob_digit: [ ]

- False
  - ob_refcnt
  - ob_type
  - ob_size: 0
  - ob_digit: [ ]

- Two integer instances.
- True with ob_size and ob_digit equals to 1.
- False with ob_size and ob_digit equals to 0.
Accessing bool

```python
>>> ctypes.c_long.from_address(id(True) + longsize * 2)
c_long(1)
>>> ctypes.c_uint.from_address(id(True) + longsize * 3)
c_uint(1)
>>> ctypes.c_long.from_address(id(False) + longsize * 2)
c_long(0)
>>> ctypes.c_uint.from_address(id(False) + longsize * 3)
c_uint(0)
```
>>> val = ctypes.c_int.from_address(id(True) + longsize * 2)
>>> val.value = 0
>>> val = ctypes.c_int.from_address(id(True) + longsize * 3)
>>> val.value = 0
>>> True == False
True
Very bad things

```python
>>> ctypes.c_long.from_address(id(True) + longsize)
c_long(140477915154496)
>>> id(bool)
140477915154496
>>> type_addr = ctypes.c_long.from_address(id(True) + longsize)
>>> type_addr.value = id(int)
>>> True
1
```
The bytes Object
bytes structure

- ob_size: Stores the number of bytes.
- ob_shash: Stores the hash of the bytes or -1.
- ob_sval: Array of bytes.
Accessing bytes

```python
>>> x = b"yep"
>>> ctypes.c_long.from_address(id(x) + longsize * 2)
c_long(3)
>>> hash(x)
954696267706832433
>>> ctypes.c_long.from_address(id(x) + longsize * 3)
c_long(954696267706832433)
>>> ctypes.c_char.from_address(id(x) + longsize * 4)
c_char(b'y')
>>> ctypes.c_char.from_address(id(x) + longsize * 4 + charsize)
c_char(b'e')
>>> ctypes.c_char.from_address(id(x) + longsize * 4 + charsize * 2)
c_char(b'p')
>>> ctypes.c_char.from_address(id(x) + longsize * 4 + charsize * 3)
c_char(b'\x00')
```
The tuple Object
tuple structure

- ob_refcnt
- ob_type
- ob_size
- ob_item[]

- ob_size: Stores the number of objects in the tuple.
- ob_item: Is an array of pointers to python objects.
(True, False) →

- `ob_refcnt`
- `ob_type`
- `ob_size`

**id**(True) **id**(False)
Accessing tuple

```python
>>> x = (True, False)
>>> ctypes.c_long.from_address(id(x) + longsize * 2)
c_long(2)
>>> ctypes.c_void_p.from_address(id(x) + longsize * 3)
c_void_p(140048684311616)
>>> ctypes.c_void_p.from_address(id(x) + longsize * 4)
c_void_p(140048684311648)
>>> id(True)
140048684311616
>>> id(False)
140048684311648
```
>>> x = (1, 2, 3)
>>> tuple_size = ctypes.c_long.from_address(id(x) + longsize * 2)
>>> tuple_size.value = 2
>>> x
(1, 2)
The list Object
list structure

- ob_size: Stores the number of objects in the list.
- ob_item: Is a pointer to an array of pointers to python objects.
- allocated: Stores the quantity of reserved memory.
list example

[True, False]

- ob_refcnt
- ob_type
- ob_size
- **ob_item
- allocated

id(True)

id(False)
Accessing list

```python
>>> x = [1,2,3]
c_long(3)
>>> ctypes.c_void_p.from_address(id(x) + longsize * 3)
c_void_p(36205328)
>>> ctypes.c_void_p.from_address(36205328 + longsize)
c_void_p(140048684735072)
>>> id(1)
140048684735040
>>> ctypes.c_void_p.from_address(36205328 + longsize)
c_void_p(140048684735072)
>>> id(2)
140048684735072
```
Very bad things

```python
>>> x = [1,2,3,4,5,6,7,8,9,10]
>>> y = [10,9,8,7]
>>> data_y = ctypes.c_long.from_address(id(y) + longsize * 3)
>>> data_x = ctypes.c_long.from_address(id(x) + longsize * 3)
>>> data_y.value = data_x.value
>>> y
[1, 2, 3, 4]
>>> x[0] = 7
>>> y
[7, 2, 3, 4]
```
The dict Object
- ma_used: Stores the number of keys in the dict.
- ma_keys: Is a pointer to a dict’s key structure.
- ma_values: Is a pointer to an array of pointers to python objects (only used in splitted tables).
**dict keys structure**

- `ob_refcnt`: Reference counter.
- `ob_type`: Type of object.
- `ma_used`: Used memory.
- `*ma_keys`: Pointer to keys array.
- `**ma_values`: Pointer to values array.
- `dk_refcnt`: Reference counter.
- `dk_size`: Total size of the hash table.
- `dk_lookup`: Slot for search function.
- `dk_usable`: Usable fraction of the dict before a resize.
- `dk_entries[]`: An array of entries entry structures.
- `me_hash`: Hash value.
- `me_key`: Key value.
- `me_value`: Value associated with the key.
dict key entry structure

- me_hash: Hash of the key
- me_key: Pointer to the key python object.
- me_value: Pointer to the value python object.
dict example (combined tables)

\{1: 2, 3: 4\}
class Test:
    pass

test = Test()
test.a = 1
Accessing dict

```python
>>> d = {1: 3, 7: 5}
>>> keys = ctypes.c_void_p.from_address(id(d) + longsize * 3).value
>>> keyentry1 = keys + longsize * 4 + longsize * hash(1) * 3
>>> keyentry7 = keys + longsize * 4 + longsize * hash(7) * 3
>>> key1 = ctypes.c_long.from_address(keyentry1 + longsize).value
>>> val1 = ctypes.c_long.from_address(keyentry1 + longsize * 2).value
>>> key7 = ctypes.c_long.from_address(keyentry7 + longsize).value
>>> val7 = ctypes.c_long.from_address(keyentry7 + longsize * 2).value
>>> ctypes.c_uint.from_address(key1 + longsize * 3)
c_long(1)
>>> ctypes.c_uint.from_address(val1 + longsize * 3)
c_long(3)
>>> ctypes.c_uint.from_address(key7 + longsize * 3)
c_long(7)
>>> ctypes.c_uint.from_address(val7 + longsize * 3)
c_long(5)
```
Extra ball
Changing integer __add__ globally

```python
>>> from ctypes import *
>>> MYFUNCTYPE = CFUNCTYPE(py_object, py_object, py_object)
>>> @MYFUNCTYPE
>>> def my_add(x, y):
...    return 42
>>> my_add_address = ctypes.c_long.from_address(id(my_add) + 8 * 10)
>>> int_address = id(int)
>>> as_number_address = ctypes.c_long.from_address(int_address + 8 * 12)
>>> add_address = ctypes.c_long.from_address(as_number_address.value)
>>> add_address.value = my_add_address.value
>>> refcnt = ctypes.c_long.from_address(id(42))
>>> refcnt.value = refcnt.value + 1
>>> print(1 + 1)
42
```
References

- Python Code: Include and Objects
- CTypes documentation: http://docs.python.org/3/library/ctypes.html
- Python C-API documentation: http://docs.python.org/3/c-api/index.html
- PEP 412 – Key-Sharing Dictionary
- Access examples code: http://github.com/jespino/cpython-objects-access
- Very bad things code: http://github.com/jespino/cpython-very-bad-things
Conclusions
Conclusions

- CPython objects are simple.
- Can be funny to play with the interpreter.
- Don’t fear the CPython source code.
Q & A