第九章 有益的探索

尝试着潜入水中，往冰山的深处扎一个小小的猛子

9.1 数据类型的底层实现

9.1.1 从奇怪的列表说起

1、错综复杂的复制

In [1]:

```
list_1 = [1, [22, 33, 44], (5, 6, 7), {"name": "Sarah"}]
```

- 浅拷贝

In [2]:

```
# list_3 = list_1           # 错误！！！
list_2 = list_1.copy()      # 或者list_1[:] \ list(list_1) 均可实习浅拷贝
```

- 对浅拷贝前后两列表分别进行操作

In [3]:

```
list_2[1].append(55)

print("list_1:  ", list_1)
print("list_2:  ", list_2)
```

```
list_1:   [1, [22, 33, 44, 55], (5, 6, 7), {'name': 'Sarah'}]
list_2:   [1, [22, 33, 44, 55], (5, 6, 7), {'name': 'Sarah'}]
```

2、列表的底层实现

引用数组的概念

列表内的元素可以分散的存储在内存中

列表存储的，实际上是这些**元素的地址！！！**——地址的存储在内存中是连续的

In [5]:

```
list_1 = [1, [22, 33, 44], (5, 6, 7), {"name": "Sarah"}]
list_2 = list(list_1)   # 浅拷贝   与list_1.copy()功能一样
```

(1) 新增元素

In [6]:

```
list_1.append(100)
list_2.append("n")

print("list_1:  ", list_1)
print("list_2:  ", list_2)
```

```
list_1:   [1, [22, 33, 44], (5, 6, 7), {'name': 'Sarah'}, 100]
list_2:   [1, [22, 33, 44], (5, 6, 7), {'name': 'Sarah'}, 'n']
```

(2) 修改元素

In [7]:

```
list_1[0] = 10
list_2[0] = 20

print("list_1:  ", list_1)
print("list_2:  ", list_2)
```

```
list_1:   [10, [22, 33, 44], (5, 6, 7), {'name': 'Sarah'}, 100]
list_2:   [20, [22, 33, 44], (5, 6, 7), {'name': 'Sarah'}, 'n']
```

(3) 对列表型元素进行操作

In [8]:

```
list_1[1].remove(44)
list_2[1] += [55, 66]

print("list_1:  ", list_1)
print("list_2:  ", list_2)
```

```
list_1:   [10, [22, 33, 55, 66], (5, 6, 7), {'name': 'Sarah'}, 100]
list_2:   [20, [22, 33, 55, 66], (5, 6, 7), {'name': 'Sarah'}, 'n']
```

(4) 对元组型元素进行操作

In [9]:

```
list_2[2] += (8,9)

print("list_1:  ", list_1)
print("list_2:  ", list_2)
```

```
list_1:   [10, [22, 33, 55, 66], (5, 6, 7), {'name': 'Sarah'}, 100]
list_2:   [20, [22, 33, 55, 66], (5, 6, 7, 8, 9), {'name': 'Sarah'}, 'n']
```

元组是不可变的！！！

(5) 对字典型元素进行操作

In [10]:

```
list_1[-2]["age"] = 18

print("list_1:  ", list_1)
print("list_2:  ", list_2)
```

```
list_1:   [10, [22, 33, 55, 66], (5, 6, 7), {'name': 'Sarah', 'age': 18}, 100]
list_2:   [20, [22, 33, 55, 66], (5, 6, 7, 8, 9), {'name': 'Sarah', 'age': 18}, 'n']
```

3、引入深拷贝

浅拷贝之后

- 针对不可变元素（数字、字符串、元组）的操作，都各自生效了
- 针对不可变元素（列表、集合）的操作，发生了一些混淆

引入深拷贝

- 深拷贝将所有层级的相关元素全部复制，完全分开，泾渭分明，避免了上述问题

In [11]:

```
import copy

list_1 = [1, [22, 33, 44], (5, 6, 7), {"name": "Sarah"}]
list_2 = copy.deepcopy(list_1)
list_1[-1]["age"] = 18
list_2[1].append(55)

print("list_1:  ", list_1)
print("list_2:  ", list_2)
```

```
list_1:   [1, [22, 33, 44], (5, 6, 7), {'name': 'Sarah', 'age': 18}]
list_2:   [1, [22, 33, 44, 55], (5, 6, 7), {'name': 'Sarah'}]
```

9.1.2 神秘的字典

1、快速的查找

In [13]:

```
import time

ls_1 = list(range(1000000))
ls_2 = list(range(500))+[-10]*500

start = time.time()
count = 0
for n in ls_2:
    if n in ls_1:
        count += 1
end = time.time()
print("查找{}个元素，在ls_1列表中的有{}个，共用时{}秒".format(len(ls_2), count,round((end-start
```

查找1000个元素，在ls_1列表中的有500个，共用时6.19秒

In [14]:

```
import time

d = {i:i for i in range(100000)}
ls_2 = list(range(500))+[-10]*500

start = time.time()
count = 0
for n in ls_2:
    try:
        d[n]
    except:
        pass
    else:
        count += 1
end = time.time()
print("查找{}个元素，在ls_1列表中的有{}个，共用时{}秒".format(len(ls_2), count,round(end-start)
```

查找1000个元素，在ls_1列表中的有500个，共用时0秒

2、字典的底层实现

通过稀疏数组来实现值的存储与访问

字典的创建过程

- 第一步：创建一个散列表（稀疏数组 N >> n）

In [97]:

```
d = {}
```

- 第一步：通过hash()计算键的散列值

In [17]:

```
print(hash("python"))
print(hash(1024))
print(hash((1,2)))
```

```
-4771046564460599764
1024
3713081631934410656
```

In [ ]:

```
d["age"] = 18     # 增加键值对的操作，首先会计算键的散列值hash("age")
print(hash("age"))
```

- 第二步：根据计算的散列值确定其在散列表中的位置

极个别时候，散列值会发生冲突，则内部有相应的解决冲突的办法

- 第三步：在该位置上存入值

键值对的访问过程

In [ ]:

```
d["age"]
```

- 第一步：计算要访问的键的散列值

- 第二步：根据计算的散列值，通过一定的规则，确定其在散列表中的位置

- 第三步：读取该位置上存储的值

    如果存在，则返回该值
    如果不存在，则报错KeyError

3、小结

（1）字典数据类型，通过空间换时间，实现了快速的数据查找

- 也就注定了字典的空间利用效率低下

（2）因为散列值对应位置的顺序与键在字典中显示的顺序可能不同，因此表现出来字典是无序的

- 回顾一下 N >> n
  如果N = n，会产生很多位置冲突

- 思考一下开头的小例子，为什么字典实现了比列表更快速的查找

9.1.3 紧凑的字符串

通过紧凑数组实现字符串的存储

- 数据在内存中是连续存放的，效率更高，节省空间

- 思考一下，同为序列类型，为什么列表采用引用数组，而字符串采用紧凑数组

9.1.4 是否可变

1、不可变类型：数字、字符串、元组

在生命周期中保持内容不变

- 换句话说，改变了就不是它自己了（id变了）

- 不可变对象的 += 操作 实际上创建了一个新的对象

In [18]:

```
x = 1
y = "Python"

print("x id:", id(x))
print("y id:", id(y))
```

```
x id: 140718440616768
y id: 2040939892664
```

In [19]:

```
x += 2
y += "3.7"

print("x id:", id(x))
print("y id:", id(y))
```

```
x id: 140718440616832
y id: 2040992707056
```

元组并不是总是不可变的

In [20]:

```
t = (1,[2])
t[1].append(3)

print(t)
```

```
(1, [2, 3])
```

2、可变类型：列表、字典、集合

- id 保持不变，但是里面的内容可以变

- 可变对象的 += 操作 实际在原对象的基础上就地修改

In [21]:

```
ls = [1, 2, 3]
d = {"Name": "Sarah", "Age": 18}

print("ls id:", id(ls))
print("d id:", id(d))
```

```
ls id: 2040991750856
d id: 2040992761608
```

In [22]:

```
ls += [4, 5]
d_2 = {"Sex": "female"}
d.update(d_2)            # 把d_2 中的元素更新到d中

print("ls id:", id(ls))
print("d id:", id(d))
```

```
ls id: 2040991750856
d id: 2040992761608
```

9.1.5 列表操作的几个小例子

【例1】 删除列表内的特定元素

- 方法1 存在运算删除法

缺点：每次存在运算，都要从头对列表进行遍历、查找、效率低

In [23]:

```
alist = ["d", "d", "d", "2", "2", "d" ,"d", "4"]
s = "d"
while True:
    if s in alist:
        alist.remove(s)
    else:
        break
print(alist)
```

['2', '2', '4']

- 方法2 一次性遍历元素执行删除

In [24]:

```
alist = ["d", "d", "d", "2", "2", "d" ,"d", "4"]
for s in alist:
    if s == "d":
        alist.remove(s)      # remove（s） 删除列表中第一次出现的该元素
print(alist)
```

['2', '2', 'd', 'd', '4']

解决方法：使用负向索引

In [25]:

```
alist = ["d", "d", "d", "2", "2", "d" ,"d", "4"]
for i in range(-len(alist), 0):
    if alist[i] == "d":
        alist.remove(alist[i])      # remove（s） 删除列表中第一次出现的该元素
print(alist)
```

['2', '2', '4']

【例2】 多维列表的创建

In [26]:

```
ls = [[0]*10]*5
ls
```

Out[26]:

```
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

In [27]:

```
ls[0][0] = 1
ls
```

Out[27]:

```
[[1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

9.2 更加简洁的语法

9.2.1 解析语法

In [28]:

```
ls = [[0]*10 for i in range(5)]
ls
```

Out[28]:

```
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

In [29]:

```
ls[0][0] = 1
ls
```

Out[29]:

```
[[1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

1、解析语法的基本结构——以列表解析为例（也称为列表推导）

[expression **for value in iterable** if conditihon]

- 三要素：表达式、可迭代对象、if条件（可选）

执行过程

（1）从可迭代对象中拿出一个元素

（2）通过if条件（如果有的话），对元素进行筛选

若通过筛选：则把元素传递给表达式

若未通过：　则进入（1）步骤，进入下一次迭代

(3) 将传递给表达式的元素，代入表达式进行处理，产生一个结果

(4) 将（3）步产生的结果作为列表的一个元素进行存储

(5) 重复（1）~（4）步，直至迭代对象迭代结束，返回新创建的列表

In [ ]:

```
# 等价于如下代码
result = []
for value in iterale:
    if condition:
        result.append(expression)
```

【例】求20以内奇数的平方

In [30]:

```
squares = []
for i in range(1,21):
    if i%2 == 1:
        squares.append(i**2)
print(squares)
```

```
[1, 9, 25, 49, 81, 121, 169, 225, 289, 361]
```

In [31]:

```
squares = [i**2 for i in range(1,21) if i%2 == 1]
print(squares)
```

```
[1, 9, 25, 49, 81, 121, 169, 225, 289, 361]
```

支持多变量

In [32]:

```
x = [1, 2, 3]
y = [1, 2, 3]

results = [i*j for i,j in zip(x, y)]
results
```

Out[32]:

```
[1, 4, 9]
```

支持循环嵌套

In [33]:

```
colors = ["black", "white"]
sizes = ["S", "M", "L"]
tshirts = ["{} {}".format(color, size) for color in colors for size in sizes]
tshirts
```

Out[33]:

['black S', 'black M', 'black L', 'white S', 'white M', 'white L']

2、其他解析语法的例子

- 解析语法构造字典（字典推导）

In [1]:

```
squares = {i: i**2 for i in range(10)}
for k, v in squares.items():
    print(k, ":  ", v)
```

```
0 :   0
1 :   1
2 :   4
3 :   9
4 :   16
5 :   25
6 :   36
7 :   49
8 :   64
9 :   81
```

- 解析语法构造集合（集合推导）

In [2]:

```
squares = {i**2 for i in range(10)}
squares
```

Out[2]:

{0, 1, 4, 9, 16, 25, 36, 49, 64, 81}

- 生成器表达式

In [34]:

```
squares = (i**2 for i in range(10))
squares
```

Out[34]:

<generator object <genexpr> at 0x000001DB37A58390>

In [35]:

```
colors = ["black", "white"]
sizes = ["S", "M", "L"]
tshirts = ("{} {}".format(color, size) for color in colors for size in sizes)
for tshirt in tshirts:
    print(tshirt)
```

```
black S
black M
black L
white S
white M
white L
```

9.2.2 条件表达式

expr1 if condition else expr2

【例】将变量n的绝对值赋值给变量x

In [37]:

```
n = -10
if n >= 0:
    x = n
else:
    x = -n
x
```

Out[37]:

10

In [38]:

```
n = -10
x = n if n>= 0 else -n
x
```

Out[38]:

10

条件表达式和解析语法简单实用、运行速度相对更快一些，相信大家会慢慢的爱上它们

9.3 三大神器

9.3.1 生成器

In [3]:

```
ls = [i**2 for i in range(1, 1000001)]
```

In [4]:

```
for i in ls:
    pass
```

缺点：占用大量内存

生成器

（1）采用惰性计算的方式

（2）无需一次性存储海量数据

（3）一边执行一边计算，只计算每次需要的值

（4）实际上一直在执行next()操作，直到无值可取

1、生成器表达式

- 海量数据，不需存储

In [ ]:

```
squares = (i**2 for i in range(1000000))
```

In [168]:

```
for i in squares:
    pass
```

- 求0~100的和

无需显示存储全部数据，节省内存

In [58]:

```
sum((i for i in range(101)))
```

Out[58]:

5050

2、生成器函数——yield

- 生产斐波那契数列

数列前两个元素为1,1 之后的元素为其前两个元素之和

In [59]:

```
def fib(max):
    ls = []
    n, a, b = 0, 1, 1
    while n < max:
        ls.append(a)
        a, b = b, a + b
        n = n + 1
    return ls


fib(10)
```

Out[59]:

[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

中间尝试

In [60]:

```
def fib(max):
    n, a, b = 0, 1, 1
    while n < max:
        print(a)
        a, b = b, a + b
        n = n + 1


fib(10)
```

```
1
1
2
3
5
8
13
21
34
55
```

构造生成器函数

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

In [61]:

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


fib(10)
```

Out[61]:

```
<generator object fib at 0x000001BE11B19048>
```

In [62]:

```
for i in fib(10):
    print(i)
```

```
1
1
2
3
5
8
13
21
34
55
```

9.3.2 迭代器

1、可迭代对象

可直接作用于for循环的对象统称为可迭代对象：Iterable

（1）列表、元组、字符串、字典、集合、文件

可以使用isinstance()判断一个对象是否是Iterable对象

In [63]:

```
from collections import Iterable

isinstance([1, 2, 3], Iterable)
```

Out[63]:

```
True
```

In [14]:

```
isinstance({"name": "Sarah"}, Iterable)
```

Out[14]:

True

In [15]:

```
isinstance('Python', Iterable)
```

Out[15]:

True

(2) 生成器

In [64]:

```
squares = (i**2 for i in range(5))
isinstance(squares, Iterable)
```

Out[64]:

True

生成器不但可以用于for循环，还可以被next()函数调用

In [65]:

```
print(next(squares))
print(next(squares))
print(next(squares))
print(next(squares))
print(next(squares))
```

0
1
4
9
16

直到没有数据可取，抛出StopIteration

In [66]:

```
print(next(squares))
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-66-f5163ac9e49b> in <module>
----> 1 print(next(squares))

StopIteration:
```

可以被next()函数调用并不断返回下一个值，直至没有数据可取的对象称为迭代器：Iterator

2、迭代器

可以使用isinstance()判断一个对象是否是Iterator对象

（1） 生成器都是迭代器

In [67]:

```
from collections import Iterator

squares = (i**2 for i in range(5))
isinstance(squares, Iterator)
```

Out[67]:

True

（2） 列表、元组、字符串、字典、集合不是迭代器

In [20]:

```
isinstance([1, 2, 3], Iterator)
```

Out[20]:

False

可以通过iter(Iterable)创建迭代器

In [21]:

```
isinstance(iter([1, 2, 3]), Iterator)
```

Out[21]:

True

for item in Iterable 等价于：

先通过iter()函数获取可迭代对象Iterable的迭代器

然后对获取到的迭代器不断调用next()方法来获取下一个值并将其赋值给item

当遇到StopIteration的异常后循环结束

（3）zip enumerate 等itertools里的函数是迭代器

In [68]:

```
x = [1, 2]
y = ["a", "b"]
zip(x, y)
```

Out[68]:

<zip at 0x1be11b13c48>

In [69]:

```
for i in zip(x, y):
    print(i)

isinstance(zip(x, y), Iterator)
```

(1, 'a')
(2, 'b')

Out[69]:

True

In [70]:

```
numbers = [1, 2, 3, 4, 5]
enumerate(numbers)
```

Out[70]:

<enumerate at 0x1be11b39990>

In [71]:

```
for i in enumerate(numbers):
    print(i)

isinstance(enumerate(numbers), Iterator)
```

(0, 1)
(1, 2)
(2, 3)
(3, 4)
(4, 5)

Out[71]:

True

(4) 文件是迭代器

In [72]:

```
with open("测试文件.txt", "r", encoding = "utf-8") as f:
    print(isinstance(f, Iterator))
```

True

(5) 迭代器是可耗尽的

In [73]:

```
squares = (i**2 for i in range(5))
for square in squares:
    print(square)
```

```
0
1
4
9
16
```

In [74]:

```
for square in squares:
    print(square)
```

(6) range()不是迭代器

In [75]:

```
numbers = range(10)
isinstance(numbers, Iterator)
```

Out[75]:

```
False
```

In [76]:

```
print(len(numbers))    # 有长度
print(numbers[0])      # 可索引
print(9 in numbers)    # 可存在计算
next(numbers)          # 不可被next()调用
```

```
10
0
True
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-76-7c59bf859258> in <module>
      2 print(numbers[0])      # 可索引
      3 print(9 in numbers)    # 可存在计算
----> 4 next(numbers)          # 不可被next()调用

TypeError: 'range' object is not an iterator
```

In [77]:

```
for number in numbers:
    print(number)
```

```
0
1
2
3
4
5
6
7
8
9
```

不会被耗尽

In [78]:

```
for number in numbers:
    print(number)
```

```
0
1
2
3
4
5
6
7
8
9
```

可以称range()为懒序列

它是一种序列

但并不包含任何内存中的内容

而是通过计算来回答问题

9.3.3 装饰器

1、需求的提出

（1）需要对已开发上线的程序添加某些功能

（2）不能对程序中函数的源代码进行修改

（3）不能改变程序中函数的调用方式

比如说，要统计每个函数的运行时间

In [ ]:

```
def f1():
    pass


def f2():
    pass


def f3():
    pass

f1()
f2()
f3()
```

没问题，我们有装饰器！！！

2、函数对象

函数是Python中的第一类对象

（1）可以把函数赋值给变量

（2）对该变量进行调用，可实现原函数的功能

In [79]:

```
def square(x):
    return x**2

print(type(square))       # square 是function类的一个实例
```

```
<class 'function'>
```

In [80]:

```
pow_2 = square            # 可以理解成给这个函数起了个别名pow_2
print(pow_2(5))
print(square(5))
```

```
25
25
```

可以将函数作为参数进行传递

3、高阶函数

（1）接收函数作为参数

（2）或者返回一个函数

满足上述条件之一的函数称之为高阶函数

In [2]:

```
def square(x):
    return x**2


def pow_2(fun):
    return fun


f = pow_2(square)
f(8)
```

Out[2]:

64

In [3]:

```
print(f == square)
```

True

4、 嵌套函数

在函数内部定义一个函数

In [83]:

```
def outer():
    print("outer is running")

    def inner():
        print("inner is running")

    inner()


outer()
```

outer is running
inner is running

5、闭包

In [84]:

```
def outer():
    x = 1
    z = 10

    def inner():
        y = x+100
        return y, z

    return inner


f = outer()                 # 实际上f包含了inner函数本身+outer函数的环境
print(f)
```

```
<function outer.<locals>.inner at 0x000001BE11B1D730>
```

In [85]:

```
print(f.__closure__)          # __closure__属性中包含了来自外部函数的信息
for i in f.__closure__:
    print(i.cell_contents)
```

```
(<cell at 0x000001BE0FDE06D8: int object at 0x00007FF910D59340>, <cell at 0x000001BE
0FDE0A98: int object at 0x00007FF910D59460>)
1
10
```

In [86]:

```
res = f()
print(res)
```

```
(101, 10)
```

闭包：延伸了作用域的函数

如果一个函数定义在另一个函数的作用域内，并且引用了外层函数的变量，则该函数称为闭包

闭包是由函数及其相关的引用环境组合而成的实体(即：闭包=函数+引用环境)

- 一旦在内层函数重新定义了相同名字的变量，则变量成为局部变量

In [87]:

```
def outer():
    x = 1

    def inner():
        x = x+100
        return x

    return inner


f = outer()
f()
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-87-d2da1048af8b> in <module>
     10
     11 f = outer()
---> 12 f()

<ipython-input-87-d2da1048af8b> in inner()
      3
      4     def inner():
----> 5         x = x+100
      6         return x
      7

UnboundLocalError: local variable 'x' referenced before assignment
```

nonlocal允许内嵌的函数来修改闭包变量

In [1]:

```
def outer():
    x = 1

    def inner():
        nonlocal x
        x = x+100
        return x
    return inner


f = outer()
f()
```

1

Out[1]:

101

6、一个简单的装饰器

嵌套函数实现

In [89]:

```python
import time

def timer(func):

    def inner():
        print("inner run")
        start = time.time()
        func()
        end = time.time()
        print("{} 函数运行用时{:.2f}秒".format(func.__name__, (end-start)))

    return inner


def f1():
    print("f1 run")
    time.sleep(1)



f1 = timer(f1)              # 包含inner()和timer的环境，如传递过来的参数func
f1()
```

```
inner run
f1 run
f1 函数运行用时1.00秒
```

语法糖

In [90]:

```python
import time

def timer(func):

    def inner():
        print("inner run")
        start = time.time()
        func()
        end = time.time()
        print("{} 函数运行用时{:.2f}秒".format(func.__name__, (end-start)))

    return inner

@timer                          # 相当于实现了f1 = timer(f1)
def f1():
    print("f1 run")
    time.sleep(1)


f1()
```

```
inner run
f1 run
f1 函数运行用时1.00秒
```

7、装饰有参函数

In [91]:

```
import time


def timer(func):

    def inner(*args, **kwargs):
        print("inner run")
        start = time.time()
        func(*args, **kwargs)
        end = time.time()
        print("{} 函数运行用时{:.2f}秒".format(func.__name__, (end-start)))

    return inner


@timer                  # 相当于实现了f1 = timer(f1)
def f1(n):
    print("f1 run")
    time.sleep(n)


f1(2)
```

```
inner run
f1 run
f1 函数运行用时2.00秒
```

被装饰函数有返回值的情况

In [92]:

```python
import time


def timer(func):

    def inner(*args, **kwargs):
        print("inner run")
        start = time.time()
        res = func(*args, **kwargs)
        end = time.time()
        print("{} 函数运行用时{:.2f}秒".format(func.__name__, (end-start)))
        return res

    return inner


@timer                    # 相当于实现了f1 = timer(f1)
def f1(n):
    print("f1 run")
    time.sleep(n)
    return "wake up"

res = f1(2)
print(res)
```

```
inner run
f1 run
f1 函数运行用时2.00秒
wake up
```

8、带参数的装饰器

装饰器本身要传递一些额外参数

- 需求：有时需要统计绝对时间，有时需要统计绝对时间的2倍

In [95]:

```python
def timer(method):

    def outer(func):

        def inner(*args, **kwargs):
            print("inner run")
            if method == "origin":
                print("origin_inner run")
                start = time.time()
                res = func(*args, **kwargs)
                end = time.time()
                print("{} 函数运行用时{:.2f}秒".format(func.__name__, (end-start)))
            elif method == "double":
                print("double_inner run")
                start = time.time()
                res = func(*args, **kwargs)
                end = time.time()
                print("{} 函数运行双倍用时{:.2f}秒".format(func.__name__, 2*(end-start)))
            return res

        return inner

    return outer


@timer(method="origin")  # 相当于timer = timer(method = "origin")   f1 = timer(f1)
def f1():
    print("f1 run")
    time.sleep(1)


@timer(method="double")
def f2():
    print("f2 run")
    time.sleep(1)


f1()
print()
f2()
```

```
inner run
origin_inner run
f1 run
f1 函数运行用时1.00秒

inner run
double_inner run
f2 run
f2 函数运行双倍用时2.00秒
```

理解闭包是关键！！！

9、何时执行装饰器

- 一装饰就执行，不必等调用

In [96]:

```
func_names=[]
def find_function(func):
    print("run")
    func_names.append(func)
    return func


@find_function
def f1():
    print("f1 run")


@find_function
def f2():
    print("f2 run")


@find_function
def f3():
    print("f3 run")


```

```
run
run
run
```

In [99]:

```
for func in func_names:
    print(func.__name__)
    func()
    print()
```

```
f1
f1 run

f2
f2 run

f3
f3 run
```

10、回归本源

- 原函数的属性被掩盖了

In [100]:

```python
import time

def timer(func):
    def inner():
        print("inner run")
        start = time.time()
        func()
        end = time.time()
        print("{} 函数运行用时{:.2f}秒".format(func.__name__, (end-start)))

    return inner

@timer                  # 相当于实现了f1 = timer(f1)
def f1():
    time.sleep(1)
    print("f1 run")

print(f1.__name__)
```

```
inner
```

- 回来

In [101]:

```python
import time
from functools import wraps

def timer(func):
    @wraps(func)
    def inner():
        print("inner run")
        start = time.time()
        func()
        end = time.time()
        print("{} 函数运行用时{:.2f}秒".format(func.__name__, (end-start)))

    return inner

@timer                  # 相当于实现了f1 = timer(f1)
def f1():
    time.sleep(1)
    print("f1 run")

print(f1.__name__)
f1()
```

```
f1
inner run
f1 run
f1 函数运行用时1.00秒
```