目录

[1. C/C++ 3](#_Toc507099866)

[1.1 const关键字 3](#_Toc507099867)

[1.2函数传值与传引用 8](#_Toc507099868)

[1.3 字符串转整型问题 9](#_Toc507099869)

[1.4 操作符重载 9](#_Toc507099870)

[2. Python 10](#_Toc507099871)

[2.1 可变类型与非可变类型 10](#_Toc507099872)

[2.2 静态方法，实例方法与类方法 10](#_Toc507099873)

[2.3 类变量与实例变量 11](#_Toc507099874)

[2.4 Python自省特性 13](#_Toc507099875)

[2.5 列表推导式与字典推导式 13](#_Toc507099876)

[2.6 深拷贝与浅拷贝 14](#_Toc507099877)

[2.7 类方法\_\_new\_\_( )与\_\_init\_\_( ) 14](#_Toc507099878)

[2.8 字符串格式化:%与.format 14](#_Toc507099879)

[2.9 Python的\*args和\*\*kwargs 16](#_Toc507099880)

[2.10 常见的几种设计模式 16](#_Toc507099881)

[2.11 Python变量的作用域 17](#_Toc507099882)

[2.12 GIL线程全局锁及协程 18](#_Toc507099883)

[2.13 闭包 18](#_Toc507099884)

[2.14 Lambda函数 19](#_Toc507099885)

[2.15 编码与解码 19](#_Toc507099886)

[2.16 迭代器与生成器 19](#_Toc507099887)

[2.17 装饰器 20](#_Toc507099888)

[2.18 Python中的重载 21](#_Toc507099889)

[2.19 Python新式类与旧式类 22](#_Toc507099890)

[2.20 邮箱地址正则表达式 23](#_Toc507099891)

[2.21 垃圾回收 23](#_Toc507099892)

[3. Algorithm 23](#_Toc507099893)

[3.1 时间复杂度计算 23](#_Toc507099894)

[3.2 二叉树 24](#_Toc507099895)

[3.2.1 二叉树的数据结构 24](#_Toc507099896)

[3.2.2 二叉树的生成 24](#_Toc507099897)

[3.2.3 二叉树的遍历算法 25](#_Toc507099898)

[3.3 最大堆 29](#_Toc507099899)

[3.4 红黑树 29](#_Toc507099900)

[3.5 B树 30](#_Toc507099901)

[3.6 线性结构-栈与队列 30](#_Toc507099902)

[3.7 线性结构-链表 30](#_Toc507099903)

[3.8 寻找链表倒数第k个节点 30](#_Toc507099904)

[3.9 快速排序 31](#_Toc507099905)

[3.10 堆排序 31](#_Toc507099906)

[3.11 无序数字列表寻找所有间隔为d的组合 31](#_Toc507099907)

[3.12 列表[a1, a2, a3, …, an]求其所有组合 31](#_Toc507099908)

[3.13 一行python代码实现1+2+3+…+10\*\*8 31](#_Toc507099909)

[3.14 长度未知的单向链表求其是否有环 31](#_Toc507099910)

[3.15 单向链表应用快速排序 31](#_Toc507099911)

[3.16 长度为n的无序数字元素列表求其中位数 32](#_Toc507099912)

[3.17 遍历一个内部未知的文件夹 32](#_Toc507099913)

[4. OS 33](#_Toc507099914)

[4.1 多线程与多进程的区别 33](#_Toc507099915)

[4.2 协程 33](#_Toc507099916)

[4.3 进程间通信方式 33](#_Toc507099917)

[4.4 虚拟存储系统中缺页计算 33](#_Toc507099918)

[4.5 并发进程不会引起死锁的资源数量计算 33](#_Toc507099919)

[4.6 常用的Linux/git命令和作用 33](#_Toc507099920)

[4.7 查看当前进程的命令 33](#_Toc507099921)

[4.8 33](#_Toc507099922)

[5. 网络 34](#_Toc507099923)

[5.1 TCP/IP协议 34](#_Toc507099924)

[5.2 OSI五层协议 34](#_Toc507099925)

[5.3 Socket长连接 34](#_Toc507099926)

[5.4 Select与epoll 34](#_Toc507099927)

[5.5 TCP与UDP协议的区别 34](#_Toc507099928)

[5.6 TIME\_WAIT过多的原因 34](#_Toc507099929)

[5.7 http一次连接的全过程描述 34](#_Toc507099930)

[5.8 http连接方式——get与post的区别 34](#_Toc507099931)

[5.9 restful 35](#_Toc507099932)

[5.10 http请求的状态码 200/403/404/504 35](#_Toc507099933)

[6. 数据库 36](#_Toc507099934)

[6.1 MySQL锁的种类 36](#_Toc507099935)

[6.2 死锁的产生 36](#_Toc507099936)

[6.3 MySQL的char/varchar/text的区别 36](#_Toc507099937)

[6.4 Join的种类与区别 36](#_Toc507099938)

[6.5 A LEFT JOIN B的查询结果中，B缺少的部分如何显示 36](#_Toc507099939)

[6.6 索引类型的种类 36](#_Toc507099940)

[6.7 BTree索引与hash索引的区别 36](#_Toc507099941)

[6.8 如何对查询命令进行优化 36](#_Toc507099942)

[6.9 NoSQL与关系型数据库的区别 36](#_Toc507099943)

[6.10 Redis常用的存储类型 37](#_Toc507099944)

# 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；

## 1.2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；

## 1.3 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；

## 1.4 操作符重载

# 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]

## 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

## 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；

\_\_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；

## 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；

## 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。

## 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；

## Lambda函数

## 编码与解码

## 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；

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

## 垃圾回收

引用计数/分代回收/孤立引用环

# 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

}

}

## 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));

}

## 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；

## 红黑树

## B树

## 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；

## 线性结构-链表

## 寻找链表倒数第k个节点

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；

## 快速排序

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

## 无序数字列表寻找所有间隔为d的组合

## 列表[a1, a2, a3, …, an]求其所有组合

## 一行python代码实现1+2+3+…+10\*\*8

## 长度未知的单向链表求其是否有环

## 单向链表应用快速排序

## 长度为n的无序数字元素列表求其中位数

## 遍历一个内部未知的文件夹

# OS

## 多线程与多进程的区别

CPU密集型适合使用多线程还是多进程；

## 协程

## 进程间通信方式

## 虚拟存储系统中缺页计算

## 并发进程不会引起死锁的资源数量计算

## 常用的Linux/git命令和作用

## 查看当前进程的命令

## 

# 网络

## TCP/IP协议

## OSI五层协议

## Socket长连接

## Select与epoll

## 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