|  |
| --- |
| ***Hsingmin.lee@yahoo.com*** |
| **Quick Start for ML** |
| ***Machine Learning Appetizers*** |

|  |
| --- |
| **黎兴民**  ***2018/3/24 Saturday*** |

目录

[1. C/C++ 4](#_Toc509819050)

[1.1 const关键字 4](#_Toc509819051)

[1.2函数传值与传引用 9](#_Toc509819052)

[1.3 字符串转整型问题 10](#_Toc509819053)

[1.4 操作符重载 10](#_Toc509819054)

[2. Python 15](#_Toc509819055)

[2.1 可变类型与非可变类型 15](#_Toc509819056)

[2.2 静态方法，实例方法与类方法 15](#_Toc509819057)

[2.3 类变量与实例变量 16](#_Toc509819058)

[2.4 Python自省特性 18](#_Toc509819059)

[2.5 列表推导式与字典推导式 18](#_Toc509819060)

[2.6 深拷贝与浅拷贝 19](#_Toc509819061)

[2.7 类方法\_\_new\_\_( )与\_\_init\_\_( ) 21](#_Toc509819062)

[2.8 字符串格式化:%与.format 22](#_Toc509819063)

[2.9 Python的\*args和\*\*kwargs 24](#_Toc509819064)

[2.10 常见的几种设计模式 24](#_Toc509819065)

[2.11 Python变量的作用域 25](#_Toc509819066)

[2.12 GIL线程全局锁及协程 26](#_Toc509819067)

[2.13 闭包 27](#_Toc509819068)

[2.14 Lambda与函数式表达式 28](#_Toc509819069)

[2.15 编码与解码(incode/decode) 28](#_Toc509819070)

[2.16 迭代器与生成器 29](#_Toc509819071)

[2.17 装饰器 30](#_Toc509819072)

[2.18 Python中的重载 31](#_Toc509819073)

[2.19 Python新式类与旧式类 32](#_Toc509819074)

[2.20 邮箱地址正则表达式 33](#_Toc509819075)

[2.21 Python内置类型list/dictionary/tuple/string 37](#_Toc509819076)

[2.22 Python中的is 39](#_Toc509819077)

[2.23 read/readline和readlines 39](#_Toc509819078)

[2.24 垃圾回收 39](#_Toc509819079)

[2.25 类继承 39](#_Toc509819080)

[2.26 方法对象 40](#_Toc509819081)

[2.27 一行代码交换参数值 41](#_Toc509819082)

[2.28 未定义方法调用 41](#_Toc509819083)

[2.29 包管理 41](#_Toc509819084)

[3. Algorithm 42](#_Toc509819085)

[3.1 时间复杂度计算 42](#_Toc509819086)

[3.2 二叉树 46](#_Toc509819087)

[3.2.1 二叉树的数据结构 46](#_Toc509819088)

[3.2.2 二叉树的生成 46](#_Toc509819089)

[3.2.3 二叉树的遍历算法 47](#_Toc509819090)

[3.3 最大堆 53](#_Toc509819091)

[3.4 红黑树 53](#_Toc509819092)

[3.5 B树 53](#_Toc509819093)

[3.6 线性结构-栈与队列 53](#_Toc509819094)

[3.7 线性结构-链表 57](#_Toc509819095)

[3.8 寻找链表倒数第k个节点 **错误!未定义书签。**](#_Toc509819096)

[3.9 快速排序 64](#_Toc509819097)

[3.10 堆排序 **错误!未定义书签。**](#_Toc509819098)

[3.11 无序数字列表寻找所有间隔为d的组合 67](#_Toc509819099)

[3.12 列表[a1, a2, a3, …, an]求其所有组合 69](#_Toc509819100)

[3.13 一行python代码实现1+2+3+…+10\*\*8 69](#_Toc509819101)

[3.14 长度未知的单向链表求其是否有环 **错误!未定义书签。**](#_Toc509819102)

[3.15 单向链表应用快速排序 66](#_Toc509819103)

[3.16 长度为n的无序数字元素列表求其中位数 66](#_Toc509819104)

[3.17 遍历一个内部未知的文件夹 69](#_Toc509819105)

[3.18 台阶问题/斐波那契 70](#_Toc509819106)

[3.19 变态台阶问题 70](#_Toc509819107)

[3.20 矩形覆盖问题 67](#_Toc509819108)

[4. OS 73](#_Toc509819109)

[4.1 多线程与多进程的区别 73](#_Toc509819110)

[4.2 协程 75](#_Toc509819111)

[4.3 进程间通信方式 79](#_Toc509819112)

[4.4 虚拟存储系统中缺页计算 79](#_Toc509819113)

[4.5 并发进程不会引起死锁的资源数量计算 80](#_Toc509819114)

[4.6 常用的Linux/git/vim命令和作用 80](#_Toc509819115)

[4.6.1 Linux常用命令 80](#_Toc509819116)

[4.6.2 git常用命令 80](#_Toc509819117)

[4.6.3 vim常用命令 80](#_Toc509819118)

[4.7 查看当前进程的命令 81](#_Toc509819119)

[4.8 任务调度算法 81](#_Toc509819120)

[5. 网络 82](#_Toc509819121)

[5.1 TCP/IP协议 82](#_Toc509819122)

[5.2 OSI五层协议 82](#_Toc509819123)

[5.3 Socket长连接 82](#_Toc509819124)

[5.4 Select与epoll 82](#_Toc509819125)

[5.5 TCP与UDP协议的区别 82](#_Toc509819126)

[5.6 TIME\_WAIT过多的原因 83](#_Toc509819127)

[5.7 http一次连接的全过程描述 83](#_Toc509819128)

[5.8 http连接方式——get与post的区别 83](#_Toc509819129)

[5.9 restful 83](#_Toc509819130)

[5.10 http请求的状态码 200/403/404/504 83](#_Toc509819131)

[6. 数据库 84](#_Toc509819132)

[6.1 MySQL锁的种类 84](#_Toc509819133)

[6.2 死锁的产生 84](#_Toc509819134)

[6.3 MySQL的char/varchar/text的区别 84](#_Toc509819135)

[6.4 Join的种类与区别 84](#_Toc509819136)

[6.5 A LEFT JOIN B的查询结果中，B缺少的部分如何显示 84](#_Toc509819137)

[6.6 索引类型的种类 84](#_Toc509819138)

[6.7 BTree索引与hash索引的区别 84](#_Toc509819139)

[6.8 如何对查询命令进行优化 84](#_Toc509819140)

[6.9 NoSQL与关系型数据库的区别 84](#_Toc509819141)

[6.10 Redis常用的存储类型 85](#_Toc509819142)

# C/C++

## const关键字

const关键字常用于数组边界和switch条件分支标号：

const type variable; //常变量等价于 type const variable

const type &reference; //常引用等价于 type const& variable

ClassName const Object; //常对象等价于const ClassName Object

ClassName::func(signature) const; //常成员函数

Type const ArrayName[size]; //常数组等价于 const type ArrayName[size]

const type\* pointer; //常指针定义方法1

type const\* pointer; //常指针定义方法2

C标准中，const变量定义为全局作用域，而C++中则要视其定义位置而定；

使用指针时涉及到两个关键要素：指针本身(地址)和指针所指向的对象(数据类型)；

将一个指针声明为const类型，是限制其指向的对象为指定类型；

将指针本身声明为type \*const pointer，则是将指针(地址)本身声明为常量；

可以将一个非const对象的地址赋给const对象的指针，反之不可；

void Func(const A \*arg); //传指针传递参数

void Func(const A &arg); //传引用传递参数

将函数传入参数声明为const，提高函数运行效率且禁止修改其内容；

在C++中使用const关键字修饰成员函数，则const对象只能访问const成员函数，而非const对象可以访问所有成员函数；const对象的成员不可修改，const成员函数不可修改对象的数据，无论对象是否具有const限定；

常量指针与指针常量：

const int \*p1 = new int(10); //指向常量的指针，不可通过p1修改其指向的内容

int \*const p2 = new int(10); //指向int类型的指针常量，不可指向其他内存

程序载入内存时，会分配常量区存储常量，可以通过直接修改变量数值或通过另外一个非常量指针进行修改：

int a = 10;

const int \*p = &a;

// error

// \*p = 100;

//直接修改变量

a = 100;

//

int \*pi = (int\*) p; //第三方指针

\*pi = 100;

函数参数中的指针常量表示不允许此指针指向其他内容：

void func(int \*const pt){

int \*p = new int(10);

pt = p; //error cannot point to a new pointer

}

函数参数中的常量指针表示此参数不允许被修改：

void func(const int \*pt){

\*pt = 100; //error cannot be modified

}

若将参数中的指针赋给一个新的指针，则会修改其指向的内容；

常量与引用的关系：引用就是变量的别名，常量引用即不允许此引用成为其他变量的别名；

int a = 10;

const int& ra = a; //常量引用，不可通过此引用改变其对应的内容

// int& const ra = a; //error

若不希望函数调用者修改参数本身的值，最可靠的方法是传递引用；

void func(const int& arg){

// arg = 100; //error cannot be modified

}

系统在加载程序时，会将内存分为4个区域：堆、栈、数据段和代码段，使用常量的方式保护数据是通过编译器语法规则限定来实现的，但是仍然可以通过以下方式完成修改：

const int a = 10;

int \*pa = (int\*) &a; //可以通过定义其他指针的方法修改常量数值

\*pa = 100;

常量函数在C++中可防止类的 数据成员被非法访问，将类的成员函数分为两类：常量成员函数和非常量成员函数；

class Test{

public:

void func() const;

private:

int intValue;

};

void Test::func() const{

intValue = 100; // raise error 常量函数尝试改变数据成员intValue的数值

//编译时引发异常

}

class Fred{

public:

void inspect() const;

void mutate();

};

void UserCode(Fred& changeable, const Fred& unChangeable){

changeable.mutate(); //非常量对象调用非常量函数

changeable.inspect(); //非常量对象调用常量函数

unChangeable.mutate(); //常量对象调用非常量函数，错误

unChangeable.inspect(); //常量对象只能调用常量函数

}

常量函数包含一个this的常量指针：

void inspect(const Fred\* this) const;

void mutate(Fred\* this);

对于常量函数，不能通过this指针修改对象对应的内存，但可以通过重新定义指针来修改内存中的内容；

void func(const int \*pt){

int \*ptr = (int\*) pt;

\*ptr = 100;

}

通过常量对象调用非常量函数将产生语法错误；对于常量函数，不能通过this指针修改对象对应的内存块；但是可以通过this指针重新定义一个指向同一内存单元的指针；

void Fred::inspect() const{

Fred \*pFred = (Fred\*) this;

pFred->intValue = 50;

}

对于常量对象，可以构造新指针，指向常量对象所在的内存单元；

C++允许在类的数据成员定义前加上mutable关键字以实现成员在常量函数中可修改：

class Fred{

public:

void inspect() const;

private:

mutable int intValue;

};

常量函数的重载问题：

class Fred{

public:

void func() const;

void func();

}

void Fred::func() const{}

viod Fred::func(){}

void UserCode(Fred& fred, const Fred& cFred){

fred.func(); //call func()

cFred.func(); //call func() const

}

int main(int argc, char\*\* argv){

Fred fred;

UserCode(fred, fred);

return 0;

}

当存在 同名同参数相同返回值的函数重载时，具体调用哪个函数取决于调用对象是常量对象还是非常量对象；

常量返回值：不希望函数调用者修改函数的返回值，将函数返回一个常量。

* Const常量与宏定义的区别：const常量有数据类型(编译器做类型检查)，宏定义没有数据类型(编译器仅作简单的字符替换)；
* Const修饰类的数据成员，const数据成员只在对象的生命周期内是常量，而对整个类而言是可变的，因此，const数据成员的初始化只能在类的构造函数的初始化列表中进行，不能在类中指定；

class A{

const int size = 100; //错误定义

int array[size]; //错误定义，未知的size

}

要在类中定义常量，应当使用enum类型：

class A{

enum{size1=100, size2=200};

int array1[size1];

int array2[size2];

}

枚举常量不占用对象的内存空间，在编译时全部求值，枚举常量隐含使用整型类型，最大数值有限，不能表示浮点型数据；

* const初始化：

T b;

const T a = b; //非指针const常量初始化

T\* p = new T();

const T\* pc = p; //指针常量初始化

const T\* pb = new T(); //同上

T r;

const T& rf = r; //rf只能访问const成员函数

const T\* c = new T();

T\* e = c; //声明指针的目的是要修改其指向的内容，但此处指向常量

T\* const c = new T();

T\* e = c; //声明指针指向的内容可变

* 参数const通常用于参数为指针或引用的情况下使用，若参数为值传递，则函数会自动产生临时变量复制，保护被传递对象的属性；
* 对于非内部数据类型的参数传递，将参数值传递改为const引用传递可以提高效率；而对于内部数据类型的参数传递，避免改为const引用传递，以免降低程序的可理解性；
* 修饰返回值的const关键字，const T func(); const T\* func();对函数返回值进行保护；
* 函数的返回值声明为const，通常用于操作符重载，若使用const修饰函数的返回值类型，则返回的const Object只能访问类中的公有数据成员和const成员函数，且无法对其进行赋值操作；指针传递函数返回值加const修饰符，则函数返回内容(指针)不能被修改，只能赋给同样加const修饰符的同类型指针；

const char\* GetString(void);

char \*str = GetString(); //compile error

const char\* str = GetString(); //compile success

* 函数返回值引用传递通常用于类的赋值函数中，实现链式表达：

class A{

A &operator = (const A& other); //赋值操作符

}

A a, b, c; //class A Object a, b, c

…

a = b = c; //all right

(a = b) = c; //legal but unusual

a.operator = (b)的返回值是const类型的引用，不可再次赋值给c；

返回值的内容不允许被修改；

* 类成员函数const关键字的使用：

class Stack{

public:

void Push(int elem);

int Pop(void);

int GetCount(void) const; //const成员函数

private:

int m\_num;

int m\_data[100];

};

//公有成员函数定义在类外

int Stack::GetCount(void) const{

++m\_num; //编译错误，企图修改数据成员

Pop(); //编译错误，企图调用非const成员函数

return m\_num;

}

* 在C中，const是一个不能被改变的普通变量，需要占用存储空间，编译器不知道编译时的数值，且数组下标必须为常量；而在C++中，将const看做编译时的常量，不为其分配存储空间，只是在编译时将其数值存储在名字列表中；
* 在C语言中，const int size; 语句正确，在C++中不正确；C编译器默认使用外部连接，将其看做声明，可在其他地方分配内存空间；C++编译器默认使用内部链接，必须在声明时初始化，将const对象默认看做文件的局部变量；
* 在C++中，是否为const 分配内存空间取决于是否添加extern关键字或者取const变量地址；

## 函数传值与传引用

函数传值：压栈的是参数的副本，函数对传递参数的操作作用在参数副本之上，不会修改原始值本身；

函数传指针(地址)：压栈的是地址的副本，但指针指向的是同一内存单元，修改操作会作用在原内存之上；

函数传引用：压栈的是引用的副本，但引用指向同一个变量地址，即同变量的两个别名，对引用的操作即对其指向的变量的操作；

值传递与引用传递的区别：

值传递(passed-by-value)：被调函数的形式参数作为被调函数的局部变量处理，为其开辟内存空间，存放传递进来的主调函数的实参数值，成为实参的一个副本，对被调函数局部变量的操作即对实参副本的操作，因此不会改变主调函数中实参的原始数值；

引用传递(passed-by-reference)：被调函数的形式参数也被当做被调函数的局部变量处理，并在堆栈中开辟内存空间，但其存储的是主调函数实参的地址，因此，被调函数实际上是对主调函数实参的间接引用，操作作用在实参原始数值之上，会改变实参的数值；

## 字符串转整型问题

将一个字符串转换成整数，如”0295”转换成0295 (Microsoft)，容易写出以下代码：

int StrToInt(char\* string){

int number = 0;

while(\*string != 0){

number = number \* 10 + \*string – ‘0’;

++string;

}

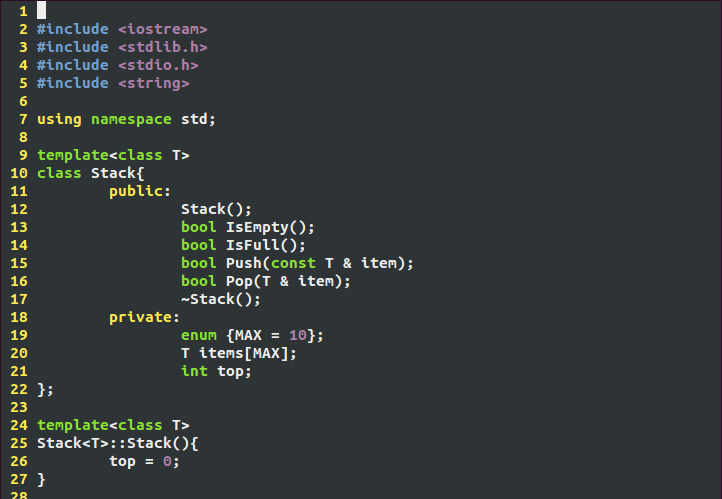
return number;

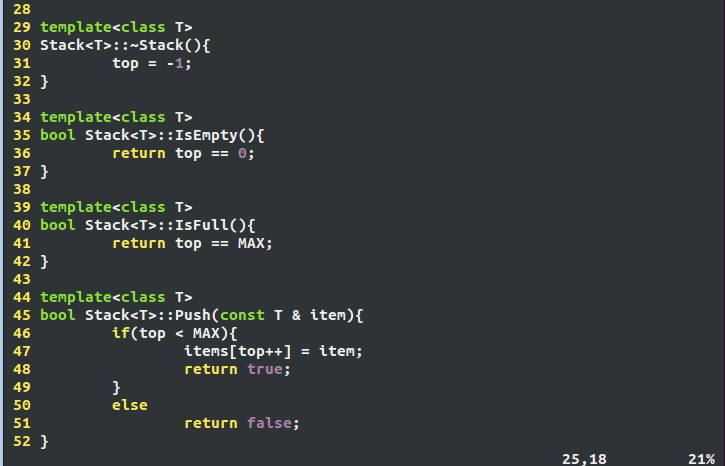
}

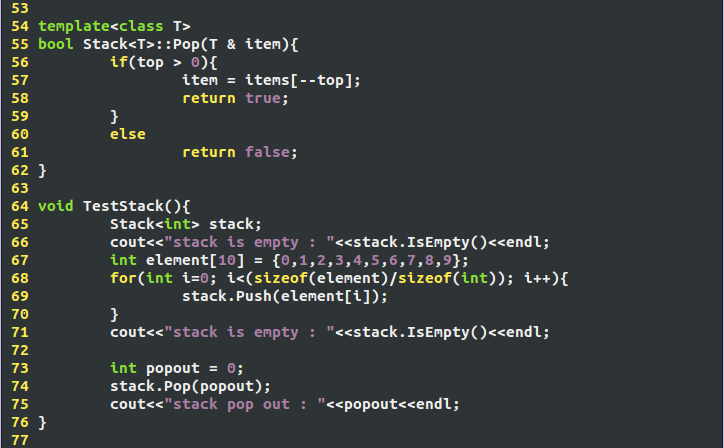
需要考虑以下问题：

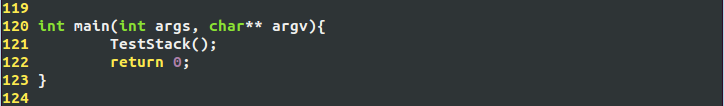
* 程序鲁棒性：函数传入空指针判断；
* 输入字符串正负号判断；
* 最大正整数，最小负整数及溢出问题；
* 非法字符输入；

## 模板









## 类与对象

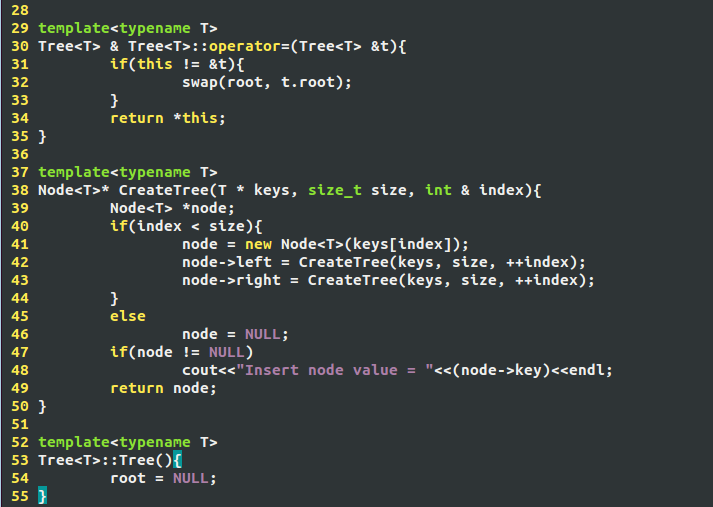
代码列表中使用了以下知识点：

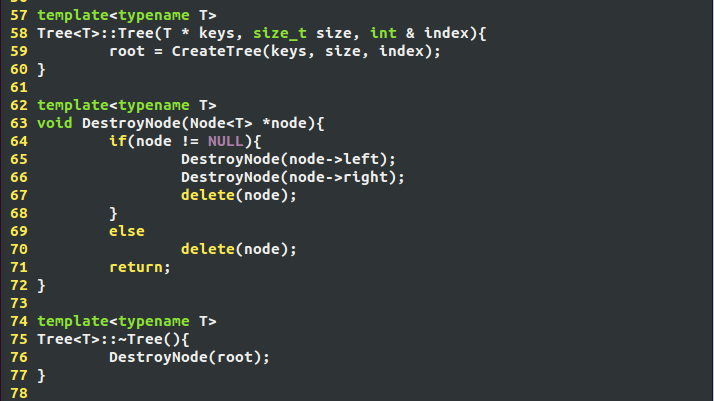
* 模板(Java中称为泛型)；
* 类的公有成员与私有成员(私有成员只能被类的private或public成员函数访问)；
* 函数初始化(参数列表)；
* 指针与引用；
* Size\_t关键字；
* 传值与传引用；
* 递归调用；
* new与delete成对使用，且使用方法要对应：new ~ delete，new [] ~ delete[]；
* C的空指针NULL，C++传统空指针0与C++ 11中的nullptr；
* 函数指针的用法 type (\*pf)(params)；
* 操作符重载；

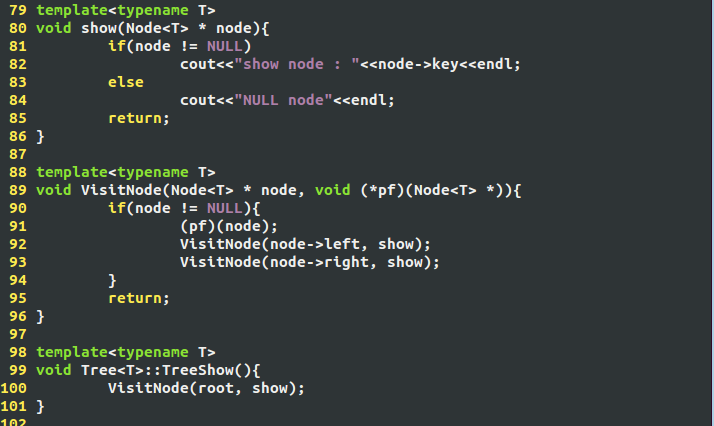
初始化参数列表：

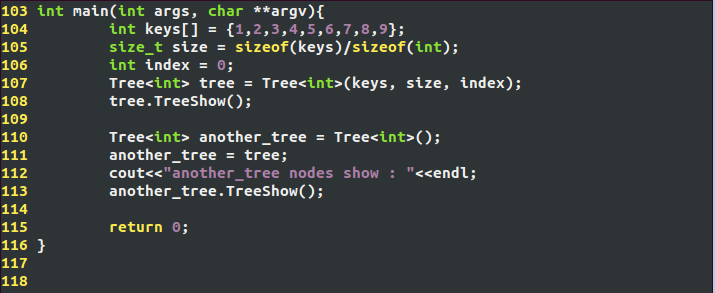


值传递传的是一份临时拷贝（t为一份临时拷贝），临时拷贝的特性在于其存活周期只限于这个函数，当函数调用完毕时，将会自动销毁，此处利用了这一特性，运行**std::swap(root, t.root)**这个语句的时候，临时拷贝t的私有变量root和 本树的私有变量this.root发生了交换，root原本的值被赋给了临时拷贝t.root，t.root的值被赋给了root，程序执行完毕，t自动销毁，比较好的利用了传递引用的特性；



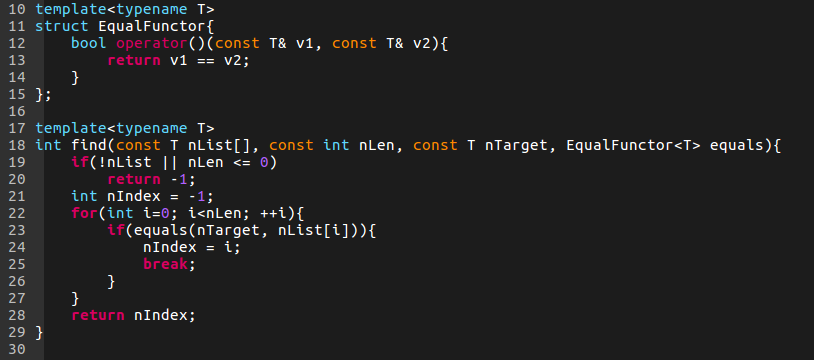




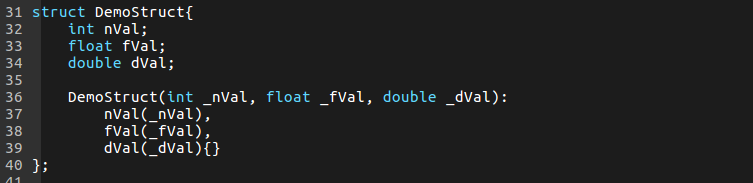


## 模板仿函数

在C++中，struct类似于class，可在其中实现对操作符的重载；此处EqualFunctor是一个使用了模板T的结构体，并重载()操作符，而在函数find()中，传入了一个EqualFunctor<T>类型的函数equals()，用于实现对nTarget和nList[i]的比较，而没有使用函数指针的形式type\_t (\*pf)(args…)；



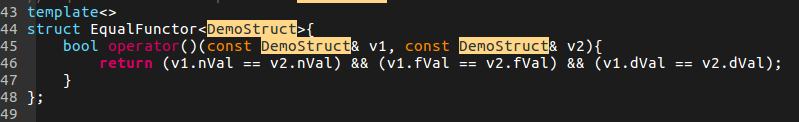
C++中的结构体，类似于C++ class，除了属性之外，还带有“构造函数”；



生成实例时，调用如下，可将实参列表传入；：

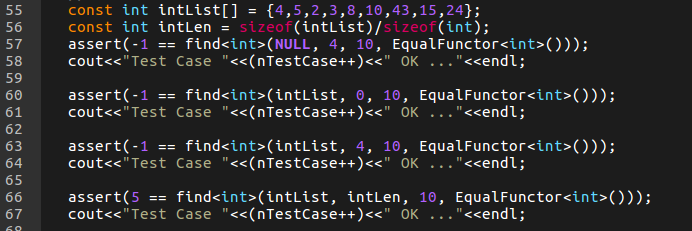


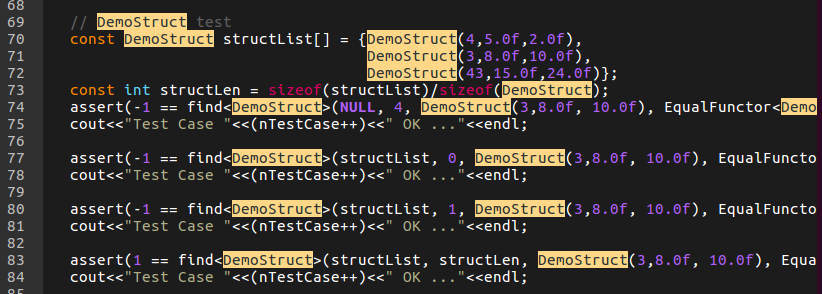
因为增加了属性，所以要重载()运算符，写法如下：



其中template<>中的typename T可省略，从struct EqualFunctor<DemoStruct>中可以推理得到；

测试代码如下：

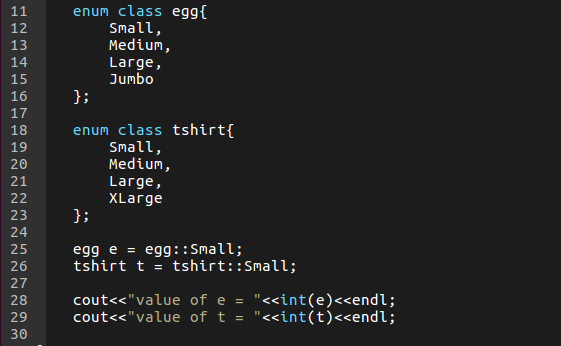




模板仿函数本质上是在类或结构体中重载operator()运算符，实现函数功能；

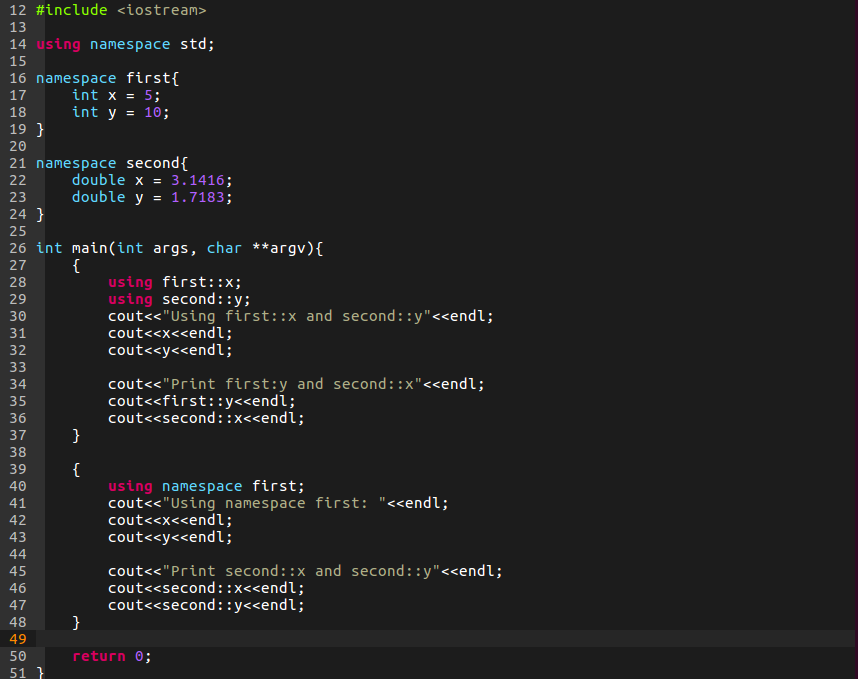
## enum枚举类型

enum类型默认数值为0,1,…，可在初始化时重新赋值；为避免多个Enum类型成员重名导致的冲突，可使用枚举类命名空间进行隔离，调用方法如下所示：



## 命名空间

命名空间用于解决变量重名问题，可使用using spacename::variable单独引入变量，也可以使用using namespace spacename使用整个命名空间；

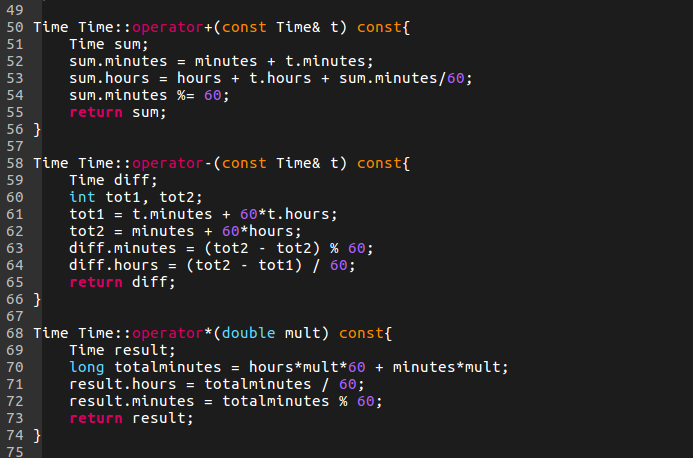


## 运算符重载

运算符重载的意义在于将标准C++运算符应用于类对象(struct)，在类中声明重载操作符：



Time operator+(const Time& t) const是常成员函数，不能修改类的数据成员；



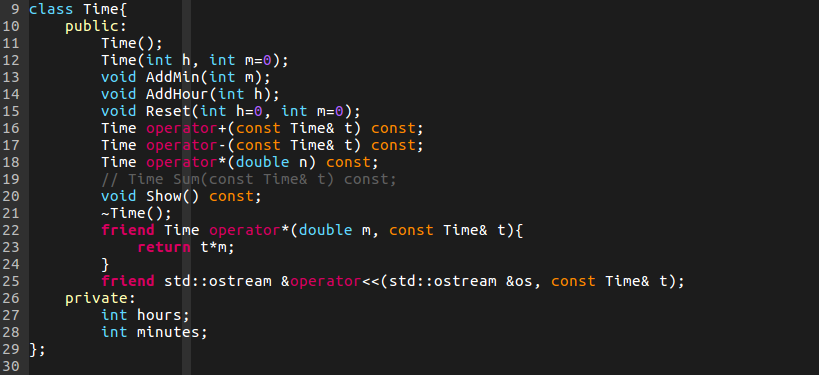
在外部实现类的成员函数，包含重载的操作符；

## 友元

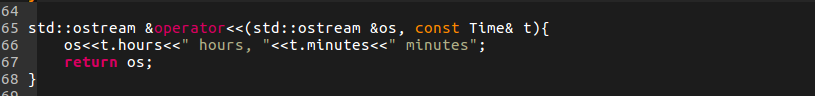
C++控制对类对象私有部分的访问，公有类的方法是唯一的访问途径，但是友元提供了另外一种方法：

* 友元类；
* 友元函数：友元函数可以允许非成员函数访问类的私有数据；
* 友元成员函数：

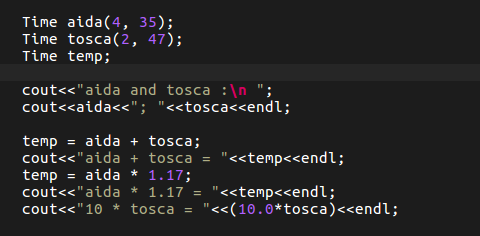
// 非成员重载运算符函数：



友元函数在类中声明，在类外定义，定义时不加成员修饰符Class::FriendFunc()：

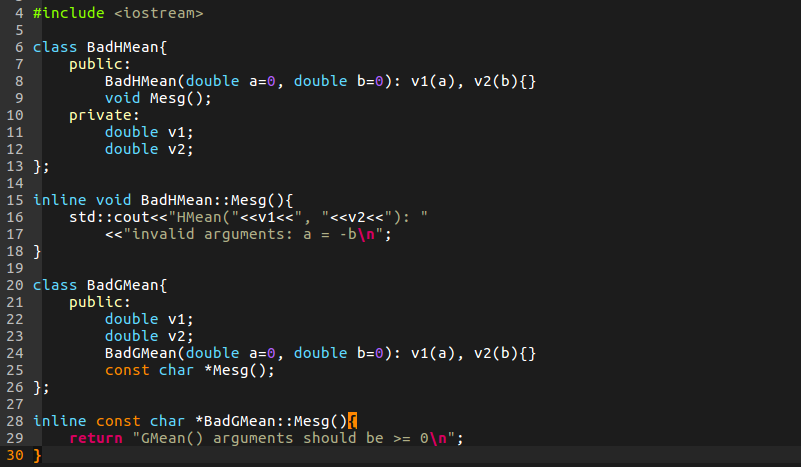


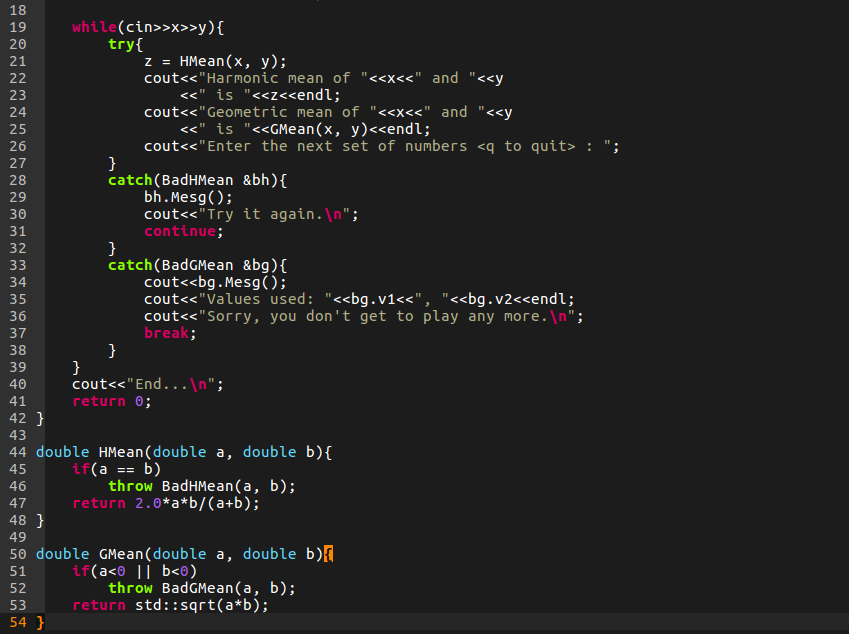
调用时，编译器根据传入参数选择使用哪一个重载函数：



## C++异常

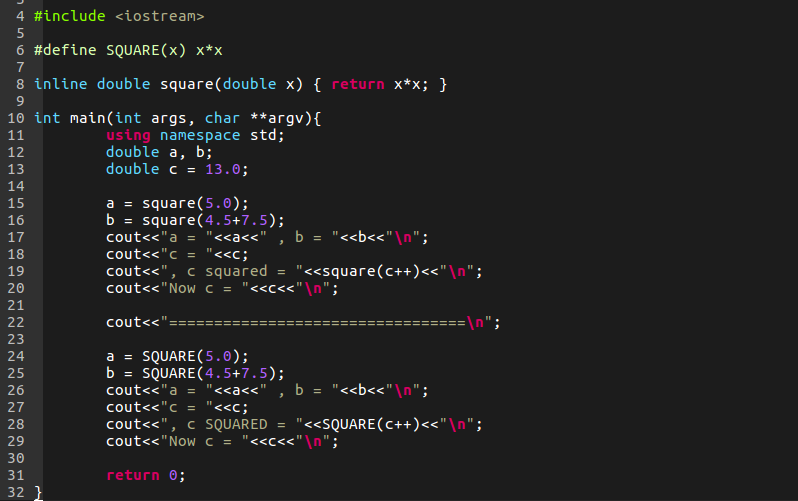
自定义异常类，try{…}代码块中调用HMean()和GMean()，当这两个函数在给定条件下抛出异常throw BadHMean(a, b)/throw BadGMean(a, b)时，被catch(Exception &e){…}捕捉到，并进行处理；



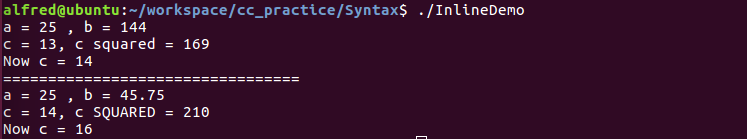


## 内联与宏

编译器使用函数代码代替函数调用，使得主调函数无需跳转，将整段代码进行替换编译，以提高编译效率；宏也是整段替换，有区别：



运行结果：



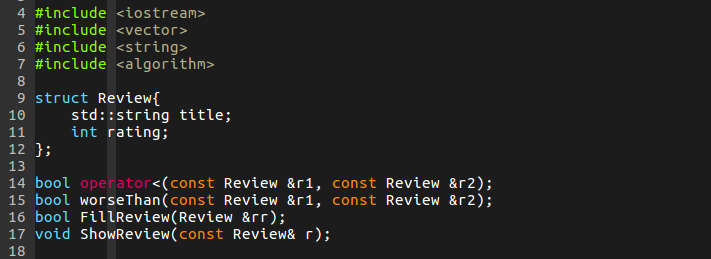
SQUARE(c++)展开可得到(c++)\*(c++)即14\*15；

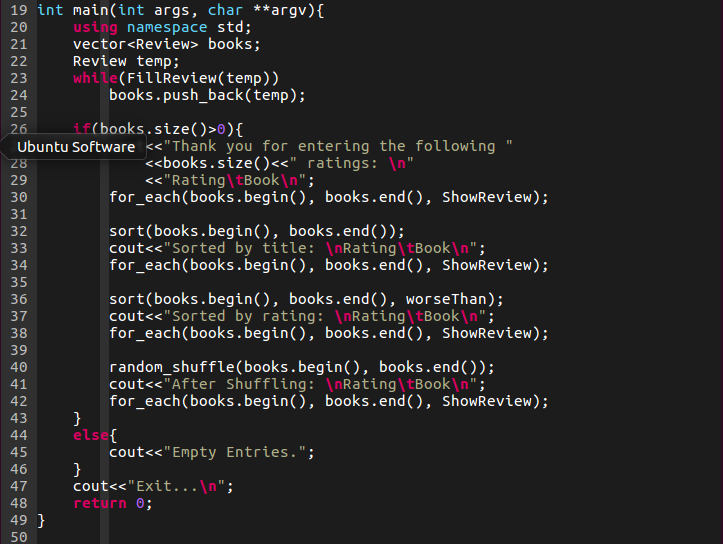
## STL标准模板库

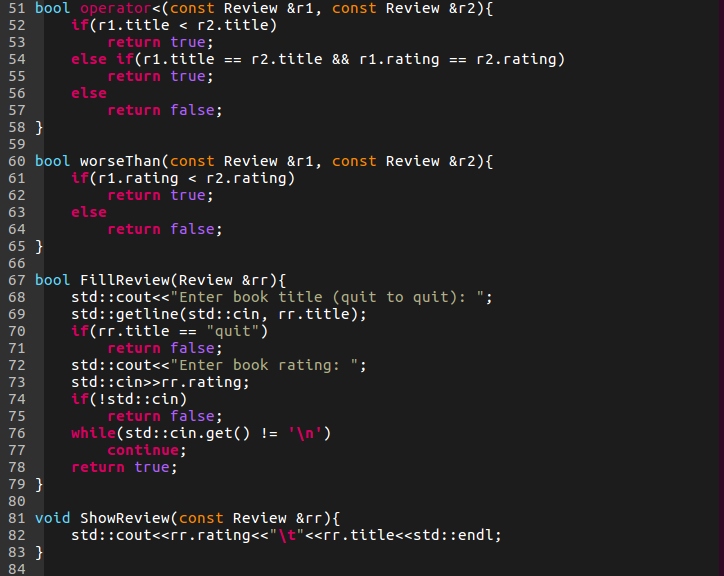
C++标准模板库提供了一组表示容器、迭代器、函数对象和算法的模板；

* 容器是一个类似于数组的单元，可以存储同质的一组数值；
* 算法是完成特定任务的处理方法；
* 迭代器用于遍历容器对象；
* 函数对象是类似于函数的对象，可以是类对象或函数指针；

STL可以用来构造各种容器(Array/Queue/LinkList…)并执行各种操作(Search/Sort/Rank…)；



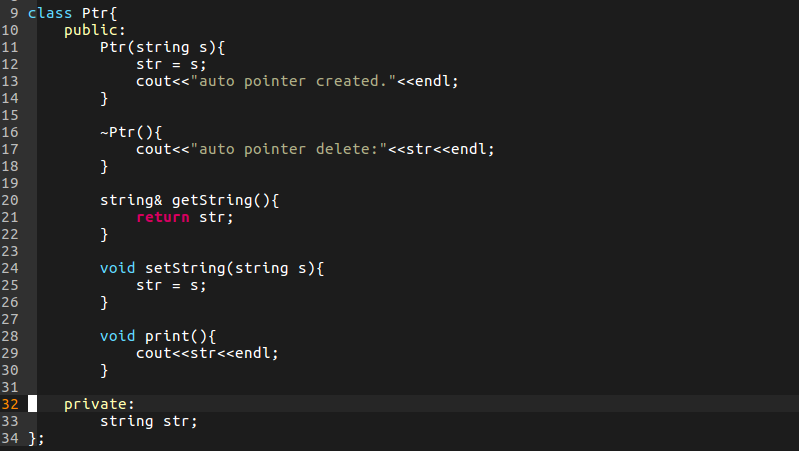


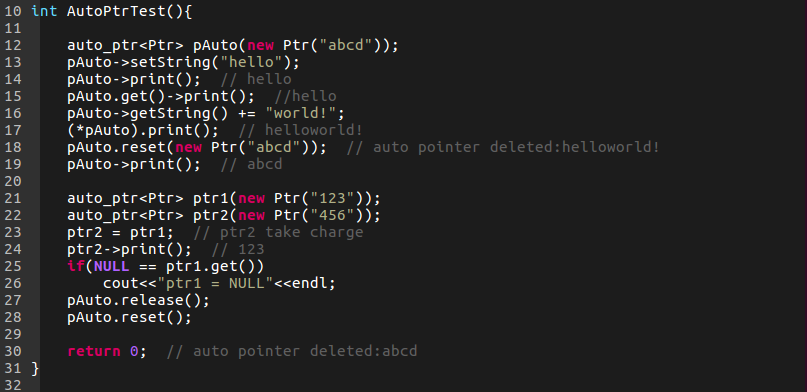


## 智能指针

auto\_ptr/shared\_ptr/weak\_ptr是C++11引入的智能指针，用来进行内存管理，使用new/delete机制时，若在delete语句之前发生程序跳转，则会出现内存泄露；而智能指针实际是一个类，当退出类的作用域时，内存就会被释放，在一定程度上可有效避免内存泄露的发生；

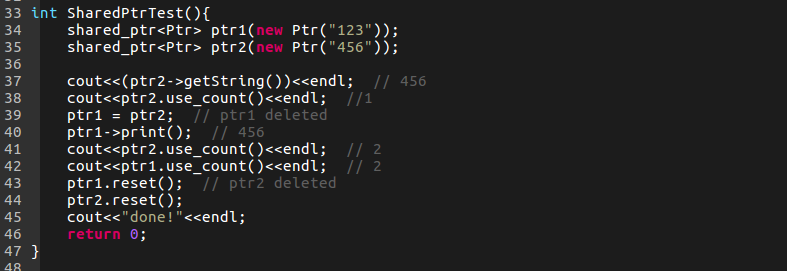
定义一个类Ptr，用于测试：

auto\_ptr测试：



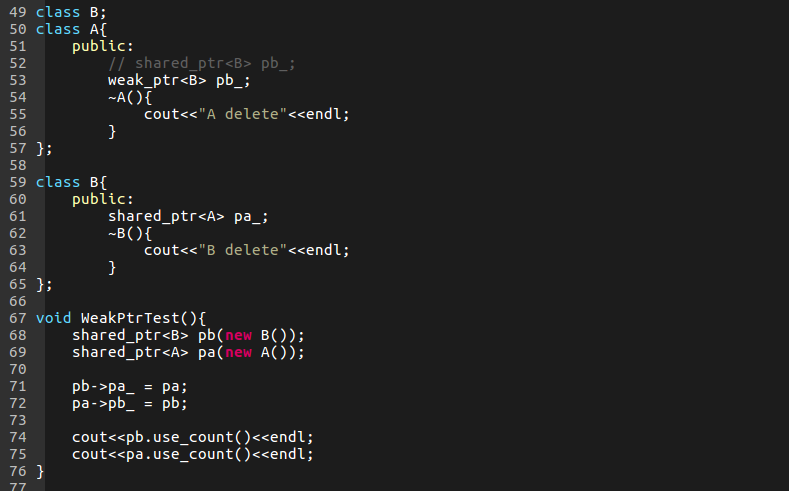
判断一个智能指针是否为空不能使用if(ptr == NULL)，应该使用if(ptr.get() == NULL)，ptr2 = ptr1实际上进行了对ptr1所指向内容的控制权转移；

Ptr.release()只是将指针赋空，并没有释放资源，释放资源使用ptr.reset()；



Ptr1 = ptr2，两个指针指向同一块内存区域，ptr.use\_count()可以给出引用数，两个shared\_ptr相互引用,那么这两个指针的引用计数永远不可能下降为0,资源永远不会释放。它是对对象的一种弱引用，不会增加对象的引用计数，和shared\_ptr之间可以相互转化，shared\_ptr可以直接赋值给它，它可以通过调用lock函数来获得shared\_ptr；

pa ，pb之间互相引用，两个资源的引用计数为2，当要跳出函数时，智能指针pa，pb析构时两个资源引用计数会减一，但是两者引用计数还是为1，导致跳出函数时资源没有被释放（A B的析构函数没有被调用），如果把其中一个改为weak\_ptr就可以了，我们把类A里面的shared\_ptr<B> pb\_; 改为weak\_ptr<B> pb\_; 运行结果如下，这样的话，资源B的引用开始就只有1，当pb析构时，B的计数变为0，B得到释放，B释放的同时也会使A的计数减一，同时pa析构时使A的计数减一，那么A的计数为0，A得到释放；



C++11中使用make\_shared替代new，并调用lock()函数避免控制权指针指向的资源被释放掉：



## 模板仿函数

## 模板仿函数

# Python

## 可变类型与非可变类型

类型属于对象而非变量即对象才有类型，而变量没有类型；对象分为可变对象(mutable)和非可变对象(immutable)两种，常见的可变对象有list/dict/set，常见的不可变对象有strings/tuples/numbers；

当不可变对象的引用传递给函数时，函数自动生成一份引用的拷贝，函数内部的操作均在拷贝上进行，不会改变外部对象；

当可变对象的引用传递给函数时，函数内的引用指向可变对象，如同指针一样，对其操作即对定位的指针地址一样，在内存中完成对对象的修改；

a = 1

def fun(a):

a = 2

fun(a)

print a # 1

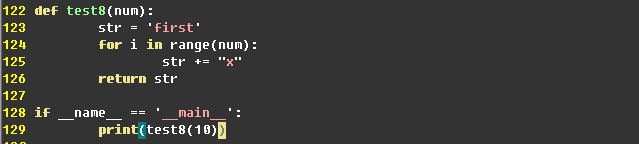
a = []

def fun(a):

a.append(1)

fun(a)

print a # [1]



String类型属于不可变对象，每次迭代都要重新生成一个新的对象存储字符串，迭代次数越多，内存消耗越大；

## 静态方法，实例方法与类方法

Python中有三种方法：静态方法@staticmethod 类方法@classmethod 和实例方法：

def foo(x):

print "executing foo(%s)"%(x)

class A(object):

def foo(self,x):

print "executing foo(%s,%s)"%(self,x)

@classmethod

def class\_foo(cls,x):

print "executing class\_foo(%s,%s)"%(cls,x)

@staticmethod

def static\_foo(x):

print "executing static\_foo(%s)"%x

a=A()

函数参数self：对实例(object)的绑定，在类中每次定义方法都要绑定实例self，foo(self, x)调用可修改实例自身；

函数参数cls：对类(class)的绑定，类方法A.class\_foo(x)传递的是类class而非实例；

静态方法与普通方法一样，不需要绑定类或实例，仅需要通过实例调用a.static\_foo(x)或通过类调用A.static\_foo(x)；

## 类变量与实例变量

类变量可在类间共享，不会单独分配给每个实例；

实例变量是指实例化后，实例单独拥有的变量；

class Test(object):

num\_of\_instance = 0

def \_\_init\_\_(self, name):

self.name = name

Test.num\_of\_instance += 1

if \_\_name\_\_ == '\_\_main\_\_':

print Test.num\_of\_instance # 0

t1 = Test('jack')

print Test.num\_of\_instance # 1

t2 = Test('lucy')

print t1.name , t1.num\_of\_instance # jack 2

print t2.name , t2.num\_of\_instance # lucy 2

class Person:

name="aaa"

p1=Person()

p2=Person()

p1.name="bbb"

print p1.name # bbb

print p2.name # aaa

print Person.name # aaa

参数传递问题，p1.name最初指向的是类变量name = “aaa”，string为不可变对象，实例单独产生拷贝，不会改变类变量本身，而list []为可变对象，每个实例均会产生对类变量的引用，实例方法对类变量的操作通过间接引用作用在类变量上，在实例的作用域内将类变量的引用改变为实例变量；

class Person:

name=[]

p1=Person()

p2=Person()

p1.name.append(1)

print p1.name # [1]

print p2.name # [1]

print Person.name # [1]

# -\*- coding: utf-8 -\*-

# -\*- versison: python 3.4.5 -\*-

# class\_method.py

class Test(object):

num\_of\_instance = 0

name\_of\_instance = "Object1"

queue\_of\_instance = []

def \_\_init\_\_(self, name):

self.name = name

Test.num\_of\_instance += 1

Test.name\_of\_instance = name

Test.queue\_of\_instance.append(name)

if \_\_name\_\_ == '\_\_main\_\_':

print("Class attribute : ")

print("Test.num\_of\_instance = ", Test.num\_of\_instance) # 0

print("Test.name\_of\_instance = ", Test.name\_of\_instance) # "Object"

print("Test.queue\_of\_instance = ", Test.queue\_of\_instance) # []

# Create object named 'jack'

t1 = Test('jack')

print("After instantiation : ")

print("Test.num\_of\_instance = ", Test.num\_of\_instance) # 1

print("Test.name\_of\_instance = ", Test.name\_of\_instance) # "jack"

print("Test.queue\_of\_instance = ", Test.queue\_of\_instance) # ['jack']

# Object t1 shares class variable

print("t1.num\_of\_instance = ", t1.num\_of\_instance) # 1

print("t1.name\_of\_instance = ", t1.name\_of\_instance) # "jack"

print("t1.queue\_of\_instance = ", t1.queue\_of\_instance) # ['jack']

# Create object named 'tom'

t2 = Test('tom')

print("Test.num\_of\_instance = ", Test.num\_of\_instance) # 2

print("Test.name\_of\_instance = ", Test.name\_of\_instance) # "tom"

print("Test.queue\_of\_instance = ", Test.queue\_of\_instance) # ['jack', 'tom']

# Object t2 shares class variable

print("t2.num\_of\_instance = ", t2.num\_of\_instance) # 2

print("t2.name\_of\_instance = ", t2.name\_of\_instance) # "tom"

print("t2.queue\_of\_instance = ", t2.queue\_of\_instance) # ['jack', 'tom']

# number, string as immutable variable

# list as mutable variable

t1.name\_of\_instance = “jack”

print("t1.num\_of\_instance = ", t1.num\_of\_instance) # 2

print("t1.name\_of\_instance = ", t1.name\_of\_instance) # "jack"

print("t1.queue\_of\_instance = ", t1.queue\_of\_instance) # ['jack', 'tom']

## Python自省特性

面向对象语言可在运行时获得对象的类型，常用自省函数：

type()/dir()/getattr()/hasattr()/isinstance()

## 列表推导式与字典推导式

列表推导式与字典推导式具有高效简短的特性：

list = [element for element in iterable]

dict = {key: value for (key, value) in iterable}

使用内建函数enumerate()赋予元素下标：

{i: el for i, el in enumerate([“one”, “two”, “three”])}

# 不使用内建函数enumerate()

lst = [“one”, “two”, “three”]

i = 0

for e in lst:

lst[i] = ‘%d: %s’ %(i, lst[i])

i += 1

将列表推导式中的[]改为()，其数据结构发生变化：

L = [x\*x for x in range(10)]

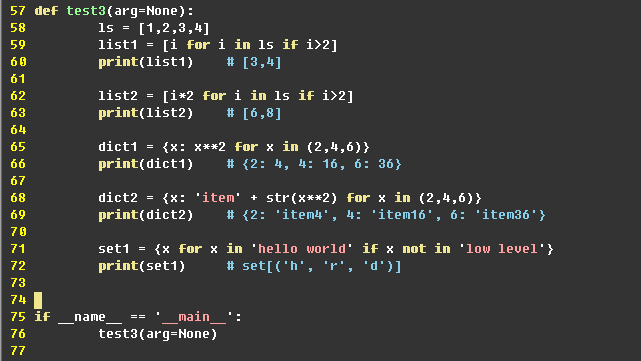
# L = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

g = (x\*x for x in range(10))

# g = <generator object <genexpr> at 0x0000028F8B774200>

即由列表对象变为生成器对象；

在Python中，常使用边循环边计算的generator机制，以节省内存空间；



## 深拷贝与浅拷贝

Python引用和copy()/deepcopy()的关系：

**import copy**

**a = [1, 2, 3, 4, ['a', 'b']] #原始对象**

**b = a #赋值，传对象的引用**

**c = copy.copy(a) #对象拷贝，浅拷贝**

**d = copy.deepcopy(a) #对象拷贝，深拷贝**

**a.append(5) #修改对象a**

**a[4].append('c') #修改对象a中的['a', 'b']数组对象**

**print 'a = ', a**

**print 'b = ', b**

**print 'c = ', c**

**print 'd = ', d**

**id(a) # 12757640**

**id(b) # 12757640**

**id(c) # 12765576**

**id(d) # 12757704**

***输出结果：***

***a = [1, 2, 3, 4, ['a', 'b', 'c'], 5]***

***b = [1, 2, 3, 4, ['a', 'b', 'c'], 5]***

***c = [1, 2, 3, 4, ['a', 'b', 'c']]***

***d = [1, 2, 3, 4, ['a', 'b']]***

在Python中，用一个变量给另一个变量赋值，就是给内存中的对象增加一个标签；浅拷贝与深拷贝都是针对组合对象来说的，组合对象是指包含了其他对象的对象，如列表、类实例等，整型、浮点型、字符串等“原子”类型则没有拷贝一说，都是对原有对象的引用；

* 浅拷贝是指创建一个新的对象，其内容是对原对象中元素的引用(拷贝组合对象不拷贝原子对象)，list对象a浅拷贝得到b，a和b都指向内存中共同的int对象：

**>>>a = [1,2,3]**

**>>>b = a**

**>>>print(id(a), id(b))**

**12757064 12765320**

**>>>for x, y in zip(x,y):**

**print(id(x), id(y))**

**1431568848 1431568848**

**1431568880 1431568880**

**1431568912 1431568912**

* 深拷贝是指创建一个对象，递归拷贝原对象包含的子对象，深拷贝出来的对象与原对象没有任何关系：

**>>> import copy**

**>>> a = [1, 2, 3]**

**>>> b = copy.deepcopy(a)**

**>>> print(id(a), id(b))**

**140601785065840 140601785066200**

**>>> for x, y in zip(a, b):**

**... print(id(x), id(y))**

**...**

***140601911441984 140601911441984***

***140601911442016 140601911442016***

***140601911442048 140601911442048***

深拷贝得到的列表元素的id之所以相同是因为对于不可变对象，当需要一个新对象时，Python可能会返回已经存在的某个类型和值都一致的对象的引用，但这种机制并不影响a和b的相互独立性，当两个元素指向同一个不可变对象时，对其中一个赋值不影响另一个；

**>>> import copy**

**>>> a = [[1, 2],[5, 6], [8, 9]]**

**>>> b = copy.copy(a) # 浅拷贝得到b**

**>>> c = copy.deepcopy(a) # 深拷贝得到c**

**>>> print(id(a), id(b)) # a 和 b 不同**

**139832578518984 139832578335520**

**>>> for x, y in zip(a, b): # a 和 b 的子对象相同**

**... print(id(x), id(y))**

**...**

***139832578622816 139832578622816***

***139832578622672 139832578622672***

***139832578623104 139832578623104***

**>>> print(id(a), id(c)) # a 和 c 不同**

**139832578518984 139832578622456**

**>>> for x, y in zip(a, c): # a 和 c 的子对象也不同**

**... print(id(x), id(y))**

**...**

***139832578622816 139832578621520***

***139832578622672 139832578518912***

***139832578623104 139832578623392***

## 类方法\_\_new\_\_( )与\_\_init\_\_( )

>>> class MyClass():

... def \_\_init\_\_(self):

... self.**\_\_superprivate** = "Hello"

... self.**\_semiprivate** = ", world!"

...

>>> mc = MyClass()

>>> print **mc.\_\_superprivate**

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: myClass instance has no attribute '\_\_superprivate'

>>> print mc.\_semiprivate

, world!

>>> print mc.\_\_dict\_\_

{'\_MyClass\_\_superprivate': 'Hello', '\_semiprivate': ', world!'}

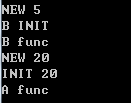
* 形如**\_\_foo\_\_：Python语言内部**的一种命名约定，用以区别其他用户自定义命名，防止命名冲突，类似\_\_init\_\_()，\_\_del\_\_()，\_\_call\_\_()，\_\_new\_\_()这些特殊方法；
* 形如**\_foo：程序员指定变量私有**，**不能使用from module import \*导入**，其他方面使用与公有变量相同；
* 形如\_\_foo：解析器使用\_classname\_\_foo替代，以区别和其他类相同的命名，无法直接像公有成员一样访问，只能通过objectname.\_classname\_\_foo()访问；

\_\_new\_\_()是一个静态方法，返回值是一个创建的实例，只有在\_\_new\_\_()返回一个cls实例之后，\_\_init\_\_()才能被调用，可以决定返回哪个cls的实例用于创建对象；

\_\_init\_\_()是一个实例方法，无返回值，创建新实例时调用\_\_new\_\_()，初始化新实例时调用\_\_init\_\_()；

\_\_metaclass\_\_在创建类时起作用，因而可以调用\_\_metaclass\_\_()，\_\_new\_\_()，\_\_init\_\_()方法，分别在类创建、实例创建和实例初始化的代码中进行改动；

以下代码返回：



类方法\_\_new\_\_(cls, arg)可以决定返回哪个对象，作用在调用实例方法\_\_init\_\_(self)创建实例之前，常用于单例模式、工厂模式等设计模式的实现；



## 字符串格式化:%与.format

"hi there %s" % name # 若name为(1,2,3)，Raise TypeError Exception

# not all arguments converted during string formatting

格式化符号%无法同时传递一个变量和元组，应当使用以下代码完成正常输出：

"hi there %s" % (name,)

此外，还可以使用.format()格式化输出：

# 使用位置参数

>>> li = ['hoho',18]

>>> 'my name is {} ,age {}'.format('hoho',18)

'my name is hoho ,age 18'

>>> 'my name is {1} ,age {0}'.format(10,'hoho')

'my name is hoho ,age 10'

>>> 'my name is {1} ,age {0} {1}'.format(10,'hoho')

'my name is hoho ,age 10 hoho'

>>> 'my name is {} ,age {}'.format(\*li)

'my name is hoho ,age 18'

# 使用关键字参数

>>> hash = {'name':'hoho','age':18}

>>> 'my name is {name},age is {age}'.format(name='hoho',age=19)

'my name is hoho,age is 19'

>>> 'my name is {name},age is {age}'.format(\*\*hash)

'my name is hoho,age is 18'

# 填充格式化 :[填充字符][对齐方式<^>][宽度]

>>> '{0:\*>10}'.format(10) ##右对齐

'\*\*\*\*\*\*\*\*10'

>>> '{0:\*<10}'.format(10) ##左对齐

'10\*\*\*\*\*\*\*\*'

>>> '{0:\*^10}'.format(10) ##居中对齐

'\*\*\*\*10\*\*\*\*'

# 数字精度与进制

>>> '{0:.2f}'.format(1/3)

'0.33'

>>> '{0:b}'.format(10) #二进制

'1010'

>>> '{0:o}'.format(10) #八进制

'12'

>>> '{0:x}'.format(10) #16进制

'a'

>>> '{:,}'.format(12369132698) #千分位格式化

'12,369,132,698'

# 使用索引

>>> li

['hoho', 18]

>>> 'name is {0[0]} age is {0[1]}'.format(li)

'name is hoho age is 18

## Python的\*args和\*\*kwargs

当不确定函数内将要传递多少参数时，使用\*args：

>>> def print\_everything(\*args):

for count, thing in enumerate(args):

... print '{0}. {1}'.format(count, thing)

...

>>> print\_everything('apple', 'banana', 'cabbage')

0. apple

1. banana

2. cabbage

\*\*kwargs允许使用事先未定义的参数名：

>>> def table\_things(\*\*kwargs):

... for name, value in kwargs.items():

... print '{0} = {1}'.format(name, value)

...

>>> table\_things(apple = 'fruit', cabbage = 'vegetable')

cabbage = vegetable

apple = fruit

## 常见的几种设计模式

单例模式：核心结构中只包含一个被称为单例类的特殊类，通过单例模式可以保证系统中一个类只有一个实例而且该实例易于外界访问，从而方便对实例个数进行控制并节约系统资源，若希望系统中某个类的对象只能存在一个，则单例模式是最适合的解决方案；

方法\_\_new\_\_()在\_\_init\_\_()之前调用，用于生成实例对象，单例模式是指创建唯一的对象，单例模式设计的类只能实例化一次：

* 使用\_\_new\_\_()方法实现：

**class Singleton(object):**

**def \_\_new\_\_(cls, \*args, \*\*kw):**

**if not hasattr(cls, ‘\_instance’):**

**orig = super(Singleton, cls)**

**cls.\_instance = orig.\_\_new\_\_(cls, \*args, \*\*kw)**

**return cls.\_instance**

**class MyClass(Singleton):**

**a = 1**

* 共享属性实现：创建实例时将实例的\_\_dict\_\_指向同一个字典，则实例具有相同的属性和方法；

**class Borg(object):**

**\_state = {}**

**def \_\_new\_\_(cls, \*args, \*\*kw):**

**ob = super(Borg, cls).\_\_new\_\_(cls, \*args, \*\*kw)**

**ob.\_\_dict\_\_ = cls.\_state**

**retrun ob**

**class MyClass1(Borg):**

**a = 1**

* 装饰器版本实现：

**def singleton(cls, \*args, \*\*kw):**

**instances = {}**

**def getinstance():**

**if cls not in instances:**

**instances[cls] = cls(\*args, \*\*kw)**

**return instances[cls]**

**return getinstance**

**@singleton**

**class MyClass:**

**…**

* import方法实现，作为python的模块是天然的单例模式；

# mysingleton.py

class My\_Singleton(object):

def foo(self):

pass

my\_singleton = My\_Singleton()

# to use

from mysingleton import my\_singleton

my\_singleton.foo()

## Python变量的作用域

* 决定Python变量作用域最小单位的是关键字def，类似于Java中的{…}，Python中能够改变变量作用域的代码段是def、class、lambda；
* if/elif/else、try/except/finally、for/while并不能涉及变量作用域的更改，即其内部代码块中的变量在外部也可访问；
* 变量的搜索路径是Local Variable 🡪 Global Variable

def scopetest():

var = 6

print(var) # 6

def innerfunc():

print(var) #6

innerfunc()

var = 5

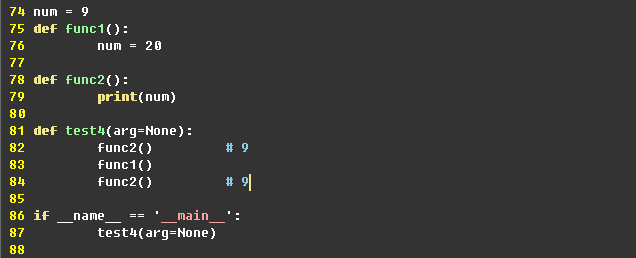
print(var) # 5

scopetest()

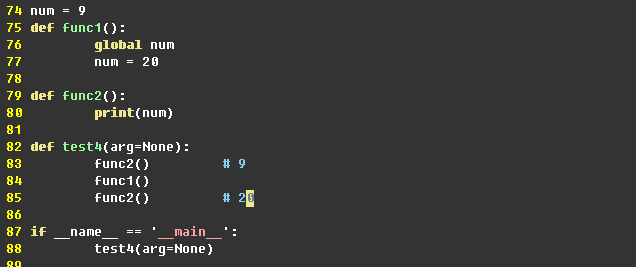
print(var) # 5

调用顺序：Local 🡪 Global，def作为变量作用域标示符，innerfunc()中的var首先在其定义域内部进行搜索，没有找到则上溯到其主调函数的作用域内搜索，找到var = 6，使用其值；

全局变量与局部变量：



在函数func1()与func2()中，仅得到num的拷贝，函数内部不会改变外部num的数值，要想改变全局变量的数值，需要在函数内部加上global关键字；



## GIL线程全局锁及协程

* 线程全局锁(Global Interpreter Lock)是Python为了保护线程安全而采取的独立线程运行的限制机制，即一个核在同一时刻只能运行一个线程，对于IO密集型任务，Python的多线程机制起到作用，对于CPU密集型任务，python多线程任务可能会因为争夺资源而变慢，解决办法即使用多进程和协程：
* 协程即用户自己控制内核态与用户态的切换，不用陷入系统的内核态，减少切换时间，Python中的yield即使用协程的思想；
* 在GIL之下，线程间的切换成为一个较大的开销，协程专注于解决I/O密集型任务；

在进程中可同时存在一个或多个线程，每个进程拥有独立的地址空间、内存、堆栈和其他数据段；

线程具有开始、顺序执行和结束三个阶段，线程是CPU调动的，没有独立的资源，多有线程共享统一进程中的资源，在Python中，为了解决多线程访问共享资源的数据保护问题，产生了线程全局锁(GIL)；

线程锁：CPU执行任务时，线程间的调度是随机进行的，并且每个线程可能只是执行n条语句就要转而执行其他线程，由于进程中多个线程间共享数据和资源很容易产生资源抢夺和脏数据，于是需要使用线程锁(GIL)限制对指定数据的访问；

Python在解释器的层面限制了程序在同一时刻只有一个线程被CPU实际执行，从而导致多线程编程效率过低，计算密集型任务推荐使用多进程，IO密集型任务推荐多线程，防止同一资源被占用的情况；

协程详解见[协程](#_协程)；

## 闭包

如果在一个内部函数中，对外部作用域(非全局作用域)的变量进行引用，则内部函数被认为是一个闭包(closure)：

>>>def addx(x):

>>> def adder(y): return x + y

>>> return adder

>>> c = addx(8)

>>> type(c)

<type 'function'>

>>> c.\_\_name\_\_

'adder'

>>> c(10)

18

adder(y)作为内部函数，对在外部作用域(非全局作用域)的变量x进行引用，x在外部作用域addx内，不在全局作用域内，adder(y)就是一个闭包；

闭包 = 函数块 + 定义函数时的环境，adder是函数块，x即环境；

闭包无法修改外部作用域的局部变量：

**def foo():**

**a = 1**

**def bar():**

**a = a+1 # 将a看做bar()的局部变量，赋值时出错**

**return a**

**return bar**

**>>> c = foo()**

**>>> print c()**

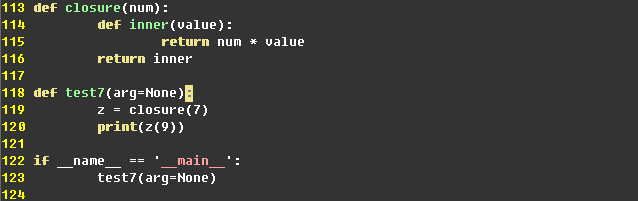
***Traceback (most recent call last):***

***File "<stdin>", line 1, in <module>***

***File "<stdin>", line 4, in bar***

***UnboundLocalError: local variable 'a' referenced before assignment***

在Python 3中，可以在增1语句a = a+1之前添加语句nonlocal a声明a非闭包局部变量即可；



以上代码实现了一个闭包，调用函数z = closure(7)返回一个函数，再通过调用z(9)将参数传入，实现相乘操作；

## Lambda与函数式表达式

* filter过滤器函数：

>>>a = [1,2,3,4,5,6,7]

>>>b = filter(lambda x: x>5, a)

>>>list(b)

[6,7]

* map依次执行函数：

>>> a = map(lambda x:x\*2,[1,2,3])

>>> list(a)

[2, 4, 6]

* reduce迭代调用函数,python 3.0.0及更高版本中，reduce()已经被移出内建函数，需要从from functools import reduce导入：

>>>from functools import reduce

>>>reduce(lambda x, y: x+y, range(1, 9))

40320

## 编码与解码(incode/decode)

编码的目的是兼容字符集之间的通用性，通常用到编码的环境有：

* 系统默认编码
* 程序运行环境编码
* 源文件自身编码
* 源代码中字符串编码

通常中文操作系统中的字符编码是gbk格式，Python运行时的字符编码默认为ASCII，源文件编码格式可指定 # -\*- coding: utf-8 -\*-

程序内容的编码可以通过Python提供的函数进行转换(Python2中字符串有两种类型unicode字符串和非unicode字符串，而Python3中只有unicode字符串)，可以使用str.encode(‘utf-8’)将指定字符串编码为’utf-8’或其他格式，使用s.decode(‘utf-8’)解码字符串；

指定的编码字符只能用指定的解码方式进行解码，否则无法还原原有的字符串；

在Python3中，取消了unicode类型，使用unicode字符类型的字符串替代，编码后变成了字节类型(bytes)：bytes 🡪 Unicode string 🡪 bytes

***u = u'中文' #显示指定unicode类型对象u***

***str = u.encode('gb2312') #以gb2312编码对unicode对像进行编码***

***str1 = u.encode('gbk') #以gbk编码对unicode对像进行编码***

***str2 = u.encode('utf-8') #以utf-8编码对unicode对像进行编码***

***u1 = str.decode('gb2312')#以gb2312编码对字符串str进行解码，以获取unicode***

***u2 = str.decode('utf-8')#如果以utf-8的编码对str进行解码得到的结果，将无法还原原来的unicode类型***

Python提供了包codecs进行文件读取，codec.open()函数可以指定编码类型：

***import codecs***

***f = codecs.open('text.text','r+',encoding='utf-8')#必须事先知道文件的编码格式，这里文件编码是使用的utf-8***

***content = f.read()#如果open时使用的encoding和文件本身的encoding不一致的话，那么这里将将会产生错误***

***f.write('你想要写入的信息')***

***f.close()***

## 迭代器与生成器

迭代器可以减少内存开销：迭代器属于临时区，安排一些元素在其中，使用时才创建临时区，一旦遍历结束即清空临时区，再次遍历时临时区失效。

# -\*- coding: utf-8 -\*-

# -\*- version: python 3.5.4 -\*-

# \_iter.py

import sys

i = iter(range(10000))

print("id(i.\_\_next\_\_()) = ", id(i.\_\_next\_\_()))

print("sys.getsizeof(i) = ", sys.getsizeof(i))

print("sys.getsizeof(i.\_\_next\_\_()) = ", sys.getsizeof(i.\_\_next\_\_()))

# 若一次性将list对象全部加载进来需要90112bytes

# 使用迭代器仅需要28bytes

e = range(10000)

print("sys.getsizeof(e) = ", sys.getsizeof(e))

print("sys.getsizeof(list(e))", sys.getsizeof(list(e)))

可以使用next()函数不断返回下一个值的对象称为迭代器iterator，生成器都是迭代器对象；list dict string虽然具有可迭代属性，但不是Iterator，可以使用iter()转换成iterator，验证是否是迭代器的方法即查看是否具有.\_\_next\_\_()方法；

生成器是一种特殊的迭代器，常在性能限制的条件下应用，readline()/readlines()就是一种常用的生成器，可在循环读取时不断处理，节省内存空间；

生成生成器的方法：

* 方法一：g = (x\*x for x in range(10)) # 类似迭代器
* 在函数中加入yield关键字用来返回值，yield遇到next()方法即返回值，再次执行时从上次yield返回处继续执行；

在函数中使用yield，则函数就变为一个生成器gernerator object，yield会生成一个序列，但不会全部返回，只有在调用.\_\_next\_\_()方法时才会返回一个值，实际使用时，使用循环将生成器所有数值返回：

def func(n):

for i in range(n):

yield i

for x in func(5):

print(x)

查看一个函数是否是生成器 ：

使用自省函数dir(g)，返回：

[‘\_\_class\_\_’,

….

‘\_\_next\_\_’,

…

‘\_\_repr\_\_’,

‘\_\_setattr\_\_’,

]

其中，魔术方法\_\_next\_\_()是生成器特有的属性，可通过调用g.\_\_next\_\_()或next(g)获得下一个生成的对象；

## 装饰器

装饰器是一种设计模式，常被应用于有切面需求的场景：插入日志、性能测试、事务处理等，装饰器的作用即为已经存在的对象添加额外的功能；

#-\*- coding: UTF-8 -\*-

import time

def foo():

print 'in foo()'

# 定义一个计时器，传入一个，并返回另一个附加了计时功能的方法

def timeit(func):

# 定义一个内嵌的包装函数，给传入的函数加上计时功能的包装

def wrapper():

start = time.clock()

func()

end =time.clock()

print 'used:', end - start

# 将包装后的函数返回

return wrapper

foo = timeit(foo)

foo()

在定义函数foo()之后，调用之前，加上foo = timeit(foo)语句，即可达到计时的目的，在这个例子中，函数进入和退出时均需要计时，成为一个横切面Aspect，这种编程方式叫做Aspect-Oriented Programming；

使用语法糖@实现，可实现在函数定义之前进行装饰：

import time

def timeit(func):

def wrapper():

start = time.clock()

func()

end =time.clock()

print 'used:', end - start

return wrapper

@timeit

def foo():

print 'in foo()'

foo()

在函数之后进行装饰；

使用装饰器实现单例模式；

## Python中的重载

函数重载为了解决两个问题：1. 可变的参数类型；2. 可变的参数个数；函数重载的设计原则是：仅当两个函数除了参数类型和参数个数不同之外，其功能完全相同，此时需要函数重载；

对于1：Python可接受任何类型的参数，若函数功能相同而参数类型不同，则在Python中代码极有可能完全相同，没有必要使用重载函数实现；

对于2：函数功能相同而参数个数不同，在Python中，使用不定长参数\*args实现，对于缺少的参数，将其设定为不定长参数即可解决问题。

函数参数类型：

* 必备参数：必备参数须以正确的顺序传入函数。调用时的数量必须和声明时的一样；
* 关键字参数：关键字参数和函数调用关系紧密，函数调用使用关键字参数来确定传入的参数值，使用关键字参数允许函数调用时参数的顺序与声明时不一致，因为 Python 解释器能够用参数名匹配参数值；
* 缺省参数：调用函数时，缺省参数的值如果没有传入，则被认为是默认值；
* **不定长参数：处理必声明时更多的参数，在声明时不命名也不固定长度，语法表示为 def functionname([format\_args], \*var\_args\_tuple)：**

#!/usr/bin/python

# -\*- coding: UTF-8 -\*-

# 可写函数说明

def printinfo( arg1, \*vartuple ):

"打印任何传入的参数"

print "输出: "

print arg1

for var in vartuple:

print var

return;

# 调用printinfo 函数

printinfo( 10 );

printinfo( 70, 60, 50 );

## Python新式类与旧式类

新式类在Python2.2之前就出现了，故旧式类问题完全是一个兼容问题，Python3中的类全部是新式类，新式类加载采用广度优先策略，旧式类加载采用深度优先策略；

class A():

def foo1(self):

print "A"

class B(A):

def foo2(self):

pass

class C(A):

def foo1(self):

print "C"

class D(B, C):

pass

d = D()

d.foo1()

# in Python2.2 or lower version : A

# in Python2.3 or higher version : C

按照经典类的查找顺序，从左到右深度优先，创建实例d，调用方法foo1()，类D中没有方法foo1()，查找class B，B中也没有方法foo1()，深度优先，继续查找类A，而非广度优先查找类C，调用类A的方法foo1()，输出”A”，而在新式类中，则会查找类C，调用重写的函数foo1()，输出”C”。

## 邮箱地址正则表达式

正则表达式是一种字符串匹配模式，可以用来检查一个子串是否含有某个子串，将匹配子串替换或从某个串中取出符合条件的子串等，例如：

* runoo+b，可以匹配 runoob、runooob、runoooooob 等，+ 号代表前面的字符必须至少出现一次（1次或多次）；
* runoo\*b，可以匹配 runob、runoob、runoooooob 等，\* 号代表字符可以不出现，也可以出现一次或者多次（0次、或1次、或多次）；
* colou?r 可以匹配 color 或者 colour，? 问号代表前面的字符最多只可以出现一次（0次、或1次）；

正则表达式的组件可以是单个字符串、字符集合、字符范围、字符间的选择或以上组件的任意组合，正则表达式有普通字符(a~z，0~9)及特殊字符(元字符)组成的文字模式，模式描述在搜索文本时要匹配一个或者多个字符串，正则表达式作为一个模板，将某个字符模式与所搜索的字符串进行匹配；

* 普通字符：普通字符包含没有显式指定为元字符的所有可打印和不可打印字符(所有大小写字母、所有数字、所有标点符号以及一些其他符号)；
* 非打印字符(转义字符)：非打印字符也可以是正则表达式的组成部分。下表列出了表示非打印字符的转义序列：

|  |  |
| --- | --- |
| **字符** | **描述** |
| **\cx** | **匹配由x指明的控制字符**。例如， \cM 匹配一个 Control-M 或回车符。x 的值必须为 A-Z 或 a-z 之一。否则，将 c 视为一个原义的 'c' 字符。 |
| **\f** | **匹配一个换页符**。等价于 \x0c 和 \cL。 |
| **\n** | **匹配一个换行符**。等价于 \x0a 和 \cJ。 |
| **\r** | **匹配一个回车符**。等价于 \x0d 和 \cM。 |
| **\s** | **匹配任何空白字符，包括空格、制表符、换页符**等等。等价于 [ \f\n\r\t\v]。 |
| **\S** | **匹配任何非空白字符**。等价于 [^ \f\n\r\t\v]。 |
| **\t** | **匹配一个制表符**。等价于 \x09 和 \cI。 |
| **\v** | **匹配一个垂直制表符**。等价于 \x0b 和 \cK。 |

* 特殊字符：具有特殊含义的字符，要在字符串中进行查找，应当对这些字符进行转义(加”\”)，如要在字符串中查找”\*”，应当在对\*进行转义：runo\\*ob；

| **特别字符** | **描述** |
| --- | --- |
| **$** | **匹配输入字符串的结尾位置**。如果设置了 RegExp 对象的 Multiline 属性，则 $ 也匹配 '\n' 或 '\r'。要匹配 $ 字符本身，请使用 \$。 |
| **( )** | **标记一个子表达式的开始和结束位置**。子表达式可以获取供以后使用。要匹配这些字符，请使用 \( 和 \)。 |
| **\*** | **匹配前面的子表达式零次或多次**。要匹配 \* 字符，请使用 \\*。 |
| **+** | **匹配前面的子表达式一次或多次**。要匹配 + 字符，请使用 \+。 |
| **.** | **匹配除换行符 \n 之外的任何单字符**。要匹配 . ，请使用 \. 。 |
| **[** | **标记一个中括号表达式的开始**。要匹配 [，请使用 \[。 |
| **?** | **匹配前面的子表达式零次或一次，或指明一个非贪婪限定符**。要匹配 ? 字符，请使用 \?。 |
| **\** | **将下一个字符标记为或特殊字符、或原义字符、或向后引用、或八进制转义符**。例如， 'n' 匹配字符 'n'。'\n' 匹配换行符。序列 '\\' 匹配 "\"，而 '\(' 则匹配 "("。 |
| **^** | **匹配输入字符串的开始位置，除非在方括号表达式中使用，此时它表示不接受该字符集合**。要匹配 ^ 字符本身，请使用 \^。 |
| **{** | **标记限定符表达式的开始**。要匹配 {，请使用 \{。 |
| **|** | **指明两项之间的一个选择**。要匹配 |，请使用 \|。 |

* 限定符：限定符用来指定正则表达式中某一给定组件必须出现多少次才能匹配，共有\* + ? {n} {n,} {n,m}六种，正则表达式的限定符有：

|  |  |
| --- | --- |
| **字符** | **描述** |
| **\*** | **匹配前面的子表达式零次或多次**。例如，zo\* 能匹配 "z" 以及 "zoo"。\* 等价于{0,}。 |
| **+** | **匹配前面的子表达式一次或多次**。例如，'zo+' 能匹配 "zo" 以及 "zoo"，但不能匹配 "z"。+ 等价于 {1,}。 |
| **?** | **匹配前面的子表达式零次或一次**。例如，"do(es)?" 可以匹配 "do" 、 "does" 中的 "does" 、 "doxy" 中的 "do" 。? 等价于 {0,1}。 |
| **{n}** | **n 是一个非负整数。匹配确定的 n 次**。例如，'o{2}' 不能匹配 "Bob" 中的 'o'，但是能匹配 "food" 中的两个 o。 |
| **{n,}** | **n 是一个非负整数。至少匹配n 次**。例如，'o{2,}' 不能匹配 "Bob" 中的 'o'，但能匹配 "foooood" 中的所有 o。'o{1,}' 等价于 'o+'。'o{0,}' 则等价于 'o\*'。 |
| **{n,m}** | **m 和 n 均为非负整数，其中n <= m。最少匹配 n 次且最多匹配 m 次**。例如，"o{1,3}" 将匹配 "fooooood" 中的前三个 o。'o{0,1}' 等价于 'o?'。请注意在逗号和两个数之间不能有空格。 |

由于章节编号在大的输入文档中会超过9，限定符可以使用/Chapter [1-9][0-9]\*/表达式匹配任何位数的章节号；\*、+限定符都是贪婪的，总是尽可能多的匹配文字，只有在其后面加上一个?才能实现非贪婪或最小匹配；

原始文档：*<H1>Chapter 1 – Introduce to Regular Expression</H1>*

/<.\*>/ ：贪婪匹配***H1>Chapter 1 – Introduce to Regular Expression</H1***

/<.\*?>/：非贪婪匹配***H1***

/<\w+?>/：非贪婪匹配***H1***

* 定位字符：将正则表达式固定到一个单词内、单词开头或单词结尾处，定位符用来描述字符串或单词的边界；

|  |  |
| --- | --- |
| **字符** | **描述** |
| **^** | **匹配输入字符串开始的位置**。如果设置了 RegExp 对象的 Multiline 属性，^ 还会与 \n 或 \r 之后的位置匹配。 |
| **$** | **匹配输入字符串结尾的位置**。如果设置了 RegExp 对象的 Multiline 属性，$ 还会与 \n 或 \r 之前的位置匹配。 |
| **\b** | **匹配一个字边界，即字与空格间的位置**。 |
| **\B** | **非字边界匹配**。 |

/^Chapter [1-9][0-9]{0,1}$/可匹配在章节号而不匹配交叉引用；

/ter\b/匹配字符串Chapter的子串”ter”；

/\Bapt/匹配字符串Chapter的子串”apt”；

* 选择：将()把所有选择项括起来，相邻选择项之间用|分隔，产生的匹配会被存储到一个临时缓冲区中，所捕获的每一个子匹配都按照在正则表达式模式中从左向右出现的顺序存储，缓冲区编号范围1~99，每个缓冲区均可使用\index访问，其中；index为缓冲区编号；
* 反向引用：对()产生的匹配缓冲区，可以使用**?:、?=或?!**重写捕获，忽略对相关匹配的保存；

**var str = "Is is the cost of of gasoline going up up";**

**var patt1 = /\b([a-z]+) \1\b/ig;**

**document.write(str.match(patt1));**

捕获的表达式由[a-z]+指定，包含一个或多个字母，\b字边界符匹配两个空格之间的子串即单词；

\1是对捕获的子匹配项的引用，即单词的第二个匹配项恰好由()表达式匹配，\1指定第一个子匹配项；

字边界元字符确保只检测整个单词；

正则表达式后面的全局标记g指定将该表达式应用到输入字符串中查找尽可能多的匹配；

结尾处的不区分大小写标记i指定不区分大小写；

**var str = "http://www.runoob.com:80/html/html-tutorial.html";**

**var patt1 = /(\w+):\/\/([^/:]+)(:\d\*)?([^# ]\*)/;**

**arr = str.match(patt1);**

**for (var i = 0; i < arr.length ; i++) {**

**document.write(arr[i]);**

**document.write("<br>");**

**}**

**(\w+)**子表达式捕获Web地址的协议部分，”**://”**之前的任意字符，匹配”**http”** ；

**([^/:]+)**子表达式捕获端口号”**:80”**之前的部分，匹配”**www.runoob.com”** ；

**(:\d\*)?**子表达式捕获端口号，匹配”**:”**之后的零个或多个数字，且只能重复一次；

**([^# ]\*)**子表达式捕获Web地址指定的路径和/或页信息，能匹配除**”#”**或空格字符之外的任何字符序列，匹配”**/html-tutorial.html”**；

**正则表达式匹配邮箱地址：**

**re = /^\w+([\.-]?\w+)\*@\w+([\.-]?\w+)\*(\.\w{2,3})+$/**

(/)斜杠之间的所有内容都是正则表达式的组成部分；

脱字符(^)表示要用这个表达式检查以特定字符串(此处为\w+)开头的字符串；

(\w)表示任意单一字符，含a~z、A~Z、0~9或下划线这些电子邮箱地址开头合法字符；

(+)表示要寻找的前面的条目(单一字符)多于一次的出现；

()表示一个组，表达式后面的内容可能要引用()中的内容，因此将其放入一个组内；

[]表示可以出现其中任意的字符，[\.-]允许邮箱地址中出现”.”点号或连字符”-”，因为”.”在正则表达式中代表单字符匹配，应当对其进行转义，此处[]表示，在邮箱地址中可以有”.”或”-”其中一个存在，但不能同时存在；

(?)表示前面的条目可以出现一次或者不出现，即电子邮箱地址的第一部分中可以有一个”.”或者”-”，也可以没有；

(?)后面的\w+表示”.”或者”-”后面必须要有其他字符；

()后出现的”\*”表示([\.-]?\w+)组内的条目可以出现0次或者多次；

“@”为其本义，邮箱地址必须有，位于电子邮箱地址和域名之间；

“@”后再次出现”\w+”，表明”@”后必须出现字符，([\.-]?\w+)表明在邮箱域名中允许出现”.”或者”-”；

(\.\w{2,3})表示在邮箱域名中，能够匹配到一个”.”其后跟随2~3个字符，而”+”表明(\.\w{2,3})组内的模式必须出现一次或者多次，形如”.com” “.cn” “.havard.edu”；

“$”表示匹配字符串到此处结束；

## Python内置类型list/dictionary/tuple/string

* list列表，即动态数组，对应于C++的vector，可以包含不同类型的元素在同一个list中；

按下标读写，当做数组处理，**list[0] list[-1]…**；

**len(list)**获得list的长度；

创建连续的list ：**list = range(1, 5) # list = [1,2,3,4]，不含右边界**

list常用方法：

**L.append(var) #追加元素**

**L.insert(index, var) #插入元素**

**L.pop(var) #返回最后一个元素，并从list中删除**

**L.remove(var) #删除第一次出现的该元素**

**L.count(var) #统计该元素在list中出现的次数**

**L.index(var) #该元素出现的位置**

**L.extend(list) #追加list到L上**

**L.sort() #排序**

**L.reverse() #倒序**

**L[1:] #片段操作符，从1~最后**

**[1,2] + [3,4] #作用同extend()**

**[2]\*4 #[2,2,2,2]**

**del L[index] #删除指定下标的元素**

**del L[l, h] #删除指定下表范围的元素**

**L1 = L #L1是L的别名，指向相同的地址**

**L1 = L[:] #L1为L的拷贝，深拷贝**

**[<expr1> for k in L if <expr2>] #list遍历**

* dictionary字典，对应于C++中的Map，键值对的格式存储dict = {‘ob1’: ‘str1’, ‘ob2’: ‘str2’, ‘ob3’: ‘str3’, …}；

常用的dict方法：

D.get(key, 0) #同D[key]，没有则返回默认值0

D.has\_key(key) #有此键则返回TRUE，否则返回FALSE

D.keys() #返回字典键的列表

D.values() #返回字典值的列表

D.items() #返回字典键值对全列表

D.update(dict2) #增加合并字典

D.popitem() #弹出第一个item并从字典中删除

D.clear() #清空字典，同del D

D.copy() #拷贝字典，浅拷贝

D.cmp(dict1, dict2) #比较字典，比较优先级元素个数🡪key🡪value，#第一个大则返回1，小则返回-1

dict1 = dict #dict1只是别名，指向同一块内存

dict2 = dict.copy() #拷贝

* tuple元组即常量数组，tuple = (‘a’, ‘b’, ‘c’, ‘d’, ‘e’)，可以使用list的[index]，:操作符提取元素，但不能修改(常量)；
* string字符串不可变对象，无法直接修改字符串中的子串，但是可以提取str[:6]，常用的字符串方法：

**substr in str #判断子串是否在字符串中**

**substr not in str**

**S.find(substring, [start [,end]]) #指定范围查找子串，返回索引**

**S.rfind(substring, [start [,end]]) #反向查找**

**S.index(substring, [start [,end]]) #同find()，查找不到则抛出异常**

**S.rindex(substring, [start [,end]])**

**S.lowercase()**

**S.capitalize() #首字母大写**

**S.lower()**

**S.upper()**

**S.swapcase()**

**S.split(string, ‘ ’) #以空格切分string成list**

**S.join(list, ‘ ’) #将list转成string，以空格连接**

**S.strip(‘c’) #移出头尾指定字符c并返回**

**len(str) #串长度**

**cmp("my friend", str) #字符串比较，第一个大返回1**

**max('abcxyz') #寻找字符串中最大的字符**

**min('abcxyz') #寻找字符串中最小的字符**

**oat(str) #变成浮点数float("1e-1") 对应0.1**

**int(str) #变成整型int("12")对应12**

**int(str,base) #变成base进制整型数，int("11",2) 对应2**

**long(str) #变成长整型**

**long(str,base) #变成base进制长整型**

**str\_format % (参数列表) #参数列表是以tuple的形式定义的，即不可运行中改变**

**>>>print ""%s's height is %dcm" % ("My brother", 180)**

**#结果显示为 My brother's height is 180cm**

**list 和 tuple 的相互转化**

**tuple(ls)**

**list(ls)**

## Python中的is

“is”方法是对比对象的地址，”==”对比的是对象的值。

## read/readline和readlines

* read()读取整个文件；
* readline()读取下一行，使用生成器方法；
* readlines()读取整个文件到一个迭代器中供遍历使用；

**fh = open('c:\\autoexec.bat')**

**for line in fh.readlines(): # 将文件读入缓冲区**

**print(line)**

## 垃圾回收

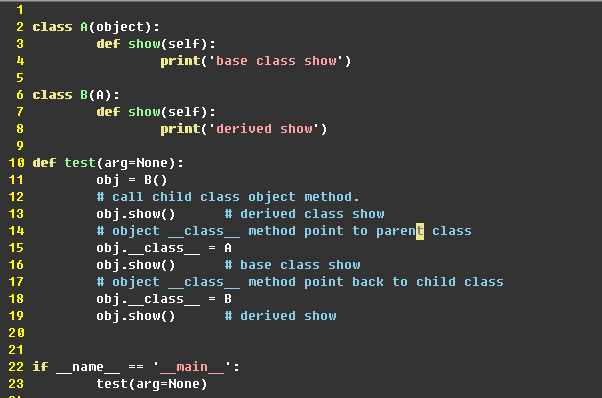
Python Garbage Collection通过引用计数Reference Counting来跟踪和回收垃圾，在引用计数的基础上，通过“标记-清除”(Mark and Sweep)解决容器对象可能产生的循环引用问题，通过分代回收(Generation Collection)以空间换时间的方法提高垃圾回收的效率；

* 引用计数：PyObject是每个对象必有的内容，其中ob\_refcnt作为引用计数，当一个对象有新的引用时，ob\_refcnt计数会增加，当引用对象被删除时，ob\_refcnt计数会减少，当ob\_refcnt清零时，该对象的生命周期结束；
* 标记-清除机制：按需分配，当没有空闲内存时，从寄存器和栈上的引用出发，遍历以对象为节点，以引用为边构成的图，将所有可以访问到的对象打上标记，清扫内存，将没有标记的对象释放；
* 分代回收：将系统中所有内存块根据其存活时间划分为不同的集合，每个集合成为一个“代”，垃圾收集频率随着“代”的存活时间增大而减小，存活时间通常利用记过几次垃圾回收来度量。

## 类继承

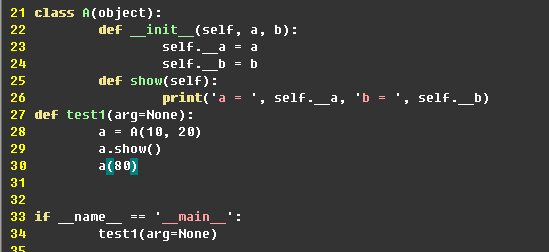
派生类(子类)调用超类(基类，父类)的方法：

派生类对象的***\_\_class\_\_***方法指向超类，可以实现超类方法的调用，调用后要将***\_\_class\_\_***方法的指向恢复：



## 方法对象

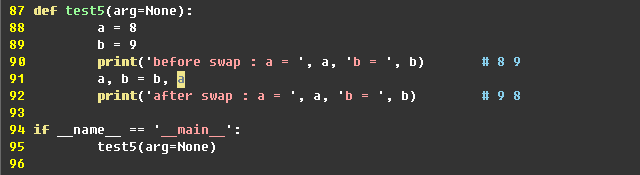
为了实现对象实例可以被直接调用，需要在类中实现\_\_call\_\_方法，以下代码会报错：***ValueError: object ‘A’ is not callable***



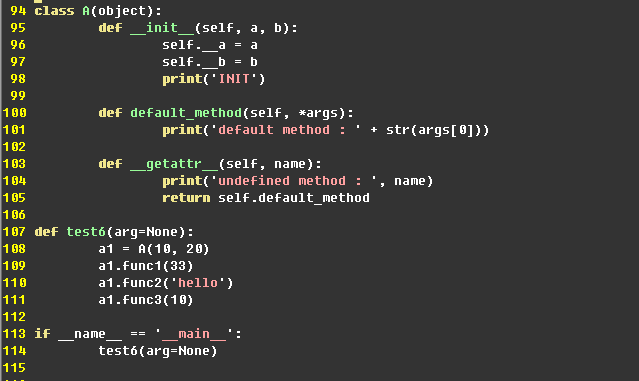
在类中直接实现***\_\_call\_\_***方法：



## 一行代码交换参数值



## 未定义方法调用

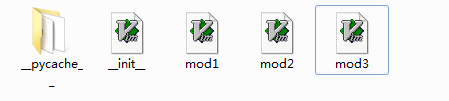


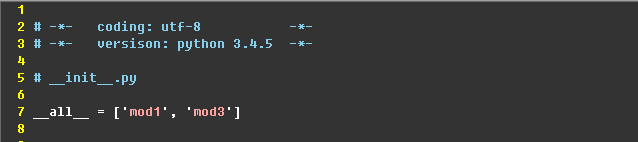
当调用未定义的对象方法时，重定向到一个默认方法上，并兼容参数列表，需要在类中实现**\_\_getattr\_\_(self, name)**方法，参数**name**可获得未定义方法名称，并返回类中实现的默认方法；

## 包管理

在python中，包package就是一个包含了\_\_init\_\_.py以及其他模块(mod.py)文件的路径，**from package import mod可以导入指定模块，而from package import \*可以导入所有模块，在\_\_init\_\_.py中增加语句**

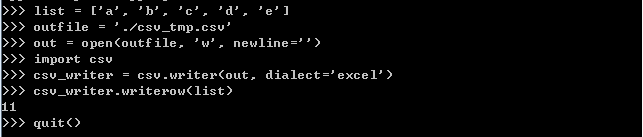
**\_\_all\_\_ = [‘mod1’, ‘mod2’, …]可以指定允许导入的模块；**





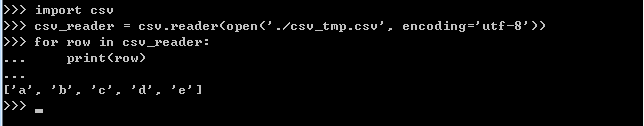
## 读写csv文件

写csv文件：



退出python后，可在本地目录下查看创建的csv文件；

读取csv文件：



读取第三方提供的csv文件时，会遭遇编码问题，参见：

[编码与解码(incode/decode)](#_编码与解码(incode/decode))；

# Algorithm

## 时间复杂度计算

待解决问题的规模为n，基本操作被重复执行的次数为n的函数，时间复杂度记作：T(n) = O[f(n)]，表示随着问题规模n的增长，算法执行的时间增长率增长率与f(n)的增长率相同；

for(i=1; i<=n; i++){

for(j=1; j<=i; j++){

++x; //执行频度为1+2+3+…+n=(n+1)n/2

a[i,j] = x;

}

}

常见的复杂度：

* 常数阶O(1)：没有循环(for/while)，仅对变量做常数范围以内的操作；
* 线性阶O(n)：一层循环

int i;

for(i=0; i<n; i++){

//operations

}

* 对数阶O(log(n))：

int count = 1;

while(count < n){

count = count \* 2;

}

未限定问题规模，2^x = n；得出x = log(n)，算法复杂度为O(log(n))；

* 平方阶O(n^2)：两重循环

int i,j;

for(i=0; i<n; i++){

for(j=0; j<n; j++){

//operations

}

}

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **排序法** | **最差时间分析** | **平均时间复杂度** | **稳定度** | **空间复杂度** |
| **冒泡排序** | **O(n2)** | **O(n2)** | **稳定** | **O(1)** |
| **快速排序** | **O(n2)** | **O(n\*log2n)** | **不稳定** | **O(log2n)~O(n)** |
| **选择排序** | **O(n2)** | **O(n2)** | **稳定** | **O(1)** |
| **二叉树排序** | **O(n2)** | **O(n\*log2n)** | **不一顶** | **O(n)** |
| **插入排序** | **O(n2)** | **O(n2)** | **稳定** | **O(1)** |
| **堆排序** | **O(n\*log2n)** | **O(n\*log2n)** | **不稳定** | **O(1)** |
| **希尔排序** | **O** | **O** | **不稳定** | **O(1)** |

1、时间复杂度   
（1）时间频度 一个算法执行所耗费的时间，从理论上是不能算出来的，必须上机运行测试才能知道。但我们不可能也没有必要对每个算法都上机测试，只需知道哪个算法花费的时间多，哪个算法花费的时间少就可以了。并且一个算法花费的时间与算法中语句的执行次数成正比例，哪个算法中语句执行次数多，它花费时间就多。一个算法中的语句执行次数称为语句频度或时间频度。记为T(n)。   
（2）时间复杂度 在刚才提到的时间频度中，n称为问题的规模，当n不断变化时，时间频度T(n)也会不断变化。但有时我们想知道它变化时呈现什么规律。为此，我们引入时间复杂度概念。 一般情况下，算法中基本操作重复执行的次数是问题规模n的某个函数，用T(n)表示，若有某个辅助函数f(n),使得当n趋近于无穷大时，T（n)/f(n)的极限值为不等于零的常数，则称f(n)是T(n)的同数量级函数。记作T(n)=Ｏ(f(n)),称Ｏ(f(n)) 为算法的渐进时间复杂度，简称时间复杂度。   
在各种不同算法中，若算法中语句执行次数为一个常数，则时间复杂度为O(1),另外，在时间频度不相同时，时间复杂度有可能相同，如T(n)=n2+3n+4与T(n)=4n2+2n+1它们的频度不同，但时间复杂度相同，都为O(n2)。 按数量级递增排列，常见的时间复杂度有：常数阶O(1),对数阶O(log2n),线性阶O(n), 线性对数阶O(nlog2n),平方阶O(n2)，立方阶O(n3),...， k次方阶O(nk),指数阶O(2n)。随着问题规模n的不断增大，上述时间复杂度不断增大，算法的执行效率越低。 2、空间复杂度 与时间复杂度类似，空间复杂度是指算法在计算机内执行时所需存储空间的度量。记作: S(n)=O(f(n)) 我们一般所讨论的是除正常占用内存开销外的辅助存储单元规模。讨论方法与时间复杂度类似，不再赘述。   
（3）渐进时间复杂度评价算法时间性能 　　主要用算法时间复杂度的数量级(即算法的渐近时间复杂度)评价一个算法的时间性能。

2、类似于时间复杂度的讨论，一个算法的空间复杂度(Space Complexity)S(n)定义为该算法所耗费的存储空间，它也是问题规模n的函数。渐近空间复杂度也常常简称为空间复杂度。   
空间复杂度(Space Complexity)是对一个算法在运行过程中临时占用存储空间大小的量度。一个算法在计算机存储器上所占用的存储空间，包括存储算法本身所占用的存储空间，算法的输入输出数据所占用的存储空间和算法在运行过程中临时占用的存储空间这三个方面。算法的输入输出数据所占用的存储空间是由要解决的问题决定的，是通过参数表由调用函数传递而来的，它不随本算法的不同而改变。存储算法本身所占用的存储空间与算法书写的长短成正比，要压缩这方面的存储空间，就必须编写出较短的算法。算法在运行过程中临时占用的存储空间随算法的不同而异，有的算法只需要占用少量的临时工作单元，而且不随问题规模的大小而改变，我们称这种算法是“就地/"进行的，是节省存储的算法，如这一节介绍过的几个算法都是如此；有的算法需要占用的临时工作单元数与解决问题的规模n有关，它随着n的增大而增大，当n较大时，将占用较多的存储单元，例如将在第九章介绍的快速排序和归并排序算法就属于这种情况。

如当一个算法的空间复杂度为一个常量，即不随被处理数据量n的大小而改变时，可表示为O(1)；当一个算法的空间复杂度与以2为底的n的对数成正比时，可表示为0(10g2n)；当一个算法的空I司复杂度与n成线性比例关系时，可表示为0(n).若形参为数组，则只需要为它分配一个存储由实参传送来的一个地址指针的空间，即一个机器字长空间；若形参为引用方式，则也只需要为其分配存储一个地址的空间，用它来存储对应实参变量的地址，以便由系统自动引用实参变量。

## 二叉树

### 3.2.1 二叉树的数据结构



二叉树的三种遍历：

* 先序遍历：ParentNode—LeftNode—RightNode；
* 中序遍历：LeftNode—ParentNode—RightNode；
* 后序遍历：LeftNode—RightNode—ParentNode；
* 层序遍历：逐层遍历；

二叉树常用作二叉查找树、二叉堆和二叉排序树；

//二叉树节点数据结构

typedef struct BiTreeNode{

// Data Field

char data;

// Child pointer

struct BiTreeNode \*lchild, \*rchild;

}BiTreeNode, \*BiTree;

### 3.2.2 二叉树的生成

// 先序生成创建二叉树

int CreateBiTree(BiTree &T){

char data;

// 按先序顺序输入二叉树节点中的数值

InputNode(&data);

if(CHARP\_VALUE == data){

T = NULL;

}

else{

T = (BiTree)malloc(sizeof(BiTreeNode));

T->data = data;

CreateBiTree(T->lchild);

CreateBiTree(T->rchild);

}

return 0;

}

### 3.2.3 二叉树的遍历算法

三种遍历均有递归实现与循环实现两种方法(广度优先/深度优先)：

广度优先：逐层遍历，从左至右依次访问，可利用FIFO队列实现广度优先搜索；

深度优先：先访问根节点，然后遍历左子树后遍历右子树，可利用栈FILO的特点，将右子树压栈后将左子树压栈；

可以使用堆(单数组加上堆的最末节点的下标)来表示完全二叉树；

二叉搜索树上的基本操作花费的时间与树的高度成正比，完全二叉树操作的时间复杂度为O(log(n))，二叉搜索树的性质：对二叉搜索树中的节点x，若y为其左子树中的节点，y.key<=x.key；若y为其右子树中的节点，y.key>=x.key；

* 二叉树遍历
* 二叉树查找
* 二叉树插入与删除

中序遍历的递归写法：

void InorderTreeWalk(pTree\* tree){

if(NULL != tree){

InorderTreeWalk(tree->pLeft);

StoreTree(tree->key);

InorderTreeWalk(tree->pRight);

}

}

先序遍历的递归写法：

void PreorderTreeWalk(pTree\* tree){

if(NULL != tree){

StoreTree(tree->key);

PreorderTreeWalk(tree->pLeft);

PreorderTreeWalk(tree->pRight);

}

}

后序遍历的递归写法：

void PostorderTreeWalk(pTree\* tree){

if(NULL != tree){

PostorderTreeWalk(tree->pLeft);

PostorderTreeWalk(tree->pRight);

StoreTree(tree->key);

}

}

先序遍历的非递归方法：先使根节点tree入栈s.push(tree)，只要栈不为空!s.empty()，即可使栈中元素出栈s.pop()，每次弹出一个节点，都要将其右孩子节点tree->pLeft入栈s.push(tree->pLeft)，再将其左孩子节点tree->pRight入栈s.push(tree->pRight)；

访问tree->key后，将tree入栈，遍历tree->pLeft，遍历左子树结束后，栈顶元素为tree，tree出栈，遍历右子树；

**void PreorderNonrecursive(BiTree T){**

**if(! T){**

**return ;**

**}**

**stack<BiTree> s; // STL**

**s.push(T); // push root node**

**while(! s.empty()){**

**BiTree temp = s.top(); // top element in stack**

**StoreNode(temp->data);**

**s.pop(); // pop root**

**if(NULL != temp->rchild){ // right child push**

**s.push(temp->rchild);**

**}**

**if(NULL != temp->lchild){ // left child push**

**s.push(temp->lchild);**

**}**

**}**

**}**

中序遍历的非递归方法：

将tree入栈，遍历左子树tree->pLeft，栈顶元素为tree，tree出栈，访问tree->key，遍历右子树tree->pRight；

引入指向当前节点的指针curr，以curr指向非NULL或栈非空为外循环条件，先将左子树全部入栈；若栈非空，则将curr指向栈顶元素，并将其出栈，访问该节点，将curr指针指向当前节点的右孩子节点；

**void InorderNonrecursive(BiTree T){**

**if(! T){**

**return ;**

**}**

**BiTree curr = T; //point to current node**

**stack<BiTree> s;**

**while(NULL != curr || ! s.empty()){**

**//左子树全部入栈**

**while(NULL != curr){**

**s.push(curr);**

**curr = curr->lchild;**

**}//while**

**if(! s.empty()){**

**curr = s.top(); // point to top element in stack**

**s.pop(); // pop element out**

**StoreNode(curr->data); // store element**

**curr = curr->rchild; // point to right child**

**}**

**}//while**

**}**

后序遍历的非递归方法：

将tree入栈，遍历左子树tree->pLeft，栈顶元素为tree，将tree出栈，遍历右子树tree->pRight；

当前节点指针curr，节点已访问标记previsited，以curr指针非NULL或栈非空为外循环条件，将左子树全部入栈；curr指向栈顶元素，若curr节点的右孩子节点为NULL或curr节点的右孩子节点已被访问：

T：访问当前节点，并标记当前节点的previsited标记，curr节点出栈，并置curr为NULL；

F：否则访问curr的右孩子节点；

**void PostorderNonrecursive(BiTree T){**

**stack<BiTree> s;**

**BiTree curr = T; // point to current node**

**BiTree previsited = NULL;**

**while(NULL != curr || ! s.empty()){**

**while(NULL != curr){**

**s.push(curr); // 左子树全部入栈**

**curr = curr->lchild;**

**} //while**

**curr = s.top(); //左子树叶子节点**

**/\* 右子树为空或已被访问过，则访问当前节点并标记当前节点已被访问**

**\* 当前节点出栈，curr指针指空；**

**\*/**

**if(NULL == curr->rchild || previstited = curr->rchild){**

**StoreNode(curr->data); //store current node**

**previsted = curr; //flag pointer**

**s.pop(); //current node pop out**

**curr = NULL; //current pointer to NULL**

**} //if**

**else{**

**curr = curr->rchild; //访问右子树中的左子树**

**} //else**

**} //while**

**}**

后续遍历的非递归遍历，双栈法：

**void PostOrderDoubleStack(BiTree T){**

**stack<BiTree> s1, s2;**

**BiTree curr; //指向当前节点**

**s1.push(T);**

**while(! s1.empty()){**

**curr = s1.top();**

**s1.pop();**

**s2.push(curr);**

**if(curr->lchild != NULL){**

**s1.push(curr->lchild);**

**}**

**if(curr->rchild != NULL){**

**s1.push(curr->rchild);**

**}**

**}//while**

**while(! s2.empty()){**

**StoreNode(s2.top()->data);**

**s2.pop();**

**}**

**}**

二叉树的先序遍历，中序遍历和后序遍历均是DFS深度优先，易于使用递归方法实现；层序遍历是BFS广度优先，易于使用队列实现非递归方法：

int visit(BiTree T){

if(T){

StoreNode(T->data);

return 1;

}

else

return 0;

}

使用队列(FIFO)实现二叉树的层序遍历具有结构优势：

**void LevelOrderWalk(BiTree T){**

**queue<BiTree> Q;**

**BiTree p; //pointer to BiTree**

**p = T;**

**if(1 == visit(p)){**

**Q.push(p);**

**}**

**while(! Q.empty()){**

**p = Q.front();**

**Q.pop();**

**if(1 == visit(p->lchild)){**

**Q.push(p->lchild);**

**}**

**if(1 == visit(p->rchild)){**

**Q.push(p->rchild);**

**}**

**}**

**}**

二叉树求深度操作：

int BiTreeDepth(BiTNode \*T){

if(! T){

return 0;

}

int d1, d2;

d1 = BiTreeDepth(T->lchild);

d2 = BiTreeDepth(T->rchild);

return (d1>d2? d1:d2) + 1;

}

二叉树求节点数操作：

int BiTreeNodeCount(BiTreeNode \*T){

if(NULL == T){

return 0;

}

return (1 + BiTreeNodeCount(T->lchild) + BiTreeNodeCount(T->rchild));

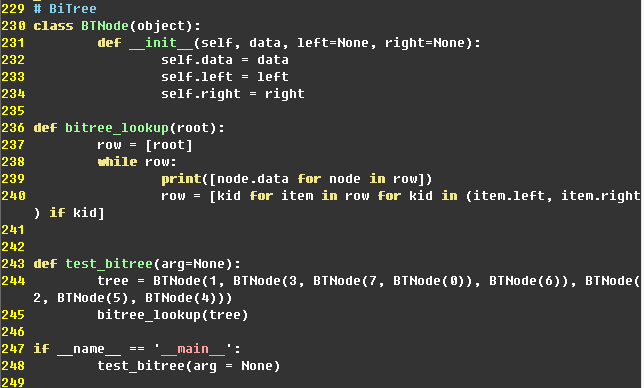
}

## 二叉树的C++实现

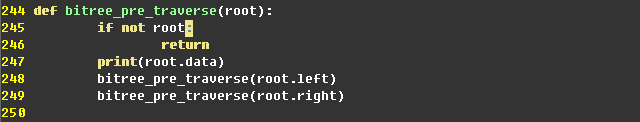
以C++类实现二叉树的创建，先序、中序和后续遍历，见[1.6 类与对象](#_1.6_类与对象)；

## 二叉树Python实现

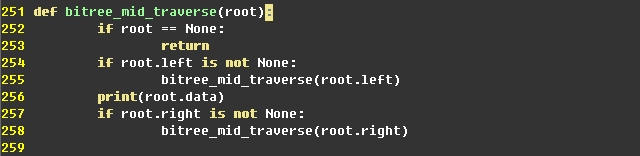
二叉树的构建与层序遍历：



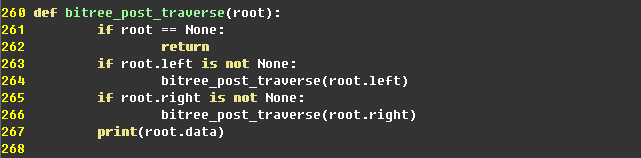
深度遍历-先序：



深度遍历-中序：



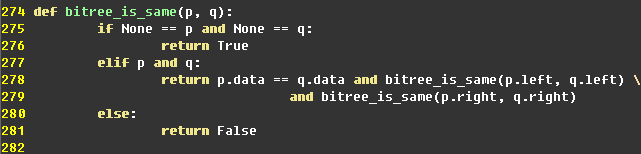
深度遍历-后序：



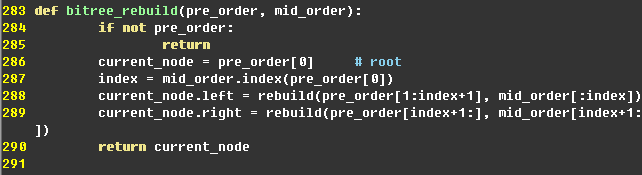
获得树的深度：



比较两棵树是否相同：



已知先序遍历列表和中序遍历列表，求后序遍历列表：



## 红黑树

## B树

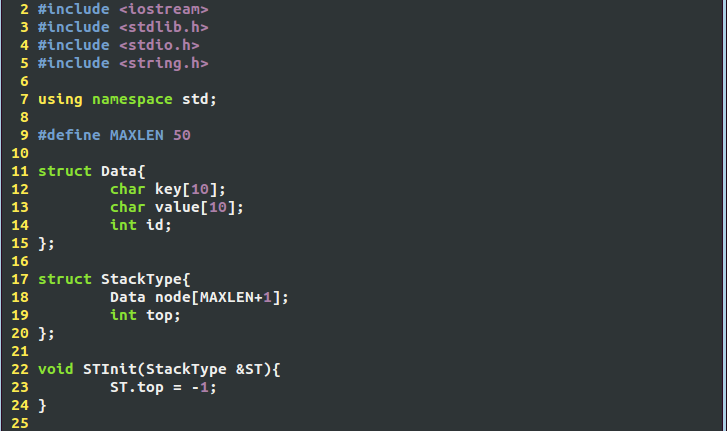
## 最大堆

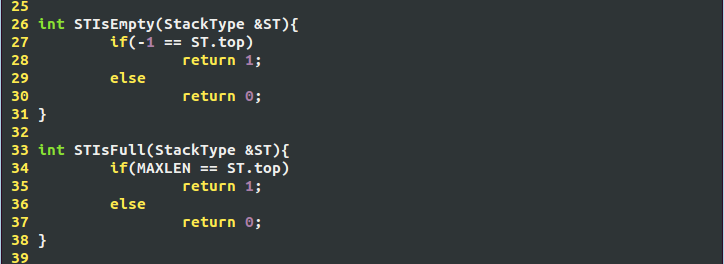
最大堆是一种完全二叉树，其父节点的关键字不小于其左右子节点的关键字；

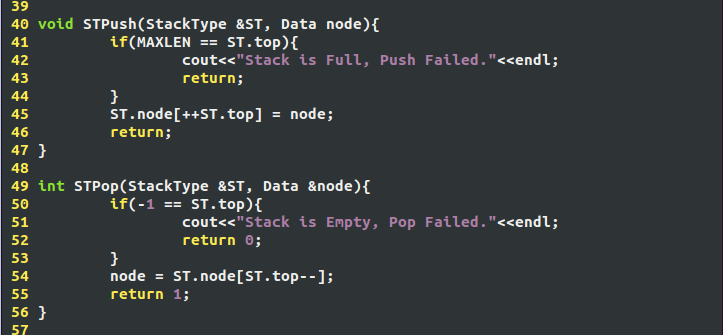
## 线性结构-栈与队列

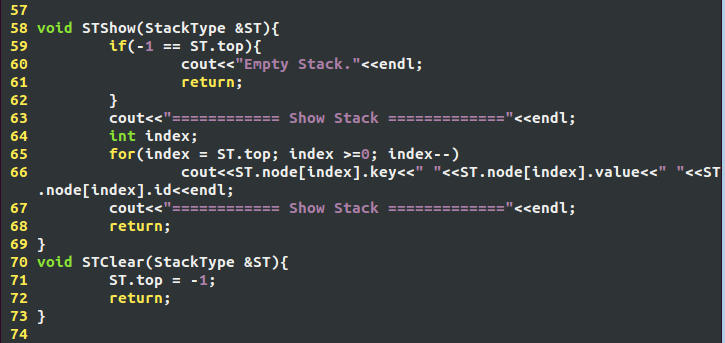
栈与队列都是动态集合，栈是***LIFO***结构，只有栈顶指针指示当前所在位置，入栈操作***PUSH***和出栈操作***POP***只能在栈顶进行，栈有上溢出和下溢出错误；队列是***FIFO***结构，分别有队头指针和队尾指针，入队操作***ENQUEUE***在队尾进行，出队操作***DEQUEUE***在队头；

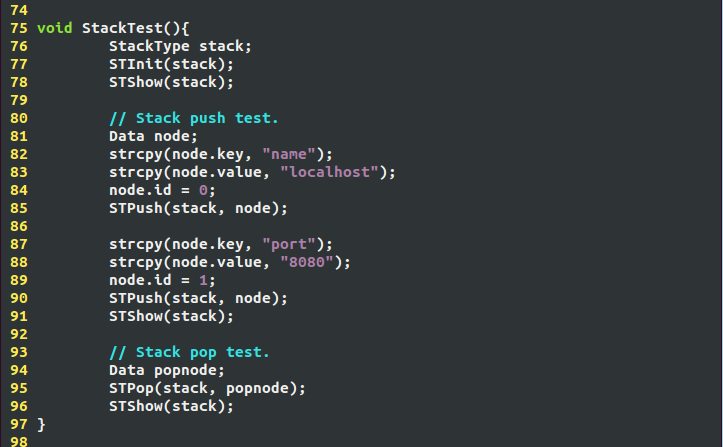
**C++(未使用Class)**实现栈：



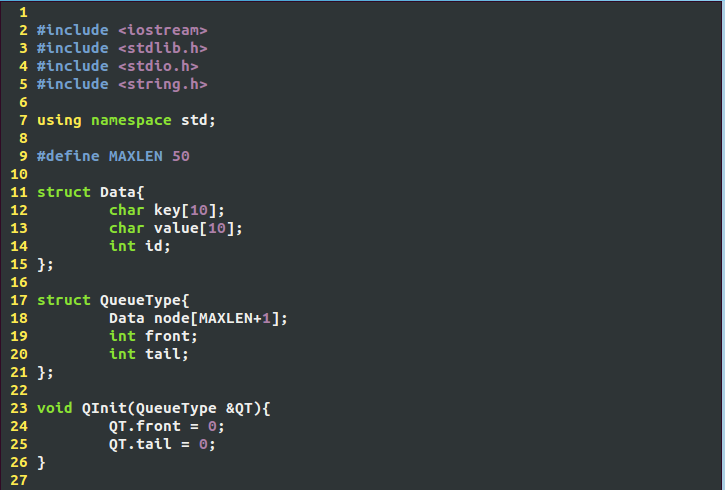


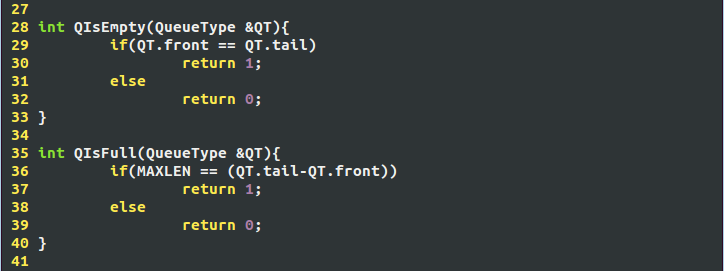


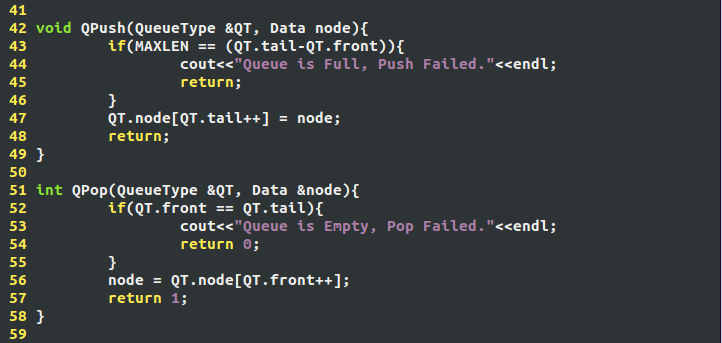


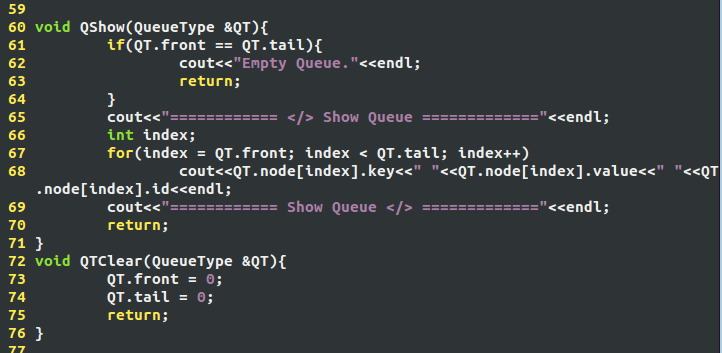


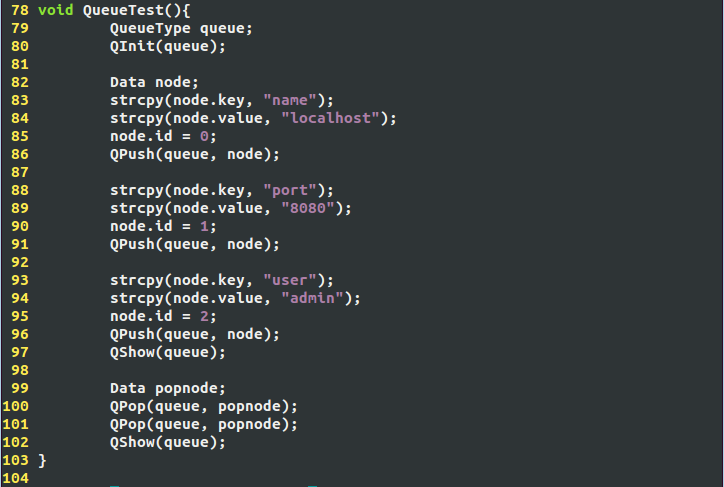
**C++(未使用Class)**实现队列：





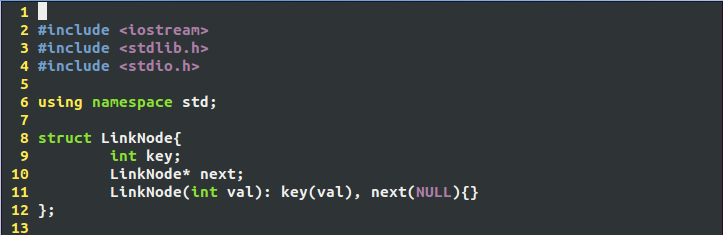




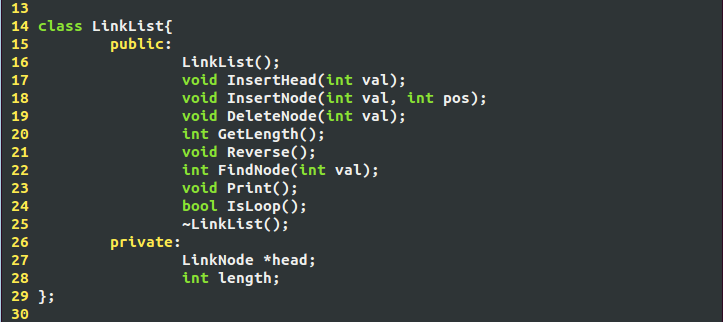


## 线性结构-链表

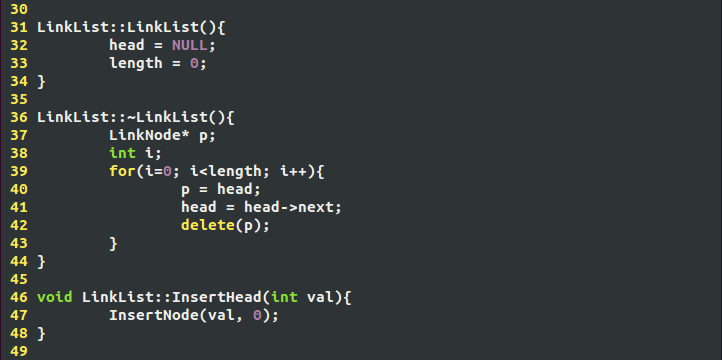
使用**C++ Class**实现一个链表类：

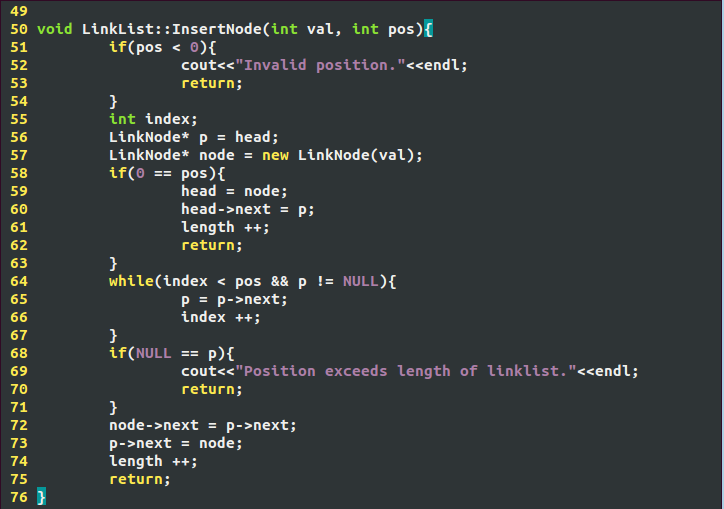


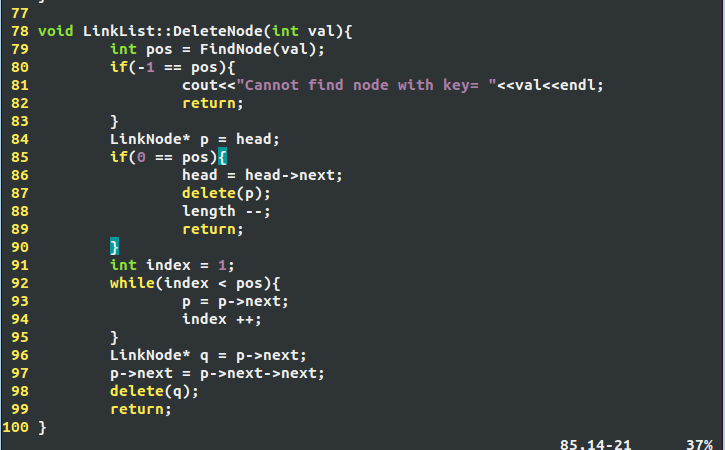
**参数列表实现构造函数**；

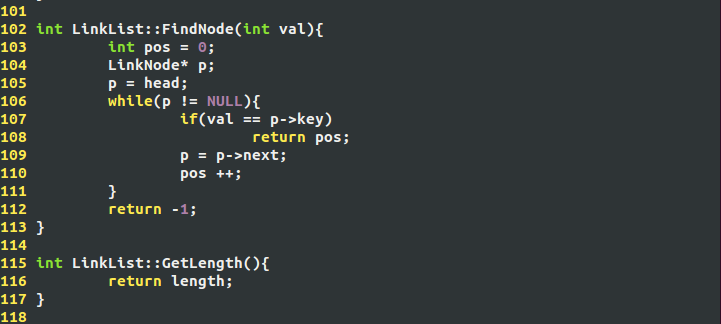


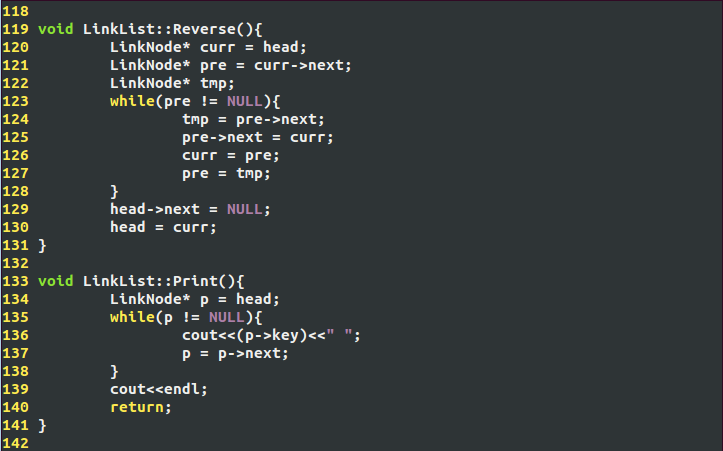
**构造函数与析构函数没有返回值**；



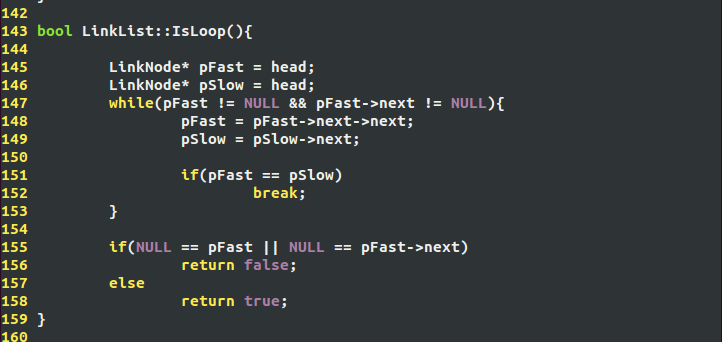


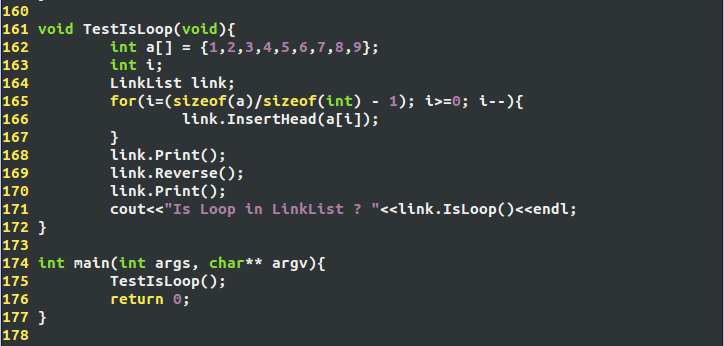






判断是否有环：





寻找链表中倒数第k个节点，定义两个指针pAhead和pBehind，使pAhead先走k-1步，然后两个指针一起移动，当pAhead移动到链表尾部时，pBehind所在位置即倒数第k个节点；容易写出以下代码：

ListNode\* FindKthToTail(ListNode\* pListHead, unsigned int k){

Unsigned int i;

if(NULL == pListHead){

return nullptr;

}

ListNode\* pAhead = pListHead;

ListNode\* pBehind = nullptr;

for(i=0; i<k-1; i++){

pAhead = pAhead->next;

}

pBehind = pListHead;

while(nullptr != pAhead->pNext){

pAhead = pAhead->pNext;

pBehind = pBehind->pNext;

}

return pBehind;

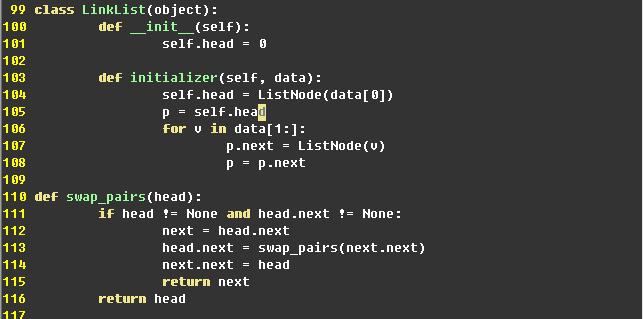
}

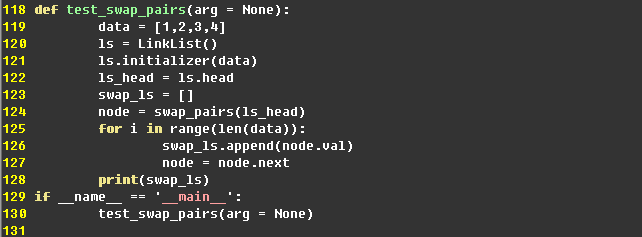
需要考虑的问题：

* 输入参数为空指针，即pListHead为nullptr；
* 输入参数k非法，如k为0；
* 链表节点总数小于k；

## 链表元素成对交换

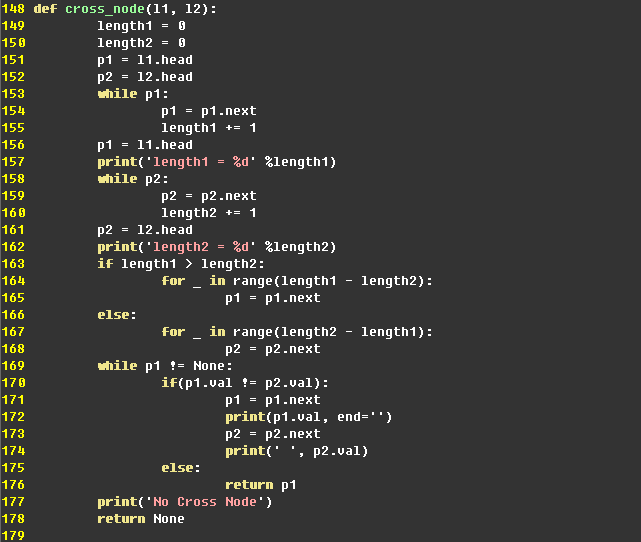
给定链表1🡪2🡪3🡪4，元素成对交换，输出2🡪1🡪4🡪3；

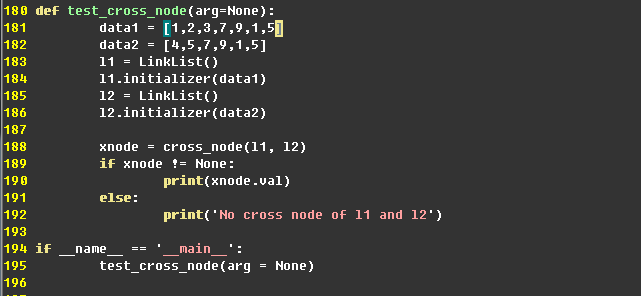




## 交叉链表求相交点

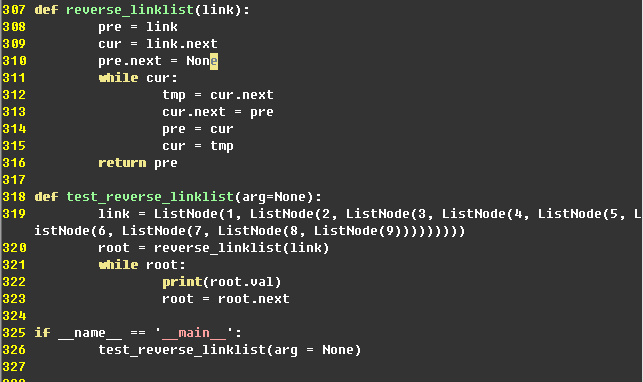
从尾部开始遍历，每个节点都相同，直至出现不同的节点，即为分叉处：





没有判断链表中是否存在环；

## 单向链表逆置

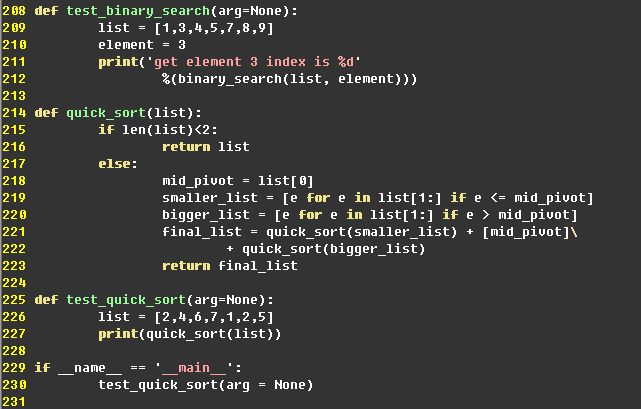


## 排序算法

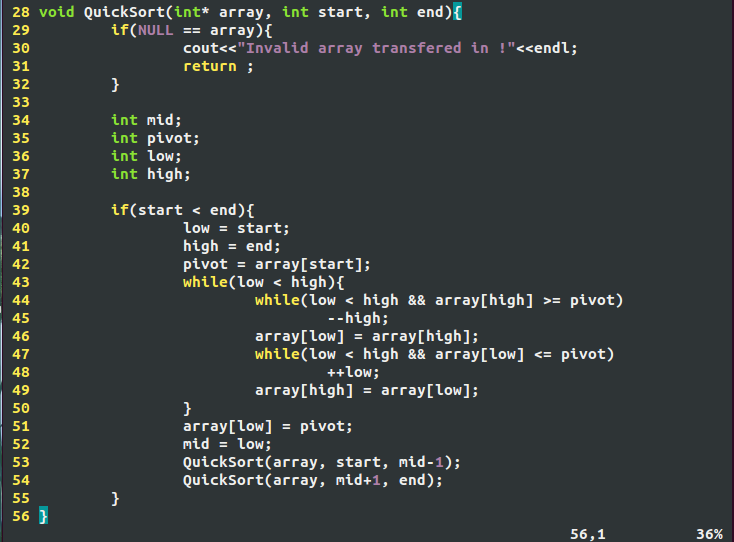
### 快速排序

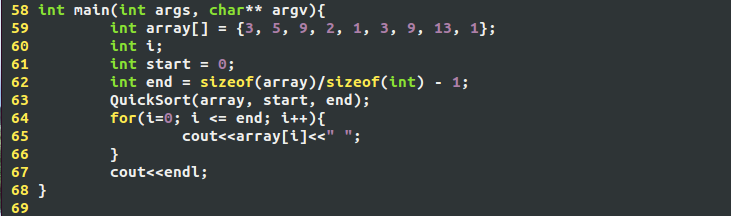
复杂度

最坏情况如何优化



**C++的递归方式实现**：





### 合并排序

复杂度

最坏情况如何优化

### 堆排序

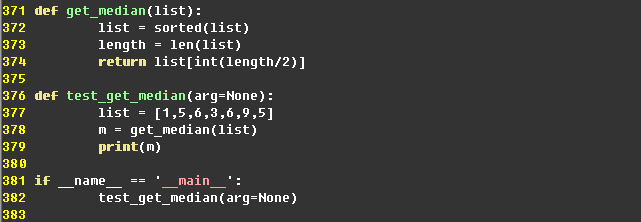
复杂度

最坏情况如何优化

### 单向链表应用快速排序

### 无序列表求其中位数

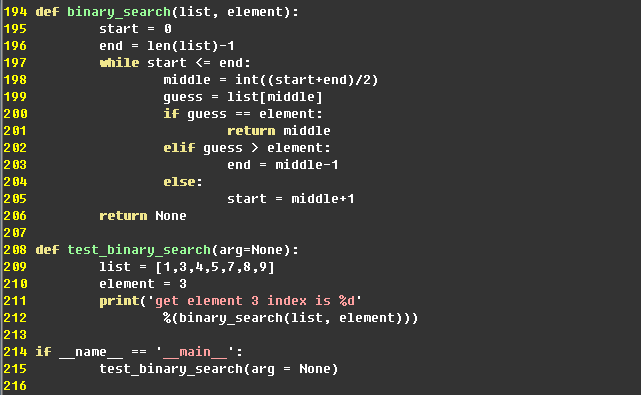
先对列表进行排序(可以选择快速排序)，然后求其中间数：



**使用C++实现排序**：**见**[**快速排序**](#_快速排序)；

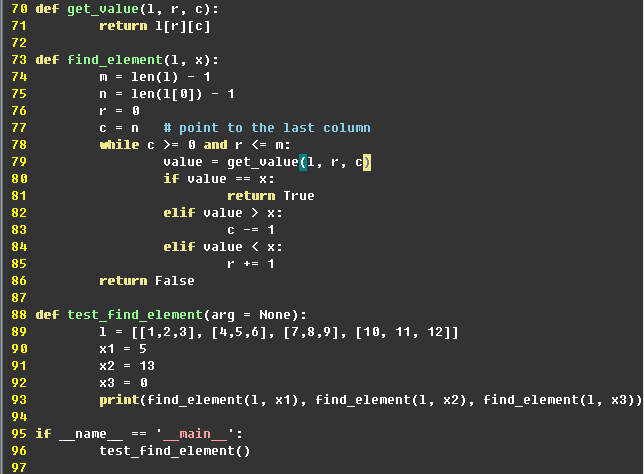
## 查找算法

### 二分查找

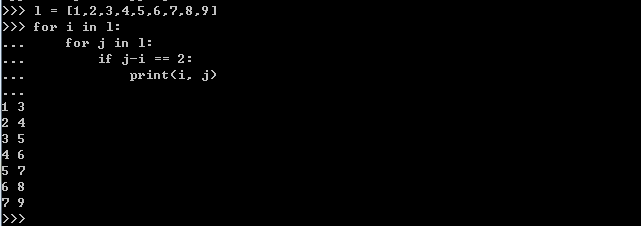


### 矩形查找问题

在一个m行n列二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该元素：

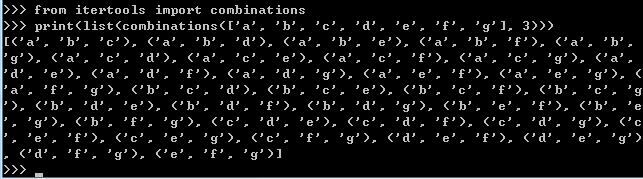


## 无序数字列表寻找所有间隔为d的组合



题干意图不明确，不知是否要求差值为定值的组合，如果是，是否还有复杂度更小(<O(n\*\*2))的算法；

## 列表[a1, a2, a3, …, an]求其所有组合

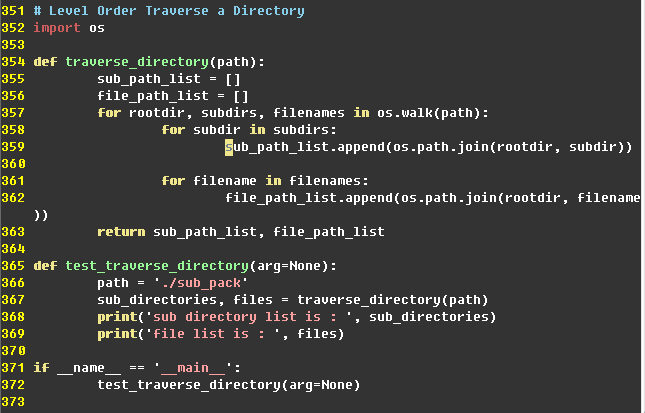


## 一行python代码实现2\*\*1000各位数求和



## 遍历目录

遍历指定目录，输出其所有子目录以及所有文件：



## DP问题

### 上台阶/矩形覆盖问题

* 青蛙上台阶，一次可以跳1级台阶也可以跳2级台阶，求上n层台阶共有多少种跳法：

**fib = lambda n: 2 if n<=2 else fib(n-1)+fib(n-2)**

* 可以使用2\*1的小矩形横向或者纵向无重叠覆盖一个2\*n的大矩形，求问总共有多少种覆盖方法(与上台阶问题相同)：

**f = lambda n: 1 if n<2 else f(n-1) + f(n-2)**

### 变态台阶问题

青蛙上台阶，一次可以跳1级台阶，2级台阶，…，n级台阶，求上n层台阶共有多少种跳法：

**n = 1 : fib(1) = 1 ;**

**n = 2 : fib(2) = fib(1) + 1 ;**

**n = 3 : fib(3) = fib(2) + fib(1) + 1 ;**

**…**

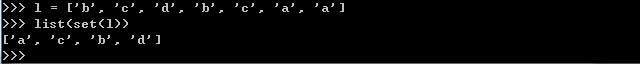
**n = n : fib(n) = fib(n-1) + fib(n-2) + … + fib(1) + 1**

**fib = lambda n : 1 if n<2 else 2\*fib(n-1)**

## 去除列表中的重复元素

给定列表，使用多种方法去除其中的重复元素：

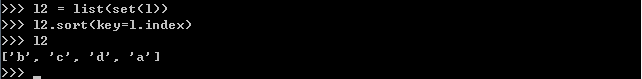
* 使用集合：



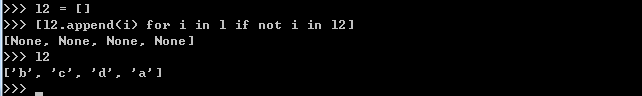
* 使用字典：



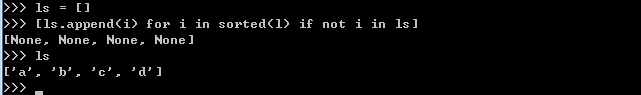
保持元素在原列表中的顺序：



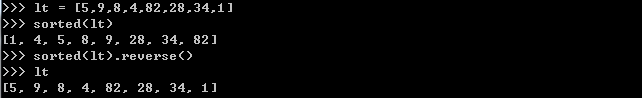
* 使用列表推导式：



带排序的列表推导式：



**列表排序：**



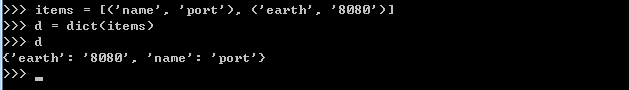


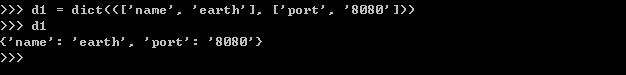
## 创建字典

* **手动创建**：

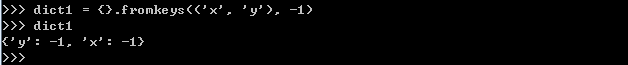


* **工厂方法**：

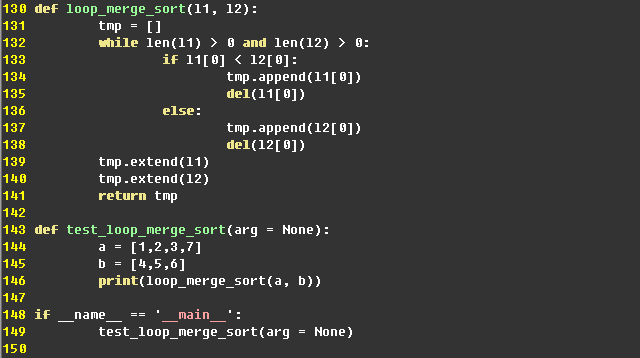




* **fromkeys()方法：**

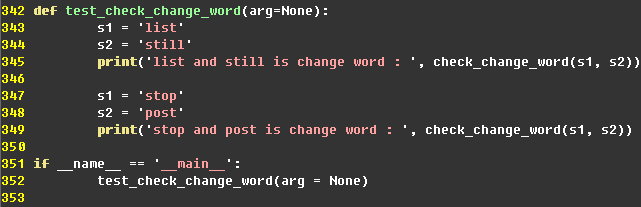


## 合并有序列表



## 变位词

检查两个字符串是否互为变位词，形如”stop”与”post”，是字符互换位置的结果；



# OS

## 多线程与多进程的区别

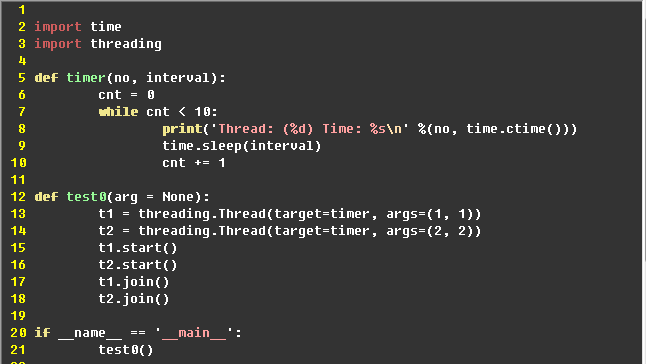
线程***(Threads)***与进程***(Process)***都是对***CPU***工作时间段的描述，进程的工作过程为：

***CPU***加载上下文 🡪 ***CPU***执行 🡪 ***CPU***保存上下文；

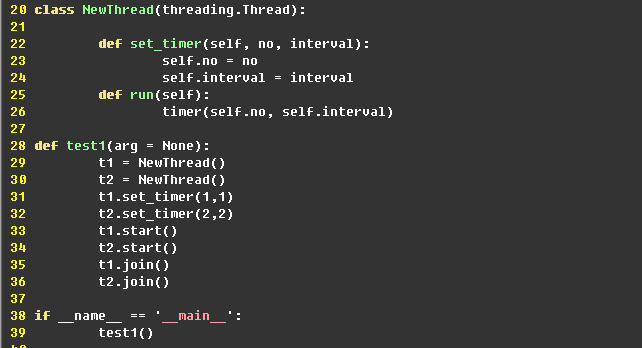
而线程是进程在***CPU***执行过程中划分的更加细小的时间段，线程间是共享上下文的；

***Python***实现多线程有两种方法：

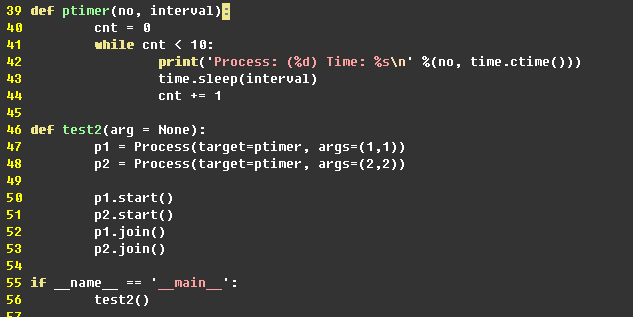
* 导入包**threading**直接在模块**threading.Thread**中运行线程体函数；



* 通过重载类**threading.Thread**的**run()**方法即可实现多线程，类似于**java**中多线程的实现；



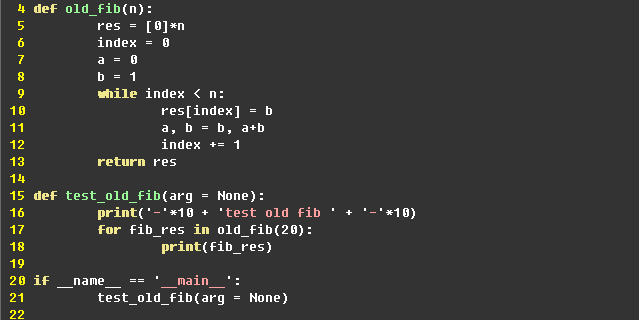
**Python**开辟进程通过导入包**multiprocessing**中的**Process**模块实现：



## 协程

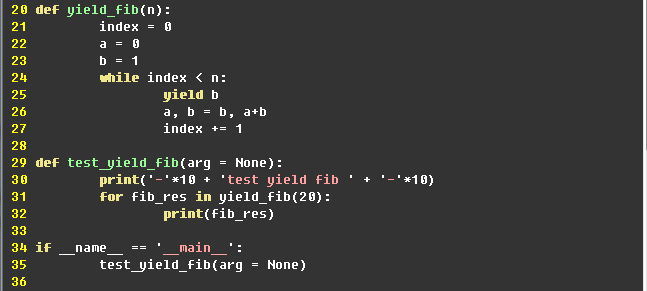
协程是为了解决受到***GIL***限制的多线程切换带来的消耗问题：

* 传统的生成斐波那契级数的方式：

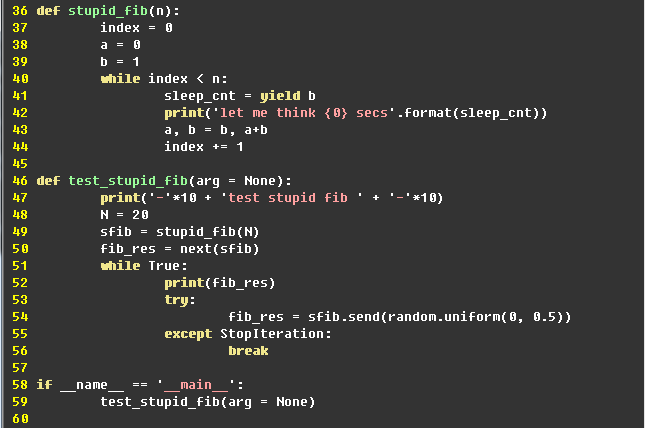


如果仅仅需要级数的第***n***项，则这种方式比较浪费内存；于是引入了**yield生成器**；

* **函数体内包含yield**语句则会变为一个生成器***(generator)***，主调函数中的yield\_fib(20)并没有直接调用函数体，而是生成一个生成器对象实例，相当于每次循环都会调用**next(fib(20))**，**yield**语句可以保存函数执行的现场，暂停计算并将计算结果返回，下一次调用next()时会唤醒生成器，直至生成器实例抛出**StopIteration**异常，被**for**循环捕获退出；

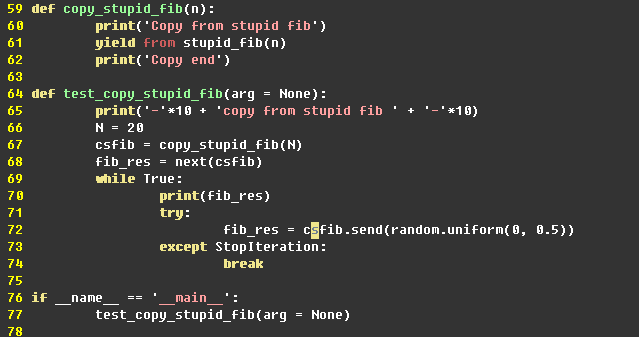


* **yield表达式可以使得生成器的数据向外传送，send()**方法可以实现向生成器内部传送数据：



主调函数可以通过**sfib.send()**函数向生成器内部传送数据，控制**stupid\_fib()**的“思考时间”，而生成器将计算结果返回给主调函数，这就实现了控制协程计算的时间，其中，第一个**next(sfib)**相当于**sfib.send(None)**，使生成器返回第一个数值，并停在**yield 1**处；

* **yield from**表达式可以重构生成器，并通过调用send()函数像管道(pipe)机制一样往内层协程中传递信息；

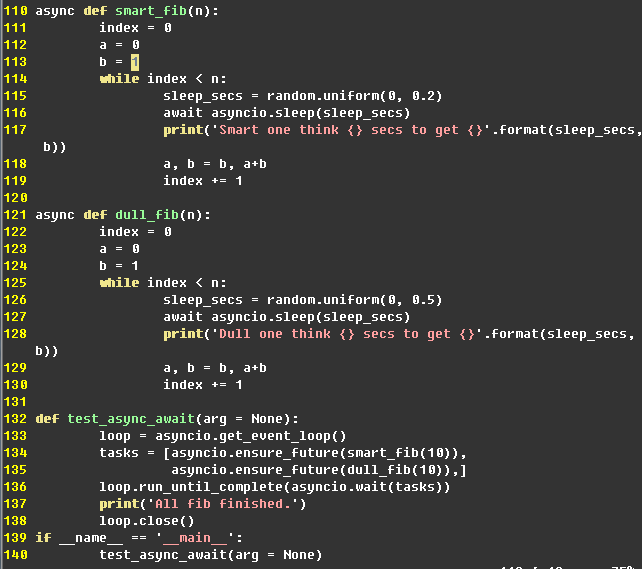


* **asyncio**是一个基于事件循环的实现异步**I/O**的模块。通过**yield from**，可以将协程**asyncio.sleep**的控制权交给事件循环，然后挂起当前协程，之后由事件循环决定何时唤醒**asyncio.sleep**，接着向后执行代码；



协程**yield from asyncio.sleep**时，事件循环其实是与Future对象建立了联系，每次事件循环调用**send(None)**时，其实都会传递到**Future**对象的**\_\_iter\_\_**函数调用；而当**Future**尚未执行完毕的时候，就会**yield self**，也就意味着暂时挂起，等待下一次**send(None)**的唤醒；

* **在python3.5中，async/await**是**asyncio.coroutine/yield from**的替代品，**async/await**让协程表面上独立于生成器而存在，将细节都隐藏于**asyncio**模块之下，语法更清晰明了：



## 进程间通信方式

* **管道(Pipe)**：管道可用于具有亲缘关系进程间的通信，允许一个进程和另一个与它有共同祖先的进程之间进行通信；
* **命名管道(named pipe)**：命名管道克服了管道没有名字的限制，因此，除具有管道所具有的功能外，它还允许无亲缘关系进程间的通信，命名管道在文件系统中有对应的文件名，命名管道通过命令**mkfifo**或系统调用**mkfifo**来创建；
* **信号(Signal)**：信号是比较复杂的通信方式，用于通知接受进程有某种事件发生，除了用于进程间通信外，进程还可以发送信号给进程本身；**linux**除了支持**Unix**早期信号语义函数**signal**外，还支持语义符合**Posix.1**标准的信号函数**sigaction**，实际上，该函数是基于**BSD**的，**BSD**为了实现可靠信号机制，又能够统一对外接口，用**sigaction**函数重新实现了**signal**函数；
* **消息(Message)队列**：消息队列是消息的链接表，包括**Posix**消息队列**system** **V**消息队列。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。消息队列克服了信号承载信息量少，管道只能承载无格式字节流以及缓冲区大小受限等缺点；
* **共享内存**：使得多个进程可以访问同一块内存空间，是最快的可用**IPC**形式。是针对其他通信机制运行效率较低而设计的。往往与其它通信机制，如信号量结合使用，来达到进程间的同步及互斥；
* **内存映射(mapped memory)**：内存映射允许任何多个进程间通信，每一个使用该机制的进程通过把一个共享的文件映射到自己的进程地址空间来实现它；
* **信号量(semaphore)**：主要作为进程间以及同一进程不同线程之间的同步手段；
* **套接口(Socket)**：更为一般的进程间通信机制，可用于不同机器之间的进程间通信。起初是由**Unix**系统的**BSD**分支开发出来的，但现在一般可以移植到其它类**Unix**系统上：**Linux**和**System V**的变种都支持套接字；

## 虚拟存储系统中缺页计算

虚拟存储系统中，若某一进程在内存中占三块，采用***FIFO***页面淘汰算法，当依次访问页面1、2、3、4、1、2、5、1、2、3、4、5、6时，将会产生多少次缺页中断：

访问页面： 缺页计数： 内存内容：

1. **缺页** 1
2. **缺页** 1,2
3. **缺页** 1,2,3
4. **缺页** 2,3,4

1 **缺页** 3,4,1

2 **缺页** 4,1,2

5 **缺页** 1,2,5

1 不缺页 2,5,1

2 不缺页 5,1,2

3 **缺页** 1,2,3

4 **缺页** 2,3,4

5 **缺页** 3,4,5

6 **缺页** 4,5,6

## 并发进程不会引起死锁的资源数量计算

* 产生死锁的原因是竞争资源和程序推进顺序不当；
* 互斥条件：

1. 请求和保持条件；
2. 不剥夺条件；
3. 环路等待条件；

* 处理死锁基本方法：

1. 预防死锁(摒弃除1以外的条件)；
2. 避免死锁(银行家算法)；
3. 检测死锁(资源分配图)；
4. 解除死锁：剥夺资源，撤销进程；

假设有5个某种资源，由4个进程共享，则每个进程最多申请多少个资源可以避免死锁：

* 当每个进程最多只能申请1个资源时，4个进程共需要4个资源，还有1个结余，此时不会发生死锁；
* 当每个进程最多可申请2个资源时，资源分配可以达到1+1+1+2的状态，总会有进程释放资源，不会形成环路等待，不会发生死锁；
* 当每个进程可以申请3个资源时，若资源分配为1+1+1+2等情况，可能会发生死锁；

## 常用的Linux/git/vim命令和作用

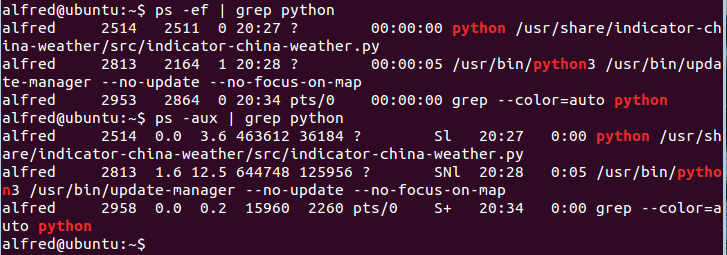
### Linux常用命令

### git常用命令

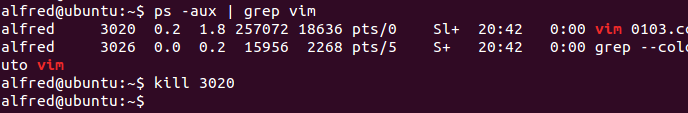
### vim常用命令

## 查看当前进程的命令

**ps** 命令可用于查看当前系统中运行的进程，配合**grep**命令使用，查看**CMD**为**python**的进程，-**aux**是显示详细的状态信息：



结束进程使用**kill PID**命令：



## 任务调度算法

根据系统的资源分配策略所规定的的资源分配算法；

常用的任务调度算法：

* 先来先服务***(First Come First Serve)FCFS***：有利于长作业，不利于短作业，利于CPU密集型作业，不利于IO密集型作业；
* 短作业优先***(Shortest Job First)SJF***：作业长短只能估算，利于短作业，不利于长作业，无法保证紧迫任务被及时处理；
* 最高优先权调度***(Priority Scheduling)***：常用于批处理系统中，又可分为抢占式/非抢占式调度；
* 时间片轮转***(Round Robin)RR***：调度时，将CPU分配给队列头保存的进程，令其执行一个时间片，计时器发出时钟中断请求时，此进程被停止，并送往就绪队列尾部，依次循环；
* 多级反馈队列调度***(Multilevel Feedback Queue Scheduling)***：

(1) 设置多个就绪队列，并为各个队列赋予不同的优先级。在优先权越高的队列中， 为每个进程所规定的执行时间片就越小。

(2) 当一个新进程进入内存后，首先放入第一队列的末尾，按***FCFS***原则排队等候调度。 如果他能在一个时间片中完成，便可撤离；如果未完成，就转入第二队列的末尾，在同样等待调度…… 如此下去，当一个长作业(进程)从第一队列依次将到第***n***队列（最后队列）后，便按第***n***队列时间片轮转运行。

(3) 仅当第一队列空闲时，调度程序才调度第二队列中的进程运行；

仅当第***1***到第 ***i-1***队列空时， 才会调度第***i***队列中的进程运行，并执行相应的时间片轮转。

(4) 如果处理机正在处理第***i***队列中某进程，又有新进程进入优先权较高的队列，则此新队列抢占正在运行的处理机，并把正在运行的进程放在第***i***队列的队尾。

# 网络

## TCP/IP协议

## OSI五层协议

## Socket长连接

## Select与epoll

Select/epoll/pollselect都是IO多路复用的机制。**I/O多路复用就通过一种机制，可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。**但select，poll，epoll本质上都是同步I/O，因为他们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的，而异步I/O则无需自己负责进行读写，异步I/O的实现会负责把数据从内核拷贝到用户空间。

**select**有3个缺点：

* 连接数受限
* 查找配对速度慢
* 数据由内核拷贝到用户态

**poll改善了第一个缺点；**

**epoll改了三个缺点；**

## TCP与UDP协议的区别

## TIME\_WAIT过多的原因

## http一次连接的全过程描述

从用户发起request到用户接收到response

## http连接方式——get与post的区别

## restful

## http请求的状态码 200/403/404/504

# 数据库

## MySQL锁的种类

## 死锁的产生

## MySQL的char/varchar/text的区别

## Join的种类与区别

## A LEFT JOIN B的查询结果中，B缺少的部分如何显示

## 索引类型的种类

## BTree索引与hash索引的区别

## 如何对查询命令进行优化

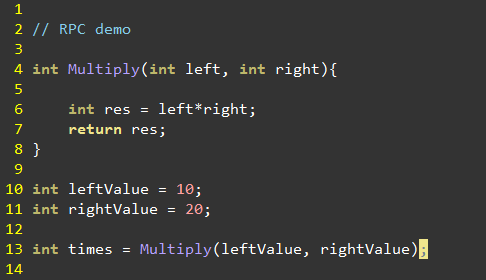
## NoSQL与关系型数据库的区别

## Redis常用的存储类型

# AI架构知识

## 分布式RPC

本地过程调用RPC就是要像调用本地的函数一样去调远程函数；假设要调用函数Multiply来计算leftValue \* rightValue的结果:



调用函数Multiply实际上执行了以下操作：将 leftValue 和 rightValue 的值压栈进入Multiply函数，取出栈中的值10 和 20，将其赋予 left 和 right执行第6行代码，计算 left \* right ，并将结果存在 res将 res 的值压栈，然后从Multiply返回第13行，从栈中取出返回值 200 ，并赋值给 times，以上就是执行本地调用的过程；

远程过程调用带来的新问题在远程调用时，需要执行的函数体是在远程的机器上的，也就是说，Multiply是在另一个进程中执行的，因此就带来了几个新问题：

* Call ID映射，告诉远程机器要调用Multiply，而不是其他函数Add或者FooBar，在本地调用中，函数体是直接通过函数指针来指定的，此过程由编译器完成；但是在远程调用中，函数指针是不行的，因为两个进程的地址空间是完全不一样的；所以，在RPC中，所有的函数都必须有自己的一个ID；这个ID在所有进程中都是唯一确定的；客户端在做远程过程调用时，必须附上这个ID。然后还需要**在客户端和服务端分别维护一个** ***{函数 <--> Call ID}*** 的对应表，**两者的表不一定需要完全相同，但相同的函数对应的Call ID必须相同；当客户端需要进行远程调用时，查表找出相应的Call ID，然后把它传给服务端，服务端也通过查表，来确定客户端需要调用的函数，然后执行相应函数的代码**；
* 序列化和反序列化：客户端怎么把参数值传给远程的函数呢？在本地调用中，只需要把参数压到栈里，然后让函数自己去栈里读就行。但是在远程过程调用时，客户端跟服务端是不同的进程，**不能通过内存来传递参数**。甚至有时候客户端和服务端使用的都不是同一种语言（比如服务端用C++，客户端用Java或者Python）。这时候就**需要客户端把参数先转成一个字节流，传给服务端后，再把字节流转成自己能读取的格式**，例如Tensorflow所使用的数据结构ProtoBuff文件就是一种典型的序列化数据，这个过程叫序列化和反序列化；同理，从服务端返回的值也需要序列化反序列化的过程；
* 网络传输：远程调用往往用在网络上，客户端和服务端是通过网络连接的，所有的数据都需要通过网络传输，因此就需要有一个**网络传输层**。网络传输层需要把**Call ID和序列化后的参数字节流传给服务端，然后再把序列化后的调用结果传回客户端。只要能完成这两者的，都可以作为传输层使用**；因此，它所使用的协议其实是不限的，能完成传输就行。尽管大部分RPC框架都使用TCP协议，但其实UDP也可以，而gRPC干脆就用了HTTP2，**Java的Netty**也属于这层的东西；

所以，要实现一个RPC框架，其实只需要把以上三点实现了就基本完成了：Call ID映射可以直接使用函数字符串，也可以使用整数ID，映射表一般就是一个哈希表；序列化反序列化可以自己写，也可以使用***Protobuf***或者***FlatBuffers***之类的；网络传输库可以自己写***socket***，或者用***asio***，***ZeroMQ***，***Netty***之类；

## Docker

基于**Docker**的神经网络服务器：常用于对服务的封装，供给业务系统使用；

Docker作为应用容器引擎，可用于将应用及其依赖包打包到一个可移植的容器中，进而完成在Linux各种发行版上的发布；

* 选取某个发行版的Docker镜像，如ubuntu 16.0.4，生成Docker容器，然后将tesseract-ocr字符识别服务安装到Docker容器中，部署一个flask服务，暴露服务端口，提供给其他业务系统使用；
* 考虑到Docker容器中的flask服务需要自启动，以及服务可能由于程序异常而造成停止服务的情况，因此，需要在容器中添加自启动程序以及服务监听程序。自启动以及监听程序可以利用linux的定时器crontab命令，添加一条定时任务（每一分钟检测一次），检测程序是否正常运行，当服务不正常时，重启flask服务。在将该容器的服务移植到其他linux服务器（宿主机）后，考虑到服务器可能断电，或者宕机的情况，也需要在**宿主机上利用crontab添加一条定时任务**，监测Docker容器是否正常运行，如果运行不正常，需要重启该Docker容器；
* 考虑到该容器移植的便利性，**利用Docker的commit和save命令，将该容器提交成新的镜像，并将该镜像保存为tar包**，部署该OCR应用容器时，只需要**将该tar包拷贝到宿主机上，利用Docker的load以及run命令，启动该容器以及自启动容器内的OCR识别服务**；

基于Docker的OCR识别系统的整个部署流程的关键操作步骤如下：

# 提交容器变化到镜像中

***docker commit 【containerId】 ocr\_images:ocr\_api***

# 保存镜像到tar包中

***docker save ocr\_images:ocr\_api > ocr\_images.tar***

# 加载OCR服务镜像

***docker load < ocr\_images.tar***

# 启动OCR服务容器

***docker run -t -d -p 0.0.0.0:8000:8000 --name ocr\_api ocr\_images:ocr\_api***

## Kubernetes

### 基于Kubernetes的分布式神经网络训练/推理框架

深度学习模型常用的两种分布式训练方式：同步更新和异步更新；同步更新模式下，所有服务器统一读取参数值，计算梯度，反向传播，更新参数，可能会出现计算快速的服务器等待计算较慢的服务器的情况，资源利用率较低；异步更新模式下，不同的服务器独自读取参数，计算梯度，反向传播并进行参数更新，可能会出现使用较旧的梯度更新参数的情况，对训练的效果产生影响；

分布式训练神经网络需要两种服务器：参数服务器和计算服务器，参数服务器用于保存神经网络的参数取值，计算服务器用于计算参数的梯度；

Tensorflow启动分布式深度学习模型训练也有两种模式：In-Graph Replication和Between-Graph Replication；In-Graph Replication会将所有参数保存到一个计算图中，在做计算时，将计算任务分配到不同的计算服务器；Between-Graph Replication模式会在所有的计算服务器创建参数，**但参数会以统一的方式分配**，适用于海量数据的处理；

Tensorflow在此处提供MapReduce计算框架，Kubernetes可提供类似Yarn的调度系统和HDFS的存储系统，以及任务调度、监控、失败重启等功能；

### Kubernetes容器集群管理框架

***Kubernetes***作为Docker生态圈中重要一员，是Google多年大规模容器管理技术的开源版本，是产线实践经验的最佳表现；

* ***Kubernetes***：

Kubernetes是Google**开源的容器集群管理系统**，其提供应用部署、维护、 扩展机制等功能，利用Kubernetes能方便地管理跨机器运行容器化的应用，其主要功能如下：

1) 使用Docker对应用程序包装(package)、实例化(instantiate)、运行(run)；

2) 以集群的方式运行、管理跨机器的容器；

3) 解决Docker跨机器容器之间的通讯问题；

4) Kubernetes的**自我修复机制**使得容器集群总是运行在用户期望的状态；

当前Kubernetes支持GCE、vShpere、CoreOS、OpenShift、Azure等平台，除此之外，也可以直接运行在物理机上；

* ***Pods*：**

**Pod是Kubernetes的基本操作单元**，把相关的一个或多个容器构成一个Pod，通常Pod里的容器运行相同的应用。Pod包含的容器运行在同一个Minion(Host)上，看作一个统一管理单元，共享相同的volumes和network namespace/IP和Port空间；

* ***Services***：

Services也是Kubernetes的基本操作单元，是真实应用服务的抽象，每一个服务后面都有很多对应的容器来支持，通过Proxy的port和服务selector决定服务请求传递给后端提供服务的容器，对外表现为一个单一访问接口，外部不需要了解后端如何运行，这给扩展或维护后端带来很大的好处；

* ***Replication Controllers***：

Replication Controller确保任何时候Kubernetes集群中有指定数量的pod副本(replicas)在运行， 如果少于指定数量的pod副本(replicas)，Replication Controller会启动新的Container，反之会杀死多余的以保证数量不变。Replication Controller使用预先定义的pod模板创建pods，一旦创建成功，pod 模板和创建的pods没有任何关联，可以修改pod 模板而不会对已创建pods有任何影响，也可以直接更新通过Replication Controller创建的pods。对于利用pod 模板创建的pods，Replication Controller根据label selector来关联，通过修改pods的label可以删除对应的pods，Replication Controller主要有如下用法：

1) **Rescheduling**

如上所述，Replication Controller会确保Kubernetes集群中指定的pod副本(replicas)在运行， 即使在节点出错时；

2) **Scaling**

通过修改Replication Controller的副本(replicas)数量来水平扩展或者缩小运行的pods；

3) **Rolling updates**

Replication Controller的设计原则使得可以一个一个地替换pods来rolling updates服务；

4) **Multiple release tracks**

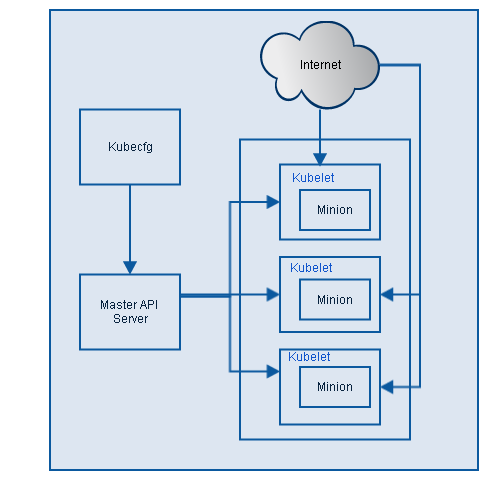
如果需要在系统中运行multiple release的服务，Replication Controller使用labels来区分multiple release tracks；

* ***Labels***：

Labels是用于区分Pod、Service、Replication Controller的key/value键值对，Pod、Service、 Replication Controller可以有多个label，但是每个label的key只能对应一个value。Labels是Service和Replication Controller运行的基础，为了将访问Service的请求转发给后端提供服务的多个容器，正是通过标识容器的labels来选择正确的容器。同样，Replication Controller也使用labels来管理通过pod 模板创建的一组容器，这样Replication Controller可以更加容易，方便地管理多个容器，无论有多少容器；

### Kubernetes构件

Kubenetes整体框架如下图，主要包括kubecfg、Master API Server、Kubelet、Minion(Host)以及Proxy；



**图7-1 Kubernetes High Level构件**

* ***Master*：**

Master定义了Kubernetes 集群Master/API Server的主要声明，包括Pod Registry、Controller Registry、Service Registry、Endpoint Registry、Minion Registry、Binding Registry、RESTStorage以及Client, 是client(Kubecfg)调用Kubernetes API，管理Kubernetes主要构件Pods、Services、Minions、容器的入口。Master由API Server、Scheduler以及Registry等组成。从下图可知Master的工作流主要分以下步骤：

1) Kubecfg将特定的请求，比如创建Pod，发送给Kubernetes Client；

2) Kubernetes Client将请求发送给API server；

3) API Server根据请求的类型，比如创建Pod时storage类型是pods，然后依此选择何种REST Storage API对请求作出处理；

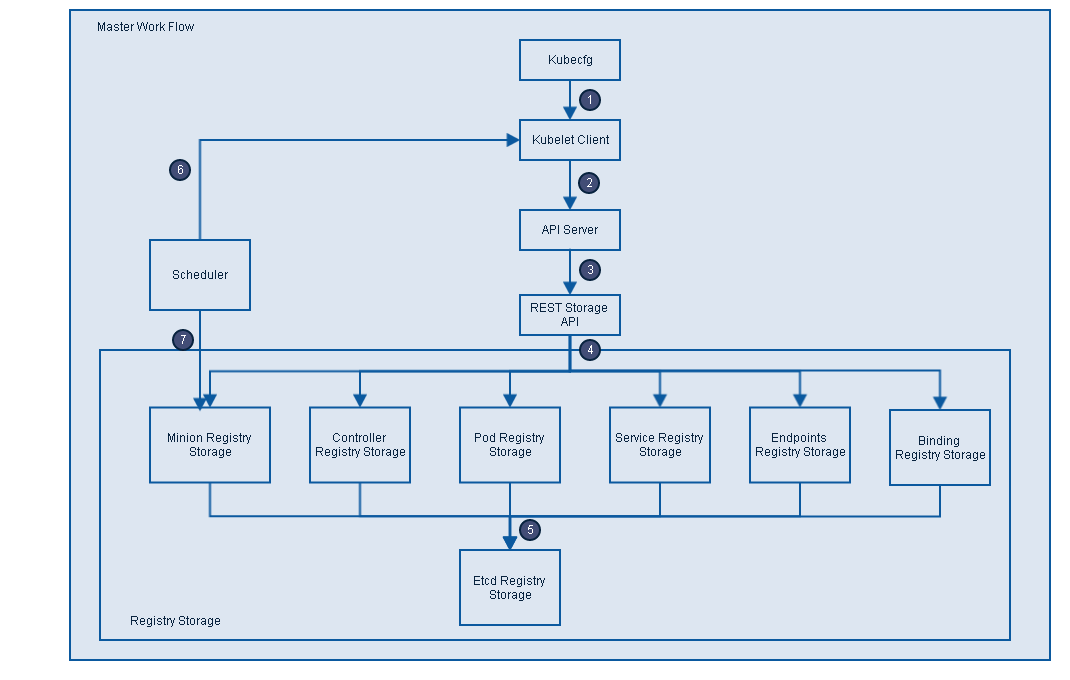
4) REST Storage API对的请求作相应的处理；

5) 将处理的结果存入高可用键值存储系统Etcd中；

6) 在API Server响应Kubecfg的请求后，Scheduler会根据Kubernetes Client获取集群中运行Pod及Minion信息；

7) 依据从Kubernetes Client获取的信息，Scheduler将未分发的Pod分发到可用的Minion节点上；

下面是Master的主要构件的详细介绍：



**图7-2 Master主要构件及工作流**

1. Minion Registry

Minion Registry负责跟踪Kubernetes 集群中有多少Minion(Host)。Kubernetes封装Minion Registry成实现Kubernetes API Server的RESTful API接口REST，通过这些API，我们可以对Minion Registry做Create、Get、List、Delete操作，由于Minon只能被创建或删除，所以不支持Update操作，并把Minion的相关配置信息存储到etcd。除此之外，Scheduler算法根据Minion的资源容量来确定是否将新建Pod分发到该Minion节点；

1. Pod Registry

Pod Registry负责跟踪Kubernetes集群中有多少Pod在运行，以及这些Pod跟Minion是如何的映射关系。将Pod Registry和Cloud Provider信息及其他相关信息封装成实现Kubernetes API Server的RESTful API接口REST。通过这些API，我们可以对Pod进行Create、Get、List、Update、Delete操作，并将Pod的信息存储到etcd中，而且可以通过Watch接口监视Pod的变化情况，比如一个Pod被新建、删除或者更新；

1. Service Registry

Service Registry负责跟踪Kubernetes集群中运行的所有服务。根据提供的Cloud Provider及Minion Registry信息把Service Registry封装成实现Kubernetes API Server需要的RESTful API接口REST。利用这些接口，我们可以对Service进行Create、Get、List、Update、Delete操作，以及监视Service变化情况的watch操作，并把Service信息存储到etcd；

1. Controller Registry

Controller Registry负责跟踪Kubernetes集群中所有的Replication Controller，Replication Controller维护着指定数量的pod 副本(replicas)拷贝，如果其中的一个容器死掉，Replication Controller会自动启动一个新的容器，如果死掉的容器恢复，其会杀死多出的容器以保证指定的拷贝不变。通过封装Controller Registry为实现Kubernetes API Server的RESTful API接口REST， 利用这些接口，我们可以对Replication Controller进行Create、Get、List、Update、Delete操作，以及监视Replication Controller变化情况的watch操作，并把Replication Controller信息存储到etcd；

1. Endpoints Registry

Endpoints Registry负责收集Service的endpoint，比如Name："mysql"，Endpoints: ["10.10.1.1:1909"，"10.10.2.2:8834"]，同Pod Registry，Controller Registry也实现了Kubernetes API Server的RESTful API接口，可以做Create、Get、List、Update、Delete以及watch操作；

1. Binding Registry

Binding包括一个需要绑定Pod的ID和Pod被绑定的Host，Scheduler写Binding Registry后，需绑定的Pod被绑定到一个host。Binding Registry也实现了Kubernetes API Server的RESTful API接口，但Binding Registry是一个write-only对象，所有只有Create操作可以使用， 否则会引起错误；

1. Scheduler

Scheduler收集和分析当前Kubernetes集群中所有Minion节点的资源(内存、CPU)负载情况，然后依此分发新建的Pod到Kubernetes集群中可用的节点。由于一旦Minion节点的资源被分配给Pod，那这些资源就不能再分配给其他Pod， 除非这些Pod被删除或者退出， 因此，Kubernetes需要分析集群中所有Minion的资源使用情况，保证分发的工作负载不会超出当前该Minion节点的可用资源范围。具体来说，Scheduler做以下工作：

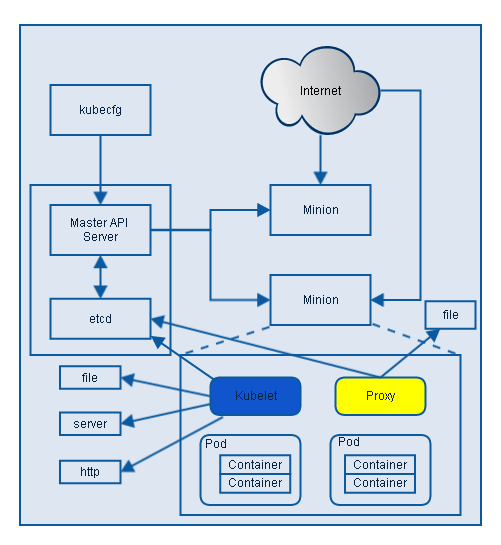
7.1) 实时监测Kubernetes集群中未分发的Pod；

7.2) 实时监测Kubernetes集群中所有运行的Pod，Scheduler需要根据这些Pod的资源状况安全地将未分发的Pod分发到指定的Minion节点上；

7.3) Scheduler也监测Minion节点信息，由于会频繁查找Minion节点，Scheduler会缓存一份最新的信息在本地；

7.4) 最后，Scheduler在分发Pod到指定的Minion节点后，会把Pod相关的信息Binding写回API Server。

* ***Kubelet***



**图7-3 Kubernetes详细构件**

根据上图，可知Kubelet是Kubernetes集群中每个Minion和Master API Server的连接点，Kubelet运行在每个Minion上，是Master API Server和Minion之间的桥梁，接收Master API Server分配给它的commands和work，与持久性键值存储etcd、file、server和http进行交互，读取配置信息。Kubelet的主要工作是管理Pod和容器的生命周期，其包括Docker Client、Root Directory、Pod Workers、Etcd Client、Cadvisor Client以及Health Checker组件，具体工作如下：

1) 通过Worker给Pod异步运行特定的Action；

2) 设置容器的环境变量；

3) 给容器绑定Volume；

4) 给容器绑定Port；

5) 根据指定的Pod运行一个单一容器；

6) 杀死容器；

7) 给指定的Pod创建network 容器；

8) 删除Pod的所有容器；

9) 同步Pod的状态；

10) 从Cadvisor获取container info、 pod info、root info、machine info；

11) 检测Pod的容器健康状态信息；

12) 在容器中运行命令；

* ***Proxy***

Proxy是为了解决外部网络能够访问跨机器集群中容器提供的应用服务而设计的，从上图3-3可知Proxy服务也运行在每个Minion上。Proxy提供TCP/UDP sockets的proxy，每创建一种Service，Proxy主要从etcd获取Services和Endpoints的配置信息，或者也可以从file获取，然后根据配置信息在Minion上启动一个Proxy的进程并监听相应的服务端口，当外部请求发生时，Proxy会根据Load Balancer将请求分发到后端正确的容器处理；

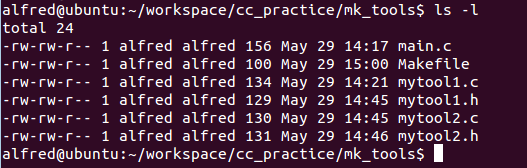
## 开发工具与脚本语言

由源文件生成.o或.obj文件的过程为编译，由.o或.obj文件生成可执行文件的过程为链接；编译时，编译器需要的是语法的正确，函数与变量的声明的正确。对于后者，通常是你需要告诉编译器头文件的所在位置（头文件中应该只是声明，而定义应该放在C/C++文件中），只要所有的语法正确，编译器就可以编译出中间目标文件。

　　链接时，主要是链接函数和全局变量，可以使用这些中间目标文件(.o文件或.obj文件）链接应用程序；链接器并不管函数所在的源文件，只管函数的中间目标文件（Object File），在大多数时候，由于源文件太多，编译生成的中间目标文件太多，而在链接时需要明显地指出中间目标文件名，这对于编译很不方便，所以，需要给中间目标文件打个包，在Windows下这种包叫“库文件”（Library File)，也就是 .lib 文件，在UNIX下，是Archive File，也就是.a 文件；

**Makefile, CMake, Bazel, bash**

* **Makefile：**

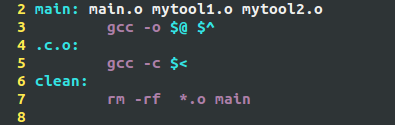


make 在当前目录下找 "Makefile"或"makefile"的文件；

如果找到，则会找文件中第一个目标文件（target）。如上例子中main；

如果 main 命令的执行,依赖后面命令执行所产生的文件，则先执行后面命令；

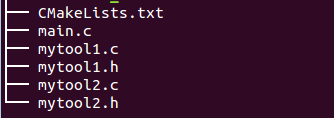
当main 命令需要的文件生成完毕，则执行main 命令 ；

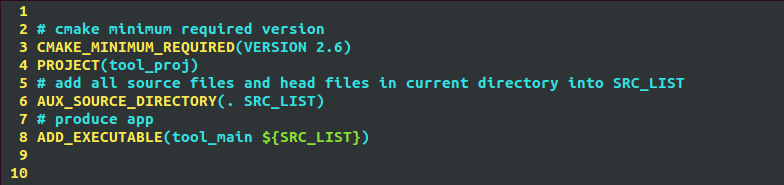


* **CMake：**

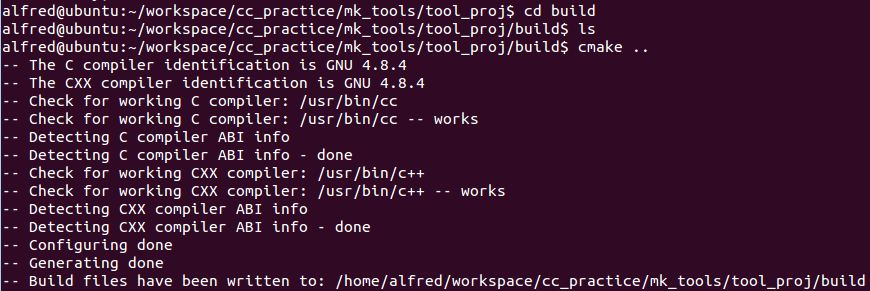
CMake是Makefile的上层工具，可用于项目代码的构建和维护，其目的是产生可移植的makefile并简化makefile的编写工作；

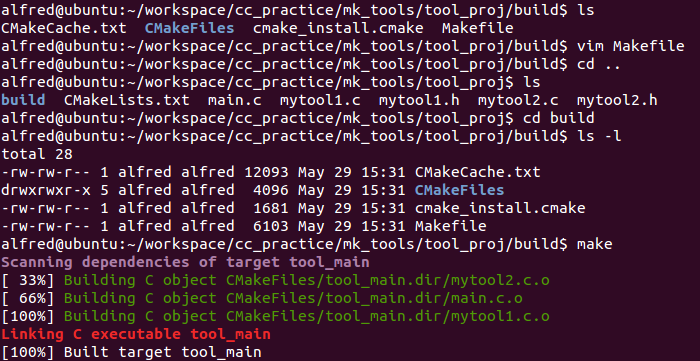
1. 在Project根目录下新建CMakeLists.txt文件：



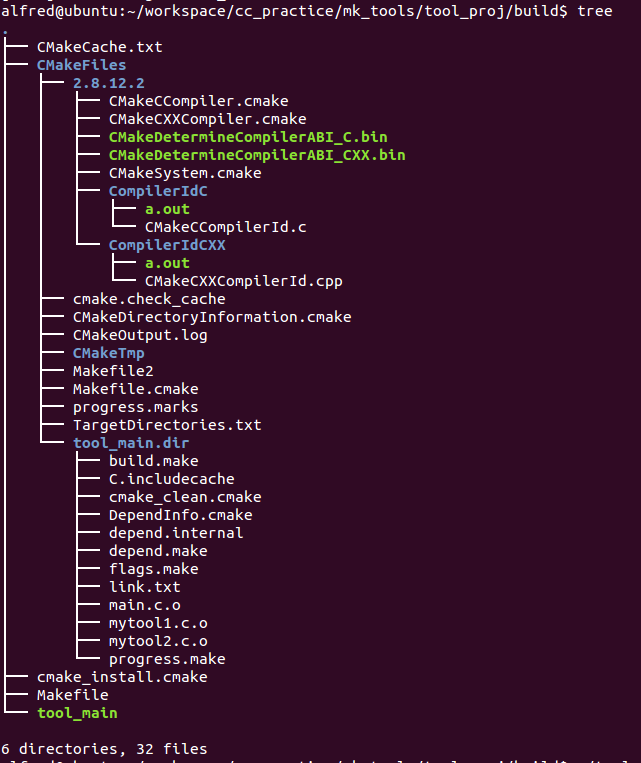


1. 在Project根目录下新建目录build用于放置编译中间生成的文件；
2. 进入build目录下，运行cmake .. ，生成Makefile；





1. 此时，Project目录结构为：此时，需要删除编译产生的中间文件，需要在build目录下运行make clean：



* **Bazel：**
* **Bash:**

Bash Shell是一个命令解释器，介于Linux Kernel与用户态之间，可以执行系统调用和系统命令，实现用户与操作系统内核之间的通信，通常大部分Linux系统预设的shell为bash；

## GPU驱动与CUDA库

CPU长于处理复杂计算步骤和复杂数据依赖的计算任务(分布式计算，数据压缩，人工智能，物理模拟等)，而GPU是将相同的指令流水线发送到众核上，并采用不同的输入数据执行；

CUDA(Compute Unified Device Architecture)，基于新的并行编程模型和指令集架构的通用计算架构，

## SDK开发

**Compiler, Debugger, Profiler**

**NPU驱动及Tensorflow后端集成；**

## 性能优化

**分析与优化分布式神经网络框架的性能；**

**测试分布式神经网络框架的方法和工具；**

## NPU模拟器

**分布式神经网络框架模拟器的设计与实现；**