Python 面向对象编程与小乌龟画图

柳银萍

- Python如同C++、Java语言一样，提供了面向对象的编程方式。面向对象的编程是以（自定义）数据结构为主，然后再编写出与这个数据结构相关的各种函数，即**方法**（methods）。
- Python本身就是面向对象编程的具体实现，Python中的列表、字符串、字典等就是“自定义”数据结构，当需要某个已经定义的数据结构时，定义一个变量，将它归属于这个数据类型，这个变量通常被称为“**对象**”（object）。
- 本章讲述Python的自定义数据结构--**类**(class)和面向对象编程的基本概念。

1.1 什么是对象

在现实生活中，随处可见的一种事物就是**对象**，对象是事物存在的实体。比如 小狗 是真实世界的一个对象，通常应该如何描述这个对象呢?

（1）**静态特征**：比如，四条腿、一条尾巴、两只耳朵……

（2）**动态特征**：比如它跑得很快，喜欢睡懒觉……

对象的静态特征称为“**属性**”，对象的动态特征称为“**方法**”。概括来讲“**对象=属性+方法**”。

1.2 体会面向对象编程的优势

- **面向过程**（Procedure-Oriented）：

  是一种以**事件**为中心的编程思想

通过把解决问题的步骤写出来，让程序一步一步去执行，直到解决问题

- **面向对象**（Object-Oriented）：

  是一种以**事物**为中心的编程思想

把数据/事物作为对象（属性+方法），使用统一的数据类及函数接口，更好的扩展性，适用于大型或多人合作

1.2 体会面向对象编程的优势

【问题描述】一个班级有20个学生，每个学生有自己的名字和学号。开学后，学生进行选课，每个学生所选的课程名称和数目可能不同。

面向过程的思想编程表示：

```
#<程序：学生信息—面向过程编程>
name=["兰兰","阿珍","小红"]
number=["1000","1001","1002"]
course=[["计算机科学导论","算法导论","图论"],
        ["计算机科学导论","图论","数据结构"],
        ["计算机操作系统","图论","计算机网络"]]
grade=[[98,97,90], [91,80,93], [90,89,78]]
```

2 介绍面向对象中的概念

- 2.1 类与对象
- 2.2 Python中的__init__()方法
- 2.3 self 变量 和 pass 关键字
- 2.4 Python中“公有”和“私有”类型的定义方式

2.1 类与对象

- 类的定义：

```
class    类名：
         属性
         方法
```

```
#<程序：Dog类1>
class Dog:
    leg=4; tail=1; ear=2 ; weight=5  #属性
    def run(self):          #方法1
        print("我正在飞快地跑")
    def sleep(self):#方法2
        print("我正在睡觉")
    def eat(self):   #方法3
        print("食物真好吃")
```

属性也称为 **成员变量**

方法也称为 **成员函数**

2.1 类与对象

- **类的实例化（Instantiate）**：通过类创建出多个类的对象，这个过程叫做**类的实例化**，创建的对象也叫作实例对象。对象都具有该类的属性和方法。可以通过“**对象名.成员**”的方式来访问 成员变量 或 成员函数。

#<程序：创建对象并访问对象的属性和方法>

```
mydog=Dog()  #创建名为mydog的对象
w1=mydog.weight   #访问mydog对象中weight属性
print("我的小狗重:%d kg"%w1)   #输出：  我的小狗重:5 kg
mydog.run()   #输出：  我正在飞快地跑
```

实例出来的对象具有与类相同的属性和方法

2.1 类与对象

思考：**通过一个类可以创建多个对象，这些对象均具有完全相同的属性和方法。**

如果修改一个对象的某个属性，是否所有对象的对应属性均发生改变呢?

```
#<程序：修改对象的属性 >
mydog=Dog(); herdog=Dog()  #创建mydog和herdog对象
mydog.weight=7 #将我的小狗的重量修改成7
w1=mydog.weight
print("修改体重后，我的小狗重:%d kg"%w1)  #输出：修改体重后，我的小狗重:7 kg
w2=herdog.weight #访问herdog中的weight属性
print("她的小狗重:%d kg"%w2)  #输出：她的小狗重:5 kg
```

对mydog对象中weight属性的修改并没有影响herdog中的weight属性。这其中蕴含了什么原理呢?

2.1 类与对象

类中的变量（属性）就好比一个母版，当通过类创建对象时，就将类中的母版拷贝（变量地址的拷贝）过去，使对象与类具有相同的成员。只有对对象中的变量进行了赋值操作，对象才会创建只属于对象本身的变量，这样对象对该变量的修改就不会影响其他的对象。

注意，母版拷贝指的是类中变量地址的拷贝。

#<程序：母版拷贝是指地址的拷贝>

```
class A:
    w=5  #属性
a1=A()
print(id(a1.w))   #输出1756720736，表示对象a1中w属性的地址
a1.w=7
print(id(a1.w))    #输出1756720800，表示修改属性后，对象a1中w属性的地址
a2=A()
print(id(a2.w))    #输出1756720736，表示对象a2中w属性的地址
```

2.1 类与对象

思考：类和对象**具有完全相同的属性，即相同的地址。**

对于string、tuple和数值这些不可变类型，当在对象中修改它们的属性时，会在对象中重新创建对应的变量值。但是对于list、dict这类可变类型，则要注意对可变类型的变量执行的是何种操作（赋值语句、append操作？）。

2.1 类与对象

```
#<程序：对象的可变类型属性的修改操作>
class Rect:
    size = [5,10]  # list类型属性
a1=Rect();  print(id(a1.size))  #输出1432633695048，对象a1中size属性的地址
a1.size=[6,9]; print(id(a1.size))  #输出1432634217224，修改属性后a1中size属性地址
a2=Rect(); print(id(a2.size))  #输出1432633695048，对象a2中size属性的地址
a2.size[1]=11; print(id(a2.size))  #输出1432633695048，对象a2中size属性的地址
a3=Rect(); print(id(a3.size))  #输出1432633695048，对象a3中size属性的地址
a3.size.append(13); print(id(a3.size)) #输出1432633695048，对象a3中size属性的地址
```

2.2 Python中的__init__()方法

【注意】：在类中定义的属性，最好是通用的属性。前面的Dog类中定义weight=5是个不很恰当的示范（每条小狗有不同的重量、昵称、主人）。非通用属性应该定义在对象中（而不是定义在类中），例如__init__ 方法中定义才为恰当。

```
#<程序：Dog类1>
class Dog:
    leg=4; tail=1; ear=2 ; weight=5   #属性
    def run(self):          #方法1
        print("我正在飞快地跑")
    def sleep(self):                    #方法2
        print("我正在睡觉")
    def eat(self):                      #方法3
        print("食物真好吃")
```

2.2 Python中的__init__()方法

- __init__是个特殊的方法，在这个方法名称中，开头和末尾各有两个下划线，每当创建对象时都会**自动调用**，该函数的参数根据需求定义。

```
#<程序：__init__的使用方法>
class Dog:
    leg=4; tail=1; ear=2
    def __init__(self, a, b):
        self.name=a
        self.owner=b
dog1=Dog("大黄","阿珍")
dog2=Dog("毛毛","兰兰")
print(dog1.owner, "的小狗叫", dog1.name) #输出：阿珍的小狗叫大黄
print(dog2.owner, "的小狗叫", dog2.name) #输出：兰兰的小狗叫毛毛
```

创建对象时需要设置的信息，定义在对象的方法中

- 创建对象时传入相应的参数
- 可通过 **对象.函数名** 来访问

2.3 self变量和pass关键字

- **类的所有实例方法都至少有一个名为 self 的参数，并且是方法的第一个参数，用于表示对象本身**，但是在调用方法时不需要为这个参数赋值。由同一个类可以生成无数对象，当一个对象的方法被调用的时候，对象会将自身作为一个参数传给该方法，这样Python就知道需要操作哪些对象的方法了。

2.3 self变量和pass关键字

下面例子中，Dog类中有两个方法：setOwner()和getOwner()，分别为设置主人的姓名和获取主人的姓名。

```
#<程序：self作用实例程序>
class Dog:
    def setOwner(self,name):     #方法1：设置对象的主人姓名
        self.owner=name
    def getOwner(self):          #方法2：获取对象的主人姓名
        print('我的主人是：%s'%self.owner)
    def run(self):               #方法3
        pass
a=Dog()           #创建了小狗对象a
a.setOwner("小红") #小红认领了小狗a
b=Dog()           #创建了小狗对象b
b.setOwner("兰兰") #兰兰认领了小狗b
a.getOwner()    #输出：我的主人是：小红
b.getOwner()    #输出：我的主人是：兰兰
```

2.3 self变量和pass关键字

思考：上面例子中通过setOwner函数创建了owner变量并给它赋值，这个变量是**对象自身的变量**，**而非类中的变量**，这种变量可以通过“对象.成员”的方式访问和修改变量，为什么还要使用 getOwner这种函数的方式呢?

```
#<程序：对象间相互影响实例 >
class A:
    list=[1,2,3]
a1=A( ); a2=A()  #创建类A的两个对象
print(a1.list); print(a2.list)  #输出它们的list属性，均为 [1,2,3]
a1.list.append(4)          #对象a1中list属性进行修改
print(a1.list); print(a2.list)  #输出两个对象的list属性，均为[1,2,3,4]
```

通过函数定义的变量才是对象特有的变量，对它的修改不会影响到其他的对象。

2.3 self变量和pass关键字

- pass 是Python提供的一个关键字，它代表空语句，目的是为了保持程序结构的完整性。

```
#<程序：pass作用实例程序>
class A:     #暂时没有确定A类的功能，使用pass占位
    pass
def demo():  #暂时没有确定函数demo功能，使用pass占位
    pass
if 5>3:      #暂时没有确定判断为真该怎么处理，使用pass占位
    pass
```

2.4 Python中“公有”和“私有”类型的定义方式

- **公有属性**：前面类中定义的属性都是公有属性，可以通过“对象名.成员”的方式来进行访问。

```
#<程序：公有属性name>
class Dog:
    name='大黄' #属性
mydog=Dog()
print(mydog.name) #输出： 大黄
```

- **私有属性**：Python定义私有类型的方式很简单，只需在变量名前加上“__”(两个下划线)。

```
#<程序：私有属性>
class Dog:
    __name='大黄' #私有属性
```

2.4 Python中“公有”和“私有”类型的定义方式

- **访问私有属性**：在类的外部不能直接访问，可以利用类内部的方法，通过“self.私有成员”进行访问。

```
#<程序：访问私有属性>
class Dog:
    __name='大黄' #私有属性
    def getName(self): #方法：获取私有属性值
        return self.__name
mydog=Dog()
print(mydog.getName()) #输出：大黄
```

私有属性是为了数据的封装和保密设计的，一般只能在类的内部访问。

2.4 Python中“公有”和“私有”类型的定义方式

- **访问私有方法**：私有方法与私有属性类似，它的名字也是以两个下划线开始的，可以在类的内部通过“self.**私有成员**” 来访问。

```
#<程序：私有方法的使用>
class Dog:
    __age=1
    def getAge(self):
        return self.__age
    def __changeAge(self,age): #私有方法
        self.__age=age
    def needChange(self,age):
        if type(age)==int:
            self.__changeAge(age)
        else:print("年龄更改不合法")
mydog=Dog()
print(mydog.getAge()) #输出：小狗的年龄为 1
mydog.needChange("毛毛") #输出：年龄更改不合法
print(mydog.getAge()) #输出：小狗的年龄为 1
```

3 了解面向对象的三大特性

- 3.1 封装（Package）
- 3.2 继承（Inheritance）
- 3.3 多态（Polymorphism）

3.1 封装（Package）

生活中处处都是**封装**的概念

- 家里的电视机，从开机，浏览节目，换台到关机，用户不需要知道电视机里面的具体细节，只需要在用的时候按下遥控器就可以完成操作；
- 用支付宝进行付款的时候，只需要把付款的二维码提供给收款方扫一下就可以完成支付，不需要知道后台是怎样处理数据的，这都体现了一种封装的概念。

- 在面向对象的编程语言中，“封装”就是将抽象得到的属性和行为相结合，形成一个有机的整体（即**类**）。

3.1 封装（Package）

一个房间有名称name、主人的名字owner、长length、宽width、高height等属性。构建Room类并实例化了一个对象，现在若需要计算房间面积，可定义函数 area( )。

```
#<程序：Room类>
class Room: #类的设计者：定义一个房间的类
    def __init__(self,n,o,l,w,h):
        self.name = n
        self.owner = o
        self.length = l #房间的长
        self.width = w  #房间的宽
        self.height = h #房间的高
#类的使用者
r1 = Room("客厅","阿珍",20,30,9)
def area(room):
 print("{0}的{1}面积是{2}".format(room.owner,room.name, room.length*room.width))
area(r1) #输出：阿珍的客厅面积是600
```

在类外定义函数计算房间的面积

3.1 封装（Package）

- **封装可以简化编程**，使用者不必了解具体的实现细节。可以直接在Room类的内部定义访问数据的函数area()。

```
#<程序：对Room类的方法进行封装>
#类的设计者
class Room: #定义一个房间的类
    def __init__(self,n,o,l,w,h):
        self.name = n
        self.owner = o
        self.length = l  #房间的长
        self.width = w   #房间的宽
        self.height = h  #房间的高
    def area(self):          #求房间的平方的功能
        print("{0}的{1}面积{2}".format(self.owner,self.name,self.length*self.width))
#类的使用者
r1 = Room("客厅","阿珍",20,30,9)
r1.area()  #输出：阿珍的客厅面积是600
```

- 在类内定义函数计算房间的面积，不必要从外面的函数去访问实例的数据，把数据和逻辑都“封装”起来
- 通过“**对象.成员**”去访问里面的属性和方法

3.1 封装（Package）

- 封装能够 **简化编程**
- 封装能够 **增强安全性**
- 封装提供良好的 **可扩展性**

思考：上述例子中增加求体积的功能，外部调用感知不到，仍然是area方法，使功能增加，代码如何改动？

类的设计者，轻松的扩展了功能，而类的使用者不需要改变自己的代码

3.1 封装（Package）

```
#<程序：对Room类中的方法进行扩展>
#类的设计者，轻松的扩展了功能，而类的使用者不需要改变自己的代码
class Room: #定义一个房间的类
  def __init__(self,n,o,l,w,h):
     self.name = n
     self.owner = o
     self.__length = l #房间的长
     self.__width = w  #房间的宽
     self.__height = h #房间的高
  def area(self): #此时我们增加了求体积
     print("{0}的{1}面积是{2}".format(self.owner,self.name,
self.__length*self.__width))
     print("体积是{0}".format(self.__length*self.__width*self.__height))
#类的使用者
r1 = Room("客厅","阿珍",20,30,9)
r1.area()#输出：  阿珍的客厅面积是600 体积是5400
```

3.2 继承（Inheritance）

简单地对动物进行分类，有狗、猫、鱼等，大部分动物的属性和方法是相似的，比如都有嘴巴、眼睛等静态特征，可以吃东西、睡觉等动态特征。

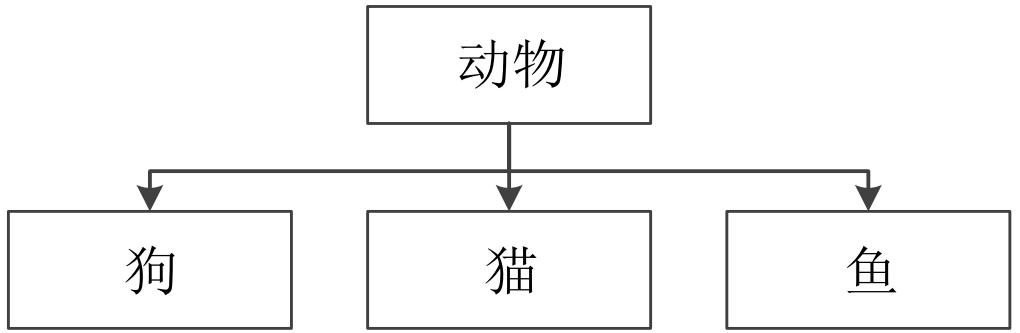
思考一个问题，如果要构建这些动物类能不能不要每次从头到尾去重新定义一个新的类呢?

如果有一种机制可以让这些相似的东西得以自动传递，就方便快捷多了，Python确实提供了这样一种机制——**继承**。

3.2 继承（Inheritance）

1、继承的基本知识

- 被继承的类称为父类，继承者称为子类。下图中，动物类为父类，狗、猫、鱼类为子类。



子类是父类的特殊化，子类继承了父类的特性，同时可以对继承到的特性进行更改，也可以拥有父类没有的特性。

3.2 继承（Inheritance）

Dog继承了父类Animal的特性，并有自己的run方法，若子类中的方法与父类中的某一方法具有相同的方法名、返回类型和参数表，则新方法将覆盖原有的方法，这称之为**方法重写**。

```
#<程序：继承的格式>
class Animal:
    def run(self):
        print("run为父类的方法")

class Dog(Animal):
    def run(self):
        print("run为子类的方法")
    def eat(self):
        print("子类特有的方法")

class Cat(Animal):
    pass
```

- 子类重写了父类的run方法，调用时就执行子类的run方法
- Dog类同时也添加了一个父类没有的eat方法

Cat类完全继承了Animal的特性

3.2 继承（Inheritance）

子类重写了父类的run方法，调用时就执行子类的，Dog类同时也添加了一个父类没有的eat方法，而Cat类完全继承了Animal的特性。

```
#<程序：继承测试示例1>
dog=Dog( )
dog.run( ) #输出：run为子类的方法
dog.eat( ) #输出：子类特有的方法

cat=Cat( )
cat.run( )  #输出：run为父类的方法
```

3.2 继承（Inheritance）

将这个例子丰富一下，Animal类增加 init方法来初始化对象开始的位置信息，run方法将对象位置的x坐标增加1，表示对象在x方向上移动了1；Dog类的init方法初始化了对象的hungry属性，表示是否要吃东西。

```
#<程序：Animal类的继承>
import random as r
class Animal:
    def __init__(self):
        self.x=r.randint(0,10)
        self.y=r.randint(0,10)
    def run(self):
        self.x+=1
        print("我的位置是：",self.x,self.y)
```

```
#<程序：Animal类的继承>
class Cat(Animal):
    pass
class Dog(Animal):
    def __init__(self):
        self.hungry=True
    def eat(self):
        if self.hungry:
            print("我正在吃东西")
            self.hungry=False
        else:print("我不想吃东西")
```

3.2 继承（Inheritance）

```
#<程序：继承测试示例2>
cat=Cat()
cat.run()
cat.run()
dog=Dog()
dog.eat()
dog.run()
```

```
我的位置是：   2 6
我的位置是：   3 6
我正在吃东西
      4 dog=Dog()
      5 dog.eat()
----> 6 dog.run()

<ipython-input-17-a1bd9c97690c> in run(self)
      6             self.y=r.randint(0,10)
      7         def run(self):
----> 8             self.x+=1
      9             print("我的位置是：  ", self.x, self.y)
     10 class Dog(Animal):

AttributeError: 'Dog' object has no attribute 'x'
```

思考： 同样是继承Animal类，cat可以通过调用run方法改变自己的位置，为什么dog一开始就报错了？

3.2 继承（Inheritance）

```
#<程序：Animal类的继承>
import random as r
class Animal:
    def __init__(self):
        self.x=r.randint(0,10)
        self.y=r.randint(0,10)
    def run(self):
        self.x+=1
        print("我的位置是：",self.x,self.y)
class Cat(Animal):
    pass
class Dog(Animal):
    def __init__(self):
        self.hungry=True
    def eat(self):
        if self.hungry:
            print("我正在吃东西")
            self.hungry=False
        else:print("我不想吃东西")
```

- **报错原因**：因为Animal中的x和y变量是__init__函数中的变量，只有当执行这个函数时才能产生，并不属于类变量。Dog类只继承Animal的__init__和run方法，而函数中的x和y没有继承下来

- **怎么解决**：在Dog类的__init__函数中调用Animal的__init__方法，产生x和y变量

3.2 继承（Inheritance）

怎么解决？ 在Dog类的__init__函数中，调用Animal的__init__方法，产生x，y变量。

- **解决办法1**

```
#<程序：继承测试示例 3>
class Dog(Animal):
    def __init__(self):
        Animal.__init__(self)
        self.hungry=True
```

- 在Dog类的__init__函数中，直接调用Animal的__init__方法，产生x，y变量

- **解决办法2**

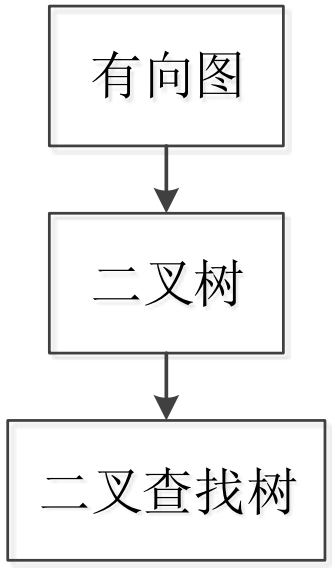
```
#<程序：继承测试示例 4>
class Dog(Animal):
    def __init__(self):
        super().__init__()
        self.hungry=True
```

- 使用super()不需要传递参数，而用父类类名的方法需要在调用的函数中传入self参数。

3.2 继承（Inheritance）

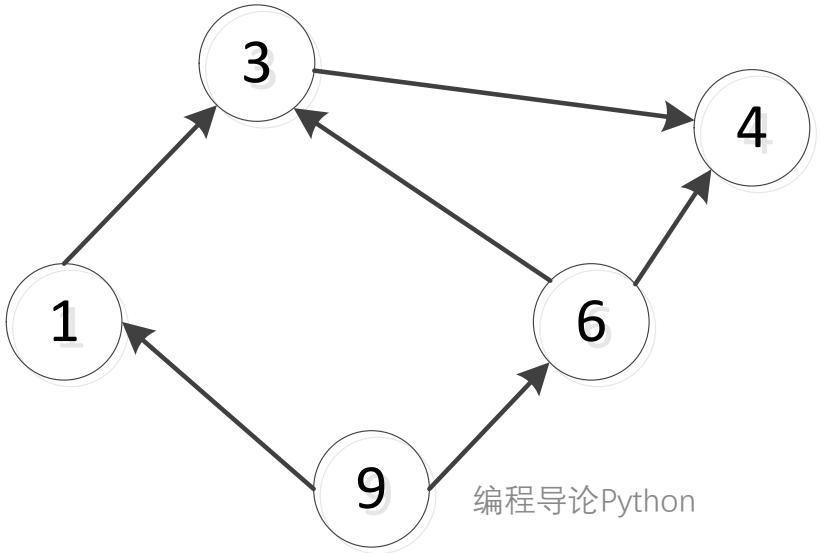
2、熟练运用继承机制编程

- **图的继承：**图的继承的类层次如下图所示，其中有向图类为二叉树类的父类，而二叉树为二叉查找树的父类。



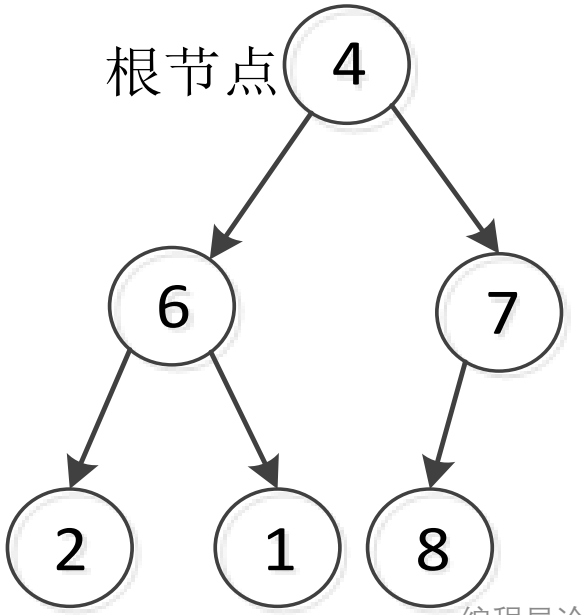
3.2 继承（Inheritance）

- **图（Graph）**是由节点（也可以称之为顶点）以及节点之间边的集合组成的，通常表示为：G(V,E)，其中，G表示一个图，V（Vertex）是图G中节点的集合，E（Edge）是图G中边的集合。
- 如果图的任意两个节点之间的边都是无向边，则称该图为**无向图**，如果图的任意两个节点之间的边都是有向边，则称该图为**有向图**。



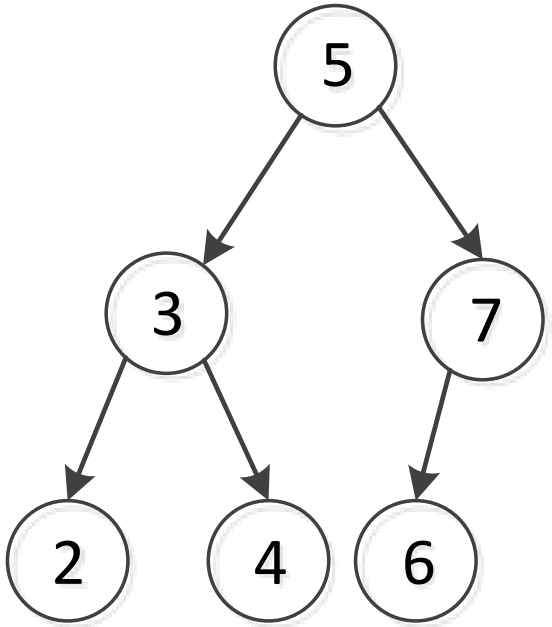
3.2 继承（Inheritance）

- **树（Tree）**也是由节点（在树中通常不称为顶点）和边组成，与图不同的是树中一定存在一个特殊的、不被任何节点所指向的节点——**根节点**，且除了树的根节点外，树中的节点只能有一个节点指向它（称为它的**父节点**）
- **二叉树** 是一种特殊的树，每个节点最多有两棵子树，称为左子树和右子树



3.2 继承（Inheritance）

- **二叉查找树 BST（Binary Search Tree）**本质上还是一棵二叉树，只不过在其上定义了一些规则：一个节点的左子树中所有的节点值都小于该节的值，而其右子树中的所有节点值都大于该节点的值。



3.2 继承（Inheritance）

图、二叉树、二叉查找树有一些相同的属性，都有节点和边。但是图中的一个节点可以和多个节点相连，而二叉树中一个节点连接的节点数量有所限制，除根节点外与每个节点直接相连的最多有两个子节点，一个父节点。二叉查找树又对二叉树进行了特殊化，对节点的左右子节点的值做了要求。

下面介绍如何用面向对象的思想描述图、二叉树、二叉查找树，以及它们之间的继承关系。

3.2 继承（Inheritance）

①首先描述一个有向图类，有向图的**静态特征**有节点和边，在有向图Graph类的构建中，我们利用两个字典结构node和edge分别记录节点和边的信息，其中node的格式为：{节点编号：节点的值}，edge的格式为：{节点编号:与该节点相邻节点的编号列表}；图的**动态特征**有：插入一个节点、判断图中是否存在具有特定值的节点、查看图。

3.2 继承（Inheritance）

- **insert()**：首先根据插入节点的顺序从小到大生成新节点的编号（node_id），并将节点的值存入到node字典中，接着更新边的信息。

- **exist_val()**：当我们需要判断图中是否存在某个值(val)时函数首先获取node字典中的所有值，进而判断val是否存在于其中，存在返回True，否则返回False。

- **show()**：实现了打印图中所有的节点信息和边信息。

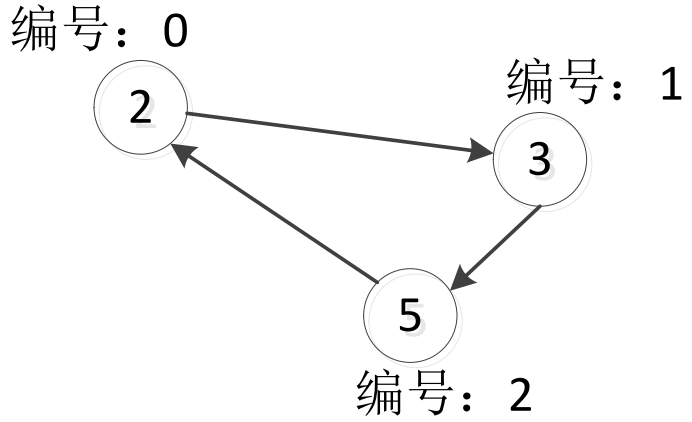
3.2 继承（Inheritance）

```
#<程序：有向图类>
class Graph:
    def __init__(self,val):
        self.node_id=0
        self.node={self.node_id:val}     #存节点的值
        self.edge={self.node_id:[]}
    def insert(self,val,L1,L2):  #L1存的该节点指向谁，L2存的是谁指向节点
        self.node_id+=1
        self.node[self.node_id] = val    #将节点记录的值存起
        for e in L2:
                self.edge[e].append(self.node_id)
        self.edge[self.node_id]=L1
    def exist_val(self,val):
        value = self.node.values() #value为字典中的所有值组成的列表
        return val in value
    def show(self):
        print("节点的信息",self.node)
        print("边的信息",self.edge)
```

3.2 继承（Inheritance）

实例化，创建一个有向图

```
#<程序：构建有向图 >
graph=Graph(2)
graph.insert(3,[],[0])#参数分别为节点值,指向哪些节点,哪些节点指向该节点
graph.insert(5,[0],[1])
graph.show()
print("值为3的节点在图中是否存在？ ", graph.exist_val(3))
```



#输出：

节点的信息 {0: 2, 1: 3, 2: 5}

边的信息 {0: [1], 1: [2], 2: [0]}

值为3的节点在图中是否存在？ True

3.2 继承（Inheritance）

②由于二叉树的静态属性和图的类似，均由节点和边组成，可以直接继承图类的属性。但图中的节点可以指向多个节点也可以被多个节点所指向，而二叉树的节点最多只能有两个子节点且只有一个父节点，因此需要重写图的insert方法，即判断插入后是否满足二叉树的条件，若满足才能执行插入操作，否则插入失败。

```
#<程序：二叉树类>
class BinaryTree(Graph):
    def insert(self,val,parent):  #parent为该节点的父节点的编号
        #首先检查是否满足二叉树的插入条件，即父节点的孩子个数是否小于2
        if len(self.edge[parent])<= 1:
            #当父节点有空位置时才能插入新节点，因此L1为空
            super().insert(val,[],[parent])
        else:print("插入失败")
```

3.2 继承（Inheritance）

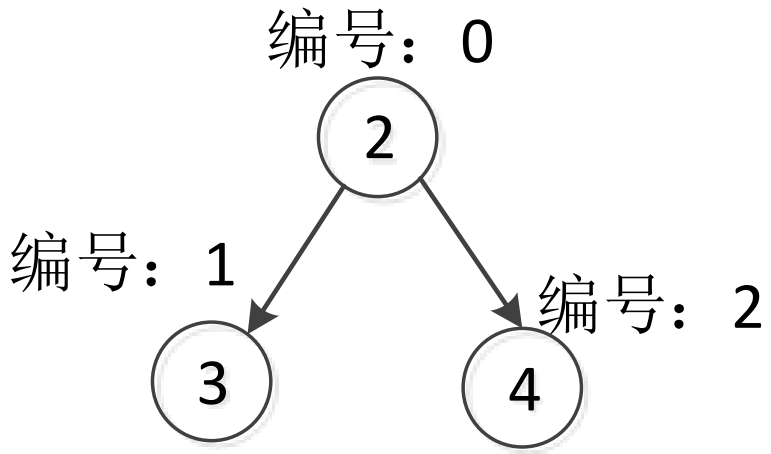
实例化，创建二叉树对象

```
#<程序：构建二叉树>
tree=BinaryTree(2)
tree.insert(3,0)  #第一个参数是节点的值，第二的参数是父节点编号
tree.insert(4,0);
tree.show()
```

#输出：

节点的信息 {0: 2, 1: 3, 2: 4}

边的信息 {0: [1, 2], 1: [], 2: []}



3.2 继承（Inheritance）

③对于二叉查找树中的节点而言，其左子树中的所有节点值都应小于该节点的值，而其右子树中的所有节点值都大于该节点的值。因此，在二叉查找树的类定义中需要区分出节点的左右子节点。

下面重新定义树的边属性：每个节点指向的子节点限定为两个，初始值为[-1,-1]，分别存储左右子树根节点的编号，需重写insert函数即可。

3.2 继承（Inheritance）

```
#<程序：二叉查找树类>
class BST(BinaryTree):
    def __init__(self,val):
        super().__init__(val)
        self.edge={self.node_id:[-1,-1]} #[-1,-1]表示没有子节点
    def insert(self,val):
      if super().exist_val(val)==0: #val没有被插入过
          self.node_id+=1
          self.node[self.node_id] =val   #将节点的值存起
          f_id,pos=self.find_pos(val,0) #返回父节点的编号及左右位置
          self.edge[f_id][pos]=self.node_id #更新父节点的边
          self.edge[self.node_id]=[-1,-1]
      else:print("该节点记录值已存在")
      ……
```

由于在插入时，首先需要根据规则为节点找一个正确的插入位置，因此这里新定义了 find_pos方法供 insert方法来调用。

3.2 继承（Inheritance）

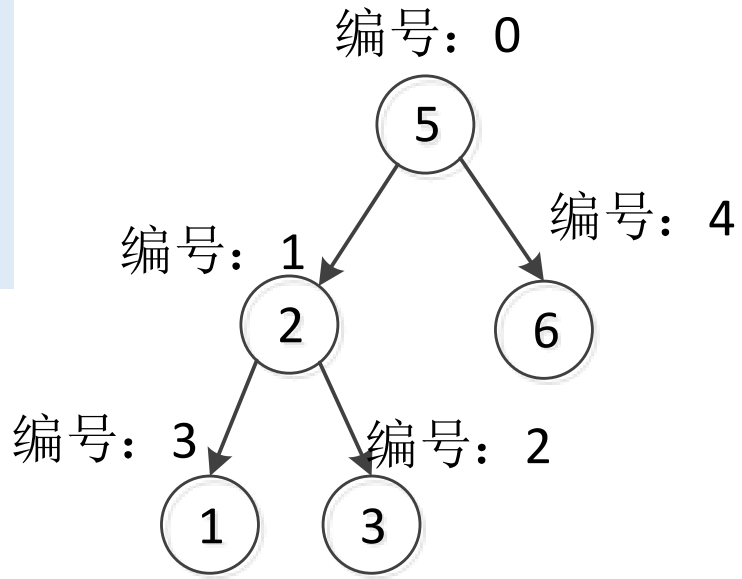
- find_pos()：当要插入的节点值小于当前节点值时，在当前节点的左子树接着找插入位置，否则，在当前节点的右子树接着找插入位置。重复这一过程，直到找到一个正确的插入位置。

```
#<程序：二叉查找树类>
    ……
     def find_pos(self,val,node_id):
            if val<self.node[node_id]: #找左子树
                if self.edge[node_id][0]==-1: #找到空位置
                    return node_id,0  #返回父节点编号,该节点为左节点
                node_id=self.edge[node_id][0] #节点左孩子节点编号
                return self.find_pos(val,node_id)
            elif val>self.node[node_id]: #找右子树
                if self.edge[node_id][1]==-1:
                    return node_id,1  #返回父节点编号,该节点为左节点
                node_id=self.edge[node_id][1] #节点右孩子节点编号
                return self.find_pos(val,node_id)
```

3.2 继承（Inheritance）

创建二叉查找树对象

```
#<程序：构建二叉查找树>
bst=BST(5)
bst.insert(2) #插入一个记录值为2的节点
bst.insert(3)
bst.insert(1)
bst.insert(6)
bst.show()
```



#输出：

节点的信息 {0: 5, 1: 2, 2: 3, 3: 1, 4: 6}

边的信息 {0: [1, 4], 1: [3, 2], 2: [-1, -1], 3: [-1, -1], 4: [-1, -1]}

3.3 多态（Polymorphism）

- 什么是**多态**？

多态是多种表现形态的意思。它是一种机制、一种能力，而非某个关键字。在面向对象的编程中，多态在类的继承和类的方法调用中得以体现。多态指的是在不清楚对象的具体类型时，根据引用对象的不同而表现出不同的行为方式。

3.3 多态（Polymorphism）

- **对象中多态的表现**：Dog类和Cat类，它们都有方法move，分别创建了dog对象和cat对象，调用move方法，然后根据其类型的不同，会表现出不同的行为。

```
#<程序：对象中方法的多态表现>
from random import choice
class Dog:
    def move(self):
        print("飞快地跑！")
class Cat:
    def move(self):
        print("慢悠悠地走。")
dog=Dog()
cat=Cat()
obj=choice([dog,cat]) #choice函数实现从列表中随机选择一个元素
print(type(obj))  #type函数可以查看对象类型
obj.move()#若obj为Cat，输出慢悠悠地走，若obj为Dog，输出飞快地跑！
```

3.3 多态（Polymorphism）

- **函数和运算符多态的表现**

```
#<程序：“+”的多态表现>
a=1
b=2
print(a+b) #输出：3

a="Hello "
b="world!"
print(a+b) #输出：Hello world!
```

```
#<程序：count函数多态>
>>> from random import choice
>>> x=choice(["Hello world!",["o","a",1,2]])
>>> x.count("o")
1
```

3.3 多态（Polymorphism）

- Python和其他静态形态检查类的语言（如C++等）不同，在参数传递时不管参数是什么类型，都会按照原有的代码顺序执行，这就很可能会出现因传递的参数类型错误而导致程序崩溃的现象。

```
#<程序：检查是否为奇数>
def is_odd_number(a):
    return not a%2==0
a=3
print("Is it an odd number?", a, is_odd_number(a))
#输出：Is it an odd number? 3 True
a=2.1
print("Is it an odd number?", a, is_odd_number(a))
#输出： Is it an odd number? 2.1 True #没报错，但结果是错的
a=[3]
print("Is it an odd number?", a, is_odd_number(a))#程序报错
```

思考：怎样避免这种因传递的参数类型错误而导致程序崩溃的现象?

3.3 多态（Polymorphism）

安全起见，需要自己编写代码来检验参数类型的正确性。

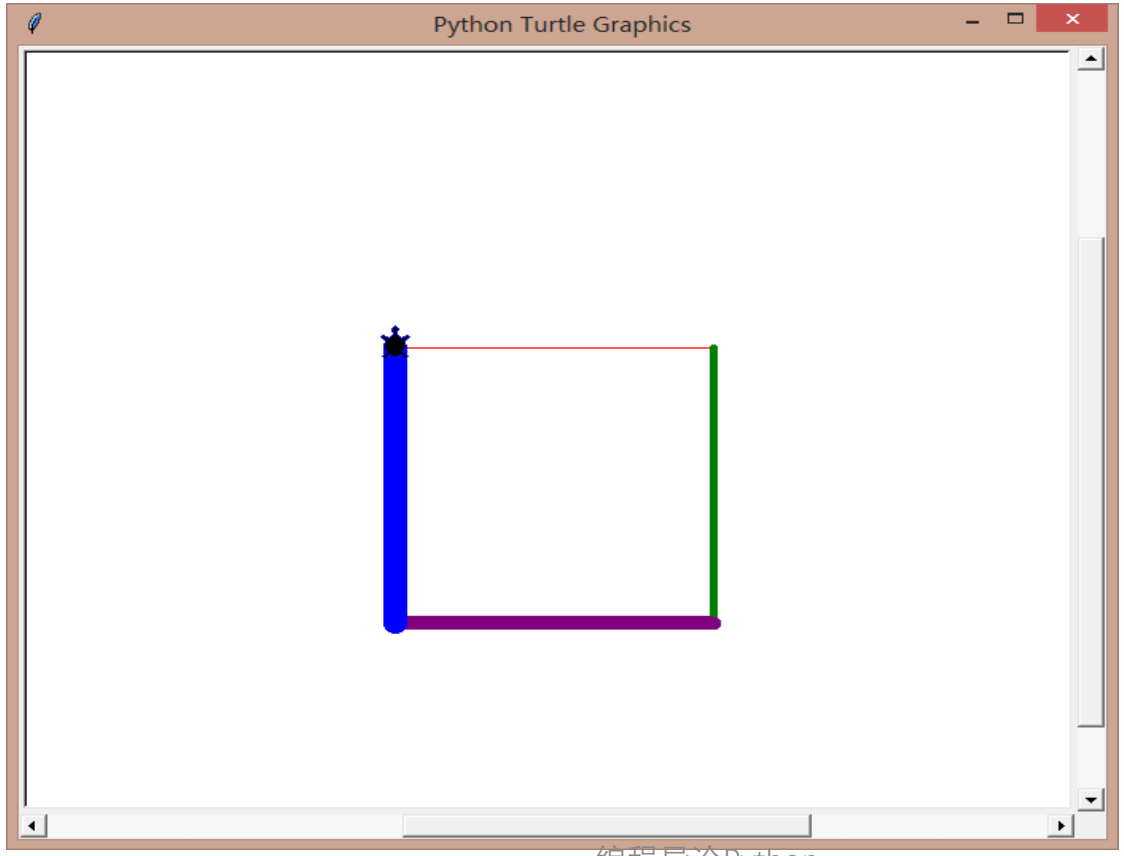
```
#<程序：检查是否为奇数前，检查数据类型>
def is_odd_number1(a):
    if type(a)!=type(1): return -1  #类型不合法，直接返回-1
    return not a%2==0
a=2.1
print("Is it an odd number?", a, is_odd_number1(a))
#输出Is it an odd number? 2.1 -1
a=[3]
print("Is it an odd number?", a, is_odd_number1(a))
#输出Is it an odd number? [3] -1
```

4 初识小乌龟

- 4.1 小乌龟的属性
- 4.2 基本图形的绘制
- 4.3 递归图形的绘制

4.1 小乌龟的属性

小乌龟（turtle）是Python提供给开发者的一个绘图的标准库，可以利用小乌龟绘画出各种各样的有趣的图形。



4.1 小乌龟的属性

1、导入turtle模块

turtle是Python标准库中的模块，有以下两种导入方式，这两种方式都会将turtle中的所有方法导入:

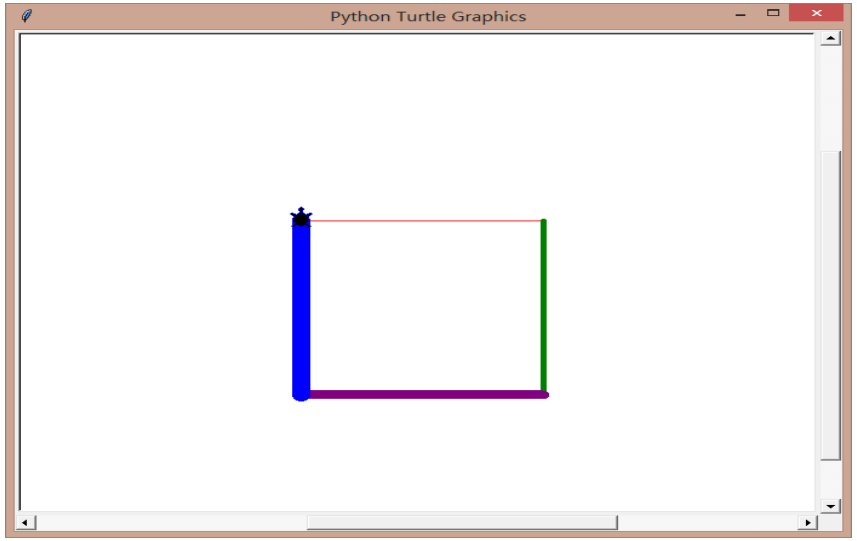
①from turtle import *

②import turtle

4.1 小乌龟的属性

2、画布的建立

turtle中画布的生成是自动的，用函数turtle.screensize()调整画布的大小，函数格式为：turtle.screensize(width,height)，其中width和height分别是画布的宽度和高度。



4.1 小乌龟的属性

3、小乌龟对象的创建

turtle中的画笔就是前面讲到的“小乌龟”，程序语句是p=Turtle()，这条语句的作用正是实例化一个小乌龟对象。实例化之后小乌龟的初始方向是水平向右的。

4.1 小乌龟的属性

4、小乌龟的形态

画笔是以小乌龟的图案出现的，叫做小乌龟形态。实际上画笔是有两种形态的：一种是小乌龟的形态，另一种则是箭头的形态。

使用turtle.shape("turtle") 时，画布上会出现一只小乌龟；

使用turtle.shape("classic")时，画布上会出现一个箭头；

当不希望任何形态的小乌龟出现在屏幕上时，用函数turtle.hideturtle()来隐藏小乌龟。

4.1 小乌龟的属性

5、小乌龟的画笔属性

函数pencolor()用来设置画笔的颜色，它的输入格式为：pencolor(colorstring)。例如pencolor("brown")，括号内填写字符串来表示颜色。

除了颜色以外，还可以利用pensize(x) 函数设置画笔粗细，其中参数x是一个表示画笔粗细的正数。例如p.pensize(1)将画笔的粗细设置为1。

4.1 小乌龟的属性

6、小乌龟的运动命令

forward(distance)：可以让小乌龟依照它当前所朝方向移动distance长度的距离，distance可以是整数或浮点数。

三个改变方向的函数:

- right(angle)：将小乌龟向右转angle角度，角度的单位默认是度数，也可以设置为弧度。
- left(angle)：将小乌龟向左转angle角度，角度的单位默认是度数，也可以设置为弧度。
- setheading(to_angle)：将小乌龟的方向设置为某个角度，其中0度的方向为水平向右，当to_angle为正角度（从0°开始逆时针旋转的度数）时，小乌龟为逆时针旋转。

4.1 小乌龟的属性

```
#<程序：小乌龟画正方形>
from turtle import *          #第1条语句
screensize(1600,800); p=Turtle() #第2~3条语句
p.shape("turtle"); p.pencolor("red"); p.pensize(1) #第4~6条语句
p.forward(200); p.right(90); p.pencolor("green"); #第7~9条语句
p.pensize(5); p.forward(200); p.left(270); #第10~12条语句
p.pencolor("purple"); p.pensize(10); p.forward(200) #第13~15条语句
p.setheading(90); p.pencolor("blue"); p.pensize(15) #第16~18条语句
p.forward(200) #第19条语句
```

4.1 小乌龟的属性

7、小乌龟的位置属性

在不知道当前位置的情况下，是否能直接移动小乌龟到期望的位置呢?

可以的。利用二维平面直角坐标系，原点(0,0)是画布的中心，水平向右是x轴的正方向，竖直向上是y轴的正方向，小乌龟默认会出现在坐标系原点，同时turtle提供了一些函数可以使小乌龟移动到期望的坐标点上:

turtle.pos()：返回小乌龟的当前位置（x，y）；

turtle.setpos(x,y)：使小乌龟移动到设定的坐标点上;

setx(x)：函数将小乌龟的横坐标设置为x，纵坐标保持不变;

setx(y)：函数将小乌龟的纵坐标设置为y，横坐标保持不变。

4.1 小乌龟的属性

小乌龟初始位置应在（0,0）点，然后将小乌龟分别移动到正方形的右上角(200,0)、右下角(200,-200)、左下角(0,-200)、左上角(0,0)，在移动的过程中小乌龟得到了一个完整的正方形：

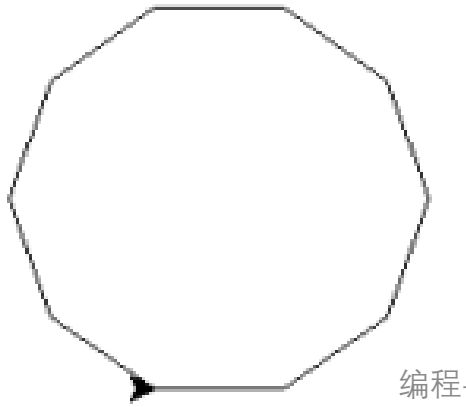
```
#<程序：小乌龟画正方形四条边（绝对位置法）>
from turtle import *
p=Turtle()
p.setpos(100,0) #小乌龟移动到正方形的右上角
p.setpos(100,-100)        #小乌龟移动到正方形的右下角
p.setpos(0,-100)  #小乌龟移动到正方形的左下角
p.setpos(0,0)             #小乌龟移动到正方形的左上角
```

4.2 基本图形的绘制

1、正多边形的绘制

①正方形的绘制方法：先用forward()函数画一边；然后向左转90度，然后forward()画第二条边；再向左转90度，forward()画第三条边；最后再向左转90度，然后forward()画第四条边。

②以此类推，任意边数的正多边形的绘制可以用一个循环k次的for循环来实现，而每次所要转动的角度其实就是正多边形的外角，即360/k。



4.2 基本图形的绘制

Screen()：该函数返回TurtleScreen类的一个对象。运行语句s = Screen()后得到当前屏幕对象。

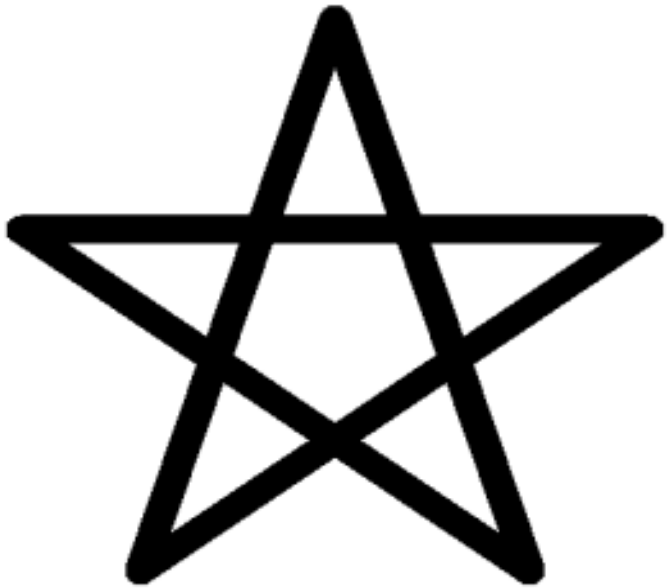
exitonclick()：当屏幕上发生鼠标点击后，关闭turtle图形窗口。

```
#<程序：画正多边形>
from turtle import *
def jumpto(x,y):
        up(); goto(x,y); down()
reset()
jumpto(-25,-25)
k=10
for i in range(k):
        forward(50)
        left(360/k)
s = Screen(); s.exitonclick()
```

4.2 基本图形的绘制

2、五角星的绘制

在纸上画五角星时，通常是先画水平边，再向左下角移动画第二条边，再向右上移动画出第三条边，再向右下角画出第四条边，最后向左上角画出第五条边。



4.2 基本图形的绘制

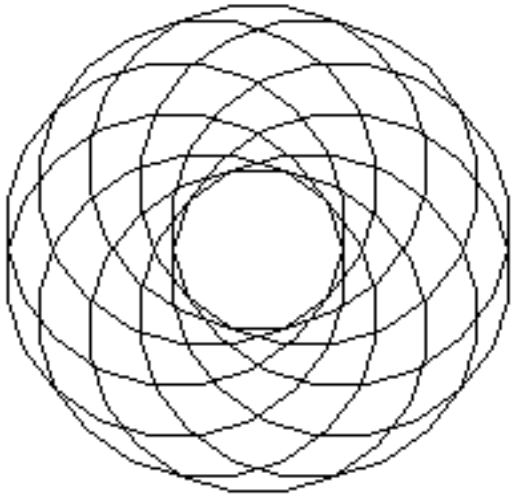
```
#<程序：五角星的绘制>
import turtle
turtle.pensize(20)
turtle.pencolor("black")
turtle.setheading(0)
length = 400
angle = 0
for i in range(5):
    turtle.forward(length)
    angle = angle - 144
    turtle.setheading(angle)
```

小乌龟的初始方向水平向右，所以可以先forward()画出水平的那条边，然后向右转动一个外角大小的角度（144°）；此时小乌龟指向了左下角的顶点，forward()画出第二条边，然后向右再转动一个外角大小的角度（144°）；此时小乌龟指向了上方的顶点，forward()画出第三条边，然后向右再转动一个外角大小的角度（144°）……这样循环五次，直到画出全部的五条边。

4.2 基本图形的绘制

3、聚合图案的绘制

聚合图案指由多个相同的图形构成的图案，下图是由20个半径相同的圆组成的圆环，每个圆的周长都是400像素，圆环最内侧的小圆的周长是200像素。这个复杂的图案是如何绘制的呢?



Click to exit

4.2 基本图形的绘制

①先画初始圆，可以用边数很多的正多边形来近似画一个圆，此处选择正四十边形，代码与前面讲的正多边形的绘制类似。

②在内圆上顺时针旋转初始圆的圆心，内圆也可以用正多边形来模拟，因为有20个圆所以可以用正二十边形来画内圆，由于内圆周长为200，所以边长为200/20。

③将上面两步整合，外层循环用来旋转圆，将循环20次；内层循环用来画圆，将循环40次。

```
#<程序：画初始圆>
IN_TIMES = 40
for j in range(IN_TIMES):
    right(360/IN_TIMES)
    forward (400/IN_TIMES)
```

4.2 基本图形的绘制

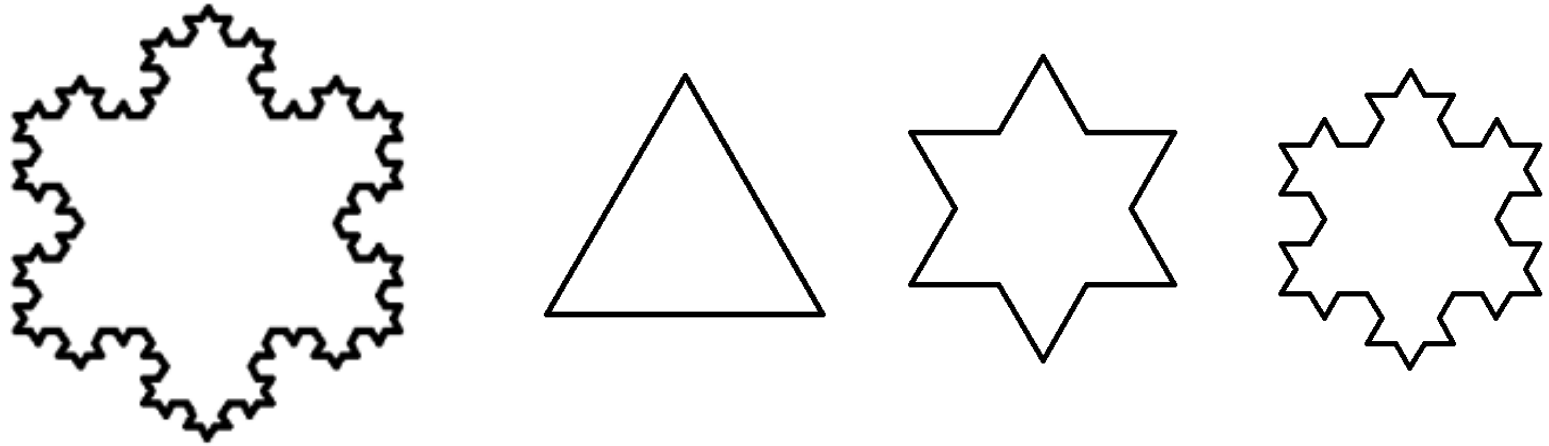
```
#<程序：多个圆形的聚合>
from turtle import *
speed('fast');IN_TIMES = 20;TIMES = 10
for i in range(TIMES):
    right(360/TIMES);forward(200/TIMES)
    for j in range(IN_TIMES):
        right(360/IN_TIMES);forward (400/IN_TIMES)
penup();setpos(-100,-200)
write("Click to exit", font = ("Courier", 12, "bold") )
s = Screen();s.exitonclick()
```

speed()：用来控制画图速度，参数可以是0到10的任意数字，也可以是字符串（包括“fastest”: 0；“fast”: 10；“normal”: 6；“slow”: 3；“slowest”: 1）。

write(arg, move=False, align="left", font=("Arial", 8, "normal"))：

4.3 递归图形的绘制

1、雪花的绘制



“雪花”是一种名为科赫曲线的几何曲线，因为形似雪花所以又称为雪花曲线。科赫曲线可以由以下步骤生成:

（1）画一个正三角形，如图a所示;

（2）把每一边三等分，并以三等分后的中间一段为边向外作正三角形，并把这“中间一段”擦掉，得到图b;

（3）重复第（2）步，画出更小的三角形，得到图c;

（4）一直重复第（2）步，所画出的曲线叫做科赫曲线。给出的雪花图案是重复3次后形成的曲线。

4.3 递归图形的绘制

如何用小乌龟画出这个曲线?

一种方法是：按照上面的四步，把每一边三等分，以三等分后的中间一段为边向外作正三角形，并把这“中间一段”擦掉。

另一种方法：先确定最终的曲线是什么样的并用某种数据结构把曲线的形状保存下来，然后依照这个结构直接画出这个曲线。

4.3 递归图形的绘制

①首先保存初始三角形的形状，假设初始三角形边长为size=243，用字符串"sftftf"来表示初始三角形，其中s表示左转60°，f表示向前243，t表示右转120°。函数的返回值是"sftftf"。

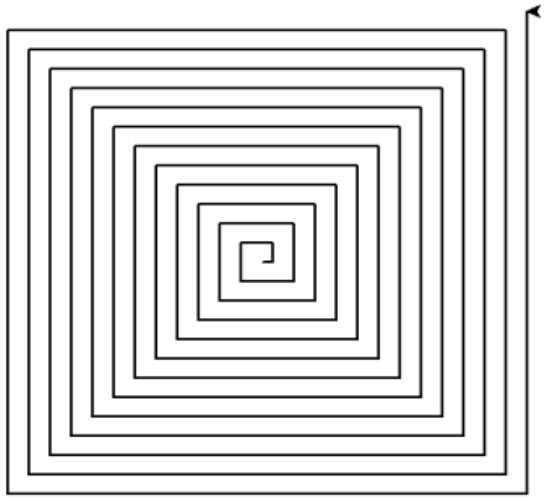
②递归函数每次重复都会将表示当前形状的字符串中的字符“f”用“fsftfsf”替换，该替换操作等同于将当前形状中每一边三等分、以三等分后的中间一段为边向外作正三角形、并把这“中间一段”擦掉。函数的返回值是新字符串。

4.3 递归图形的绘制

```
#<程序：雪花的绘制>
from turtle import *
import time
def draw(s,size):
    for i in s:
        if i == 's': t.left(60)
        elif i == 'f': t.forward(size)
        else: t.left(-120)
def koch_curve(n):
    if n==1: return "sftftf"
    else: return koch_curve(n-1).replace("f","fsftfsf")
t=Pen();size=243
for i in range(1,5):
    time.sleep(3); t.reset();time.sleep(1)
    t.penup();t.goto(-200,200);t.pensize(5)
    t.pendown();t.write(i,font=("Arial", 30, "normal"))
    t.hideturtle();t.penup();t.goto(-120,-70)
    t.pendown();s=koch_curve(i);draw(s,size);size=size/3
```

4.3 递归图形的绘制

2、螺旋线的绘制



图中螺旋线由50条线段组成，其中最短的线段长为8，接下来每一段都比上一段长8，每两条相邻平行线的间隔也是8。这个螺旋线也可以递归的绘制出，思路如下：

①初始时小乌龟在画布中心向前移动8（即forward(8)）绘制出第一条线段；

②递归函数每次都在前面绘制完的基础上左转90°，然后向前移动比上一次移动的距离还要长8的距离。

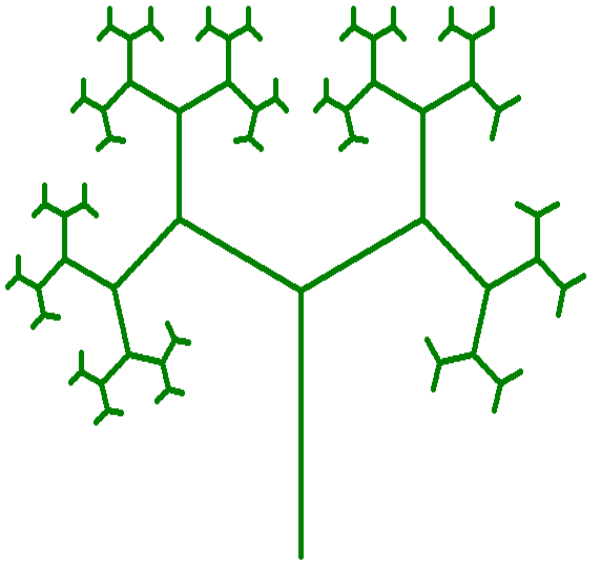
4.3 递归图形的绘制

函数draw(n)用于绘制有n条边的螺旋线，该函数会先递归调用draw(n-1)绘制出有n-1条边的螺旋线，在此基础上左转90°绘出第n条边。

```
#<程序：螺旋线的绘制>
from turtle import *
speed("fastest")
pensize(2)
def draw(n):
    if n==1:
        forward(8*n);left(90)
    else:
        draw(n-1);forward(8*n);left(90)
draw(50)
```

4.3 递归图形的绘制

3、树



树也是用小乌龟递归的画出的，其中树根的长度为200，每一层分支的长度都是上一层分支长度的0.6375，最短的分支的长度大于5。绘制思路如下：先用一个画笔画出最下面的树根；再将当前这一个画笔克隆成两个画笔，在树根的基础上，用这两个画笔画出第二层的两个分支；再将当前这两个画笔克隆成四个画笔，在第二层分支的基础上画出第三层的四个分支……一直画到使最后一次分支的长度小于5为止。

4.3 递归图形的绘制

绘制函数是tree(plist, l, a, f)，plist是存有当前所有画笔的列表，l表示当前分支的长度，a表示两个分支之间夹角度数，f是分支长度的缩短率。

```
#<程序：树的绘制>
from turtle import Turtle
def tree(plist, l, a, f):
    if l > 5:
        lst = []
        for p in plist:
            p.forward(l);q = p.clone() #复制一个与画笔p属性一样的画笔q
            p.left(a);q.right(a);lst.append(p);lst.append(q)
        tree(lst, l*f, a, f)
def main():
    p = Turtle();p.color("green");p.pensize(5)
    p.hideturtle();p.speed(1);p.left(90)
    p.penup();p.goto(0,-200);p.pendown()
    t = tree([p], 200, 65, 0.6375)
main()
```

5 多个小乌龟的动图绘制

- 5.1 过河游戏
- 5.2 小老鼠走迷宫

5.1 过河游戏

1、画河与两岸

河与两岸可以简单的用两条竖线表示，思路就是：将小乌龟移到画布右上角，竖直向下画出一条竖线作为右岸，再将小乌龟移到画布左上角，竖直向下画出一条竖线作为左岸。

根据这个思路写出画河与两岸的函数SetupRiver()，其中SetupRiver()函数将会建立河的外框，main()会调用SetupRiver()函数，从而使小乌龟在画布上画出河与两岸。

5.1 过河游戏

```
#<程序：过河游戏-画河与两岸>
from turtle import *
def SetupRiver():                #建立河的外框
    pensize(6);penup();goto(300,300)
    pendown();right(90);forward(500)
    penup();goto(-300,300);pendown();forward(500)
def main():
    tracer(False)    #使多个绘制对象同时显示
    SetupRiver()
    tracer(True)
if __name__ == "__main__":
    main()
```

tracer(False)：关闭动画开关，使接下来的绘图都不显示绘制的过程，直接显示绘制结果。

tracer(True)：打开动画开关，使接下来的绘图都显示绘制的过程，从而形成动态的图像。

5.1 过河游戏

2、在河上画小船

①利用s=Shape(“compound”)创建一个类型为“compound”的空Shape对象，“compound”表明这是一个复合Shape对象。接下来给这个对象添加小船多边形：分析它所有顶点的二维坐标是多少，用一个元组顺序存储所有坐标，这个元组就代表了小船的形状。然后用addcomponent函数将这个代表多边形的元组添加到空Shape对象中，setboat(name)函数会将我们创建的小船形状命名为name。

②函数Init()中用语句boat = Turtle()创建了一个小乌龟对象boat，用语句boat.shape("boat")将小乌龟boat的形状设置为我们自定义的图形“boat”。main()函数会调用Init()使小船出现在画布上，同时main()函数也调用了SetupRiver()函数，故河与小船会同时出现在画布上。

5.1 过河游戏

```
#<程序：过河游戏-画小船>
from turtle import *
def SetupRiver():    #函数体与<程序：过河游戏-画河与两岸>相同，故省略
def main():                          #修改main()函数
    tracer(False)        #使多个绘制对象同时显示
    SetupRiver();  Init();  tracer(True)
def setboat(name):  #定义函数setboat(name)
    s = Shape("compound")
    poly1=((-229,270),(-250,295),(-250,220),
           (-325,220),(-260,195),(-260,220),(-250,220),
           (-250,145),(-225,170),(-225,270))#存储小船顶点的元组
    s.addcomponent(poly1, "white", "black")
    register_shape(name, s)
def Init():                          #定义函数Init()
    global boat
    setboat(“boat”);boat = Turtle();boat.shape("boat")
if __name__ == "__main__":
    main()
```

5.1 过河游戏

3、确定两岸的菜鸡狼

首先要确定菜鸡狼在不同时间所处的位置，画出三个地方的菜鸡狼——左岸、船上和右岸，在实现前面的过河游戏之后，得到的结果是存储在列表中的字串，如下所示:[["狼,鸡,菜"], ["船"],[],True, ["狼,菜"], ["船,鸡"],[],False,……]，以此作为输入赋值给列表animals，animals中每四个元素构成了游戏过程中的一个状态，例如：["狼,鸡,菜"]、["船"]、[]、True四个元素表示河的左岸是菜鸡狼，小船上为空，右岸为空，小船当前方向是右岸到左岸，而["狼,菜"]、["船,鸡"]、[]、False表示河的左岸是狼菜，小船上有鸡，右岸为空，小船当前方向是左岸到右岸。因此每次只需要获取animals中的连续四个元素，故设计了函数animal(t)，t表示需要获取第几个状态的数据，函数将返回第t个状态的数据，即animals[4*t:4*t+4]。

5.1 过河游戏

```
#<程序：过河游戏-确定两岸的菜鸡狼>
def animal(t):       #定义函数animal(t)
    return animals[4*t:4*t+4]
def Init(): #在Init()中添加如下代码:
            …
    global animals
    animals = [["狼,鸡,菜"], ["船"],[],True,
           ["狼,菜"], ["船,鸡"],[],False,
           ["狼,菜"], ["船"],["鸡"],True,
           ["狼"], ["船,菜"],["鸡"],False,
           ["狼"], ["船,鸡"],["菜"],True,
           ["鸡"], ["船,狼"],["菜"],False,
           ["鸡"], ["船"],["狼,菜"],True,
           [], ["船,鸡"],["狼,菜"],False,
           [], ["船"],["狼,鸡,菜"],False]
            …
```

5.1 过河游戏

4、画菜鸡狼

同时画出菜鸡狼在三个地方——左岸、船上和右岸的分布，需要创建三个turtle对象。

```
#<程序：过河游戏-画两岸的菜鸡狼>
def Init():  #在Init()中添加如下代码段
    global printer,printer1,printer2
    printer = Turtle()  #建立输出文字Turtle——printer
    printer.hideturtle()
    printer.penup()
    printer1 = Turtle()  #建立输出文字Turtle——printer1
    printer1.hideturtle()
    printer1.penup()
    printer2 = Turtle()  #建立输出文字Turtle——printer2
    printer2.hideturtle()
    printer2.penup()
```

5.1 过河游戏

5、动态显示

最后让所有的turtle对象动起来：按照输入的列表控制各个turtle对象，例如animal(t)返回的第一组数据是[“狼,鸡,菜”]、[“船”]、[]、True，那么表示左岸动物的turtle要写文字“狼，鸡，菜”，表示小船的turte写文字“船”，表示右岸的turtle不写文字，小船turtle从画布中河的右岸移向左岸这部分动态功能由函数Tick()来实现。

```
#<程序：过河游戏-动态显示>
def Tick():
    animal_len=len(animals)//4
    for i in range (animal_len):
        tmpanimal=animal(i)
        tracer(False)#不显示绘制的过程，直接显示绘制结果
        printer.reset();printer.hideturtle();printer.up()
        printer.goto(-350,190)
         ……
```

5.1 过河游戏

```
……
if len(tmpanimal[0])>0:
            printer.write(tmpanimal[0][0],align="center",
font=("Courier",14,"bold"))
        printer.down();printer1.reset();printer1.hideturtle()
        printer1.up();printer1.goto(350,190)
        if len(tmpanimal[2])>0:
            printer1.write(tmpanimal[2][0],align="center",
font=("Courier",14,"bold"))
        printer1.down();tracer(True)
        if tmpanimal[3]==True:
            time.sleep(2);boat.up();boat.goto(-441,-0)
            printer2.reset();printer2.hideturtle()
            printer2.up();printer2.goto(-250,190)
            if len(tmpanimal[1])>0:
                printer2.write(tmpanimal[1][0],align="center",
                font=("Courier",14,"bold"))
……
```

5.1 过河游戏

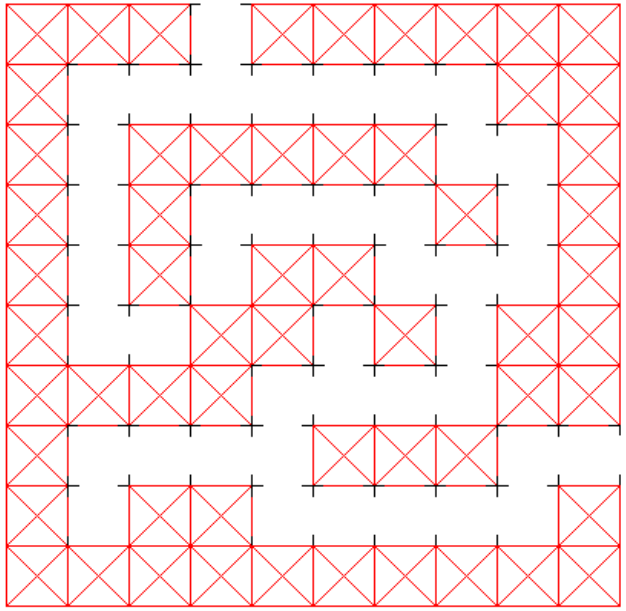
```
……
            time.sleep(2);tmp=animal(i+1)
            printer2.clear();printer.clear()
            if len(tmp[1])>0:
                printer2.write(tmp[1][0],align="center",
                font=("Courier",14,"bold"))
            if len(tmp[0])>0:
                printer.write(tmp[0][0],align="center",
                font=("Courier",14,"bold"))
        elif tmpanimal[3]==False:
            boat.up();boat.goto(0,0);printer2.reset()
            printer2.hideturtle();printer2.up()
            printer2.goto(250,190)
            if len(tmpanimal[1])>0:
                printer2.write(tmpanimal[1][0],align="center",
                font=("Courier",14,"bold"))
            time.sleep(2);tmp=animal(i+1)
……
```

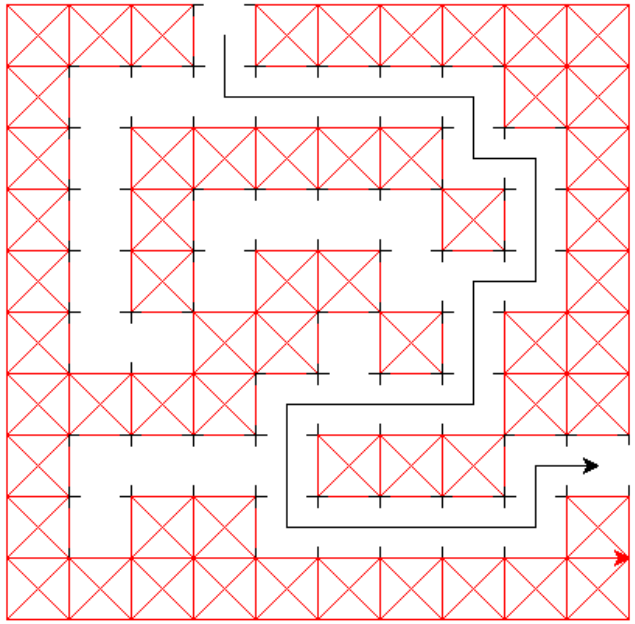
5.1 过河游戏

```
……
            printer2.clear()
            printer1.clear()
            if len(tmp[1])>0:
                printer2.write(tmp[1][0],align="center",
                font=("Courier",14,"bold"))
            if len(tmp[2])>0:
                printer1.write(tmp[2][0],align="center",
                font=("Courier",14,"bold"))
        time.sleep(2)
```

5.2 小老鼠走迷宫

前面章节讲了小老鼠走迷宫游戏，本小节将以动图的方式展示小老鼠走迷宫的动态过程。





5.2 小老鼠走迷宫

1、迷宫的绘制

列表中的每一个1或0代表了迷宫中的一个方块，根据当前方块所在行数与列数，计算出每个方块各自的起始位置，然后就可以绘制方块了。如果当前方块的表示为1则画墙，否则则画通道。其中墙用一个正方形加“×”表示，通道用没有封闭的正方形表示。

```
#<程序：迷宫输入>
m=[[1,1,1,0,1,1,1,1,1,1],
[1,0,0,0,0,0,0,0,1,1],
[1,0,1,1,1,1,1,0,0,1],
[1,0,1,0,0,0,0,1,0,1],
[1,0,1,0,1,1,0,0,0,1],
[1,0,0,1,1,0,1,0,1,1],
[1,1,1,1,0,0,0,0,1,1],
[1,0,0,0,0,1,1,1,0,0],
[1,0,1,1,0,0,0,0,0,1],
[1,1,1,1,1,1,1,1,1,1]]
```

5.2 小老鼠走迷宫

绘制迷宫的函数draw_myth()，它会对输入的迷宫列表进行遍历，根据遍历到的1或0调用drawBox(x, y, size, blocked)函数绘制相应的墙或通道。

```
#<程序：迷宫中的墙与通道绘制>
def jumpto(x,y):
    up();goto(x,y); down()
def drawBox(x,y,size,blocked):
    color("black"); jumpto(x,y)
    if blocked:
        color("red")
        for i in range(4): forward(size); right(90)
        goto(x+size,y-size); jumpto(x,y-size); goto(x+size,y)
    else:
        for i in range(4):
            forward(size/6); up(); forward(size/6*4); down()
            forward(size/6); right(90)
……
```

5.2 小老鼠走迷宫

```
#<程序：迷宫中的墙与通道绘制>
……
from turtle import *
def draw_myth():
    global m;reset();speed('fast');size=40
    for i in range(0,len(m)):
        for j in range(0,len(m[i])):
            drawBox(-200+j*size,200-i*size,size,m[i][j])
```

5.2 小老鼠走迷宫

2、绘制小老鼠走迷宫

小老鼠每搜索一个方块，就走到相应点的位置，此时会留下黑色的运动轨迹；当发现当前方块不可行时，就走回当前方块的上一个方块（即遍历当前方块之前遍历的方块），此时会留下白色的轨迹，之前的错误轨迹就被擦除。

```
#<程序：小老鼠走迷宫绘制>
sta1=0;sta2=3;fsh1=7;fsh2=9; success=0;path=[];size=40
r=(size-10)/2;global mouse
mouse= Turtle()
x=-200+3*size+size/2;y=200-0*size-size/2
mouse.up();mouse.goto(x,y);mouse.speed(1);mouse.down()
……
```

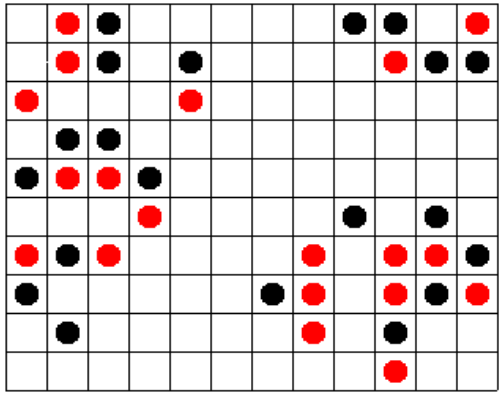
5.2 小老鼠走迷宫

```
#<程序：小老鼠走迷宫绘制
……
  from turtle import *
  def LabyrinthRat():
      global m
      print("显示迷宫：")
      for i in range(len(m)):
          print(m[i])
      print("入口：m[%d][%d]：出口
  m[%d][%d]"%(sta1,sta2,fsh1,fsh2))
      if (visit(sta1,sta2,sta1,sta2))==0:
          print("没有找到出口")
      else:print("显示路径：")
      for i in range(10):print(m[i])
  ……
```

5.2 小老鼠走迷宫

```
#<程序：小老鼠走迷宫绘制
……
  def visit(i,j,p,q):
      global m;m[i][j]=2;x=-200+j*size+size/2;y=200-i*size-size/2;
      mouse.pencolor("black");mouse.goto(x,y)
      global success,path;path.append([i,j])
      if(i==fsh1)and(j==fsh2): success=1
      if(success!=1)and(m[i-1][j]==0): visit(i-1,j,i,j)
      if(success!=1)and(m[i+1][j]==0): visit(i+1,j,i,j)
      if(success!=1)and(m[i][j-1]==0): visit(i,j-1,i,j)
      if(success!=1)and(m[i][j+1]==0): visit(i,j+1,i,j)
      if success!=1:
          m[i][j]=3; x=-200+q*size+size/2;
          y=200-p*size-size/2;mouse.pencolor("white");
          mouse.goto(x,y)
      return success
  if(__name__=="__main__"):
              tracer(False);draw_myth();tracer(True);LabyrinthRat()
              print(path)
```

练习题 6.1 Life



[问题描述] Life（生命游戏）又称细胞自动机，这个游戏由一个二维矩形世界构成，矩阵中的每个方格居住着一个活着或死了的细胞。一个细胞在下一个时刻的生死取决于相邻八个方格中活着或死了的细胞的数量，具体规则如下：

（1）“人口过少”：任何活细胞如果活邻居少于2个，则死掉。

（2）“正常”：任何活细胞如果活邻居为2个或3个，则继续活。

（3）“人口过多”：任何活细胞如果活邻居大于3个，则死掉。

（4）“繁殖”：任何死细胞如果活邻居正好是3个，则活过来。

现在请用小乌龟绘制细胞自动机的游戏过程。

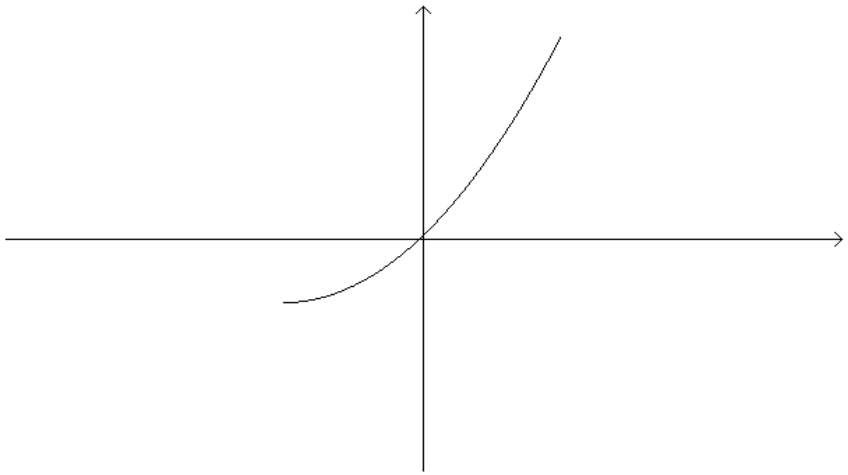
练习题6.1 Life

[解题思路]根据上述规则可以看出，这个游戏会一直循环下去，那么小乌龟的绘制也会随之循环下去。具体绘制思路如下：

首先绘制一个二维矩阵世界，假设它由ROW行、COL列个方块组成，那么就绘制一个ROW行、COL列的二维方格表；然后再随机的产生这个二维矩阵世界的初始状态，也就是在矩阵方格中随机产生活细胞（每个方格中可能产生一个活细胞或者没有活细胞），用1表示有活细胞的方块，0表示没有活细胞的方块，把所有1和0用一个二维列表存储起来，接下来根据这个列表把活细胞在画布的二维矩阵上画出；根据当前二维矩阵世界的状态，依照上述四条规则，计算出下一时刻二维矩阵世界的状态，把下一时刻的状态存储到列表中，根据这个列表把活细胞在画布的二维矩阵上画出；下下一时刻又会重复这一过程……就这样循环下去。

练习题6.2：函数图像的绘制

[问题描述] 写一个turtle程序，用户希望画出输入函数的图像，其中x轴的范围是（-100,100），y轴范围不限制，用户输入对应项系数来表示函数。例如用户希望画出$y=x^2+2*x$的图像，那么在键盘输入对应项系数“0，2，1”即可，输入的系数可以是浮点数也可以是整数，第一个数表示x^0项的系数，最后一个数是最高次项的系数；回车后程序将画出函数$y= x^2+2*x$的图像，程序将画出函数图像。



练习题6.2：函数图像的绘制

[解题思路]该程序的具体实现过程如下：

首先要画出x轴和y轴，建立好坐标系。画x轴比较简单，就是画一条范围是（-100,100）的水平线，再在x轴正方向画出箭头；画y轴之前要先计算出y轴的范围，然后画出相应范围的竖直线（此处事先计算出函数的y值范围，是为了根据y值范围动态调整y轴长度，避免部分函数图像超过画布），再在y轴正方向画箭头，即完成xy轴的绘制。

画好了坐标系，接下来要在坐标系上画出函数图像了。函数图像的绘制可以用点连线的方法:先把小乌龟放在函数x值取-100的点处，然后再让小乌龟移动到函数值x取-99的点处，就这样把小乌龟向每一个点依次移动，最后让小乌龟移动到函数值x取100的点处，这样小乌龟的移动轨迹近似是函数图像。