Chapter1
Python BASICS

—

By Weihan Jiang, Master student and Research Assistant at Univ. of Washington

1.1 RunTime Environment - Is Python interpreted or compiled or both?

The popular question about Python is that "Is Python interpreted or compiled or both?"

Practically speaking, Python is compiled.

However Not compiled to machine code ahead of time, "only" compiled to to bytecode, but it's still compilation with at least some of the benefits.

For example, the the following function,

```
>>> def func1():

...     print "this is function 1!"
```

Gets compiled into the following bytecode,

```
>>> dis.dis(func1)

  2           0 LOAD_CONST               1 ('this is function 1!')

              3 PRINT_ITEM

              4 PRINT_NEWLINE

              5 LOAD_CONST               0 (None)

              8 RETURN_VALUE
```

It's actually less readable and a bit more low-level.

Interpreting this is faster than interpreting from a higher-level representation.

So the conclusion is that the bytecode is either interpreted (note that there's a difference, both in theory and in practical performance, between interpreting directly and first compiling to some intermediate representation and interpret that), as with the reference implementation (CPython), or both interpreted and compiled to optimized machine code at runtime(Just-in-time Compiler), as with PyPy.

The Python VM(Pyobject* Stack) will execute the bytecode by following the steps below,

1.2 Types and objects

Type comes ahead of instance.

Everything in Python is treated as object, including type.

Every object, including type, has a head "PyObject_HEAD", where the type of the object is specified.

PyObject_HEAD consists of two things,

- reference count
- type pointer

Let's take int type as an example,

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

Let’s have a check with the reference count,

```
>>> import sys
>>> x = 200
>>> sys.getrefcount(x)  #means int 200 has 4 refcount
4
>>> y = x               #add one more
>>> sys.getrefcount(x)
5                      #refcount has increased by 1
```

Pointer “`*ob_type`” points to the object of type.

```
>>> x = 20
>>> type(x) is types.IntType
True
```

id will show the address of the variables,

```
>>> hex(id(x)), hex(id(y))
('0x10030bc00', '0x10030bc00')
```

and it also shows that keyword “int”, “float”, “bool” are just alias of types.

```
>>> hex(id(int)), hex(id(types.IntType))
('0x1001792c0', '0x1001792c0')
```

1.3 Namespaces

Namespaces are one of the key concepts to help you understand Python.

```
>>> x

NameError: name 'x' is not defined
```

We get used to think that “x” here is a variable, but in fact it is just a name.

In C language, the variable name is the address of the variable.

In Python, it is not.

In Python a name is a string object, and in name space, we have pair the string object with the object it points to in a format of {name: object}

Python has multiple namespaces. For example, namespace “globals” for module, “locals” for the stack frame of a function, namespace “class”, and namespace“instance”. Different namespace has different lifecycle and scale.

```
>>> globals()

{'__builtins__': <module '__builtin__' (built-in)>, '__package__':
None, 'x': 123, 'y': 20, '__name__': '__main__', '__doc__': None,
'types': <module 'types' from
'/Users/weihan/anaconda/lib/python2.7/types.pyc'>}
```

We can observe that namespace is infact a dictionary. You can edit namespace in a more hacky way as below.

```
>>> globals()['z'] = "helloworld!"

>>> globals()

{'__builtins__': <module '__builtin__' (built-in)>, '__package__':
None, 'x': 123, 'y': 20, '__name__': '__main__', 'z': 'helloworld!',
'__doc__': None, 'types': <module 'types' from
'/Users/weihan/anaconda/lib/python2.7/types.pyc'>}
```

“Name does not have type, but objects do.” -- comments in source code of Cpython.

So you can use the same name to bound different types of objects,

```
>>> y = 20

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

>>> y = "helloworld"
```

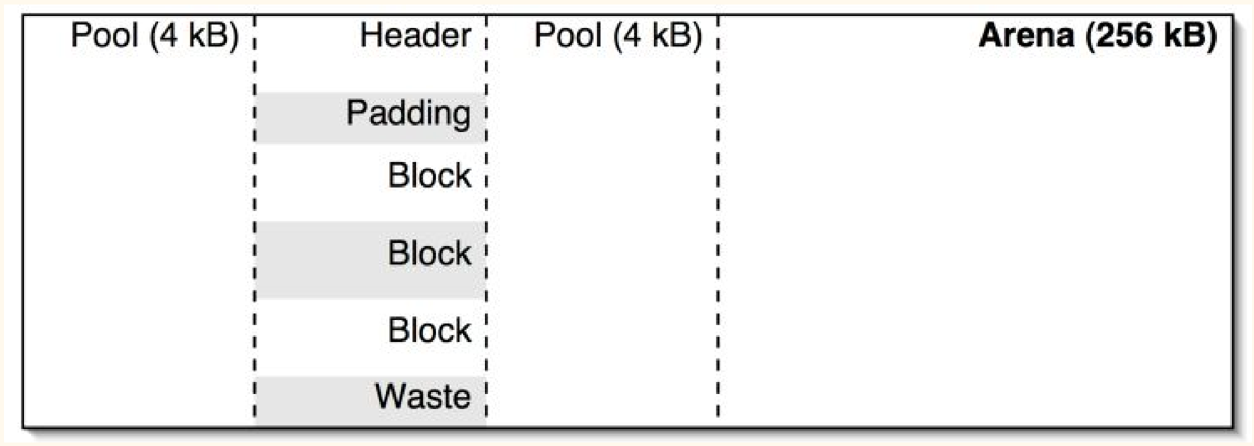
1.4 Memory and Recycle

The pymalloc Allocator

Python's memory allocator, called pymalloc, was written by Vladimir Marangozov and originally was an experimental feature is Python 2.1 and 2.2, before becoming enabled by default in 2.3. Python uses a lot of small objects that get created and destroyed frequently, and calling malloc() and free() for each one introduces significant overhead. To avoid this, pymalloc allocates memory in 256 kB chunks, called arenas. The arenas are divided into 4 kB pools, which are in turn subdivided into fixed sized blocks, as shown in Figure 1. The blocks are returned to the application. To understand the details of the allocator, we will step through the process that occurs when a block is needed, and when it is freed.

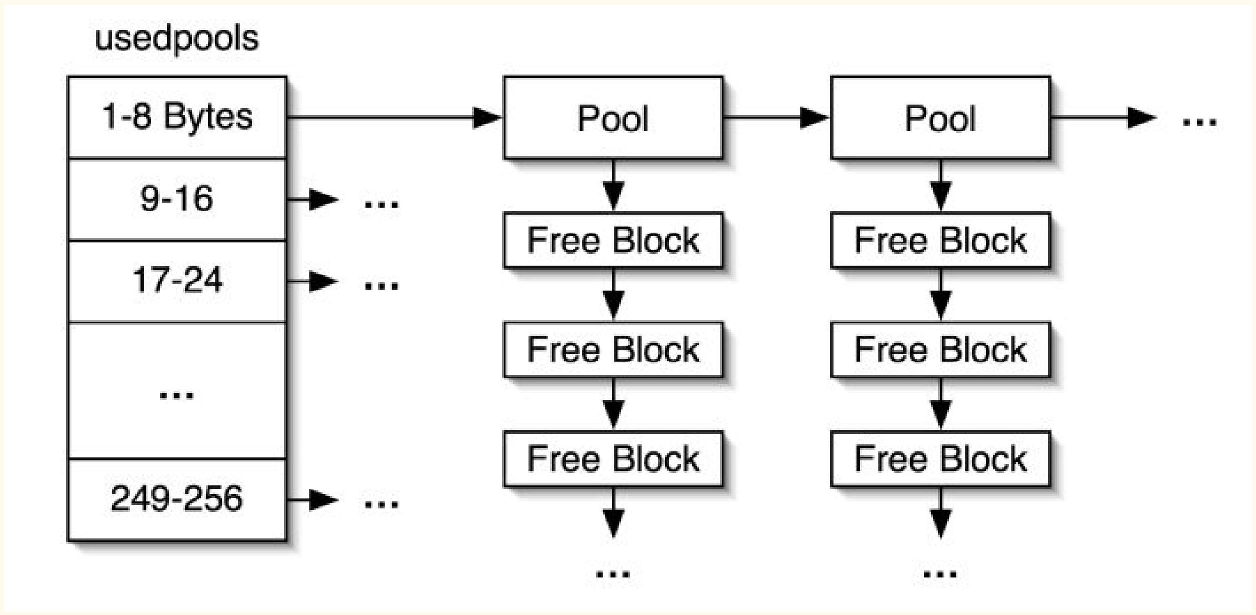
Note: that “block” is the minimum storage unit in Python. Each block is multiple of 8 bytes. So if you need memory to store an object of size 13 bytes, you will still need occupy a block.

Note: for object larger than 256 bytes, Python will use malloc to get memory from heap(exactly like you use malloc in C language), but there is much fewer chance to do so.



Allocating Memory

When allocating a new object, the first allocator looks to see if there are any pools that have already been divided into blocks of the required size. The `usedpools` array, shown in Figure 2, contains linked lists of pools that have blocks of each size. Each of the pools has a singly linked list of available blocks. If there is a pool, we pop a block off of its list. This is the most common case and it is very fast as it only requires a few memory reads and one write. If the block was the last one in the pool, then it is now completely allocated. In this case, we also pop the pool off. Finally, we return the block to the application.



If there are no pools of the correct size, we need to find an available pool. The freepools linked list contains available pools. If there is a pool on the list, we pop it off. Otherwise, we need to create a new pool. If there is space left at the end of the last arena we allocated, we can cut another pool off using the arenabase pointerIf there is no space, we call malloc() to allocate a new arena. Now that we finally have a pool, we divide it into fixed size blocks, place the pool in usedpools and finally return one block to the application.

Pass by Reference

Python copies pointers to make different names to bound the same object. Arena is stored on heap, so very single type of object is stored on heap. So there is NO such concept like “value”(on stack) or “reference value” (on heap). All object in Python has a standard and complete head.

Below example showed that all address are the same, hence it showed that all variable is on heap.

```
>>> a = object()

>>> b = a

>>> a is b

True

>>> hex(id(a)), hex(id(b))

('0x1006560a0', '0x1006560a0')

>>> def test(x):

...     print hex(id(x))

...

>>> test(a)

0x1006560a0
```

Also example below shows the copy mechanism of Python,

```
>>> import copy

>>> x = object()

>>> lst = [x]

>>> lst2 = copy.copy(lst)

>>> lst2 is lst

False
```

```
>>> lst2[0] is lst[0]

True

>>> lst3 = copy.deepcopy(lst)

>>> lst3 is lst

False

>>> lst3[0] is lst[0]

False
```

Reference Count

By default, Python uses reference to manage the recycle of objects.

When the reference count of an object dropped to 0, Python will either mark the block, where the object previously resides, as free; Or it will return the memory to OS.

In following example, we will use __del__ to monitor the recycle of an object.

```
>>> class User(object):

...     def __del__(self):

...             print "will be dead!"

>>> a = User()                    #create an instance

>>> b = a                         #reference +1

>>> import sys

#note getrefcount itself will lead refcount +1

>>> sys.getrefcount(a)

3

>>> del a

>>> sys.getrefcount(a)
>>> sys.getrefcount(b)

2

#del the last ref, will get refcount to 0, and trigger recycle

>>> del b

will be dead!
```

Garbage Collection

In face, rather than the reference count, Python maintains another recycle mechanism -- GC.

GC is introduced basically to deal with “Reference Cycle Garbage Collection”, for example,

recursive reference of list, set, or object.

For example,

```
>>> l = [1, 2, 3]

>>> l.append(l)

>>> l

[1, 2, 3, [...]]
```

So for these types, Python will PYGC_Head to track these object for the purpose of GC.

However, if you are sure your code does not involve recursive reference, you can choose to disable GC for a faster execution speed.

for example,

```
>>> import gc

>>> class User(object):

...     def __del__(self):

...             print hex(id(self)), "will be head!"
```

```
...
>>> gc.disable()                    #disable GC
>>> a = User()
>>> del a
0x10213bed0 will be head!
>>>
```

Like .NET and Java, Python's GC will categorize objects into the following three types,

- GEN0 - newly added object
- GEN1 - objects that persisted after last GC operation
- GEN2 - objects that persisted after more than two GC operations

When GEN0 count exceed a certain threshold, it will trigger GC operation.

GNE1, and GEN2 objects will be treated by special mechanisms.

1.5 Compile

Python implements a "stack-based VM" structure to process the bytecodes.

In short, Python uses a virtual machine under the covers, from a user's perspective, one can ignore this detail most of the time.

To execute a program, Python interpreter will compile the source code to bytecodes, and store the bytecode in .pyc files. Then the .pyc code is executed by Python's virtual machine.

Compilation happens when the module is imported.

if there is .pyc file existing, Python will go through the following step,

1. check the .pyc file's Magic mark
2. checking time stamp to decide whether recompilation is needed.
3. import the module.

If there is no .pyc file existing,

1. AST analysis
2. convert result from step1 to PyCodeObject
3. Save the PyCodeObject to .pyc file together with the Magic Mark and time stamp.
4. Import Module.

1.6 Execution

There are two ways to execute code,

- evel
- exec

There are two difference between these methods,

1. eval returns the value that is returned by the evaluated byte code / the resulting value of the evaluated expression. exec ignores the return value and always returns None (in Python 2 it is a statement and cannot be used as an expression, so it really does not return anything).

   ```
   >>> eval('42')

   42

   >>> exec('42')
   ```

2. If a code object (which contains Python bytecode) is passed to exec or eval, they behave identically, except for the return value (eval returns the value returned, exec returns None).

   If a str object (which contains Python source code) is passed to exec / eval, it is internally compiled to bytecode using compile(source, '<string>', mode) where mode is exec or eval respectively. This is where the differences really come from.

   Below is an example of the different behaviors of the two modes.

   ```
   >>> eval('for i in range(3): print(i)')

   Traceback (most recent call last):

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

```
  File "<string>", line 1

    for i in range(3): print(i)

      ^

SyntaxError: invalid syntax

>>> exec('for i in range(3): print(i)')

0

1

2
```

by default, eval will read current namespace,

```
>>> x = 100

>>> eval("x+200")

300
```

Also we can assign a namespace for eval,

```
>>> ns = dict(x=10, y=20)

>>> eval("x+y", ns)

30
```

We could also assign namespace to exec the same way, if we need avoid pollution to current namespaces.