Chapter2
Python BuiltIn Types

By Weihan Jiang (updated 16 Nov 2015)

2.1 Numbers

Bool

In Python, None, 0, empty string, empty container will all be treated as False. Otherwise, it is True.

```
>>> map(bool, [None, 0, "", u"", list(), tuple(), dict(), set(), frozenset()])
```

```
[False, False, False, False, False, False, False, False, False]
```

A little bit weird, but False and True could be used as numbers in Python...

```
>>> int(True)

1

>>> int(False)

0

>>> int(3>1)

1

>>> lst = [1, 2, 3]

>>> lst[True]

2
```

Int

On 64-bit Machine, int if of type 64-bit int,

```
>>> sys.maxint

9223372036854775807
```

which makes int big enough to cover most of the scenarios.

Additionally, Python VM provided optimizations for int type,

- Python applies PyIntblock to store the int objects from heap.
- for small ints between [-5, 257], Python does not create new objects but using index to get int from the cached list.
- PyIntBlock will not be returned to OS, until “shutdown”(Pyint_fini)

if interested, the source code is
https://hg.python.org/cpython/file/4de09cbd3b97/Objects/intobject.c

Let’s see how big and small int behaves differently,

```
>>> a = 15
```

```
>>> b = 15

>>> a is b

True

>>> import sys

>>> sys.getrefcount(a)

15

>>> sys.getrefcount(b)

16

>>> a = 257

>>> b = 257

>>> a is b

False

>>> sys.getrefcount(a)

2
```

And because PyIntBlock does not get released, maintaining large amount of big int will create spike of memory consumption,

```
def test(): x=0

    for i in range(10000000):  #xrange

        x += i

return x
```

Below are the RSS result for range and xrange,

```
range: meminfo(rss=93339648L, vms=2583552000L    #80M

xrange: meminfo(rss=8638464L, vms=2499342336L       #8M
```

Note that, in Python3, there is xrange() has replaced range().

Long

When int continues growing beyond the sysmaxint limit, it will be cast automatically into long type.

```
>>> a = 9223372036854775807

>>> type(a)

<type 'int'>

>>> a += 1

>>> a

9223372036854775808L

>>> type(a)

<type 'long'>
```

As long as your memory can hold, long type could be bigger than you could think of...

```
>>> 1 << 3000

1230231922161117176931558813276752514640713895736833715766118029160058800614672948775360067838593459582429649254051804908512884180898236823585082482065348331234959350355845017413023320111360666922624728239756880416434478315693675013413090757208690376793296658810662941824493488451726505303712916005346747908623702673480919353936813105736620402352744776903840477883651100322409301983488363802930540482487909763484098253940728685132044408863734754271212592471778643949486688511721051561970432780747454823776808464180697103083861812184348565522740195796680
```

```
26222055118455120805520103100502558015893496459280011337454742207150136834139075427790637598338761013542351842450966700421607206294115815023712480084304471848420986103205804179922066622473287221220885136436839076703602091626536706411309369970021705006755013747239987660058275793007232534748906122501351718891748990799112915123997738721785190182299893761
```

Different from Int, Long type was not optimized by Python.

Float

A few points bear mentioning of Float type in Python.

To your surprise, Floating point numbers cannot handle small decimal numbers.

```
      >>> 3/2

1

>>> float(3)/2

1.5

>>> 3 * 0.1 == 0.3

False

>>> 3 * 0.1

0.30000000000000004

>>> round(2.657, 2)

2.66

In this case, try to use “Decimal” moduel,

>>> Decimal('0.1')*3 == Decimal('0.3')

True
```

Note that, similarly to Int, Python uses the same mechanism PyFloatBlock to cache floats. But Python does not cache list for small floats.

String

Things get complicated when touching “strings”, for example, “intern” and “encode”.

Some basic operations,

```
>>> 'a'+'b'
'ab'
>>> 'a'*3
'aaa'
>>> ''.join(['a', 'b', 'c'])
'abc'
>>> "a,b,c".split(',')
['a', 'b', 'c']
>>> "a\nb\nc".splitlines()
['a', 'b', 'c']
>>> "abc".startwith("ab")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'str' object has no attribute 'startwith'
>>> "abc".startwith("ab")
True
>>> "abc".upper()
'ABC'
>>> "abcde".find("bc")
1
```

```
>>> " abc".lstrip(), "abc ".rstrip(), " abc ".strip()

('abc', 'abc', 'abc')

>>> "abc".strip("ac")

'b'

>>> "abcabc".replace("bc", "BC")

'aBCaBC'
```

Python has implemented string Interning mechanism to avoid creating too many objects for string, which will eventually lead to have too many “__name__”, “__doc__” names. This will help reduce the memory consumption, and improve performance.

```
>>> "string" is "string"

True

>>> "strin" + "g" is "string"

True

>>> s1 = "strin"

>>> s2 = "string"
```

In the second example, the expression "strin"+"g" is evaluated at compile time, and is replaced with "string". This makes the first two examples behave the same.

The third example involves a run-time concatenation, the result of which is not automatically interned:

```
>>> s1 + "g" is s2

False

>>> s3 = s1 + "g"

>>> s3 is s2

False
```

If you were to manually intern() the result of the third expression, you'd get the same object as before:

```
>>> intern(s3) is s2

True
```

Let's try some problem solvings,

https://leetcode.com/problems/length-of-last-word/

Given a string *s* consists of upper/lower-case alphabets and empty space characters `' '`, return the length of last word in the string.

If the last word does not exist, return 0.

Note: A word is defined as a character sequence consists of non-space characters only.

For example,

Given *s* = `"Hello World"`,

return `5`.

List

Python's list is more like "Vector" rather than "Linked List".

- element object and its index is stored separately. The index is stored in index list on heap.
- index list is dynamically allocated (provisioned more space than the actual elements)

Creating lists,

```
>>> []          #empty list
```

```
[]
>>> ['a', 'b'] *3          #multiplying
['a', 'b', 'a', 'b', 'a', 'b']
>>> ['a', 'b'] + ['c']          #comcatinating
['a', 'b', 'c']
>>> list("abcd")                #casting iterables into list
['a', 'b', 'c', 'd']
>>> [x for x in xrange(10)]               #create list from
expression
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Common operations,

```
>>> l = list(xrange(10))
>>> l[2:-2]                    #slicing
[2, 3, 4, 5, 6, 7]
>>> l = list("abcabc")         #searching from given position, returning index
>>> l.index("a", 2)
3
>>> l = list("abc")
>>> l.append("d")              #appending element
>>> l
['a', 'b', 'c', 'd']
>>> l.insert(1, 100)           #insertion of lelement
>>> l
```

['a', 100, 'b', 'c', 'd']

>>> l.remove(100) #remove the first encountered matching element

>>> l

['a', 'b', 'c', 'd']

Performance, there nothing gained for free(of course!).

Python list uses C language realloc() to re-allocate the "index list" of a list. Copying, insertion, deleting, recursively moving of elements. There are potential compromises to performance.

So when we encounter large scale list, which requires frequent deletion/insertion of elements, we should turn to linked list.

Let's look at one example,

```
>>> import itertools, gc

>>> gc.disable()

>>> def test(n):

... return len([0 for i in xrange(n)])          #appending element to
list

>>> def test2(n):

...     return len(list(itertools.repeat(0, n)))          #creating
list at once

>>> timeit test(10000)

1000 loops, best of 3: 810 us per loop

>>> timeit test2(10000)

10000 loops, best of 3: 89.5 us per loop
```

Sometimes, we could consider using array in Python.

Tuple

Looking from facade, tuple is just the immutable version of list.

In fact, in Python the implementation of list and tuple are very different.

From the lower layer's point of view, element objects and their index in tuple are stored sequentially rather than separately.

From programming point of view, tuple uses memory better, and tuple is much more friendly for multiprocessing and distributed systems.

common operations,

```
>>> a = (4)                #it is just a number

>>> type(a)

<type 'int'>

>>> a = (4,)               #it is now a tuple

>>> type(a)

<type 'tuple'>

>>> t = tuple("abcdefg")

>>> t

('a', 'b', 'c', 'd', 'e', 'f', 'g')

>>> t.count('a')          #counting number

1

>>> t.index('a')          #find index

0
```

Standard Python library included "namedtuple", which allows you to use name to get the element.

>>> from collections import namedtuple

>>> User = namedtuple("User", "name age")

>>> u = User("user1", 10)

>>> u.name

'user1'

>>> u.age

10

However it is not infact a tuple, but a self-defined class... and performance was not very good(tested).

Dictionary

Yes, it is a hash mapping or hash table. You can read a description of python's dict implementation, as written by Tim Peters, here → https://mail.python.org/pipermail/python-list/2000-March/048085.html.

Some interesting facts,

- Python’s dictionary maintains a 8-size hashtable after initializing, when grow beyand 8, it applies memory to store the hash table in heap.
- Re-sizing the dictionary will result in re-allocating memory, and re-hashing
- Deleting element will not immediately shrunk the memeory

Creating dictionary,

```
>>> {}

{}

>>> {'a':1, 'b':1}

{'a': 1, 'b': 1}
```

```
>>> {'a':1, 'b':2}
{'a': 1, 'b': 2}
>>> dict(a=1, b=2)
{'a': 1, 'b': 2}
>>> dict(zip("ab", [1, 2]))
{'a': 1, 'b': 2}
```

Common operations,

```
>>> d = {"a":1, "b":2}
>>> 'b' in d          #check whether key exists
True
>>> del d['b']        #deletion
>>> d
{'a': 1}
>>> d = {"a":1}
>>> d.update({"c": 3})          #combination of two dic
>>> d
{'a': 1, 'c': 3}
>>> d = {"a":1, "b":2}          #pop from dic
>>> d.pop("b")
2
>>> d
{'a': 1}
>>> d = {"a":1, "b":2}          #pop whole item
```

```
>>> d.popitem()
('a', 1)
>>>
```

Some interesting tips,

```
>>> d = {"a":1, "b":2}
>>> d['c']      #try to get the value of c, which is not in dic
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'c'
>>> d.get('c')        #now using get
>>> print d.get('c')
None
>>> print d.get('c', "not exsits")         #set default value for
get()
not exsits
```

Iterator-related,

```
>>> d = {"a":1, "b":2}
>>> d.keys()
['a', 'b']
>>> d.values()
[1, 2]
>>> d.items()
[('a', 1), ('b', 2)]
```

>>> for k in d: print k, d[k]

...

a 1

b 2

For huge-size dictionaries, items(), keys(), values() will create huge list, so we should use iterators to avoid problem when encountering these scenarios,

```
>>> d = {"a":1, "b":2}

>>> d.iterkeys

<built-in method iterkeys of dict object at 0x10137d910>

>>> d.iterkeys()

<dictionary-keyiterator object at 0x1013b7890>

>>> for k in d.iterkeys():

...     print k

...
```

Set

Set is used to store non-sequential, and non-duplicated elements.

Common operations

```
>>> s = set('abc')

>>> s

set(['a', 'c', 'b'])

>>> {v for v in "abc"}

set(['a', 'c', 'b'])
```

```
>>>

>>> {'a', 'b', 'c'}

set(['a', 'c', 'b'])

>>> s.add('d')

>>> s.remove('b')

>>> s

set(['a', 'c', 'd'])

>>> s.remove('e')

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

KeyError: 'e'

>>> s.discard('e')

>>> s

set(['a', 'c', 'd'])
```

Summary

Data structure is important, the Python builtin types are no where from being sufficient to cover scenarios encountered when programming. C, data structure, some basic algorithms are all “must-haves” for programmers.