《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

OOP总结：Why OOP?

- 面向对象编程（Object Oriented Programming， OOP）是一种程序设计思想。
  - OOP把对象作为程序的基本单元，一个对象包含了数据和操作数据的函数
  - OOP把计算机程序视为一组对象的集合，而每个对象都可以接收其他对象发过来的消息，并处理这些消息，计算机程序的执行就是一系列消息在各个对象之间传递。
  - 实现了数据的抽象和封装
    - 对象（实例）与客观世界中的的实体对应
  - 可复用性
  - 可拓展性

OOP总结：核心概念

- 类与实例
  - 实例（对象）
    - 保存在计算机内存中的一个object
    - 对数据进行抽象和封装
    - 数据保存在**实例属性**里
  - 从求解问题的角度，类是实例的**抽象**
    - 例：Michael，Kristen都是Student
      - 注意与Dog，Cat都是Animal区分
  - 从程序实现的角度，类是实例的**模板**
    - 通过__init__函数绑定实例共有的属性
    - 通过定义方法，定义实例共有的功能
      - 方法被保存在**类属性**中

```
In [15]: 1 animal
Out[15]: <__main__.Dog at 0x7fb6b106d9a0>
```

```
class Animal(object):
    name = 'Animal'
    def __init__(self):
        print('This is an animal class.')

    def live(self):
        print("{}'s life:".format(self.name))
        self.eat()
        self.play()
        self.sleep()

    def play(self):
        print('Animal is playing')

    def eat(self):
        print('Animal is eating')

    def sleep(self):
        print('Animal is sleeping')
```

OOP总结：核心概念

- 继承：
  - 在OOP 中定义一个class 时，可以从某个现有的class继承，新的class 称为子类(subclass)，而被继承的class 称为基类、父类或超类(base class，super class)。
  - 继承：一个派生类(derived class)继承基类的字段和方法，即子类获得了父类的全部功能。
  - 举例：Dog类继承Animal类

[1]:
```
class Animal(object):
    def run(self):
        print('Animal is running')

class Dog(Animal):
    pass

class Cat(Animal):
    pass

animal = Dog()
animal.run()
```

```
Animal is running
```

OOP总结：核心概念

- 多态：
  - 为不同数据类型的实体提供统一的**接口**。
  - 相同的消息给予不同的对象会引发不同的动作

```
class Dog(Animal):
    def play(self):
        print('Dog like playing frisbee!')

class Husky(Dog):
    def play(self):
        print('Husky like playing sofa!')

animal1 = Husky()
animal1.play()
animal2 = Dog()
animal2.play()
```

```
This is an animal class.
Husky like playing sofa!
This is an animal class.
Dog like playing frisbee!
```

```
[4]: class Dog(Animal):

        def run(self):
            print('Dog is running...')

        def eat(self):
            print('Eating meat...')
```

```
[5]: class Cat(Animal):

        def run(self):
            print('Cat is running...')

        def eat(self):
            print('Eating meat...')
```

```
def run_animal(animal):
    animal.run()

run_animal(Animal())
run_animal(Dog())
run_animal(Cat())
```

```
Animal is running...
Dog is running...
Cat is running...
```

OOP总结：实现原理

- 类属性vs.实例属性
  - 类属性：绑定在类上的属性
    - 将函数绑定在类上，即可定义方法
  - 实例属性：绑定在实例上的属性
    - 用来保存和实例有关的数据和信息
- 属性（和方法）查找过程:
  - 首先，在实例属性中查找（所以同名实例属性会覆盖类属性）
  - 然后，在类属性中查找
  - 然后，在父类属性中查找（继承的实现机制）
  - 若在类或父类属性中找到一个类型为**函数**的属性，则自动返回一个类型为**方法**的属性
  - 注意：该查找过程是在程序运行时**动态完成**的
    - 动态绑定
    - 动态查找

OOP总结：实现原理

- 对象的创建：
  - 在创建对象时，解释器会自动调用一个特殊的方法__init__，该方法被称为构造函数
  - 如果不提供构造函数，则会自动使用父类的构造函数
  - 通常我们会在构造函数里给对象绑定必须的实例属性

```
In [1]: class A(object):
            def __init__(self):
                print("A() is created")

        a = A()
```

```
A() is created
```

```
In [3]: class B(A):
            pass

        b = B()
```

```
A() is created
```

OOP总结：实现原理

- 对象的创建：
  - 如果子类有构造函数，如果需要调用父类的构造函数，需要显式调用

In [43]:
```
class B3(A):
    def __init__(self):
        super().__init__()
        print('I am called')
        #pass

b3 = B3()
```

```
A() is created
I am called
```

OOP总结：实现原理

- 对象的创建：
  - 如果类的构造函数已经包含了参数，创建对象时需给与正确的参数

```
In [44]: class A2(object):
             def __init__(self, name):
                 self.name = name
                 print("A() is created, ", name)

         a1 = A2('A1')
         a2 = A2()
```

```
A() is created,  A1
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-44-b78d6ae782e9> in <module>
      5
      6 a1 = A('A1')
----> 7 a2 = A()

TypeError: __init__() missing 1 required positional argument: 'name'
```

OOP总结：实现原理

- 对象创建：
  - 如果父类的构造函数已经包含了参数，子类调用父类的构造函数时需要给与正确的参数

In [47]:
```
class B2(A2):
    def __init__(self, name):
        super().__init__(name)

b = B2('hello')
```

```
A() is created,  hello
```

OOP总结：实现原理

- 如果子类没有正确创建父类呢?

In [54]:

```
class D(object):
    def __init__(self, name):
        self.name = name
        print("D() is created ", name)
    def hello(self):
        print("hello world ", self.name)

d = D("Dragon")
d.hello()
#print(d.name)
```

```
D() is created  Dragon
hello world  Dragon
```

In [55]:

```
class C(D):
    def __init__(self):
        #super().__init__('xxx')
        print("C()")
        pass
c = C()
c.hello()
```

```
C()
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-55-e0bd3b9ae103> in <module>
      5         pass
      6 c = C()
----> 7 c.hello()

<ipython-input-54-a0ae7fadaa4f> in hello(self)
      4         print("D() is created ", name)
      5     def hello(self):
----> 6         print("hello world ", self.name)
      7
      8 d = D("Dragon")

AttributeError: 'C' object has no attribute 'name'
```

OOP总结：实现原理

- 鸭子类型：
  - 在程序设计中，鸭子类型(英语:duck typing)是动态类型(Dynamic Typing)的一种风格。
  - 在这种风格中， 一个对象有效的语义，不是由继承自特定的类或实现特定的接口，而是由”当前方法和属性的集合”决定。
  - 来源于由 James Whitcomb Riley 提出的鸭子测试:
    - 当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子。
  - 相较于继承来说更加灵活
    - 可以通过重载__xxx__等特殊方法来改变自定义类的行为

实现二维列表类

- 通过重载__xxx__等特殊方法来改变自定义类的行为
  - 目的：使其行为更像python自带的list对象
  - 思考这样有什么优点和缺点
- 重载：
  - __repr__： 打印实例时输出字符串
  - __len__： 内置函数len()的返回值
  - __getitem__: []操作符
  - __setitem__: 赋值语句中的[]操作符
  - __iter__ and __next__： for循环中使用

for循环

- for x in y:

  <循环体语句块>

- 执行步骤：
  - 调用y.__iter__()获得一个迭代器（iterator） iter
  - 每次循环：
    - x = iter.__next__()
    - 执行<循环体语句块>
    - 直到抛出StopIteration异常

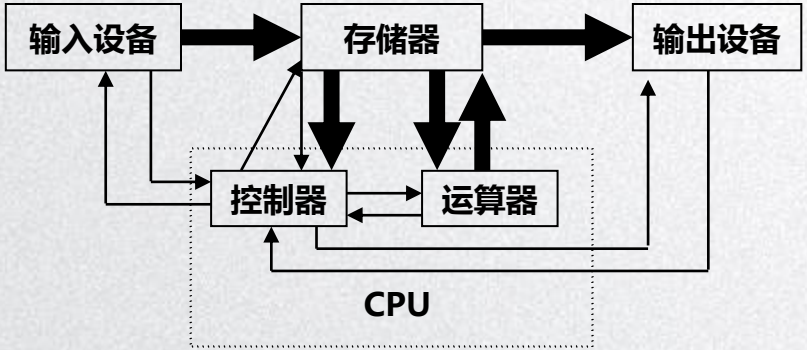
Python进阶知识补充

提纲

- 变量、对象引用

回顾：计算机的基本组成

- 冯•诺依曼体系结构



- 计算机是如何储存数据的?
  - 以**二进制**的形式保存在**存储器**（内存）中
  - 每个存储器单元有一个地址
    - 内存地址：
  - CPU可以通过**内存地址**找到内存单元，并访问、修改内存中保存的数据

变量

- 在**机器语言**中：
  - 需要使用**内存地址**来访问和修改内存中保存的数据
- 在**高级语言**(如C语言、Python)中：
  - 允许我们使用**变量**来访问和修改数据
  - 相较于内存地址，使用变量更方便程序员理解和编写程序

类型

- 与变量相关的一个概念是**类型(type)**
  - 内存里保存的相同的二进制数据可能代表不同的含义
  - 1100001
    - 整数：97
    - ASCII编码的字符： ‘a’
  - 所以我们需要通过变量或者对象的类型，来确定程序如何理解和使用内存里保存的二进制数据
  - 在C语言里，每个变量都有自己的类型，在声明变量时确定
    - 静态类型：变量的类型在**编译时确定**
  - 在Python语言里，每个对象都有自己的类型
    - 不需要提前声明变量和变量类型，变量的类型就是其引用的对象的类型
    - 动态类型：对象（和变量）的类型在**运行时确定**

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

Python中的变量

- Python: “贴标签式” “起名字式”

```
>>> a=3
>>> print(a)
3
>>> a='3'
>>> print(a)
3
>>> a=(3,)
>>> print(a)
(3,)
>>> a={3}
>>> print(a)
{3}
>>> a=(3)
>>> print(a)
3
```

动态类型

- Python不像C，C++，Java等语言一样，可以不用事先声明变量类型而直接对变量进行赋值。

- 对Python语言来讲，变量的类型是在运行时确定的。

- a = 1
  - 变量为a，1是对象

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

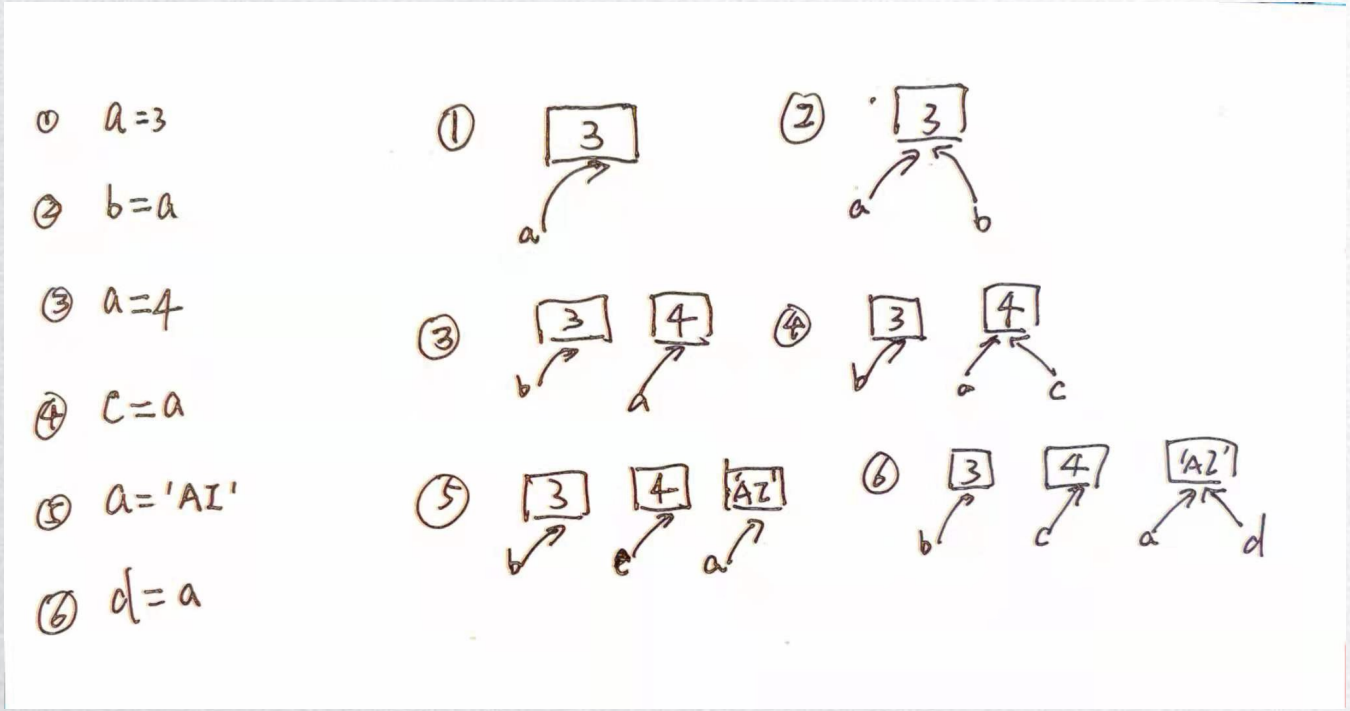
```
[1]: import this

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!
```

Python中的变量

- Python: “贴标签式”



变量引用对象

- Python中的变量可以看作是对象的引用
  - **python中的变量没有类型信息，类型的概念存在于对象中而不是变量中。**
    - 变量是通用的，它只是引用了一个特定的对象，即只是恰巧在某个时间点上引用了当时的特定对象而已。
    - 就比如说在表达式中，我们用的那个变量会立马被它当时所引用的特定对象所替代。
  - 思考：Python中**类型**到底指什么？

相等性

- ==：比较对象所存储的数据的值是否相等
- is: 比较两个变量是否都引用了同一个对象

相等性

- ==：比较对象所存储的数据的值是否相等
- is: 比较两个变量是否都引用了同一个对象

```
>>> a=12313
>>> b=12313
>>> a==b
True
>>> a is b
False
>>> a=b
>>> a == b
True
>>> a is b
True
>>>
```

id()函数

- **id()** 函数用于获取对象（变量所指的对象）的内存地址（标识符）。

```
>>> a=12313
>>> b=12313
>>> id(a)
37656880
>>> id(b)
37656944
>>> a=b
>>> id(a)
37656944
>>> id(b)
37656944
>>>
```

id(), is, ==的区别与联系

- 别名：Lewis Carroll 是 Charles Lutwidge Dodgson 教授的笔名。Carroll 先生指的就是 Dodgson 教授，二者是同一个人

```
charles = {'name': 'Charles L. Dodgson', 'born': 1832, 'balance': 900}
lewis = charles
print(lewis is charles)
print(id(lewis), id(charles))
lewis['balance'] = 950
print(charles)
```

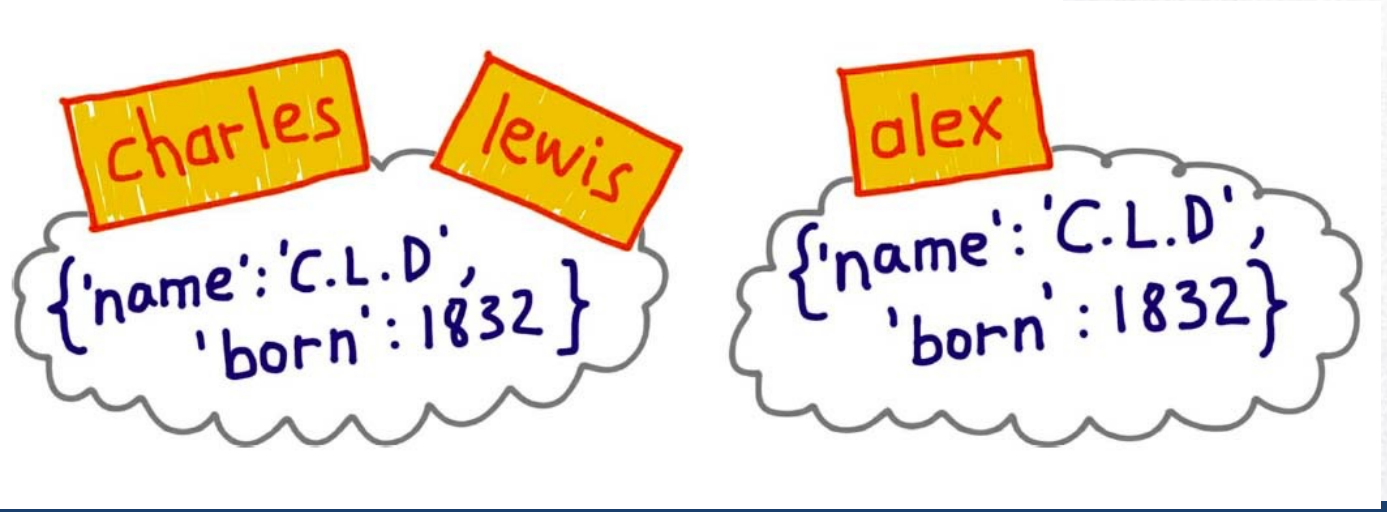
```
True
1773855252736 1773855252736
{'name': 'Charles L. Dodgson', 'born': 1832, 'balance': 950}
```

id(), is, ==的区别与联系

- 冒充者（Alexander Pedachenko 博士）声称他是 Charles L. Dodgson。这个冒充者的证件可能一样，但是不是 Dodgson 教授

```
alex = {'name': 'Charles L. Dodgson', 'born': 1832, 'balance': 950}
print(lewis == alex)
print(alex is not lewis)
```

```
True
True
```



别忘记另一个有用的函数

- type()

```
# 一个参数实例
>>> type(1)
<type 'int'>
>>> type('runoob')
<type 'str'>
>>> type([2])
<type 'list'>
>>> type({0:'zero'})
<type 'dict'>
>>> x = 1
>>> type( x ) == int    # 判断类型是否相等
True

# 三个参数
>>> class X(object):
...     a = 1
...
>>> X = type('X', (object,), dict(a=1))  # 产生一个新的类型
>>> X
<class '__main__.X'>
```

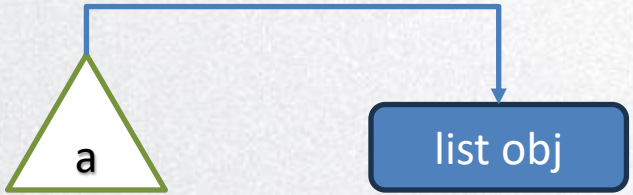
https://www.runoob.com/python/python-func-type.html

对象的“回收”

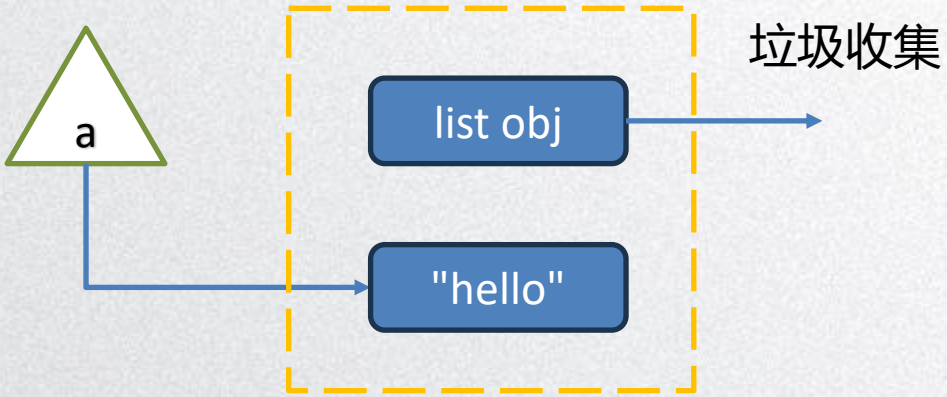
- 类型属于对象，并且对象中包含了一个引用计数器，用于记录当前有多少个变量在引用这个对象。
  - 一旦引用计数器为0，那么该对象就会被系统自动回收（这里有例外，python中缓存了一些小的常用的对象）

对象的“回收”

- 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
  - 给出一些例子?

- 不可变对象无法修改，包括数字、字符串str，元组tuple
  - 给出一些例子?

可变对象和不可变对象

- **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

可变对象和不可变对象

- 不可变对象发生修改：

```
name="Python"
name+="AI"
```

- 不可变对象：
  - 第一句创建一个字符串对象，并让变量name引用该对象。
  - 按照C++中的理解，第二句试图修改name这个字符串。
  - 但是在python中，其实新建了一个值为”PythonAI”的字符串对象，并让name引用该对象。

如何验证这一情况？

可变对象和不可变对象

- 不可变对象发生修改：

```
name="Python"
name+="AI"
```

- 不可变对象：

```
>>> name = 'python'
>>> id(name)
39862128
>>> name2 = name
>>> print(name2)
python
>>> id(name2)
39862128
>>> name += ' AI'
>>> print(name)
python AI
>>> id(name)
39866416
>>> id(name2)
39862128
>>> print(name2)
python
>>>
```

  - 第一句创建一个字符串对象，并让变量name引用该对象。
  - 按照C++中的理解，第二句试图修改name这个字符串。
  - 但是在python中，其实新建了一个值为”PythonAI”的字符串对象，并让name引用该对象。

如何验证这一情况？

可变对象和不可变对象

- 不可变对象发生修改：
  - 更多例子

```
>>> a = (2,3)
>>> id(a)
39742592
>>> a += (4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate tuple (not "int") to tuple
>>> a += (4,)
>>> a
(2, 3, 4)
>>> id(a)
39510528
>>>
```

```
>>> a = 323
>>> id(a)
37656880
>>> b = 323
>>> a += 1
>>> id(a)
37656944
>>> print(a)
324
>>> print(b)
323
>>> id(b)
37656496
>>>
```

可变对象和不可变对象

- 可变对象发生修改：
  - 不再创建新的对象，在原始的对象上进行修改
  - 因此对象的地址不会发生改变

```
>>> a = []
>>> id(a)
39823424
>>> a.append(1)
>>> a
[1]
>>> id(a)
39823424
```

```
>>> a = dict()
>>> id(a)
39866624
>>> a['3'] = 32
>>> a
{'3': 32}
>>> id(a)
39866624
```

可变对象和不可变对象

- 可变与不可变指的是**顶层对象**不可变
  - 不可变是指所包含对象的标识符不能发生改变
    - 也就是每个元素对应的id值不动
  - 但是如果元素本身可以被修改，则不限制

```
>>> a = (1, 3, [])
>>> a[1] = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>>
```

```
>>> a = (1, 3, [])
>>> [id(x) for x in a]
[8790970992304, 8790970992368, 39823360]
>>> a[-1].append(1)
>>> [id(x) for x in a]
[8790970992304, 8790970992368, 39823360]
>>>
```

可变对象和不可变对象

- 元组的相对不可变性
  - 元组是不可变对象
  - 与多数 Python 组合数据类型（列表、字典、集，等等）一样，元组保存的是对象的引用。
    - 如果引用的元素是可变的，即便元组本身不可变，元素依然可变。
    - 也就是说，元组的不可变性其实是指 tuple 数据结构的物理内容（即保存的引用）不可变，与引用的对象无关。
- 元组的值会随着引用的可变对象的变化而变。元组中不可变的是元素的标识。

```
t1 = (1, 2, [30, 40])
t2 = (1, 2, [30, 40])
print(t1 == t2)
```

```
True
```

```
id(t1[-1])
```

```
1773854962688
```

```
t1[-1].append(1000)
print(t1)
print(t1 == t2)
print(id(t1[-1]))
```

```
(1, 2, [30, 40, 1000])
False
1773854962688
```

可变对象（更多例子）

- list1、list2[1]和dict1[‘list1’]都是同一个list对象的引用，并且由于list对象是可变对象，通过上面三个变量中的任意一个变量修改该list对象都会影响到其余的变量

```
list1=[1,2,3,4]
list2=['hello',list1]
dict1={'list1':list1}

list1[0] = 0
print(list2)
print(dict1)
```

```
['hello', [0, 2, 3, 4]]
{'list1': [0, 2, 3, 4]}
```

可变对象（更多例子）

```
l1 = [1,2,3]
l2 = l1
l1[0] = 0
print(l1)
print(l2)
print(l1 == l2)
print(l1 is l2)
```

```
[0, 2, 3]
[0, 2, 3]
True
True
```

```
l1 = [1,2,3]
l3 = [1,2,3]
l1[0] = 0
print(l1)
print(l3)
```

```
[0, 2, 3]
[1, 2, 3]
```

```
l1 = [1,2,3]
l3 = [1,2,3]
print(l1 == l3)
print(l1 is l3)
```

```
True
False
```

```
old = [1, 2, 3, 4, 5]
new = old
old = [6]
print(old)
print(new)
```

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

```
print(id(old))
print(id(new))
```

```
1773854129216
1773855362112
```

相等性

- 例外

```
a = 2
b = 2
print(a == b)
print(a is b)
```

```
True
True
```

- a和b两个变量不是引用两个不同的对象?
- Python在底层做了一定的优化，对于使用过整数以及字符串都会被缓存起来。所以上述b引用的应该是被缓存过的2
- python中数字和字符串一经创建都是不可修改的

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

- 复制列表（或多数内置的可变集合）最简单的方式是使用内置的类型构造方法；默认是浅复制

```
l1 = [3, [55, 44], (7, 8, 9)]
print(id(l1))
l2 = list(l1)
l3 = l1[:]
print(l2,id(l2))
print(l3,id(l3))
print(l1 == l2 == l3)
print(l2 is l1)
print(l3 is l1)
```

```
1773855374528
[3, [55, 44], (7, 8, 9)] 1773855424640
[3, [55, 44], (7, 8, 9)] 1773855374144
True
False
False
```

```
from copy import copy
l4 = l1.copy()
print(id(l4))
print(l4 is l1)
print(id(l1[1]))
print(id(l4[1]))
```

```
1773854123328
False
1773855374400
1773855374400
```

浅复制

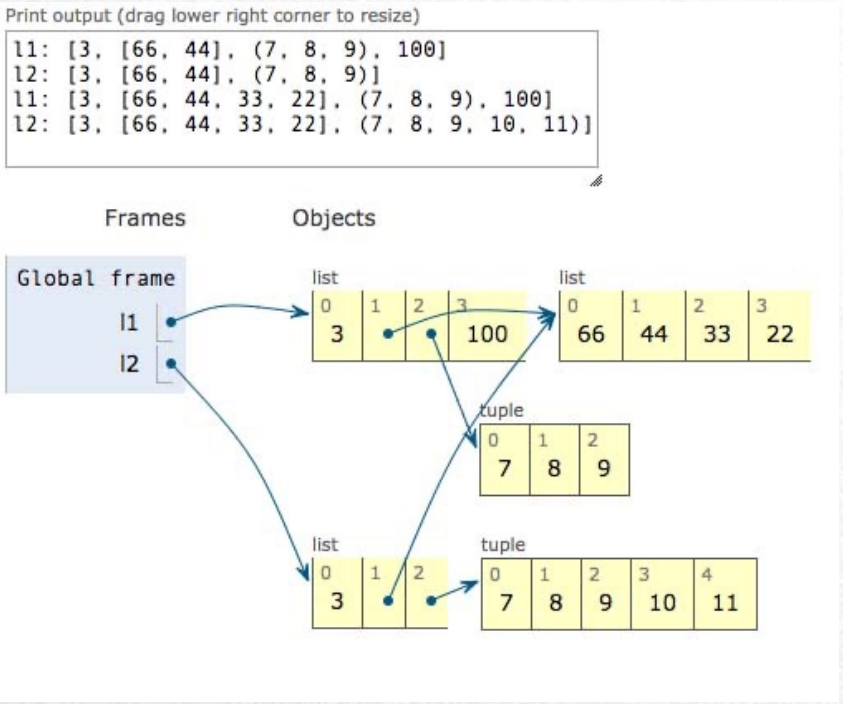
- l2 是 l1 的浅复制副本

```
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

```python
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)
```

```python
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”

谢谢！