Python赋值与动态类型

Python的动态类型, 是这门语言简洁性和灵活性的基础, 这跟强数据类型语言如C JAVA等很不相同. 在Python中, 我们并没有声明脚本中所使用的变量的确切类型, 但是变量仍然可以工作.

赋值的基本形式是在等号的左边写赋值的目标, 而要赋值的对象位于右侧, 赋值语句虽然简单, 但是一些特性还是需要记住.

如: 当定义变量a = 3时, 经过了哪些过程?

1. 变量创建: 一个变量例如a, 当代码第一次给它赋值时就被创建了, 之后的赋值将会改变已创建的变量的值.
2. 变量类型 变量没有任何的和它相关联的类型信息或约束, 类型的概念是存在于对象中而不是变量中, 变量只是在一个特定的时间点, 引用了一个特定的对象而已
3. 变量使用 变量出现在表达式中时, 它会马上被引用的对象所代替, 无论这个对象是什么类型; 此外, 所有的变量必须在其使用前明确地赋值, 使用未赋值的变量会产生错误。

总而言之变量在赋值的时候才创建, 它可以引用任何类型的对象, 并且必须在引用之前赋值; 引用一个未赋值的变量会报错.

In [1]:
```
print a
```

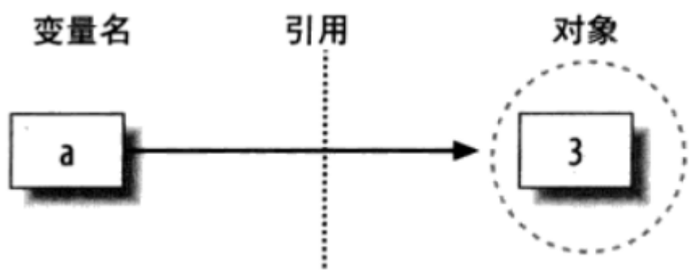
```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-da1608c9d425> in <module>()
----> 1 print a

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

Python会执行三个不同的步骤去完成这个创建变量的请求:

1. 创建一个对象代表值3
2. 创建一个变量a，如果它还没有创建的话
3. 将变量与新的对象3连接（引用，指针指向）

Python的变量更像是指针, 而不是数据存储区域; 变量和对象保存在内存中的不同部分, 并且通过引用相关联.

1. 对象是被分配的一块内存, 有足够的空间去表现他们所代表的值.
2. 引用是自动形成的从变量到对象的指针



In [2]:
```
#示例代码:
a = 3
a = 'python'
a = 3.14
```

以上代码对a赋值了三次, 第一次是整数, 第二次是字符串, 第三次是浮点数, 这样的赋值对于习惯了C,C++,JAVA语言的童鞋来说很不习惯; 可能他们会认为a居然能从整数变成了字符串, 这样的操作很奇怪. 因为在python中, 变量名是没有类型的, 类型是属于对象的. 所以对上面的例子我们可以这样理解, 第一次a指向一个整形对象, 第二次a指向一个字符串对象, 第三次a指向一个浮点型对象. 所以我们实际上并没有改变变量a的类型, 只是让它指向了不同类型的对象而已. 让我们再一次记住, python中的类型是与对象相关联, 而不是与变量相关联.

垃圾收集

每一个对象都有两个标准的头部信息:

1. 类型标识符: 标示这个对象的类型.
2. 引用计数器: 用来决定是不是可以回收这个对象, 当引用计数器为0时回收这个对象

在Python中, 每当一个变量名被赋予一个新的对象后, 如果之前的对象如果没有被其他的变量引用的话, 那个对象占用的空间就会被收回。这种回收对象空间的技术就叫做垃圾收集。

垃圾收集最直接的,可感受得到的好处就是可以在脚本中任意使用对象而不需要顾虑空间的释放.

In [4]:
```
import sys
a = ['x', 'apple', 'meizu']
sys.getrefcount(a)
```

Out[4]: 2

In [4]:
```
#赋值语句的形式

a = 1 #基本形式
a, b = 'x', 'y' #元组赋值运算
print(a, b)
[a, b] = ["x", "y"] #列表赋值
print(a, b)
a,b,c,d = 'spam'  #序列赋值
print(a, b, c, d)
a, *b = 'spam' #扩展序列解包
print(a, b)
a = b = 'apple' #多目标赋值
print(a, b)
a = 0
a += 1  #增强赋值运算(相当于a = a + 1)
print(a)
```

```
x y
x y
s p a m
s ['p', 'a', 'm']
apple apple
1
```

In [7]:
```
#扩展序列赋值在for循环中依然有效

for a, b in ((1, 2), (3, 4)):
    print(a, b)

for (a, b) in ((1, 2), (3,4)):
    print(a, b)

for(a, *b, c) in ((1,2,3,4), (4,5,6,7)):
    print(a, b, c)
```

```
1 2
3 4
1 2
3 4
1 [2, 3] 4
4 [5, 6] 7
```

共享引用

刚刚我们看到的是一个变量被赋值引用了多个对象的情况，那么下面我们看一个对象被多个变量引用的情况.

In [5]:
```
a = 3
b = a
```

变量名和对象，在运行赋值语句 b = a 之后，变量 a,b 都指向了对象 3 的内存空间

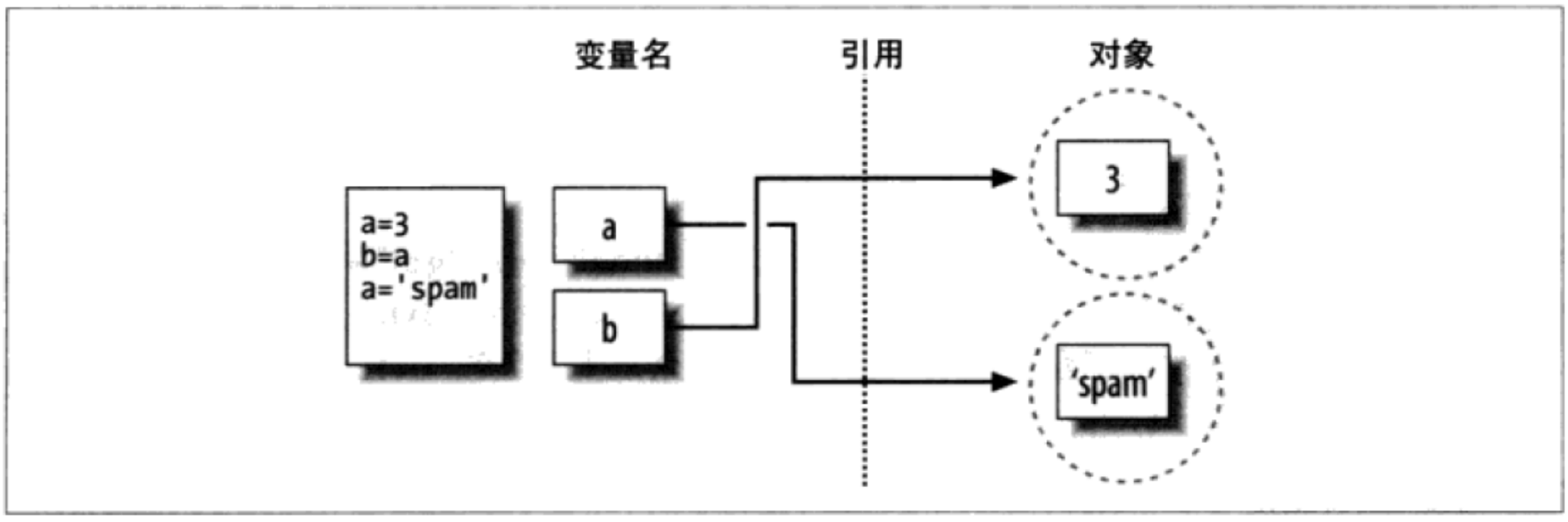
变量名和对象, 在运行赋值语句b = a之后, 变量a,b都指向了对象3的内存空间.

In [6]:
```
a = 'spam'
```

这时执行 a = 'spam', a将指向刚创建的字符串对象，但是不会影响到B的值.

In [7]:
```
print a
print b
```

```
spam
3
```



共享引用列表与共享引用字符串有所不同

In [9]:
```
l1 = [1, 2, 3, 4]
l2 = l1
print l2
l1[0] = 'python'
print l1
print l2
print id(l1)
print id(l2)
```

```
[1, 2, 3, 4]
['python', 2, 3, 4]
140171964561672
140171964561672
```

l1改变了所引用的对象中的一个元素, 这类修改会覆盖列表对象中的某一个部分, l2=l1 时, 由于他们将指向同一个内存空间. 当l1[0]改变指向时, l2也随着发生了变化. 这种结果并不是我们想要的, 如果你不想要这样的现象发生，则需要Python的对象拷贝, 而不是创建引用.

对象拷贝:

In [12]:
```
#示例代码1.浅拷贝
l3 = [1, 2, 3, 4]
l4 = l3[:]       #使用对象拷贝而不是创建引用
l5 = list(l3)    #使用对象拷贝而不是创建引用

print l3 == l4
print l3 is l4

l3[0] = 'python'
print l3
print l4
print l5
```

```
True
False
['python', 2, 3, 4]
[1, 2, 3, 4]
[1, 2, 3, 4]
```

对l3的修改不会影响到L4, 因为l4所引用的是l3所引用对象的一个拷贝, 两个变量指向的是不同的区域, 可以通过id()函数来查看对象所引用的内存空间.

In [13]:
```
print id(l3)
print id(l4)
print id(l5)
```

```
140172125018880
140172124611216
140172124613736
```

In [14]:
```
#示例代码2: 深拷贝
x = [[1,2,3,4], 5, 6]
y = x
z = list(x)

print id(x)
print id(y)
print id(z)
```

```
140171964503464
140171964503464
140172124612080
```

In [16]:
```
print [id(j) for j in x]
print [id(j) for j in y]
print [id(j) for j in z]
```

```
[140172124613016, 11202840, 11202816]
[140172124613016, 11202840, 11202816]
[140172124613016, 11202840, 11202816]
```

In [17]:
```
x[1] = 'apple'
print x
print y
print z
print [id(j) for j in x]
print [id(j) for j in y]
print [id(j) for j in z]
```

```
[[1, 2, 3, 4], 'apple', 6]
[[1, 2, 3, 4], 'apple', 6]
[[1, 2, 3, 4], 5, 6]
[140172124613016, 140171963126000, 11202816]
[140172124613016, 140171963126000, 11202816]
[140172124613016, 11202840, 11202816]
```

In [18]:
```
x[0][0] = 'python-training'
print x
print y
print z
```

```
[['python-training', 2, 3, 4], 'apple', 6]
[['python-training', 2, 3, 4], 'apple', 6]
[['python-training', 2, 3, 4], 5, 6]
```

In [19]:
```
import copy
w = copy.deepcopy(x)
print [id(j) for j in x]
print [id(j) for j in y]
print [id(j) for j in z]
print [id(j) for j in w]
```

```
[140172124613016, 140171963126000, 11202816]
[140172124613016, 140171963126000, 11202816]
[140172124613016, 11202840, 11202816]
[140172124612512, 140171963126000, 11202816]
```

In [20]:
```
x[0][1] = 'xxyyzz'
print x
print y
print z
print w
```

```
[['python-training', 'xxyyzz', 3, 4], 'apple', 6]
[['python-training', 'xxyyzz', 3, 4], 'apple', 6]
[['python-training', 'xxyyzz', 3, 4], 5, 6]
[['python-training', 2, 3, 4], 'apple', 6]
```

In [ ]:
```

```