《人工智能与Python程序设计》—Python变量进阶知识补充

人工智能与Python程序设计 教研组

C和Python中的变量

- C语言：“装盒子式”
  - int a;
  - a = 2;
- C语言的对这一条语句的处理是：
  - ①在内存中为变量a找到一片供存储的内存空间；
  - ②往这一片内存空间中填上二进制10
- Python：“贴标签式”或者“起名字式”
  - a = 2
  - 先在内存空间中找到一片区域存储2，之后再把a作为一个标签贴在2这一片区域上
  - a这个标签/名称 “引用/指向”2这个对象

变量引用对象

- 变量赋值：本质上是将变量“贴”在对象上（给对象起名字），运行使用变量来访问、修改对象
  - 对象是一块内存空间，内存空间里存储它们所表示的值；
  - 变量是到内存空间的一个标签或引用，也就是拥有指向对象存储的空间；
  - 引用就是自动形成的从变量到对象的映射关系（类似指针）
    - 引用可以看成对象的别名，通过别名可以直接操纵对象
    - 但与C语言中的指针不同，我们无法直接修改指针的值
      - 只能通过变量访问对象，或通过赋值更改变量指向的对象
  - **赋值是将一个变量标签与一个实际对象建立关联的过程**

命名空间（namespace）

- *namespace*（命名空间）是从名称到对象的映射
  - 例如：
    - 内置函数名到对应函数对象的映射（print，input，...）
    - 全局变量名到对象的映射
    - 函数中的局部变量名到对象的映射
    - 实例属性名到属性值的映射
    - 类属性名到属性值的映射
  - 命名空间存在嵌套关系
    - 函数作用域：先查找局部名称，再查找全局名称，最后查找内置函数 ……
    - 类和实例：实例属性->类属性->父类属性->...
  - 大多数命名空间都使用 Python 字典（dict）实现

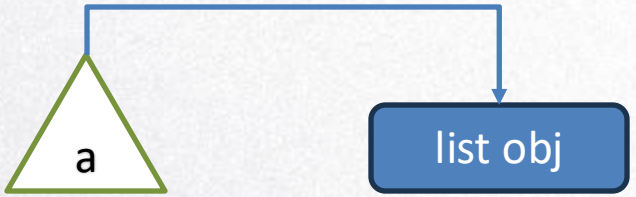
```
[1]: import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

对象的“回收”

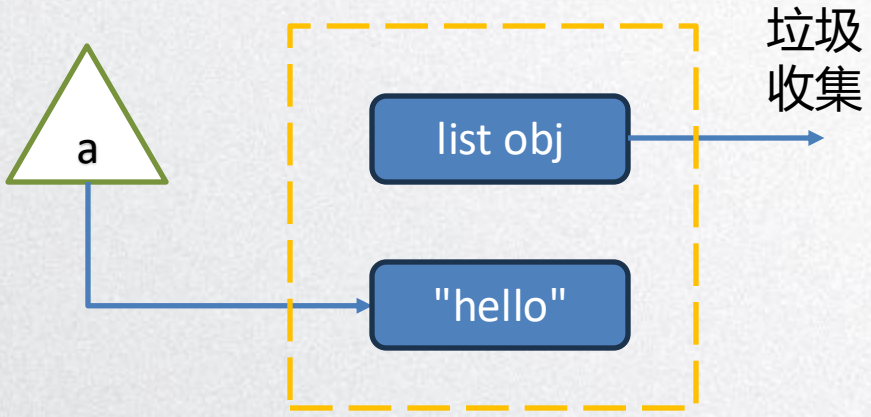
- 类型属于对象，并且对象中包含了一个引用计数器，用于记录当前有多少个变量在引用这个对象。
  - 一旦引用计数器为0，那么该对象就会被系统自动回收（这里有例外，python中缓存了一些小的常用的对象）

对象的“回收”

- 类型属于对象，并且对象中包含了一个引用计数器，用于记录当前有多少个变量在引用这个对象。
  - 一旦引用计数器为0，那么该对象就会被系统自动回收（这里有例外，python中缓存了一些小的常用的对象）



- a = list()　　#完成了变量a对内存空间中的一个list对象的引用



- a = "hello"
  - 没有别的变量引用list对象
  - list对象被加入垃圾收集

可变对象和不可变对象

- 可变对象可以被修改，包括列表list、字典dict、集合set

- 不可变对象无法修改，包括数字、字符串str，元组tuple

- **思考：**设置可变对象和不可变对象的目的和优势是？
  - 效率与维护
    - 哪个在系统底层实现更容易
  - 功能与方法
    - 哪个在使用上功能更强大

复制

- = 赋值并不会新建对象，b 和 a 引用的是同一个对象。

- copy模块

```
from copy import copy, deepcopy
```

- copy 方法会新建对象，b 和 a 引用的是不同的对象，但里面的可变对象（列表 y）依然引用的是同一个对象。也就是说 copy 方法只会复制最外面一层，里面的不会新建对象而是直接用原对象，是浅层复制。
- deepcopy 方法会新建对象，里面的可变对象也会新建对象。实际上deepcopy是递归copy，是深层复制。

浅复制

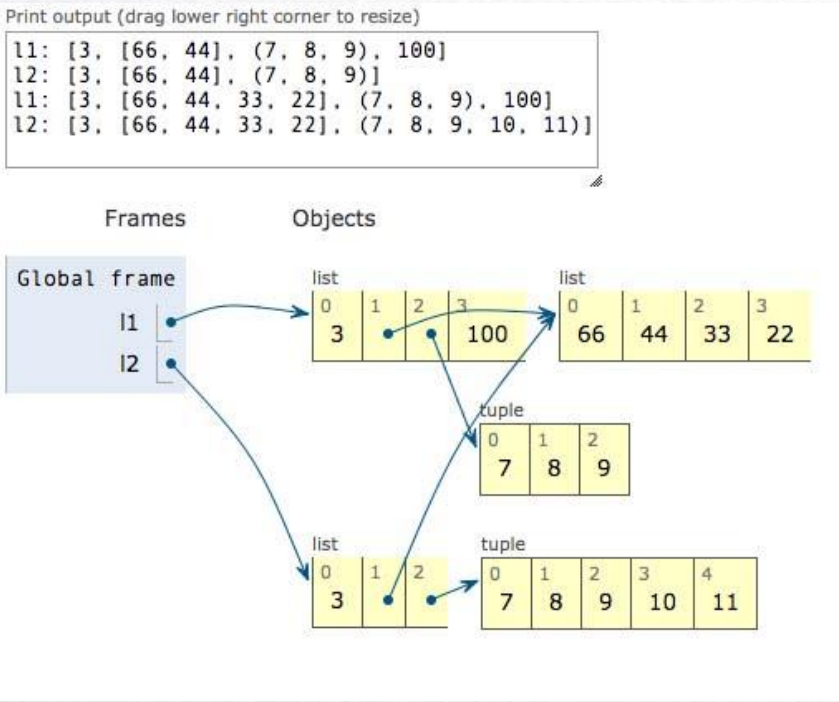
- l2 是 l1 的浅复制副本

```
l1 = [3, [66, 55, 44], (7, 8, 9)]
l2 = list(l1)     #浅复制了l1
l1.append(100)     #l1列表在尾部添加数值100
l1[1].remove(55)    #移除列表中第1个索引的值
print('l1:', l1)
print('l2:', l2)
l2[1] += [33, 22]    #l2列表中第1个索引做列表拼接
l2[2] += (10, 11)    #l2列表中的第2个索引做元祖拼接
print('l1:', l1)
print('l2:', l2)
```

```
l1: [3, [66, 44], (7, 8, 9), 100]
l2: [3, [66, 44], (7, 8, 9)]
l1: [3, [66, 44, 33, 22], (7, 8, 9), 100]
l2: [3, [66, 44, 33, 22], (7, 8, 9, 10, 11)]
```

Print output (drag lower right corner to resize)

```
l1: [3, [66, 44], (7, 8, 9), 100]
l2: [3, [66, 44], (7, 8, 9)]
l1: [3, [66, 44, 33, 22], (7, 8, 9), 100]
l2: [3, [66, 44, 33, 22], (7, 8, 9, 10, 11)]
```



深复制

```
class Bus:
    def __init__(self, passengers=None):
        if passengers is None:
            self.passengers = []
        else:
            self.passengers = list(passengers)

    def pick(self, name):
        self.passengers.append(name)

    def drop(self, name):
        self.passengers.remove(name)
```

```
from copy import copy, deepcopy

bus1 = Bus(['Alice', 'Bill', 'Claire', 'David'])
bus2 = copy(bus1)       #bus2浅复制的bus1
bus3 = deepcopy(bus1)      #bus3深复制了bus1
print(id(bus1), id(bus2), id(bus3))  #查看三个对象的内存地址

bus1.drop('Bill')       #bus1的车上Bill下车了
print('bus2:', bus2.passengers)    #bus2中的Bill也没有了!
print(id(bus1.passengers), id(bus2.passengers), id(bus3.passengers))
#审查 passengers 属性后发现，bus1和bus2共享同一个列表对象，因为bus2是bus1的浅复制副本

print('bus3:', bus3.passengers)
#bus3是bus1 的深复制副本，因此它的 passengers 属性指代另一个列表
```

```
1773854639680 1773854638912 1773854638528
bus2: ['Alice', 'Claire', 'David']
1773855364608 1773855364608 1773855409600
bus3: ['Alice', 'Bill', 'Claire', 'David']
```

深复制

- 循环引用：b 引用 a，然后追加到 a 中；deepcopy 会想办法复制 a
  - 尽量避免递归复制

```
a = [10, 20]
b = [a, 30]
a.append(b)
print(a)

from copy import copy, deepcopy
c = deepcopy(a)
e = a.copy()
print(c)
print(e)
```

```
[10, 20, [[...], 30]]
[10, 20, [[...], 30]]
[10, 20, [[10, 20, [...]], 30]]
```

函数的参数作为引用

- Python 唯一支持的参数传递模式是传递对象的引用的复本
  - 共享传参（call by sharing）

- 共享传参指函数的各个形式参数获得实参中各个引用的副本。也就是说，函数内部的形参是实参的别名。
  - 传了一个指向相同对象的“标签”

- 函数可能会修改接收到的任何可变对象

```
def f(a, b):
    a += b
    return a

x = 1
y = 2
print(f(x, y))
print(x, y)

a = [1, 2]
b = [3, 4]
print(f(a, b))
print(a, b)

t = (10, 20)
u = (30, 40)
print(f(t, u))
print(t, u)
```

```
3
1 2
[1, 2, 3, 4]
[1, 2, 3, 4] [3, 4]
(10, 20, 30, 40)
(10, 20) (30, 40)
```

函数的参数作为引用

- **思考**：什么时候会修改函数外的数值?

```
>>> a = 523432
>>> id(a)
37656816
>>> def f(x):
...     x *= 3
...     return x
...
>>> f(a)
1570296
>>> id(a)
37656816
>>> print(a)
523432
```

```
>>> b = [1]
>>> id(b)
39823360
>>> f(b)
[1, 1, 1]
>>> id(b)
39823360
>>> print(b)
[1, 1, 1]
>>>
```

当传入参数指向可变对象时才可能会被修改

函数的参数作为引用

- 是 Python 函数定义时，可选参数可以有默认值，这的一个很棒的特性，这样我们的 API 在进化的同时能保证向后兼容。
- 然而，应该避免使用可变的对象作为参数的默认值。

危险的可变默认值

- 幽灵车

```
class HauntedBus:
    '''
     备受折磨的幽灵车
    '''
    def __init__(self, passengers=[]):
        self.passengers = passengers

    def pick(self, name):
        self.passengers.append(name)

    def drop(self, name):
        self.passengers.remove(name)

bus1 = HauntedBus(['Alice', 'Bill'])
print('bus1上的乘客：', bus1.passengers)
bus1.pick('Charlie')    #bus1上来一名乘客Charile
bus1.drop('Alice')     #bus1下去一名乘客Alice
print('bus1上的乘客：', bus1.passengers)    #打印bus1上的乘客

bus2 = HauntedBus()     #实例化bus2
bus2.pick('Carrie')     #bus2上来一名课程Carrie
print('bus2上的乘客：', bus2.passengers)
```

```
bus3 = HauntedBus()
print('bus3上的乘客：', bus3.passengers)
bus3.pick('Dave')
print('bus2上的乘客：', bus2.passengers)
#登录到bus3上的乘客Dava跑到了bus2上面

print('bus2是否为bus3的对象：', bus2.passengers is bus3.passengers)
print('bus1上的乘客：', bus1.passengers)
```

```
bus1上的乘客： ['Alice', 'Bill']
bus1上的乘客： ['Bill', 'Charlie']
bus2上的乘客： ['Carrie']
bus3上的乘客： ['Carrie']
bus2上的乘客： ['Carrie', 'Dave']
bus2是否为bus3的对象： True
bus1上的乘客： ['Bill', 'Charlie']
```

幽灵车

- 实例化 HauntedBus 时，如果传入乘客，会按预期运作。
- 如果不为 HauntedBus 指定乘客的话，奇怪的事就发生了，这是因为 self.passengers 变成了 passengers 参数默认值的别名。
- 出现这个问题的根源是，默认值在定义函数时计算（通常在加载模块时），因此默认值变成了函数对象的属性。
- 如果默认值是可变对象，而且修改了它的值，那么后续的函数调用都会受到影响。

思考

- 幽灵车解决方案?

```
class HauntedBus:
    '''
     备受折磨的幽灵车
    '''
    def __init__(self, passengers=[]):
        self.passengers = list(passengers)

    def pick(self, name):
        self.passengers.append(name)

    def drop(self, name):
        self.passengers.remove(name)
```

变量的使用“法则”

- **在同一作用域下**，使用不同变量进行计算，可以不用理解背后的原理，正常进行编码即可。
  - 绝大部分情况下也是如此
- **在函数内（或嵌套作用域），**对于不可变对象的修改对于外部一般来说不会产生影响，而对于可变对象的修改会对外部产生影响。
  - 如果期望函数修改外部的参数所引用的对象，应该传可变对象
  - 否则传不可变对象
- 特别的地方要特殊考虑
  - 如global、nonlocal等关键字，不推荐使用太多的全局变量
  - 如默认参数尽量不使用可变对象
  - None：空值，但是与0，”“，list(), set()等不同

如何将这门课学好

- 动手是学会编程的唯一途径
  - 确保所有课件上的代码，完全手敲一遍
  - 确保所有的作业认真完成，做之前认真研读课件和提示代码
  - 没有动手意味着没有“真正的学习”
- 阅读一些升级知识，提升对于一些难点的理解
  - 知乎、stackoverflow、python文档
  - 图书不要乱买，可以查询高频推荐的图书
- 不要轻易相信任何网络、书籍、课件上的知识
  - Test it by yourself with a python interpreter!!!
- 阅读一些较好的代码、标程，理解后复写、分模块改写

“写而不思则罔，思而不写则殆”

习题讲解

选项	百分比%	小计
第二次_H_实现分词函数word_cut	75.00%	15
第二次_I_奇怪的字典序	55.00%	11
第一次_G 判断字符串是否由字典中的词组成	50.00%	10
第三次_J_auto_market	45.00%	9
第二次_J_面向对象方法实现宠物店管理	40.00%	8
第一次_H 采用面向对象方法实现列表元素计数	25.00%	5
第三次_D_elementwise_product	10.00%	2
第二次_G_实现函数min_in_list_2	10.00%	2
.....	.....	.....

谢谢！