**[python -- lambda表达式](https://www.cnblogs.com/hf8051/p/8085424.html)**

通常是在**需要一个函数，但是又不想费神去命名一个函数**的场合下使用，也就是指**匿名函数**。

lambda所表示的匿名函数的内容应该是很简单的，如果复杂的话，干脆就重新定义一个函数了，使用lambda就有点过于执拗了。

lambda就是用来定义一个匿名函数的，如果还要给他绑定一个名字的话，就会显得有点画蛇添足，通常是直接使用lambda函数。如下所示：

add = lambda x, y : x+y

add(1,2) # 结果为3

**1、应用在函数式编程中**

Python提供了很多函数式编程的特性，如：map、reduce、filter、sorted等这些函数都支持函数作为参数，lambda函数就可以应用在函数式编程中。如下：

# 需求：将列表中的元素按照绝对值大小进行升序排列

list1 = [3,5,-4,-1,0,-2,-6]

sorted(list1, key=lambda x: abs(x))

当然，也可以如下：

list1 = [3,5,-4,-1,0,-2,-6]

def get\_abs(x):

return abs(x)

sorted(list1,key=get\_abs)

只不过这种方式的代码看起来不够Pythonic

**2、应用在闭包中**

def get\_y(a,b):

return lambda x:a+b

y1 = get\_y(1,1)

y1(1) # 结果为2

当然，也可以用常规函数实现闭包，如下：

def get\_y(a,b):

def func(x):

return ax+b

return func

y1 = get\_y(1,1)

y1(1) # 结果为2

Python之禅中有这么一句话：Explicit is better than implicit（明了胜于晦涩），就是说那种方式更清晰就用哪一种方式。

[**C/C++-- lambda表达式**](https://www.cnblogs.com/hf8051/p/8085424.html)

##大前提：捕获列表里变量的确定时机。

**捕获列表和参数列表有区别，捕获列表里的变量，是在捕获的时间点就确定了，而不是在lambda调用时确定，参数列表是在调用时才确定。所以当捕获了一个int i，i=12，然后在lambda后面的代码又改变i为22，但是当调用lambda的时候，i值还是12。**

## 剖析点：

**1，值捕获，即使在lambda后面改变了该值，在调用lambda时，这个值还是捕获时的值。**

**2，引用捕获，在lambda后面改变了该值，在调用lambda时，这个值不是捕获时的值，而是改变后的值。**

**3，隐式捕获：**

* [=]代表全部采用值捕获
* [&]代表全部采用引用捕获
* [=, &val]代表val为引用捕获，其余为值捕获
* [&,val]代表val为值捕获，其余为引用捕获

**4，可变lambda，当想在lambda函数体里，修改一个值捕获的变量是，需要*mutable*关键字。**

**5，lambda的返回类型，函数体是单一的return语句的话，可以在声明lambda时，省略返回值的类型。**

### 由剖析点2：引用捕获，会引发很多血案。比如，被捕获的引用或者指针指向的对象已经不存在了，然后调用lambda时，就会出现致命错误。

**警告：当以引用或者指针方式捕获一个变量时，必须保证在lambda执行时变量是存在的。**

**建议：**

**1，捕获一个普通变量时，如int, string或其他非指针类型，通常可以采用简单的值捕获方式。所以，只需关注变量在捕获时，值是否是所需的值就行。**

**2，如果捕获一个指针或迭代器，或引用，就必须保证在lambda被执行的时候，绑定到迭代器，指针或引用的对象仍然存在，而且，需要保证对象是预期的值。因为，有可能在捕获的时候，是预期的值，但是在执行lambda之前有代码改变了绑定对象的值，在执行lambda时，就变成不是预期的值了。**

**3，一般来说，尽量减少捕获的数据量，来避免潜在的捕获导致的问题。而且，如果可能的话，尽量避免捕获指针或引用。**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int main(){

//test1 值捕获

/\*

int i = 1;

auto f = [i]{return i;};

i = 10;

int j = f();

cout << j << endl;

\*/

//test2 引用捕获

/\*

int i = 1;

auto f = [&i]{return i;};

i = 10;

int j = f();

cout << j << endl;//3

\*/

//test3 隐式值捕获

/\*

int i = 1;

int j = 2;

auto f = [=]{return i + j;};

i = 3;

int m = f();

cout << m << endl;

\*/

//test4 隐式引用捕获

/\*

int i = 1;

int j = 2;

auto f = [&]{return i + j;};

i = 3;

int m = f();

cout << m << endl;//5

\*/

//test5 隐式，显式混合1

/\*

int i = 1;

int j = 2;

//i为值捕获，j为引用捕获

auto f = [=,&j]{return i + j;};

i = 3;

int m = f();

cout << m << endl;//3

\*/

//test5 隐式，显式混合2

/\*

int i = 1;

int j = 2;

//i为引用捕获，j为值捕获

auto f = [&,j]{return i + j;};

i = 3;

int m = f();

cout << m << endl;//5

\*/

//test6 可变lambda

/\*

int i = 10;

auto f = [i] () mutable{return ++i;};

int j = f();

cout << j << endl;

\*/

/\*

const int i = 10;

//编译错误，因为i为const

auto f = [i] () mutable{return ++i;};

int j = f();

cout << j << endl;

\*/

//test7 lambda的返回类型

vector<int> ivec{-12,2,-22,3,0};

//改变ivec里的值，负数变成整数

//此lambda不写返回类型没有问题.

//transform(ivec.begin(),ivec.end(),ivec.begin(),

// [](int i){return i < 0 ? -i : i;});

//此lambda不写返回类型也没有问题.

transform(ivec.begin(),ivec.end(),ivec.begin(),

[](int i){if(i < 0) return -i;

else return i;});

for(const auto &s : ivec){

cout << s << " ";

}

cout << endl;

}