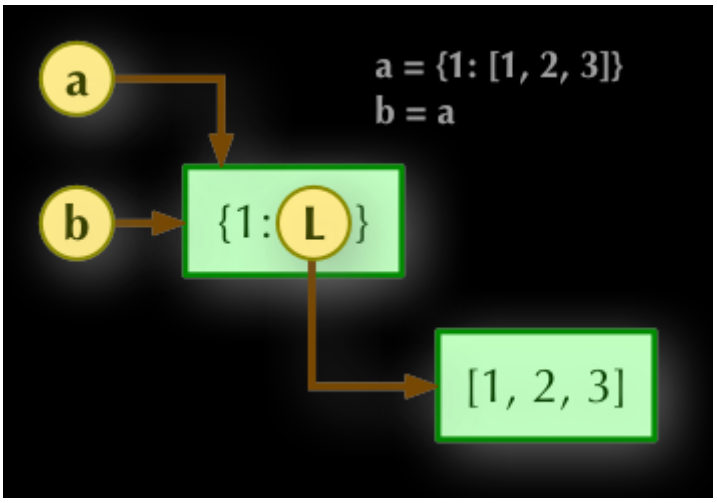
一、浅拷贝+深拷贝

浅拷贝就是对引用的拷贝，所谓深拷贝就是对对象的资源的拷贝。

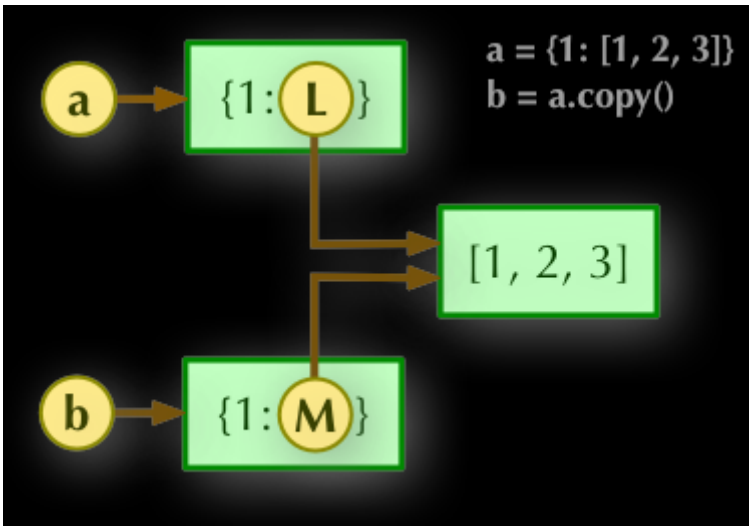
0. 赋值引用



1. 浅拷贝

仅仅复制了源容器中初始元素的地址

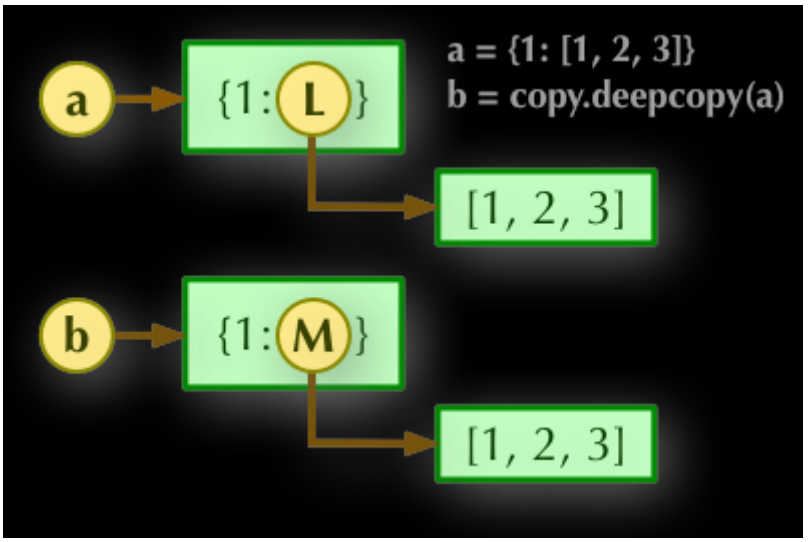
浅拷贝是在另一块地址中创建一个新的变量或容器，但是容器内的元素的地址均是源对象的元素的地址的拷贝。也就是说新的容器中指向了旧的元素（ 新瓶装旧酒 ）。



2. 深拷贝

完全拷贝了一个副本，容器内部元素地址都不一样

深拷贝是在另一块地址中创建一个新的变量或容器，同时容器内的元素的地址也是新开辟的，仅仅是值相同而已，是完全的副本。也就是说（ 新瓶装新酒 ）



3.赋值操作

我们要有以下认识：

1. 赋值是将一个对象的地址赋值给一个变量，让变量指向该地址（旧瓶装旧酒）。
2. 修改不可变对象（str、tuple）需要开辟新的空间
3. 修改可变对象（list等）不需要开辟新的空间

4. 测试代码如下：

```
import copy
a=[1,2,['a','b']
b=copy.copy(a)  # 浅拷贝
>>> [id(x) for x in a]
[140714547533472, 140714547533504, 1474744245888]
>>> [id(x) for x in b]
[140714547533472, 140714547533504, 1474744245888]
# ------------------------------------------------
# 进行操作:a.append(5)、a[1]='str'、a[2].append('c')
# 操作1
a.append(5)
>>> [id(x) for x in a]   # a的地址发生了改变，增加了一个
[140714547533472, 140714547533504, 1474744245888, 140714547533600]
>>> [id(x) for x in b]
[140714547533472, 140714547533504, 1474744245888]
# 操作2
a[1]='str'
>>> [id(x) for x in a]   # a[1]的地址发生了改变
[140714547533472, 1474696587696, 1474744245888, 140714547533600]
>>> [id(x) for x in b]
[140714547533472, 140714547533504, 1474744245888]
# 操作3
a[2].append('c')        # a[2]的地址未改变
```

```
>>> [id(x) for x in a]
[140714547533472, 1474696587696, 1474744245888, 140714547533600]
>>> [id(x) for x in b]
[140714547533472, 140714547533504, 1474744245888]
```

5. 总结

所谓浅拷贝就是对引用的拷贝，所谓深拷贝就是对对象的资源的拷贝。

二、python中常用的几个容器

容器，是Python 中的一个抽象概念，是对专门用来装其他对象的数据类型的统称。

在Python 中，有四类最常见的内建容器类型： 列表（list） 、 元组（tuple） 、字典（dict） 、 集合（set） 。 通过单独或是组合使用它们，可以高效的完成很多事情。

三、高级特性

1. 生成器

01. 生成器是一个返回迭代器的函数。

不需要像迭代器的类一样写__iter__()和__next__()方法，只需要一个yiled关键字，每次遇到yield时函数会暂停并保存当前所有的运行信息，返回yield的值,并在下一次执行next()方法时从当前位置继续运行。

02. 创建generator方法：

（1） **把一个列表生成式的[]改成()**

```
>>> L = [x * x for x in range(10)]
>>> L
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> g = (x * x for x in range(10))
>>> g
<generator object <genexpr> at 0x1022ef630>
```

创建L和g的区别仅在于最外层的[]和()，L是一个list，而g是一个generator。通过next()函数获得generator的下一个返回值：

```
>>> next(g)
0
>>> next(g)
1
>>> next(g)
4
>>> next(g)
```

```
9
>>> next(g)
16
>>> next(g)
25
>>> next(g)
36
>>> next(g)
49
>>> next(g)
64
>>> next(g)
81
>>> next(g)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

*可用for循环便捷化输出。

（2）range(数组)、斐波拉且数列

```
>>> g = (x * x for x in range(10))
>>> for n in g:
...     print(n)
...     yield n


def fib(max):
    n, a, b = 0, 0, 1
    while n < max:
        yield b
        a, b = b, a + b
        n = n + 1
    return 'done'

>>> f = fib(6)
>>> f
<generator object fib at 0x104feaaa0>
```

（3）难点：generator和函数的执行流程不一样

函数是顺序执行，遇到`return`语句或者最后一行函数语句就返回。

而变成generator的函数，在每次调用`next()`的时候执行，遇到`yield`语句返回，再次执行时从上次返回的`yield`语句处继续执行。

```
def odd():
    print('step 1')
    yield 1
    print('step 2')
    yield(3)
```

```
    print('step 3')
    yield(5)

>>> o = odd()
>>> next(o)
step 1
1
>>> next(o)
step 2
3
>>> next(o)
step 3
5
>>> next(o)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

2. 迭代器

迭代器是一个可以记住遍历的位置的对象。迭代器对象从集合的第一个元素开始访问，直到所有的元素被访问完结束。迭代器只能往前不会后退。迭代器有两个基本的方法：iter() 和 next()

凡是可作用于for循环的对象都是Iterable类型；

凡是可作用于next()函数的对象都是Iterator类型，它们表示一个惰性计算的序列；

集合数据类型如list、dict、str，set等是Iterable但不是Iterator，不过可以通过iter()函数获得一个Iterator对象。

Python的for循环本质上就是通过不断调用next()函数实现的，例如：

```
for x in [1, 2, 3, 4, 5]:
    pass

# 等价于:
# 首先获得Iterator对象:
it = iter([1, 2, 3, 4, 5])
# 循环:
while True:
    try:
        # 获得下一个值:
        x = next(it)
    except StopIteration:
```

```
        # 遇到StopIteration就退出循环
        break
```

四、函数式编程

1. 装饰器

装饰器本质上是一个Python函数，它可以让其它函数在不作任何变动的情况下增加额外功能，装饰器的返回值也是一个函数对象。比如：**插入日志、性能测试、事务处理、缓存、权限校验**等。

有了装饰器我们就可以抽离出大量的与函数功能无关的雷同代码进行重用。装饰器其实就是一个闭包。

2. 代码

```
# 定义装饰器函数：
def log(func):
    def wrapper(*args, **kw):
        print('call %s():' % func.__name__)
        return func(*args, **kw)
    return wrapper

# 使用装饰器：
@log
def now():
    print('2015-3-25')

# 执行函数:
now()
#输出结果：
call now():
2015-3-25

# ------------------------------------------
如果decorator本身需要传入参数，那就需要编写一个返回decorator的高阶函数。
但写出来会更复杂。比如，要自定义log的文本：
def log(text):
    def decorator(func):
        def wrapper(*args, **kw):
            print('%s %s():' % (text, func.__name__))
            return func(*args, **kw)
        return wrapper
    return decorator

@log('execute')
def now():
```

```
    print('2015-3-25')

# 输出结果
execute now():
2015-3-25
```

完整的decorator的写法如下：

```
import functools
# 不带参数的decorator:
def log(func):
    @functools.wraps(func)
    def wrapper(*args, **kw):
        print('call %s():' % func.__name__)
        return func(*args, **kw)
    return wrapper

# 带参数的decorator:
def log(text):
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kw):
            print('%s %s():' % (text, func.__name__))
            return func(*args, **kw)
        return wrapper
    return decorator
```

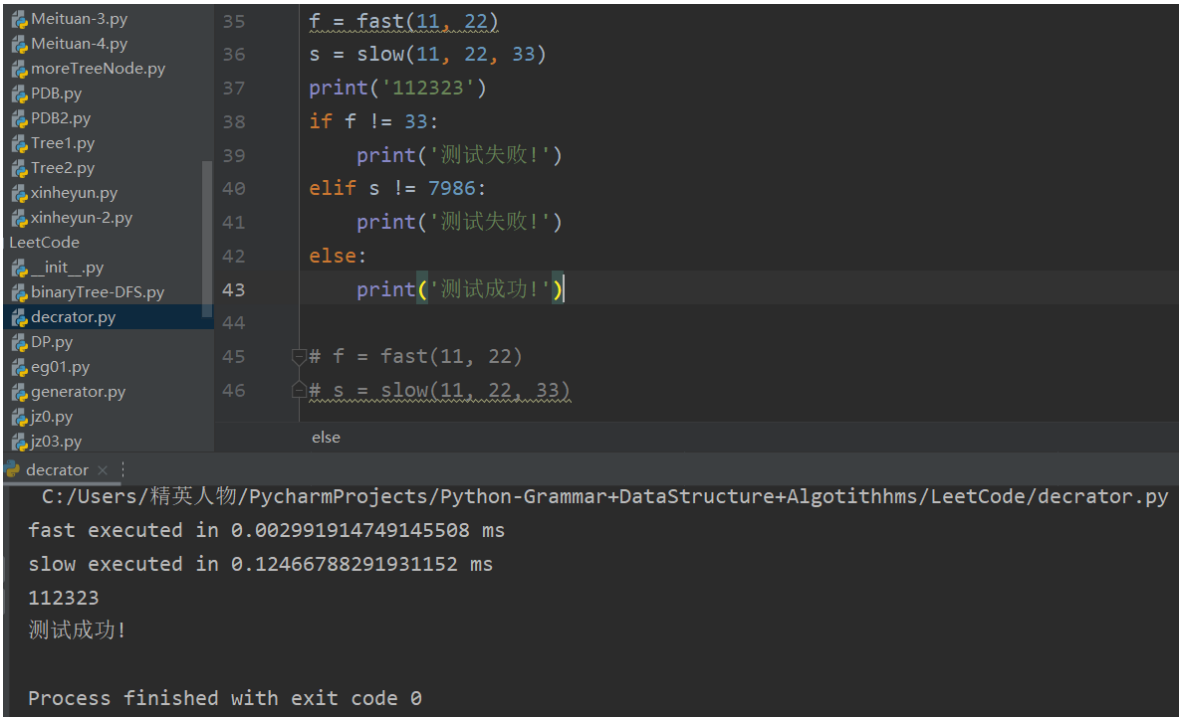
3.例子

测试某个函数之的时间装饰器

```python
def metric(fn):
    @functools.wraps(fn)
    def wraps(*args,**kw):
        t0 = time.time()
        res = fn(*args,**kw)
        t1 = time.time()
        print('%s executed in %s ms' % (fn.__name__, t1-t0))
        return res
        # return fn(*args,**kw)
    return wraps

# 测试
@metric
def fast(x, y):
    time.sleep(0.0012)
    return x + y

@metric
def slow(x, y, z):
    time.sleep(0.1234)
    return x * y * z
```



五、进程和线程

六、hashmap原理

哈希表，根据键Key，直接对储存位置进行访问的数据结构。存储位置可直接通过哈希函数计算。 常用的哈希函数:

1. 直接定置法. 取关键字的某个线性函数值为散列地址. hash(k)=ak+b, a,b为常数.
2. 除留余数法. 取关键字除p的余数为哈希地址. hash(k) = k mod p, p<=m. m 为哈希表长度.
3. 数字分析法. 取关键字的若干数位组成哈希地址.

随着数据的增加,当通过哈希函数计算的存储地址已经有值了,会发生哈希冲突. python 通过开放定址法解决哈希冲突，JAVA hashMap通过拉链法，此外还有再构建哈希函数等方法.

- 开放定址法: 产生哈希冲突时, python 通过一个二次探测函数(增量序列:线性,平方,伪随机), 计算下一个候选位置, 当下一个位置可用, 将数据插入该位置, 如果不可用则再次调用探测函数, 获得下一个候选位置.

开放定址法存在的问题: 通过多次使用二次探测函数f(增量序列)，每一个位置对上一个位置都有依赖，这形成了一个 ‘冲突探测链’，当需要删除探测链上中间的某个数据时，会导致探测链断裂，无法访问到后序位置.

所以采用开放定地法，删除链路上的某个元素时，不能真正的删除元素，只能‘伪删除’. python字典的三种状态 Unused, Active, Dummy. 当字典中的 key 和 value 被删除后字典不能从Active 直接进入 Unused 状态 否则会出现冲突链路中断,实际上python进行删除字典元素时，会将key的状态改为 Dummy ,这就是 python的 ‘伪删除’.

- 拉链法: 将通过哈希函数映射到同一个存储位置的所有元素保存在一个链表中. JAVA 1.8之后, 当链表长度超过阈值时, 将链表转为红黑树.

* 载荷因子 = 填入表中的元素个数 / 散列表的长度 对于开放定址法，载荷因子很重要，应严格限制在0.7-0.8以下。超过0.8，查表时的缓存不命中（cache missing）按照指数曲线上升。超过载荷因子阈值，需要resize扩容哈希表。(扩容后hashcode需要重新计算)

七、*、**的作用 （To Do List）

1. 函数的可变参数

当函数的参数前面有一个星号*的时候表示这是一个可变的位置参数，两个星号**表示是可变的关键字参数。

2. unpack参数

星号*把序列/集合解包（unpack）成位置参数，两个星号**把字典解包成关键字参数。下面通过示例来进一步加深理解：