23 用字节码彻底征服面试官-慕课专栏

请问 Python 程序是怎么运行的？是编译成机器码后在执行的吗？试着与 *C*、*C++* 、*Java*、*Shell* 等常见语言比较说明。

不少初学者对 *Python* 存在误解，以为它是类似 *Shell* 的解释性脚本语言，其实并不是。虽然执行 *Python* 程序的 *python* 命令也被称为 *Python* 解释器，但它其实包含一个 **编译器** 和一个 **虚拟机** 。

当我们在命令行敲下 `python xxxx.py` 时，*python* 命令中的编译器首先登场，将 *Python* 代码编译成 **代码** 对象。**代码** 对象包含 **字节码** 以及执行字节码所需的 **名字** 以及 **常量** 。

当编译器完成编译动作后，接力棒便传给 **虚拟机** 。**虚拟机** 维护执行上下文，逐行执行 **字节码** 指令。执行上下文中最核心的 **名字空间** ，便是由 **虚拟机** 维护的。

因此，*Python* 程序的执行原理其实更像 *Java*，可以用两个词来概括—— **虚拟机**和**字节码** 。不同的是，*Java* 编译器 *javac* 与 虚拟机 *java* 是分离的，而 *Python* 将两者整合成一个 *python* 命令。此外，*Java* 程序执行前必须先完整编译，而 *Python* 则允许程序启动后再编译并加载需要执行的模块。

pyc 文件保存什么东西，有什么作用？

Python 程序执行时需要先由 **编译器** 编译成 **代码** 对象，然后再交由 **虚拟机** 来执行。不管程序执行多少次，只要源码没有变化，编译后得到的代码对象就肯定是一样的。因此，*Python* 将代码对象序列化并保存到 *pyc* 文件中。当程序再次执行时，*Python* 直接从 *pyc* 文件中加载代码对象，省去编译环节。当然了，当 *py* 源码文件改动后，*pyc* 文件便失效了，这时 *Python* 必须重新编译 *py* 文件。

如何查看 Python 程序的字节码？

Python 标准库中的 *dis* 模块，可以对**代码**对象以及 **函数** 对象进行反编译，并显示其中的 **字节码**。

例如，对于函数 *add* ，通过 *__code__* 字段取到它的 **代码** 对象，并调用 *dis* 进行反编译：

```
>>> def add(x, y):
...     return x + y
...
>>> dis.dis(add.__code__)
  2           0 LOAD_FAST                0 (x)
              2 LOAD_FAST                1 (y)
              4 BINARY_ADD
              6 RETURN_VALUE
```

当然了，直接将 **函数** 对象传给 *dis* 也可以：

```
>>> dis.dis(add)
  2           0 LOAD_FAST                0 (x)
              2 LOAD_FAST                1 (y)
              4 BINARY_ADD
              6 RETURN_VALUE
```

Python 中变量交换有两种不同的写法，示例如下。这两种写法有什么区别吗？那种写法更好？

```
a, b = b, a
```

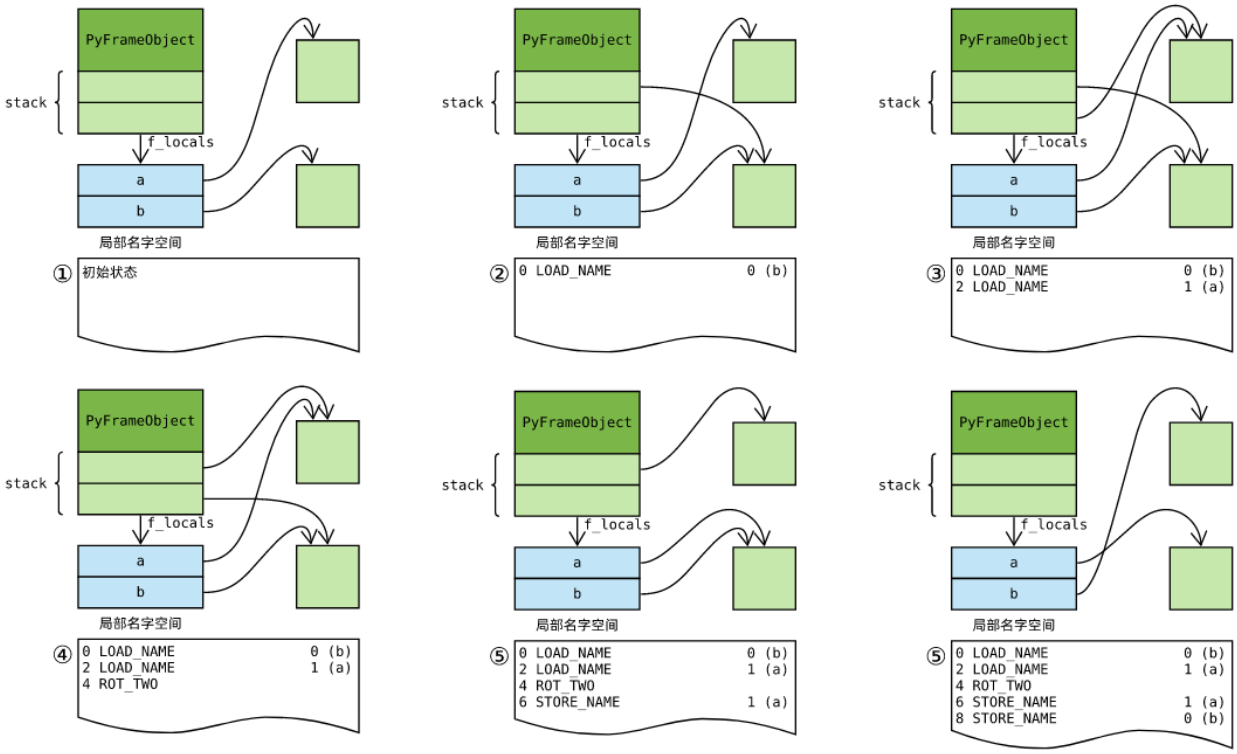
```
tmp = a
a = b
b = tmp
```

这两种写法都能实现变量交换，表面上看第一种写法更加简洁明了，似乎更优。那么，在优雅的外表下是否隐藏着不为人知的性能缺陷呢？想要找打答案，唯一的途径是研究字节码：

```
  1           0 LOAD_NAME                0 (b)
              2 LOAD_NAME                1 (a)
              4 ROT_TWO
              6 STORE_NAME               1 (a)
              8 STORE_NAME               0 (b)
```

```
  1           0 LOAD_NAME                0 (a)
              2 STORE_NAME               1 (tmp)

  2           4 LOAD_NAME                2 (b)
              6 STORE_NAME               0 (a)

  3           8 LOAD_NAME                1 (tmp)
             10 STORE_NAME               2 (b)
```

从字节码上看，第一种写法需要的指令条目也更少：先将两个变量依次加载到栈，然后一条 *ROT_TWO* 指令将栈中的两个变量交换，最后再将变量依次写回去。注意到，变量加载的顺序与 = 右边一致，写回顺序与 = 左边一致。



而且，*ROT_TWO* 指令只是将栈顶两个元素交换位置，执行起来比 *LOAD_NAME* 和 *STORE_NAME* 都要快。

至此，我们可以得到结论了—— **第一种变量交换写法更优** ：

- 代码简洁明了，不拖泥带水；
- 不需要辅助变量 *tmp* ，节约内存；
- *ROT_TWO* 指令比一个 *LOAD_NAME STORE_NAME* 指令对更有优势，执行效率更高；

请解释 is 和 == 这两个操作的区别。

```
a is b
a == b
```

我们知道 *is* 是 **对象标识符** (*object identity*)，判断两个引用 **是不是同一个对象** ，等价于 *id(a) == id(b)* ；而 == 操作符判断两个引用 **是不是相等** ，等价于调用魔法方法 *a.__eq__(b)* 。因此，== 操作符可以通过 *__eq__* 魔法方法进行覆写(*overriding*)，而 *is* 操作符无法覆写。

从字节码上看，这两个语句也很接近，区别仅在比较指令 *COMPARE_OP* 的 **操作数** 上：

```
 1           0 LOAD_NAME                0 (a)
             2 LOAD_NAME                1 (b)
             4 COMPARE_OP               8 (is)
             6 POP_TOP
```

```
 1           0 LOAD_NAME                0 (a)
             2 LOAD_NAME                1 (b)
             4 COMPARE_OP               2 (==)
             6 POP_TOP
```

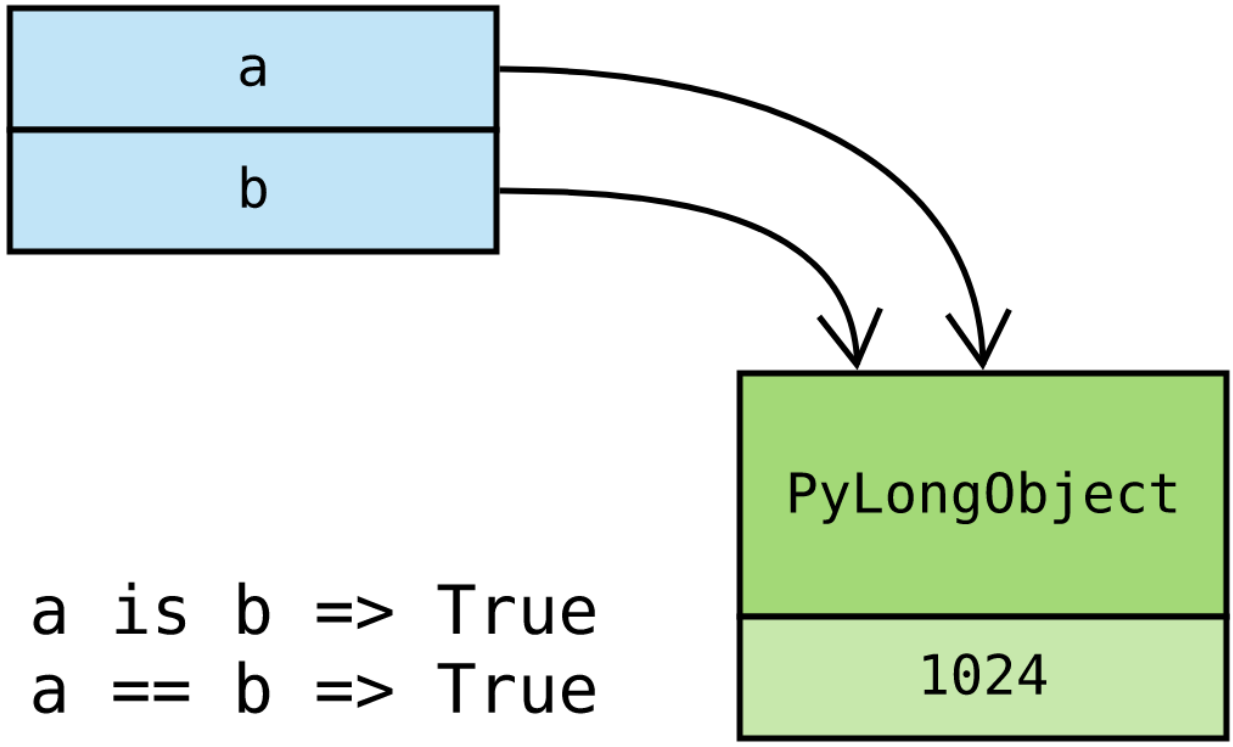
COMPARE_OP 指令处理逻辑在 *Python/ceval.c* 源文件中实现，关键函数是 *cmp_outcome* ：

```
static PyObject *
cmp_outcome(int op, PyObject *v, PyObject *w)
{
    int res = 0;
    switch (op) {
    case PyCmp_IS:
        res = (v == w);
        break;
    // ...
    default:
        return PyObject_RichCompare(v, w, op);
    }
    v = res ? Py_True : Py_False;
    Py_INCREF(v);
    return v;
}
```

源码表明，*is* 操作操作符仅需比较两个对象的 **地址** (指针)是不是相同(第 *7* 行)。包括 == 在内的其他比较操作，则需要调用 *PyObject_RichCompare* 函数，而 *PyObject_RichCompare* 将进一步调用对象的魔法方法进行判断。

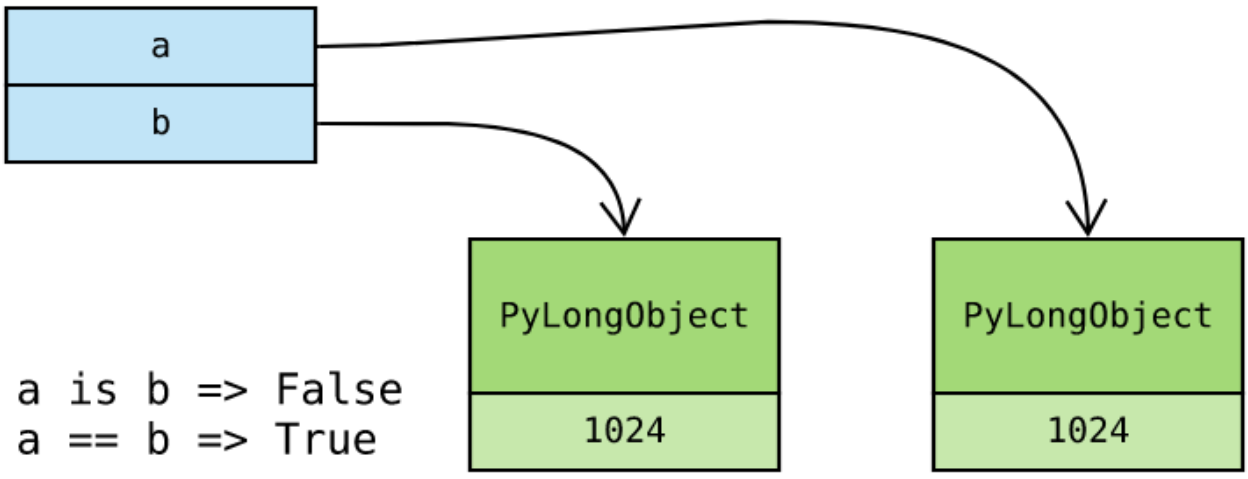
这个例子，*a* 和 *b* 均引用同一个对象，*is* 和 == 操作均返回 *True* ：

```
>>> a = 1024
>>> b = a
>>> a is b
True
>>> a == b
True
```



另一个例子，*a* 和 *b* 引用不同的两个 **整数** 对象，整数对象数值均为 *1024* ：

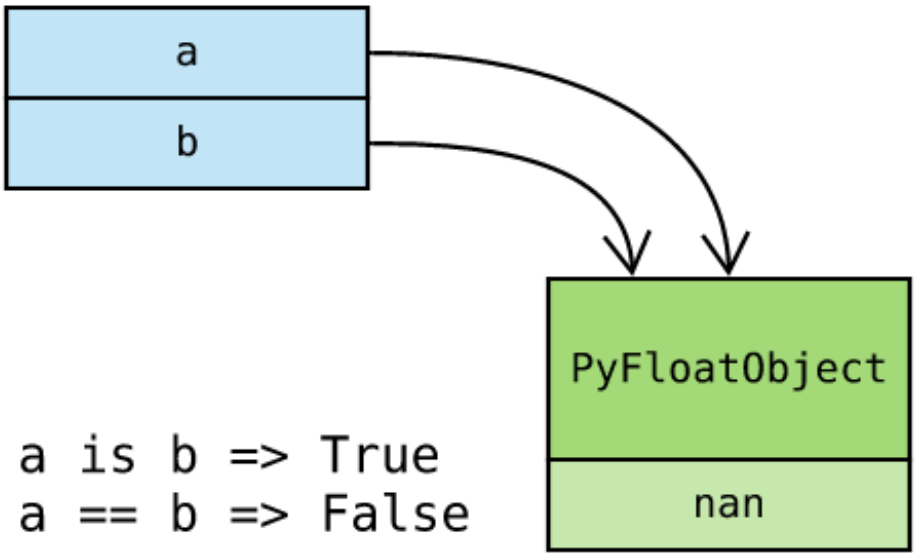
```
>>> a = 1024
>>> b = int('1024')
>>> a is b
False
>>> a == b
True
```



显然，由于背后对象是不同的， *is* 操作结果是 *False* ；而对象值相同，== 操作结果是 *True* 。

一般而言，对象与自身比较，结果是相等的。因此，如果 *a is b* 结果为 *True* ，那么 *a == b* 多半也是 *True* 。但这并不是一个永远成立的铁律，至少存在一个例外—— **浮点数** *nan* 。*nan* 是一个特殊的 **浮点数** ，用于表示 **异常值** ，即不存在或者非法的值。不管 *nan* 跟任何浮点(包括自身)做何种数学比较，结果均为 *False* 。

```
>>> a = float('nan')
>>> b = a
>>> a is b
True
>>> a == b
False
```



在 Python 中与 None 比较时，为什么要用 is None 而不是 == None ？

None 是一种特殊的内建对象，它是 **单例** 对象，在整个程序中只有一个实例。因此，如果一个变量是 *None* ，它和 *None* 一定指向同一个实例，内存地址肯定也相同。这就是要用 *is* 操作符与 *None* 比较的原因。

Python 中的 == 操作符对两个对象进行相等性比较，背后调用 *__eq__* 魔法方法。在自定义类中，__eq__ 方法可以被覆写，实现：

```
>>> class Foo(object):
...     def __eq__(self, other):
...         return True
...
>>> f = Foo()
>>> f is None
False
>>> f == None
True
```

由于 *is None* 只需比较对象的 **内存地址** ，而 == *None* 需要调用对象的魔法函数进行判断，因此 *is None* 更有性能优势。下面这个程序尖锐地揭开这两种比较间微妙的性能差距：

```
import time

class Foo(object):

    def __eq__(self, other):
        return False

def compare_with_eq(n):
    nones = 0

    f = Foo()
    for _ in range(n):
        if f == None:
            nones += 1

    return nones

def compare_with_is(n):
    nones = 0

    f = Foo()
    for _ in range(n):
        if f is None:
            nones += 1

    return nones

def test(func, n):
    start_ts = time.time()

    func(n)

    print('call function {name} with {n} times in {seconds:.2f}s'.format(
        name=func.__name__,
        n=n,
        seconds=time.time()-start_ts,
    ))

if __name__ == '__main__':
    N = 100000000

    test(compare_with_eq, N)
    test(compare_with_is, N)
```

例子程序将自定义类 *Foo* 对象与 *None* 进行一亿次比较，在我的 MacBook 上表现分别如下：

```
call function compare_with_eq with 100000000 times in 15.91s
call function compare_with_is with 100000000 times in 4.28s
```

- *== None* 方式耗时大概 *16* 秒；
- *is None* 方式耗时大概 4 秒；
- 两者性能相差整整 4倍！

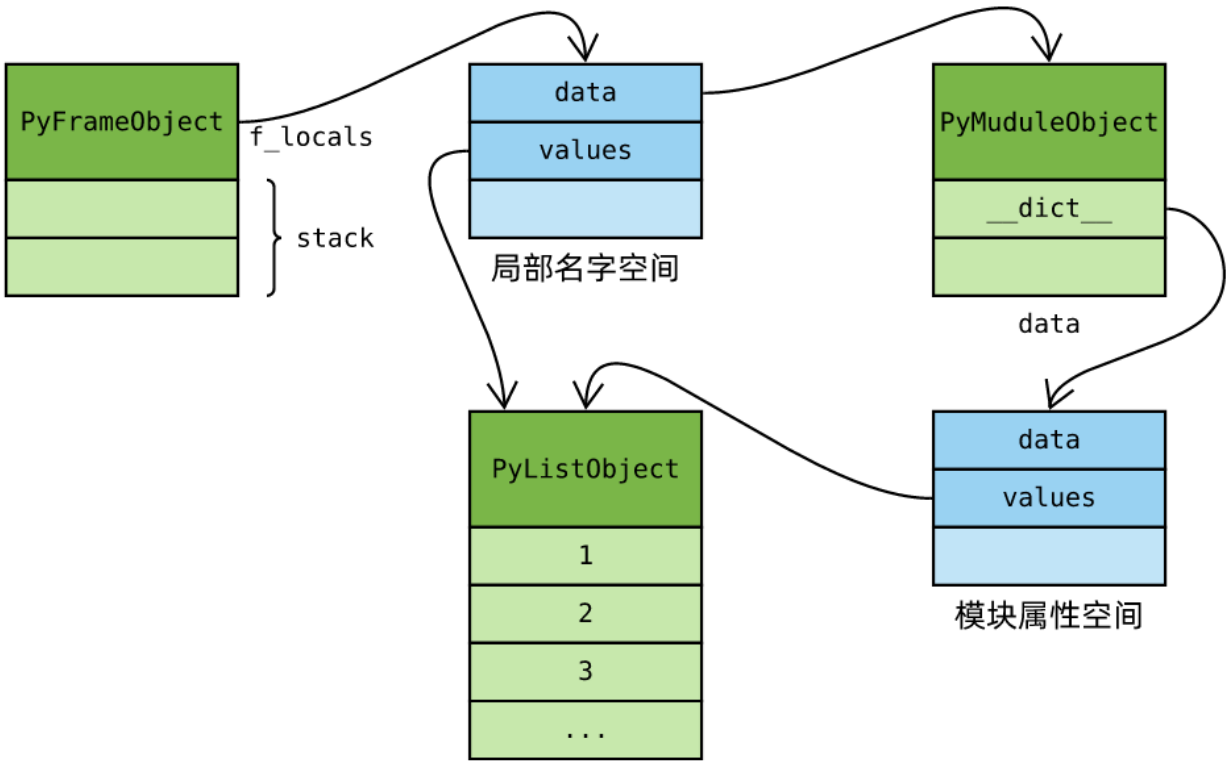
请问以下程序输出什么？为什么？

```
values = [1, 2, 3]

import data
from data import values

values.pop()
print(data.values)

values = [3.14, 2.71]
print(data.values)
```

在 **模块加载机制** 一节，我们学习到 *from import* 语句的原理：先加载模块对象，在从模块对象中取出指定属性，并引入到当前 **局部名字空间** 。与 *from data import values* 相对应的字节码如下：

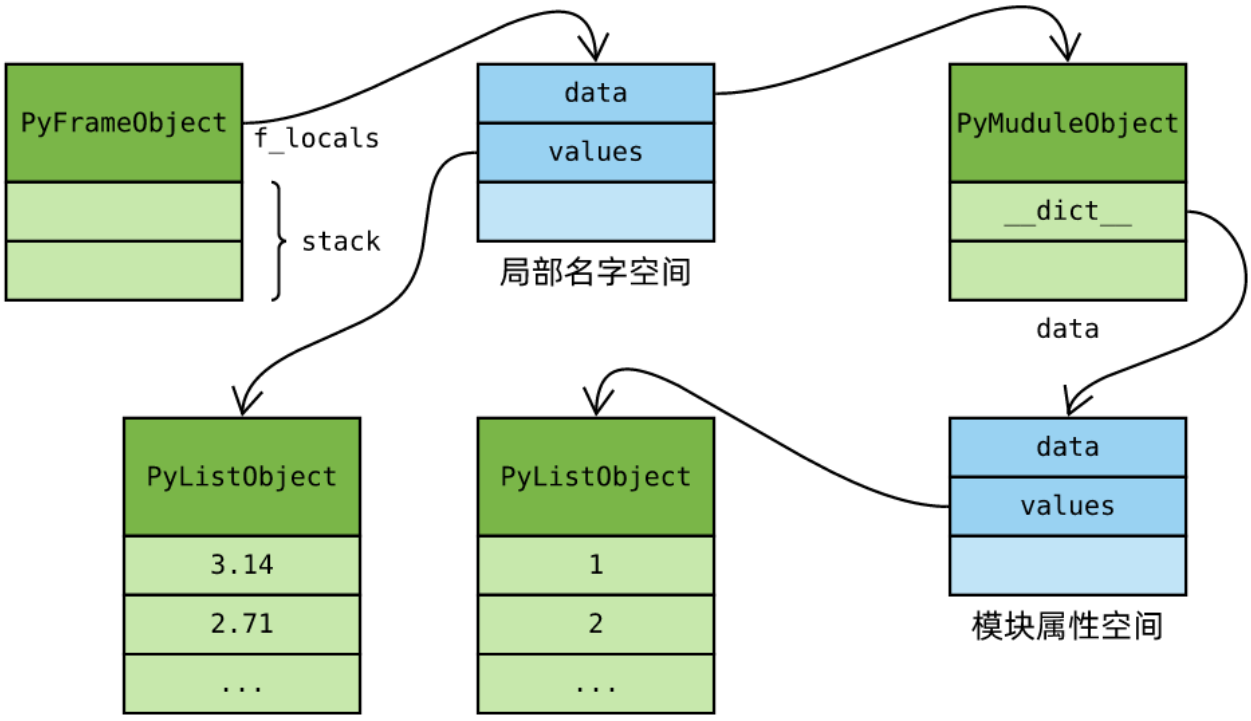
```
1           0 LOAD_CONST               0 (0)
            2 LOAD_CONST               1 (('values',))
            4 IMPORT_NAME              0 (data)
            6 IMPORT_FROM              1 (values)
            8 STORE_NAME               1 (values)
           10 POP_TOP
```

因此，当两个 *import* 语句执行完毕后，虚拟机中的状态是这样的：



局部名字空间中 *values* 指向的 *list* 对象与模块 *data* 中 *values* 是同一个。对局部名字空间的 *values* 进行操作，等价于对 data 模块中的 values 进行操作。因此，第一个 print 语句输出 `[1, 2]` 。

当代码第 *7* 行对局部名字空间中的 *values* 进行重新赋值，指向一个新的 *list* 对象后，虚拟机中的状态是这样的：



这时，局部名字空间中的 *values* 与模块中的保存独立，不再有任何关联了。因此，整个程序输出内容如下：

```
[1, 2]
[1, 2]
```