Hsingmin.lee@yahoo.com

Quick Start for ML

Machine Learning Appetizers

黎兴民

2018/3/24 Saturday

目录

1.C/C++

1.1 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 模板

```
#include <iostream>
#include <stdlib.h>
#include <stdio.h>
#include <string>

using namespace std;

template<class T>
class Stack{
        public:
                Stack();
                bool IsEmpty();
                bool IsFull();
                bool Push(const T & item);
                bool Pop(T & item);
                ~Stack();
        private:
                enum {MAX = 10};
                T items[MAX];
                int top;
};

template<class T>
Stack<T>::Stack(){
        top = 0;
}
```

```
28
29 template<class T>
30 Stack<T>::~Stack(){
31         top = -1;
32 }
33
34 template<class T>
35 bool Stack<T>::IsEmpty(){
36         return top == 0;
37 }
38
39 template<class T>
40 bool Stack<T>::IsFull(){
41         return top == MAX;
42 }
43
44 template<class T>
45 bool Stack<T>::Push(const T & item){
46         if(top < MAX){
47                 items[top++] = item;
48                 return true;
49         }
50         else
51                 return false;
52 }
                                                    25,18         21%
```

```
53
54 template<class T>
55 bool Stack<T>::Pop(T & item){
56         if(top > 0){
57                 item = items[--top];
58                 return true;
59         }
60         else
61                 return false;
62 }
63
64 void TestStack(){
65         Stack<int> stack;
66         cout<<"stack is empty : "<<stack.IsEmpty()<<endl;
67         int element[10] = {0,1,2,3,4,5,6,7,8,9};
68         for(int i=0; i<(sizeof(element)/sizeof(int)); i++){
69                 stack.Push(element[i]);
70         }
71         cout<<"stack is empty : "<<stack.IsEmpty()<<endl;
72
73         int popout = 0;
74         stack.Pop(popout);
75         cout<<"stack pop out : "<<popout<<endl;
76 }
77
```

```
119
120 int main(int args, char** argv){
121         TestStack();
122         return 0;
123 }
124
```

1.5 操作符重载

1.6 类与对象

2.Python

2.1 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]
```

```
122 def test8(num):
123         str = 'first'
124         for i in range(num):
125                 str += "x"
126         return str
127
128 if __name__ == '__main__':
129         print(test8(10))
```

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；

2.2 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)；

2.3 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']
```

2.4 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()

2.5 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；

```
57 def test3(arg=None):
58         ls = [1,2,3,4]
59         list1 = [i for i in ls if i>2]
60         print(list1)    # [3,4]
61
62         list2 = [i*2 for i in ls if i>2]
63         print(list2)    # [6,8]
64
65         dict1 = {x: x**2 for x in (2,4,6)}
66         print(dict1)    # {2: 4, 4: 16, 6: 36}
67
68         dict2 = {x: 'item' + str(x**2) for x in (2,4,6)}
69         print(dict2)    # {2: 'item4', 4: 'item16', 6: 'item36'}
70
71         set1 = {x for x in 'hello world' if x not in 'low level'}
72         print(set1)     # set[('h', 'r', 'd')]
73
74
75 if __name__ == '__main__':
76         test3(arg=None)
77
```

2.6 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
```

2.7 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：

```
NEW 5
B INIT
B func
NEW 20
INIT 20
A func
```

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；

```
35 class B(object):
36         def func(self):
37                 print('B func')
38         def __init__(self):
39                 print('B INIT')
40
41 class A(object):
42         def func(self):
43                 print('A func')
44         def __new__(cls, a):
45                 print('NEW', a)
46                 if a > 10:
47                         return super(A, cls).__new__(cls)
48                 return B()
49         def __init__(self, a):
50                 print('INIT', a)
51
52 def test2(arg=None):
53         a1 = A(5)       # NEW 5 , B INIT
54         a1.func()       # B func
55         a2 = A(20)      # NEW 20 , INIT 20
56         a2.func()       # A func
57
58 if __name__ == '__main__':
59         test2(arg=None)
```

2.8 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

2.9 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
```

2.10 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()
```

2.11 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：

```
num = 9
def func1():
        num = 20

def func2():
        print(num)

def test4(arg=None):
        func2()         # 9
        func1()
        func2()         # 9

if __name__ == '__main__':
        test4(arg=None)

```

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；

```
num = 9
def func1():
        global num
        num = 20

def func2():
        print(num)

def test4(arg=None):
        func2()         # 9
        func1()
        func2()         # 20

if __name__ == '__main__':
        test4(arg=None)

```

2.12 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；

2.13 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；

```
113 def closure(num):
114         def inner(value):
115                 return num * value
116         return inner
117
118 def test7(arg=None):
119         z = closure(7)
120         print(z(9))
121
122 if __name__ == '__main__':
123         test7(arg=None)
124
```

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；

2.14 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

2.15 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()
```

2.16 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；

2.17 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；

2.18 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 );
```

2.19 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”。

2.20 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、runooooooob 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、runooooooob 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；

2.21 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)

2.22 Python 中的 is

“is”方法是对比对象的地址，”==”对比的是对象的值。

2.23 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)
```

2.24 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。

2.25 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：

```
1
2 class A(object):
3         def show(self):
4                 print('base class show')
5
6 class B(A):
7         def show(self):
8                 print('derived show')
9
10 def test(arg=None):
11         obj = B()
12         # call child class object method.
13         obj.show()      # derived class show
14         # object __class__ method point to parent class
15         obj.__class__ = A
16         obj.show()      # base class show
17         # object __class__ method point back to child class
18         obj.__class__ = B
19         obj.show()      # derived show
20
21
22 if __name__ == '__main__':
23         test(arg=None)
```

2.26 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***

```
21 class A(object):
22         def __init__(self, a, b):
23                 self.__a = a
24                 self.__b = b
25         def show(self):
26                 print('a = ', self.__a, 'b = ', self.__b)
27 def test1(arg=None):
28         a = A(10, 20)
29         a.show()
30         a(80)
31
32
33 if __name__ == '__main__':
34         test1(arg=None)
```

在类中直接实现***__call__***方法：

```
27         def __call__(self, num):
28                 print('call : ', self.__a + num)
29
```

2.27 一行代码交换参数值

```
87 def test5(arg=None):
88         a = 8
89         b = 9
90         print('before swap : a = ', a, 'b = ', b)       # 8 9
91         a, b = b, a
92         print('after swap : a = ', a, 'b = ', b)        # 9 8
93
94 if __name__ == '__main__':
95         test5(arg=None)
96
```

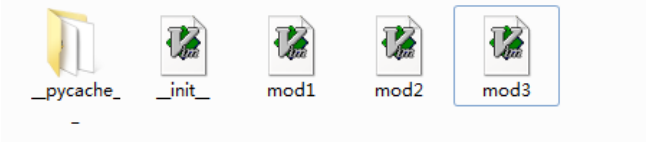
2.28 未定义方法调用

```
94 class A(object):
95         def __init__(self, a, b):
96                 self.__a = a
97                 self.__b = b
98                 print('INIT')
99
100        def default_method(self, *args):
101                print('default method : ' + str(args[0]))
102
103        def __getattr__(self, name):
104                print('undefined method : ', name)
105                return self.default_method
106
107 def test6(arg=None):
108        a1 = A(10, 20)
109        a1.func1(33)
110        a1.func2('hello')
111        a1.func3(10)
112
113 if __name__ == '__main__':
114        test6(arg=None)
115
```

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；

2.29 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；




```
1
2 # -*-   coding: utf-8           -*-
3 # -*-   versison: python 3.4.5  -*-
4
5 # __init__.py
6
7 __all__ = ['mod1', 'mod3']
8
```

3.Algorithm

3.1 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。

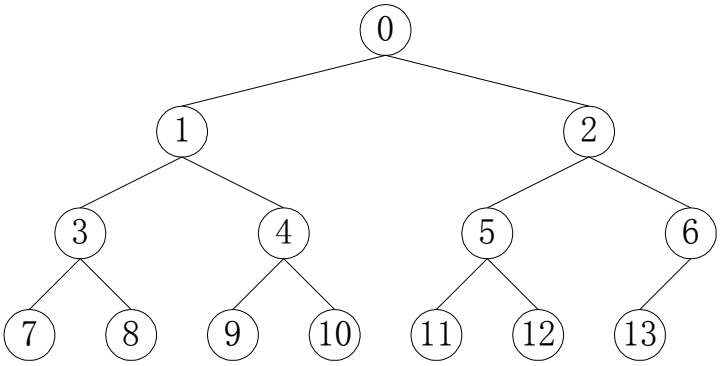
2、类似于时间复杂度的讨论，一个算法的空间复杂度(Space Complexity)S(n)定义为该算法所耗费的存储空间，它也是问题规模 n 的函数。渐近空间复杂度也常常简称为空间复杂度。

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。

3.2 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));
}
```

3.3 二叉树的 C++实现

3.4 二叉树 Python 实现

二叉树的构建与层序遍历：

```
229 # BiTree
230 class BTNode(object):
231         def __init__(self, data, left=None, right=None):
232                 self.data = data
233                 self.left = left
234                 self.right = right
235
236 def bitree_lookup(root):
237         row = [root]
238         while row:
239                 print([node.data for node in row])
240                 row = [kid for item in row for kid in (item.left, item.right
    ) if kid]
241
242
243 def test_bitree(arg=None):
244         tree = BTNode(1, BTNode(3, BTNode(7, BTNode(0)), BTNode(6)), BTNode(
    2, BTNode(5), BTNode(4)))
245         bitree_lookup(tree)
246
247 if __name__ == '__main__':
248         test_bitree(arg = None)
249
```

深度遍历-先序：

```
244 def bitree_pre_traverse(root):
245         if not root:
246                 return
247         print(root.data)
248         bitree_pre_traverse(root.left)
249         bitree_pre_traverse(root.right)
250
```

深度遍历-中序：

```
251 def bitree_mid_traverse(root):
252         if root == None:
253                 return
254         if root.left is not None:
255                 bitree_mid_traverse(root.left)
256         print(root.data)
257         if root.right is not None:
258                 bitree_mid_traverse(root.right)
259
```

深度遍历-后序：

```
260 def bitree_post_traverse(root):
261         if root == None:
262                 return
263         if root.left is not None:
264                 bitree_post_traverse(root.left)
265         if root.right is not None:
266                 bitree_post_traverse(root.right)
267         print(root.data)
268
```

获得树的深度：

```
269 def bitree_get_deepth(root):
270         if not root:
271                 return 0
272         return max(bitree_get_deepth(root.left), bitree_get_deepth(root.righ
    t))+1
273
```

比较两棵树是否相同：

```
274 def bitree_is_same(p, q):
275         if None == p and None == q:
276                 return True
277         elif p and q:
278                 return p.data == q.data and bitree_is_same(p.left, q.left) \
279                                 and bitree_is_same(p.right, q.right)
280         else:
281                 return False
282
```

已知先序遍历列表和中序遍历列表，求后序遍历列表：

```
283 def bitree_rebuild(pre_order, mid_order):
284         if not pre_order:
285                 return
286         current_node = pre_order[0]     # root
287         index = mid_order.index(pre_order[0])
288         current_node.left = rebuild(pre_order[1:index+1], mid_order[:index])
289         current_node.right = rebuild(pre_order[index+1:], mid_order[index+1:
    ])
290         return current_node
291
```

3.5 红黑树

3.6 B 树

3.7 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；

3.8 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：

```cpp
#include <iostream>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

using namespace std;

#define MAXLEN 50

struct Data{
        char key[10];
        char value[10];
        int id;
};

struct StackType{
        Data node[MAXLEN+1];
        int top;
};

void STInit(StackType &ST){
        ST.top = -1;
}

```

```cpp

int STIsEmpty(StackType &ST){
        if(-1 == ST.top)
                return 1;
        else
                return 0;
}

int STIsFull(StackType &ST){
        if(MAXLEN == ST.top)
                return 1;
        else
                return 0;
}

```

```cpp

void STPush(StackType &ST, Data node){
        if(MAXLEN == ST.top){
                cout<<"Stack is Full, Push Failed."<<endl;
                return;
        }
        ST.node[++ST.top] = node;
        return;
}

int STPop(StackType &ST, Data &node){
        if(-1 == ST.top){
                cout<<"Stack is Empty, Pop Failed."<<endl;
                return 0;
        }
        node = ST.node[ST.top--];
        return 1;
}

```

```
57
58 void STShow(StackType &ST){
59         if(-1 == ST.top){
60                 cout<<"Empty Stack."<<endl;
61                 return;
62         }
63         cout<<"============ Show Stack ============="<<endl;
64         int index;
65         for(index = ST.top; index >=0; index--)
66                 cout<<ST.node[index].key<<" "<<ST.node[index].value<<" "<<ST
   .node[index].id<<endl;
67         cout<<"============ Show Stack ============="<<endl;
68         return;
69 }
70 void STClear(StackType &ST){
71         ST.top = -1;
72         return;
73 }
74
```

```
74
75 void StackTest(){
76         StackType stack;
77         STInit(stack);
78         STShow(stack);
79
80         // Stack push test.
81         Data node;
82         strcpy(node.key, "name");
83         strcpy(node.value, "localhost");
84         node.id = 0;
85         STPush(stack, node);
86
87         strcpy(node.key, "port");
88         strcpy(node.value, "8080");
89         node.id = 1;
90         STPush(stack, node);
91         STShow(stack);
92
93         // Stack pop test.
94         Data popnode;
95         STPop(stack, popnode);
96         STShow(stack);
97 }
98
```

C++(未使用 Class)实现队列：

```
 1
 2 #include <iostream>
 3 #include <stdlib.h>
 4 #include <stdio.h>
 5 #include <string.h>
 6
 7 using namespace std;
 8
 9 #define MAXLEN 50
10
11 struct Data{
12         char key[10];
13         char value[10];
14         int id;
15 };
16
17 struct QueueType{
18         Data node[MAXLEN+1];
19         int front;
20         int tail;
21 };
22
23 void QInit(QueueType &QT){
24         QT.front = 0;
25         QT.tail = 0;
26 }
27
```

```
27
28 int QIsEmpty(QueueType &QT){
29         if(QT.front == QT.tail)
30                 return 1;
31         else
32                 return 0;
33 }
34
35 int QIsFull(QueueType &QT){
36         if(MAXLEN == (QT.tail-QT.front))
37                 return 1;
38         else
39                 return 0;
40 }
41
```

```
41
42 void QPush(QueueType &QT, Data node){
43         if(MAXLEN == (QT.tail-QT.front)){
44                 cout<<"Queue is Full, Push Failed."<<endl;
45                 return;
46         }
47         QT.node[QT.tail++] = node;
48         return;
49 }
50
51 int QPop(QueueType &QT, Data &node){
52         if(QT.front == QT.tail){
53                 cout<<"Queue is Empty, Pop Failed."<<endl;
54                 return 0;
55         }
56         node = QT.node[QT.front++];
57         return 1;
58 }
59
```

```
59
60 void QShow(QueueType &QT){
61         if(QT.front == QT.tail){
62                 cout<<"Empty Queue."<<endl;
63                 return;
64         }
65         cout<<"============ </> Show Queue ============="<<endl;
66         int index;
67         for(index = QT.front; index < QT.tail; index++)
68                 cout<<QT.node[index].key<<" "<<QT.node[index].value<<" "<<QT
   .node[index].id<<endl;
69         cout<<"============ Show Queue </> ============="<<endl;
70         return;
71 }
72 void QTClear(QueueType &QT){
73         QT.front = 0;
74         QT.tail = 0;
75         return;
76 }
77
```

```
78 void QueueTest(){
79         QueueType queue;
80         QInit(queue);
81
82         Data node;
83         strcpy(node.key, "name");
84         strcpy(node.value, "localhost");
85         node.id = 0;
86         QPush(queue, node);
87
88         strcpy(node.key, "port");
89         strcpy(node.value, "8080");
90         node.id = 1;
91         QPush(queue, node);
92
93         strcpy(node.key, "user");
94         strcpy(node.value, "admin");
95         node.id = 2;
96         QPush(queue, node);
97         QShow(queue);
98
99         Data popnode;
100        QPop(queue, popnode);
101        QPop(queue, popnode);
102        QShow(queue);
103 }
104
```

3.9 线性结构-链表

使用 **C++ Class** 实现一个链表类：

```
 1
 2 #include <iostream>
 3 #include <stdlib.h>
 4 #include <stdio.h>
 5
 6 using namespace std;
 7
 8 struct LinkNode{
 9         int key;
10         LinkNode* next;
11         LinkNode(int val): key(val), next(NULL){}
12 };
13
```

参数列表实现构造函数；

```
13
14 class LinkList{
15         public:
16                 LinkList();
17                 void InsertHead(int val);
18                 void InsertNode(int val, int pos);
19                 void DeleteNode(int val);
20                 int GetLength();
21                 void Reverse();
22                 int FindNode(int val);
23                 void Print();
24                 bool IsLoop();
25                 ~LinkList();
26         private:
27                 LinkNode *head;
28                 int length;
29 };
30
```

构造函数与析构函数没有返回值；

```
30
31 LinkList::LinkList(){
32         head = NULL;
33         length = 0;
34 }
35
36 LinkList::~LinkList(){
37         LinkNode* p;
38         int i;
39         for(i=0; i<length; i++){
40                 p = head;
41                 head = head->next;
42                 delete(p);
43         }
44 }
45
46 void LinkList::InsertHead(int val){
47         InsertNode(val, 0);
48 }
49
```

```
49
50 void LinkList::InsertNode(int val, int pos){
51         if(pos < 0){
52                 cout<<"Invalid position."<<endl;
53                 return;
54         }
55         int index;
56         LinkNode* p = head;
57         LinkNode* node = new LinkNode(val);
58         if(0 == pos){
59                 head = node;
60                 head->next = p;
61                 length ++;
62                 return;
63         }
64         while(index < pos && p != NULL){
65                 p = p->next;
66                 index ++;
67         }
68         if(NULL == p){
69                 cout<<"Position exceeds length of linklist."<<endl;
70                 return;
71         }
72         node->next = p->next;
73         p->next = node;
74         length ++;
75         return;
76 }
```

```
77
78 void LinkList::DeleteNode(int val){
79         int pos = FindNode(val);
80         if(-1 == pos){
81                 cout<<"Cannot find node with key= "<<val<<endl;
82                 return;
83         }
84         LinkNode* p = head;
85         if(0 == pos){
86                 head = head->next;
87                 delete(p);
88                 length --;
89                 return;
90         }
91         int index = 1;
92         while(index < pos){
93                 p = p->next;
94                 index ++;
95         }
96         LinkNode* q = p->next;
97         p->next = p->next->next;
98         delete(q);
99         return;
100 }
```

85,14-21 37%

```
101
102 int LinkList::FindNode(int val){
103         int pos = 0;
104         LinkNode* p;
105         p = head;
106         while(p != NULL){
107                 if(val == p->key)
108                         return pos;
109                 p = p->next;
110                 pos ++;
111         }
112         return -1;
113 }
114
115 int LinkList::GetLength(){
116         return length;
117 }
118
```

```
118
119 void LinkList::Reverse(){
120         LinkNode* curr = head;
121         LinkNode* pre = curr->next;
122         LinkNode* tmp;
123         while(pre != NULL){
124                 tmp = pre->next;
125                 pre->next = curr;
126                 curr = pre;
127                 pre = tmp;
128         }
129         head->next = NULL;
130         head = curr;
131 }
132
133 void LinkList::Print(){
134         LinkNode* p = head;
135         while(p != NULL){
136                 cout<<(p->key)<<" ";
137                 p = p->next;
138         }
139         cout<<endl;
140         return;
141 }
142
```

判断是否有环：

```
142
143 bool LinkList::IsLoop(){
144
145         LinkNode* pFast = head;
146         LinkNode* pSlow = head;
147         while(pFast != NULL && pFast->next != NULL){
148                 pFast = pFast->next->next;
149                 pSlow = pSlow->next;
150
151                 if(pFast == pSlow)
152                         break;
153         }
154
155         if(NULL == pFast || NULL == pFast->next)
156                 return false;
157         else
158                 return true;
159 }
160
```

```
160
161 void TestIsLoop(void){
162         int a[] = {1,2,3,4,5,6,7,8,9};
163         int i;
164         LinkList link;
165         for(i=(sizeof(a)/sizeof(int) - 1); i>=0; i--){
166                 link.InsertHead(a[i]);
167         }
168         link.Print();
169         link.Reverse();
170         link.Print();
171         cout<<"Is Loop in LinkList ? "<<link.IsLoop()<<endl;
172 }
173
174 int main(int args, char** argv){
175         TestIsLoop();
176         return 0;
177 }
178
```

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；

3.10 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；

```
 99 class LinkList(object):
100         def __init__(self):
101                 self.head = 0
102
103         def initializer(self, data):
104                 self.head = ListNode(data[0])
105                 p = self.head
106                 for v in data[1:]:
107                         p.next = ListNode(v)
108                         p = p.next
109
110 def swap_pairs(head):
111         if head != None and head.next != None:
112                 next = head.next
113                 head.next = swap_pairs(next.next)
114                 next.next = head
115                 return next
116         return head
117
```

```
118 def test_swap_pairs(arg = None):
119         data = [1,2,3,4]
120         ls = LinkList()
121         ls.initializer(data)
122         ls_head = ls.head
123         swap_ls = []
124         node = swap_pairs(ls_head)
125         for i in range(len(data)):
126                 swap_ls.append(node.val)
127                 node = node.next
128         print(swap_ls)
129 if __name__ == '__main__':
130         test_swap_pairs(arg = None)
131
```

3.11 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：

```
148 def cross_node(l1, l2):
149         length1 = 0
150         length2 = 0
151         p1 = l1.head
152         p2 = l2.head
153         while p1:
154                 p1 = p1.next
155                 length1 += 1
156         p1 = l1.head
157         print('length1 = %d' %length1)
158         while p2:
159                 p2 = p2.next
160                 length2 += 1
161         p2 = l2.head
162         print('length2 = %d' %length2)
163         if length1 > length2:
164                 for _ in range(length1 - length2):
165                         p1 = p1.next
166         else:
167                 for _ in range(length2 - length1):
168                         p2 = p2.next
169         while p1 != None:
170                 if(p1.val != p2.val):
171                         p1 = p1.next
172                         print(p1.val, end='')
173                         p2 = p2.next
174                         print(' ', p2.val)
175                 else:
176                         return p1
177         print('No Cross Node')
178         return None
179
```

```
180 def test_cross_node(arg=None):
181         data1 = [1,2,3,7,9,1,5]
182         data2 = [4,5,7,9,1,5]
183         l1 = LinkList()
184         l1.initializer(data1)
185         l2 = LinkList()
186         l2.initializer(data2)
187
188         xnode = cross_node(l1, l2)
189         if xnode != None:
190                 print(xnode.val)
191         else:
192                 print('No cross node of l1 and l2')
193
194 if __name__ == '__main__':
195         test_cross_node(arg = None)
196
```

没有判断链表中是否存在环；

3.12 单向链表逆置

```
307 def reverse_linklist(link):
308         pre = link
309         cur = link.next
310         pre.next = None
311         while cur:
312                 tmp = cur.next
313                 cur.next = pre
314                 pre = cur
315                 cur = tmp
316         return pre
317
318 def test_reverse_linklist(arg=None):
319         link = ListNode(1, ListNode(2, ListNode(3, ListNode(4, ListNode(5, L
    istNode(6, ListNode(7, ListNode(8, ListNode(9)))))))))
320         root = reverse_linklist(link)
321         while root:
322                 print(root.val)
323                 root = root.next
324
325 if __name__ == '__main__':
326         test_reverse_linklist(arg = None)
327
```

3.13 排序算法

3.13.1 快速排序

复杂度
最坏情况如何优化

```
208 def test_binary_search(arg=None):
209         list = [1,3,4,5,7,8,9]
210         element = 3
211         print('get element 3 index is %d'
212                 %(binary_search(list, element)))
213
214 def quick_sort(list):
215         if len(list)<2:
216                 return list
217         else:
218                 mid_pivot = list[0]
219                 smaller_list = [e for e in list[1:] if e <= mid_pivot]
220                 bigger_list = [e for e in list[1:] if e > mid_pivot]
221                 final_list = quick_sort(smaller_list) + [mid_pivot]\
222                         + quick_sort(bigger_list)
223                 return final_list
224
225 def test_quick_sort(arg=None):
226         list = [2,4,6,7,1,2,5]
227         print(quick_sort(list))
228
229 if __name__ == '__main__':
230         test_quick_sort(arg = None)
231
```

C++的递归方式实现：

```
28 void QuickSort(int* array, int start, int end){
29         if(NULL == array){
30                 cout<<"Invalid array transfered in !"<<endl;
31                 return ;
32         }
33
34         int mid;
35         int pivot;
36         int low;
37         int high;
38
39         if(start < end){
40                 low = start;
41                 high = end;
42                 pivot = array[start];
43                 while(low < high){
44                         while(low < high && array[high] >= pivot)
45                                 --high;
46                         array[low] = array[high];
47                         while(low < high && array[low] <= pivot)
48                                 ++low;
49                         array[high] = array[low];
50                 }
51                 array[low] = pivot;
52                 mid = low;
53                 QuickSort(array, start, mid-1);
54                 QuickSort(array, mid+1, end);
55         }
56 }
                                                              56,1          36%
```

```
58 int main(int args, char** argv){
59         int array[] = {3, 5, 9, 2, 1, 3, 9, 13, 1};
60         int i;
61         int start = 0;
62         int end = sizeof(array)/sizeof(int) - 1;
63         QuickSort(array, start, end);
64         for(i=0; i <= end; i++){
65                 cout<<array[i]<<" ";
66         }
67         cout<<endl;
68 }
69
```

3.13.2 合并排序

复杂度
最坏情况如何优化

3.13.3 堆排序

复杂度
最坏情况如何优化

3.13.4 单向链表应用快速排序

3.13.5 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：

```
371 def get_median(list):
372         list = sorted(list)
373         length = len(list)
374         return list[int(length/2)]
375
376 def test_get_median(arg=None):
377         list = [1,5,6,3,6,9,5]
378         m = get_median(list)
379         print(m)
380
381 if __name__ == '__main__':
382         test_get_median(arg=None)
383
```

使用 C++实现排序：见[快速排序]；

3.14 查找算法

3.14.1 二分查找

```
194 def binary_search(list, element):
195         start = 0
196         end = len(list)-1
197         while start <= end:
198                 middle = int((start+end)/2)
199                 guess = list[middle]
200                 if guess == element:
201                         return middle
202                 elif guess > element:
203                         end = middle-1
204                 else:
205                         start = middle+1
206         return None
207
208 def test_binary_search(arg=None):
209         list = [1,3,4,5,7,8,9]
210         element = 3
211         print('get element 3 index is %d'
212                 %(binary_search(list, element)))
213
214 if __name__ == '__main__':
215         test_binary_search(arg = None)
216
```

3.14.2 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：

```
70 def get_value(l, r, c):
71         return l[r][c]
72
73 def find_element(l, x):
74         m = len(l) - 1
75         n = len(l[0]) - 1
76         r = 0
77         c = n   # point to the last column
78         while c >= 0 and r <= m:
79                 value = get_value(l, r, c)
80                 if value == x:
81                         return True
82                 elif value > x:
83                         c -= 1
84                 elif value < x:
85                         r += 1
86         return False
87
88 def test_find_element(arg = None):
89         l = [[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]]
90         x1 = 5
91         x2 = 13
92         x3 = 0
93         print(find_element(l, x1), find_element(l, x2), find_element(l, x3))
94
95 if __name__ == '__main__':
96         test_find_element()
97
```

3.15 无序数字列表寻找所有间隔为 d 的组合

```
>>> l = [1,2,3,4,5,6,7,8,9]
>>> for i in l:
...     for j in l:
...         if j-i == 2:
...             print(i, j)
...
1 3
2 4
3 5
4 6
5 7
6 8
7 9
>>>
```

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；

3.16 列表[a1, a2, a3, ..., an]求其所有组合

```
>>> from itertools import combinations
>>> print(list(combinations(['a', 'b', 'c', 'd', 'e', 'f', 'g'], 3)))
[('a', 'b', 'c'), ('a', 'b', 'd'), ('a', 'b', 'e'), ('a', 'b', 'f'), ('a', 'b',
'g'), ('a', 'c', 'd'), ('a', 'c', 'e'), ('a', 'c', 'f'), ('a', 'c', 'g'), ('a',
'd', 'e'), ('a', 'd', 'f'), ('a', 'd', 'g'), ('a', 'e', 'f'), ('a', 'e', 'g'), (
'a', 'f', 'g'), ('b', 'c', 'd'), ('b', 'c', 'e'), ('b', 'c', 'f'), ('b', 'c', 'g
'), ('b', 'd', 'e'), ('b', 'd', 'f'), ('b', 'd', 'g'), ('b', 'e', 'f'), ('b', 'e
', 'g'), ('b', 'f', 'g'), ('c', 'd', 'e'), ('c', 'd', 'f'), ('c', 'd', 'g'), ('c
', 'e', 'f'), ('c', 'e', 'g'), ('c', 'f', 'g'), ('d', 'e', 'f'), ('d', 'e', 'g')
, ('d', 'f', 'g'), ('e', 'f', 'g')]
>>>
```

3.17 一行 python 代码实现 2**1000 各位数求和

```
>>> sum(map(int, str(2**1000)))
1366
```

3.18 遍历目录

遍历指定目录，输出其所有子目录以及所有文件：

```
351 # Level Order Traverse a Directory
352 import os
353
354 def traverse_directory(path):
355         sub_path_list = []
356         file_path_list = []
357         for rootdir, subdirs, filenames in os.walk(path):
358                 for subdir in subdirs:
359                         sub_path_list.append(os.path.join(rootdir, subdir))
360
361                 for filename in filenames:
362                         file_path_list.append(os.path.join(rootdir, filename
    ))
363         return sub_path_list, file_path_list
364
365 def test_traverse_directory(arg=None):
366         path = './sub_pack'
367         sub_directories, files = traverse_directory(path)
368         print('sub directory list is : ', sub_directories)
369         print('file list is : ', files)
370
371 if __name__ == '__main__':
372         test_traverse_directory(arg=None)
373
```

3.19 DP 问题

3.19.1 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)

3.19.2 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)

3.20 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：

```
>>> l = ['b', 'c', 'd', 'b', 'c', 'a', 'a']
>>> list(set(l))
['a', 'c', 'b', 'd']
>>>
```

- 使用字典：

```
>>> l2 = {}.fromkeys(l).keys()
>>> l2
dict_keys(['a', 'c', 'b', 'd'])
>>>
```

保持元素在原列表中的顺序：

```
>>> l2 = list(set(l))
>>> l2.sort(key=l.index)
>>> l2
['b', 'c', 'd', 'a']
>>>
```

- 使用列表推导式：

```
>>> l2 = []
>>> [l2.append(i) for i in l if not i in l2]
[None, None, None, None]
>>> l2
['b', 'c', 'd', 'a']
>>>
```

带排序的列表推导式：

```
>>> ls = []
>>> [ls.append(i) for i in sorted(l) if not i in ls]
[None, None, None, None]
>>> ls
['a', 'b', 'c', 'd']
>>>
```

列表排序：

```
>>> lt = [5,9,8,4,82,28,34,1]
>>> sorted(lt)
[1, 4, 5, 8, 9, 28, 34, 82]
>>> sorted(lt).reverse()
>>> lt
[5, 9, 8, 4, 82, 28, 34, 1]
```

```
>>> sorted(lt, reverse=True)
[82, 34, 28, 9, 8, 5, 4, 1]
>>>
```

3.21 创建字典

- 手动创建：

```
>>> dict = {'name': 'earth', 'port': '8080'}
>>> dict
{'port': '8080', 'name': 'earth'}
>>>
```

- 工厂方法：

```
>>> items = [('name', 'port'), ('earth', '8080')]
>>> d = dict(items)
>>> d
{'earth': '8080', 'name': 'port'}
>>>
```

```
>>> d1 = dict([['name', 'earth'], ['port', '8080']])
>>> d1
{'name': 'earth', 'port': '8080'}
>>>
```

- **fromkeys()方法：**

```
>>> dict1 = {}.fromkeys(('x', 'y'), -1)
>>> dict1
{'y': -1, 'x': -1}
>>>
```

3.22 合并有序列表

```
130 def loop_merge_sort(l1, l2):
131         tmp = []
132         while len(l1) > 0 and len(l2) > 0:
133                 if l1[0] < l2[0]:
134                         tmp.append(l1[0])
135                         del(l1[0])
136                 else:
137                         tmp.append(l2[0])
138                         del(l2[0])
139         tmp.extend(l1)
140         tmp.extend(l2)
141         return tmp
142
143 def test_loop_merge_sort(arg = None):
144         a = [1,2,3,7]
145         b = [4,5,6]
146         print(loop_merge_sort(a, b))
147
148 if __name__ == '__main__':
149         test_loop_merge_sort(arg = None)
150
```

3.23 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；

```
342 def test_check_change_word(arg=None):
343         s1 = 'list'
344         s2 = 'still'
345         print('list and still is change word : ', check_change_word(s1, s2))
346
347         s1 = 'stop'
348         s2 = 'post'
349         print('stop and post is change word : ', check_change_word(s1, s2))
350
351 if __name__ == '__main__':
352         test_check_change_word(arg = None)
353
```

4.OS

4.1 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；

```
1
2 import time
3 import threading
4
5 def timer(no, interval):
6         cnt = 0
7         while cnt < 10:
8                 print('Thread: (%d) Time: %s\n' %(no, time.ctime()))
9                 time.sleep(interval)
10                cnt += 1
11
12 def test0(arg = None):
13        t1 = threading.Thread(target=timer, args=(1, 1))
14        t2 = threading.Thread(target=timer, args=(2, 2))
15        t1.start()
16        t2.start()
17        t1.join()
18        t2.join()
19
20 if __name__ == '__main__':
21        test0()
```

- 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；

```
20 class NewThread(threading.Thread):
21
22         def set_timer(self, no, interval):
23                 self.no = no
24                 self.interval = interval
25         def run(self):
26                 timer(self.no, self.interval)
27
28 def test1(arg = None):
29         t1 = NewThread()
30         t2 = NewThread()
31         t1.set_timer(1,1)
32         t2.set_timer(2,2)
33         t1.start()
34         t2.start()
35         t1.join()
36         t2.join()
37
38 if __name__ == '__main__':
39         test1()
```

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

```
39 def ptimer(no, interval):
40         cnt = 0
41         while cnt < 10:
42                 print('Process: (%d) Time: %s\n' %(no, time.ctime()))
43                 time.sleep(interval)
44                 cnt += 1
45
46 def test2(arg = None):
47         p1 = Process(target=ptimer, args=(1,1))
48         p2 = Process(target=ptimer, args=(2,2))
49
50         p1.start()
51         p2.start()
52         p1.join()
53         p2.join()
54
55 if __name__ == '__main__':
56         test2()
```

4.2 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：

```
 4 def old_fib(n):
 5         res = [0]*n
 6         index = 0
 7         a = 0
 8         b = 1
 9         while index < n:
10                 res[index] = b
11                 a, b = b, a+b
12                 index += 1
13         return res
14
15 def test_old_fib(arg = None):
16         print('-'*10 + 'test old fib ' + '-'*10)
17         for fib_res in old_fib(20):
18                 print(fib_res)
19
20 if __name__ == '__main__':
21         test_old_fib(arg = None)
22
```

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；

```
20 def yield_fib(n):
21         index = 0
22         a = 0
23         b = 1
24         while index < n:
25                 yield b
26                 a, b = b, a+b
27                 index += 1
28
29 def test_yield_fib(arg = None):
30         print('-'*10 + 'test yield fib ' + '-'*10)
31         for fib_res in yield_fib(20):
32                 print(fib_res)
33
34 if __name__ == '__main__':
35         test_yield_fib(arg = None)
36
```

- **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：

```
36 def stupid_fib(n):
37         index = 0
38         a = 0
39         b = 1
40         while index < n:
41                 sleep_cnt = yield b
42                 print('let me think {0} secs'.format(sleep_cnt))
43                 a, b = b, a+b
44                 index += 1
45
46 def test_stupid_fib(arg = None):
47         print('-'*10 + 'test stupid fib ' + '-'*10)
48         N = 20
49         sfib = stupid_fib(N)
50         fib_res = next(sfib)
51         while True:
52                 print(fib_res)
53                 try:
54                         fib_res = sfib.send(random.uniform(0, 0.5))
55                 except StopIteration:
56                         break
57
58 if __name__ == '__main__':
59         test_stupid_fib(arg = None)
60
```

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；

```
59 def copy_stupid_fib(n):
60         print('Copy from stupid fib')
61         yield from stupid_fib(n)
62         print('Copy end')
63
64 def test_copy_stupid_fib(arg = None):
65         print('-'*10 + 'copy from stupid fib ' + '-'*10)
66         N = 20
67         csfib = copy_stupid_fib(N)
68         fib_res = next(csfib)
69         while True:
70                 print(fib_res)
71                 try:
72                         fib_res = csfib.send(random.uniform(0, 0.5))
73                 except StopIteration:
74                         break
75
76 if __name__ == '__main__':
77         test_copy_stupid_fib(arg = None)
78
```

- **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；

```
77 @asyncio.coroutine
78 def smart_fib(n):
79         index = 0
80         a = 0
81         b = 1
82         while index < n:
83                 sleep_secs = random.uniform(0, 0.2)
84                 yield from asyncio.sleep(sleep_secs)
85                 print('Smart one think {} secs to get {}'.format(sleep_secs,
    b))
86                 a, b = b, a+b
87                 index += 1
88
89 @asyncio.coroutine
90 def dull_fib(n):
91         index = 0
92         a = 0
93         b = 1
94         while index < n:
95                 sleep_secs = random.uniform(0, 0.5)
96                 yield from asyncio.sleep(sleep_secs)
97                 print('Dull one think {} secs to get {}'.format(sleep_secs,
   b))
98                 a, b = b, a+b
99                 index += 1
100
101 def test_asyncio_coroutine(arg = None):
102         loop = asyncio.get_event_loop()
103         tasks = [asyncio.async(smart_fib(10)),
104                  asyncio.async(dull_fib(10)),]
105         loop.run_until_complete(asyncio.wait(tasks))
106         print('All fib finished.')
107         loop.close()
108
```

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：

```
async def smart_fib(n):
        index = 0
        a = 0
        b = 1
        while index < n:
                sleep_secs = random.uniform(0, 0.2)
                await asyncio.sleep(sleep_secs)
                print('Smart one think {} secs to get {}'.format(sleep_secs,
 b))
                a, b = b, a+b
                index += 1

async def dull_fib(n):
        index = 0
        a = 0
        b = 1
        while index < n:
                sleep_secs = random.uniform(0, 0.5)
                await asyncio.sleep(sleep_secs)
                print('Dull one think {} secs to get {}'.format(sleep_secs,
 b))
                a, b = b, a+b
                index += 1

def test_async_await(arg = None):
        loop = asyncio.get_event_loop()
        tasks = [asyncio.ensure_future(smart_fib(10)),
                 asyncio.ensure_future(dull_fib(10)),]
        loop.run_until_complete(asyncio.wait(tasks))
        print('All fib finished.')
        loop.close()
if __name__ == '__main__':
        test_async_await(arg = None)
```

4.3 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；

4.4 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

4.5 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；

4.6 常用的 Linux/git/vim 命令和作用

4.6.1 Linux 常用命令

4.6.2 git 常用命令

4.6.3 vim 常用命令

4.7 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：

```
alfred@ubuntu:~$ ps -ef | grep python
alfred     2514   2511  0 20:27 ?        00:00:00 python /usr/share/indicator-ch
ina-weather/src/indicator-china-weather.py
alfred     2813   2164  1 20:28 ?        00:00:05 /usr/bin/python3 /usr/bin/upda
te-manager --no-update --no-focus-on-map
alfred     2953   2864  0 20:34 pts/0    00:00:00 grep --color=auto python
alfred@ubuntu:~$ ps -aux | grep python
alfred     2514  0.0  3.6 463612 36184 ?        Sl   20:27   0:00 python /usr/sh
are/indicator-china-weather/src/indicator-china-weather.py
alfred     2813  1.6 12.5 644748 125956 ?       SNl  20:28   0:05 /usr/bin/pytho
n3 /usr/bin/update-manager --no-update --no-focus-on-map
alfred     2958  0.0  0.2  15960  2260 pts/0    S+   20:34   0:00 grep --color=a
uto python
alfred@ubuntu:~$
```

结束进程使用 **kill PID** 命令：

```
alfred@ubuntu:~$ ps -aux | grep vim
alfred     3020  0.2  1.8 257072 18636 pts/0    Sl+  20:42   0:00 vim 0103.c
alfred     3026  0.0  0.2  15956  2268 pts/5    S+   20:42   0:00 grep --col
uto vim
alfred@ubuntu:~$ kill 3020
alfred@ubuntu:~$
```

4.8 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。

5.网络

5.1 TCP/IP 协议

5.2 OSI 五层协议

5.3 Socket 长连接

5.4 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；

5.5 TCP 与 UDP 协议的区别

5.6 TIME_WAIT 过多的原因

5.7 http 一次连接的全过程描述

从用户发起 request 到用户接收到 response

5.8 http 连接方式——get 与 post 的区别

5.9 restful

5.10 http 请求的状态码 200/403/404/504

6.数据库

6.1 MySQL 锁的种类

6.2 死锁的产生

6.3 MySQL 的 char/varchar/text 的区别

6.4 Join 的种类与区别

6.5 A LEFT JOIN B 的查询结果中，B 缺少的部分如何显示

6.6 索引类型的种类

6.7 BTree 索引与 hash 索引的区别

6.8 如何对查询命令进行优化

6.9 NoSQL 与关系型数据库的区别

6.10 Redis 常用的存储类型