List, Tuple, Dict under the hood

stoworow

stoworow@gmail.com

PyGDA 29 May 2016

Agenda

1. Lists
2. Tuples
3. Dicts
4. Miscellaneous

List

List - creating empty

```
>>> import sys
>>> my_list = []
>>> print sys.getsizeof(my_list)
72
```

sys.getsizeof()

Return the size of an object in bytes. The object can be any type of object. All built-in objects will return correct results, but this does not have to hold true for third-party extensions as it is implementation specific.

List - appending

```
>>> my_list.append(1)
>>> print sys.getsizeof(my_list)
104
```

List - appending

```
>>> my_list.append(2)
>>> print sys.getsizeof(my_list)
104
```

List - appending

```
>>> my_list.append(3)
>>> my_list.append(4)
>>> my_list
[1, 2, 3, 4]
>>> sys.getsizeof(my_list)
104
>>> my_list.append(5)
>>> sys.getsizeof(my_list)
136
```

List - removing

```
>>> my_list
[1, 2, 3, 4, 5]
>>> sys.getsizeof(new_list)
136
>>> my_list.pop()
>>> my_list
[1, 2, 3, 4]
>>> sys.getsizeof(new_list)
136
```

List - removing

```
>>> my_list.pop()
>>> my_list
[1, 2, 3]
>>> sys.getsizeof(new_list)
120
>>> my_list.pop()
>>> sys.getsizeof(new_list)
112
>>> my_list
[1]
>>> sys.getsizeof(new_list)
104
```

Allocation equation

$$new\_allocated = (newsize >> 3) + (newsize < 9?3 : 6)$$

N	0	1-4	5-8	9-16	17-25	26-35	36-46	...	991-1120
M	0	4	8	16	25	35	46	...	1120

Python's interpreter code note

The over-allocation is mild, but is enough to give linear-time amortized behavior over a long sequence of appends() in the presence of a poorly-performing system realloc().

List - defining non-empty

```
>>> new_list = [1,2,3]
>>> sys.getsizeof(new_list)
96
>>> my_list.append(4)
>>> sys.getsizeof(new_list)
128
```

List - dealing with overallocation

```
>>> new_list = []
>>> new_list.extend([1,2,3,4,5,6,7,8,9,10])
>>> sys.getsizeof(new_list)
208
>>> new_list = [None] * 10
>>> sys.getsizeof(new_list)
152
```

List - max size

```
>>> import sys
>>> print sys.maxsize
9223372036854775807
```

PY_SSIZE_T_MAX / sizeof(PyObject)

PY_SSIZE_T_MAX is defined in pyport.h to be ((size_t) -1)>>1

List

Summary:

1. On a 64-bit machine empty list occupy 72 B of memory.
2. List pre-allocate some memory to be able to perform quick appends.
3. Every "slot" occupy the same amount of memory (it'll be only a reference to actual object which is put in list) - 8 B.
4. For C code look at: https://hg.python.org/cpython/file/273e17260d25/Objects/listobject.c#l12

Tuple

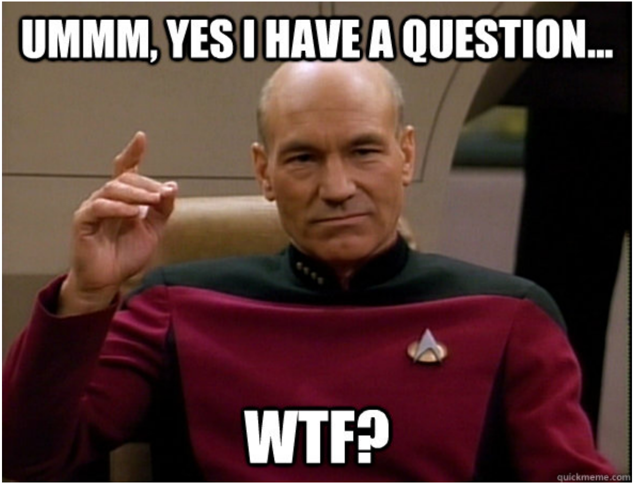
Tuple

```
>>> my_tuple = ()
>>> sys.getsizeof(my_tuple)
56
```

Tuple

```
>>> my_tuple = (1)
>>> sys.getsizeof(my_tuple)
24 !
>>> my_tuple = ('1')
>>> sys.getsizeof(my_tuple)
38 !
```

Tuple



Tuple

```
>>> sys.getsizeof(int)
24
>>> sys.getsizeof(str)
38
```

Tuple

```
>>> my_tuple = (1, 2)
>>> sys.getsizeof(my_tuple)
72
>>> my_tuple = (1, 2, 3)
>>> sys.getsizeof(my_tuple)
80
>>> my_tuple = (1, 2, 3, 4)
>>> sys.getsizeof(my_tuple)
88
>>> my_tuple = ('1', '2', '3', '4')
>>> sys.getsizeof(my_tuple)
88
```

Tuple

Summary:

1. One element tuple is a reference to particular object (still unmutable) thus actual size is lower than empty tuple (no need for whole container overhead).
2. Every chunk of memory in a tuple costs 8 B of memory.

Dict

Dict

```
>>> my_dict = {}
>>> sys.getsizeof(my_dict)
280
```

Dict

```
>>> my_dict['key1'] = 1
>>> sys.getsizeof(my_dict)
280
>>> my_dict['key2'] = 1
>>> my_dict['key3'] = 1
>>> my_dict['key4'] = 1
>>> my_dict['key5'] = 1
>>> sys.getsizeof(my_dict)
280
```

Dict

```
>>> my_dict['key6'] = 1
>>> sys.getsizeof(my_dict)
1048
```

Dict flow

1. Empty dict allocate 8 chunks of memory at the beginning.
2. Every time insert operation is performed dict checks whether it needs to be re-sized.
3. Re-sizing takes place when 2/3 of allocated memory is full.
4. When re-sizing it allocates 4x more memory if there are less than 50.000 elements.
5. When dict contains more than 50.00 it re-sizes 2x.
6. 8, 32, 128, 512, 2048, 8192, 32768, 131072, 262133, ...

Dict - fitting element

1. Assume calculate hash = 12345
2. index = hash AND mask
3. mask = allocated elements - 1
4. 0b001 = 0b0011000000111001 AND 0b111
5. index = 1

Dict - collisions & probing

1. If index $= 1$ is empty - put pair of elements (key and value) here.
2. If index $= 1$ is not empty - search for new empty location.
3. new index $=$ index $+ 1$ (linear probing).

Dict - re-sizing

1. During re-sizing all elements are copied to a new dict.
2. Old calculated indexes may not correlate with the new ones (new mask).
3. Python use quite simple hashing algorithm making it fast - is probably faster to re-alloc memory and recalculate hashes rather than perform any complicated hashing.

Dict - removing elements

```
>>> my_dict.pop('key6')
>>> my_dict.pop('key5')
>>> my_dict.pop('key4')
>>> my_dict.pop('key3')
>>> sys.getsizeof(my_dict)
1048
```

Dict

Summary:

1. Dict re-allocates when it's 2/3 full.
2. Linear probing - technique for solving problem with collisions.
3. Re-sizing requires re-calculating all indexes.

Miscellaneous

Miscellaneous

Python 2.7 sizes of elements:
'decimal', 80
'dict', 280
'float', 24
'int', 24
'list', 72
'object', 16
'set', 232
'str', 38
'tuple', 56
'unicode', 56

Miscellaneous

Python 3.3 sizes of elements:
'decimal', 104
'dict', 296
'float', 24
'int', 24
'list', 72
'object', 16
'set', 232
'str', 50
'tuple', 56
'unicode', 50

References

1. "High Performance Python" - Micha Gorelick & Ian Ozsvald
2. http://www.laurentluce.com/posts/python-list-implementation/

Thank you