从面向过程走向面向对象

早期的计算机编程方法很自然地应用了面向过程的编程方法，也称作结构化程序设计方法。这种方法天生地特别适用于解决科学计算类问题。比如，先输入用于计算的数据，根据计算机需求逐步计算机，最后输出计算结果。计算步骤之间具有很强的连续性、逻辑性，所以早期使用的机器语言、汇编语言、BASIC、C、Pascal等都是面向过程的语言。

实际上，早在20世纪40年代，在对数字模拟的分析研究中就引入了“对象”的概念。随后经过多年的发展，在70年代初期，以Smalltalk为代表的纯面向对象程序设计语言环境诞生，它体现了纯粹的面向对象思想，堪称面向对象的经典。随后的80年代和90年代中产生了许多面向对象的语言，比如C++、JAVA、C#等。

面向对象语言的发展是计算机技术发展到一定阶段的产物。随着计算机技术发展，需要解决的问题越来越复杂，计算机软件规模的扩张也很快。这样一来，软件的代码量愈来愈大，软件的开发、代码的维护、功能的修改变得越来越困难，很自然地过渡到了面向对象的编程方法。

面向对象的编程是类似于模拟现实世界的事物，现实宇宙中的万物都处于不断运动变化之中，事物的属性、能动性时刻都可能有所变化。而它们的运动变化既有相互作用，也有自身演变。程序描述的事物一旦具有以上特性，将与真实世界达到极强的拟合度，更贴近自然界的运行方式。

从代码的组织方面来看，面向对象程序的代码组织比面向过程程序的代码组织更合理，更易于代码重用和维护。面向过程程序是由一系列相互独立的常量、变量和函数组成，他们之间没有明显的关联和隶属关系。任意函数之间可以相互调用，任意函数也可以使用全局的常量、变量，甚至可以改变全局的变量。面向对象程序的代码则是由一系列对象组成，每个对象包含其所属的常量、变量、函数（为与面向过程函数区分，常称为方法），一般来说，引用对象内部状态（对象内部常量）与修改对象内部状态（对象内部变量）必须通过对象自身来进行，不能或不允许直接从对象的外部修改和控制对象的状态。这样，就可以避免面向过程程序中的变量发生不可预料的改变或超出其范围的变化。

从以上内容看出，面向对象程序主要优点是易维护、易复用、易扩展，可以设计出低耦合的系统，使系统更加灵活、更加易于维护。面向对象的三个基本特征是封装、继承、多态。

面向对象的编程方法确实具有一定的优越性，但并不能取代面向过程的的编程方法。当今IT界两种方式的编程语言都是很活跃的，甚至有可以混合使用两种编程方法的高级语言，如Python、PHP等。

为程序建立模型及面向对象编程基本过程

为什么要讨论模型——编程时类的定义是建模的结果。

模型是某种事物的一个简单表示。

从定义可以看出模型与其表示的事物并不完全相同。类是建模的结果。对模型的理解要注意以下几点：

1、模型是真实对象的表现，模型可以具有真实对象不具有的性质，也可以不具有真实对象的某些性质。这要看建立模型的目的或需求。如同样一名学生，在公安部门需要的身份信息与学校需要的身份信息不可能一样，建模时其模型就不可能相同。

2、模型可以做真实事物不能做到的或不容易做的。如用积木搭建房子比用真实的砖、瓦建材要容易的多；真实的学生成绩表不能我们不可能每天任意时刻都可以查询，而在计算机中的学生成绩表就可以每天任意时刻查询。

3、定义中的“表示”说明模型是表示事物的某种方法而已，而且表示方法可以不尽相同。如演示地球形状的模型可以用木头制作，也可以用泥塑制作，还可以用塑料制作。说明我们在建模时，对同一事物最后的建模可能不同，哪怕是同一个目的。

4、建模决定事物表示的优劣。优秀的建模能实现真实事物不能达到的目的，又能在使用时得心应手。编程者知识面越宽，建模就会越好，因为他能真正认识建模的事物。

宇宙中事物多种多样，人们总是习惯分门别类去认识它们。分门别类有利于找到相同点与不同点，通过总体的概括认识事物比一个一个独立认识它们效率高出很多，而这种概括同样可以使编程变得更有效、更容易。

建立模型的过程中所分析出的相关类是指具有相同属性与能力的建模对象。而同一个类中的不同具体的对象就是实例。

同一类表现基本相同，即具有相同的属性和能动性，而一类中的不同实例的状态则不可能完全相同，就像世界上不可能两片完全相同的叶子一样。如同属于兔子类（类）两只兔子（实例）可能一只在吃食物，另一只在睡觉；而这两只兔子各项生理特征也不可能完全相同，如心跳速度、毛色、体重等等。

注意：“对象”一词所指的东西是根据上下文不同既可指类，也可指实例，在此应该说是一个非准确的用语。

面向对象编程基本过程也是以建模分析和建立类为主要过程的。

1、搜集与分析客户要求：采用多种方法对客户的要求进行搜集与分析，搜集的越全面、越准确，对以后的编程越好。

2、设计系统，即面向对象的分析与设计。其中贯彻了“万物皆对象”的思想，即建模。

3、编写代码。根据系统的设计，定义系统中用到的类，并构建相应的实例，完成软件代码的编写。

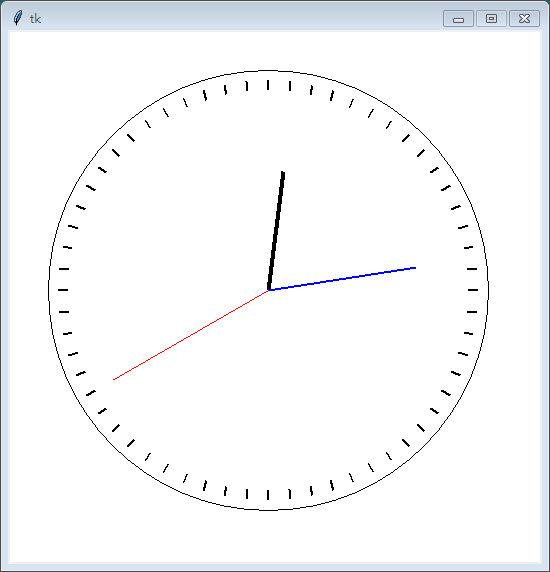
4、测试代码。测试代码的准确性，即能否在任何情况下都能正确的工作。

5、分发、使用与维护软件。

由以上步骤也可以看出，以上步骤中后四个几乎都是在围绕类的定义与使用开展工作。

最简单的面向对象分析与类的定义

本文后续将以一个简单的图形化实例来展示面向对象编程中的一些概念和方法。现在我们要实现一个简陋的图形化的时钟程序，运行的效果如下图所示：



首先我们来分析下基本界面中的涉及哪些对象？不难看出，其中包括刻度(整点)、刻度(分钟)、时针、分针、秒针。

对于初学者或者对类的概念了解不深的话，有可能在建立模型的时候为上述每个对象建立一个类，并编写相应的实现代码。

如果深入一点的话，整点的刻度标记和分钟的刻度标记都是一条短的线段，其区别在于其粗细或颜色不同、显示的位置不同，这些可以作为同一个类的不同状态，应该建立一个类来代表他们。对于时针、分针、秒针，是长短不同的线段，其颜色也不相同，也可以作为同一个类的不同状态。不同指针在指示时间时旋转的速度与角度不同，也只不过是同一个类的不同旋转方法而已。所以，正确的建模方式应该是两个类就可以了。一个是刻度类、另一个是时针类。

再深入一点呢，虽然指针可以转动，而刻度线不需要转动，但外观形象相同的，它们都可以看作是长短不一、颜色不同的线段。

如果按照最简单的形式来实现的话，我们可以定义如下的一个类：

class MyLine:

def \_\_init\_\_(self,canvas,width=1,color='black'):

self.canvas = canvas

self.width = width

self.color = color

self.widget\_id = None

类MyLine非常简单，只有四个实例属性（canvas、width、color、widget\_id），分别用来表示画布（在tkinter库中用来显示内容的对象）、线段宽度、线段颜色和线段实例的ID（用来引用已经显示的内容，默认的引用为空，因为这时候引用内容还没有建立）。

一个构造方法（\_\_init\_\_，字母的前后是两个下划线，在Python中表示这是个特殊的方法）。构造方法带有五个参数：self是个特殊的参数，专指该类建立的实例本身，调用时该参数自动传入；第二个参数、第三个参数是初始化类实例必传的参数，后两个是带有默认值的参数。

实例化该类的形式之一可以如下：

myline\_a = MyLine(canvas,2,’green’)

myline\_b= MyLine(canvas)

此时，我们称myline\_a, myline\_b为类MyLine类的实例，类实例化时，会自动调用其构造方法（\_\_init\_\_），来初始化实例，这里表现为给三个实例属性赋值。实例myline\_a表示的线段的宽度为2，线段的颜色是绿色。实例myline\_b表示的线段的宽度为1，线段的颜色是黑色，都为默认值，所以实例化时并没有传递这两个参数。就是说同一个类经过不同的实例化后的实例可以千差万别。因此类就是生成实例的模板，我们也可以简单地把类理解为一枚印章，用不同的印泥可以印出不同颜色的图样，而类的实例化则更为复杂。

通过MyLine类的定义，我们还可以看出面向对象编程的另一个特征，即封装性。在该类中定义的属性和方法都定义在MyLine类体内部，而面向过程编程的代码就没有这种组织代码的方式。封装性还表现在要修改类的属性，原则上要通过类实例的自身进行修改，而不是在类实例的外部随意的修改；同时，要调用类的方法执行某一动作，同样是通过类实例去调用他。此外，我们也可以通过方法来控制对属性的赋值，比如赋值的类型、大小范围等。所以封装起来的类，使用时只有从外部调用类实例或类的方法来改变其自身的状态，这样可以避免一些误操作。就好像我们驾驶汽车一样，只能通过汽车提供的油门、档位、离合、刹车、各种开关来控制汽车，你不能随意的加大油门，或把档位调整后更高。

再谈类定义

上文中定义了一个包含一个构造方法和四个实例属性的类MyLine。对于Python中的类，还有其他相关的概念需要我们了解。

Python中实例属性包含实例变量和实例方法。

1.实例变量的定义

作为动态语言，Python中的类不要求预先定义实例变量，实例变量赋值即视为定义，你也可以在类的其他普通方法中定义和使用实例变量。请看如下交互式环境下运行的代码：

>>> class Foo:

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

self.x = x

def testa(self,y):

self.y = y

def getxy(self):

print(self.x,self.y)

>>> f = Foo(3)

>>> f.getxy()

Traceback (most recent call last):

File "<pyshell#86>", line 1, in <module>

f.getxy()

File "<pyshell#84>", line 9, in getxy

print(self.x,self.y)

AttributeError: 'Foo' object has no attribute 'y'

>>> f.testa(9)

>>> f.getxy()

3 9

第一次调用f.getxy()时，试图使用实例变量y，Python给出的错误信息很明确，此时，Foo对象的实例f还不具有实例变量y。当执行完f.testa(9)时，实际上就是运行了方法中的语句self.y = 9，这时，通过赋值即定义的形式使f这个实例具有了y这个实例变量。所以，再次调用f.getxy()时，成功地输出了x和y的属性值。

一般情况下，实例变量尽量在类的构造方法中定义和赋值，如果能保证其实例变量是在使用前被定义和赋值，则也可以在一般方法中定义实例变量，实现的方法可以是在构造方法中调用定义实例变量的一般方法一次。

2.实例方法的定义

通过上文中的代码，也可以看出一般实例方法的定义和构造方法定义形式是类似的，只不过方法名称不使用双下划线了。

一般方法的定义时，第一个参数也是self，用于对实例自身的引用，其他参数可以和面向过程的函数一样，由用户根据实际需求定义。但需要说明的是，在使用类实例去调用普通的实例方法时，self这个参数是不用提供的，由Python自动传入。

类的公共属性

前面的代码中，我们只给类定义了实例变量，同一个类建立的不同实例的实例属性可以不同。要实现相同类的实例共享属性，可以给类定义类属性和类方法。

1.类属性

顾名思义，类属性就是这个属性是属于类的，而不是属于某一个类实例的。例如以下类定义：

class Foo:

bar = 7

类Foo中没有定义构造方法，但定义了一个类属性。类属性是所有这个类实例的仅有属性，看具体操作：

>>> class Foo:

bar = 7

>>> foo = Foo()

>>> foo2 = Foo()

>>> Foo.bar

7

>>> foo.bar

7

>>> foo2.bar

7

>>> Foo.bar = 0

>>> foo.bar

0

>>> foo2.bar

0

以上是再Python交互环境下运行的结果。可看到，定义Foo类和实例化之后，不管是Foo.bar，还是查看其实例的bar的值都为7。当调用Foo.bar = 0之后，其实例的bar属性也都改变成了0。

注意：如果执行foo.bar = 3，则此时执行的是为foo这个实例添加了一个实例变量bar并赋值为3，并没有改变Foo的类属性bar的值，Foo.bar和foo2.bar返回的是类属性，其值依然是0。而再引用foo.bar则访问的是foo这个实例的实例变量，所以结果为3。

>>> foo.bar =3

>>> foo2.bar

0

>>> foo.bar

3

>>> Foo.bar

0

>>> foo.bar

3

实际上，foo的实例变量bar和类属性bar重名了，引用foo.bar时，优先引用了实例变量。此时我们可以理解为foo的实例变量遮盖了其类属性。而我们一般避免使用foo.bar = 3的形式来改变类的实例，同时也不使用与类属性同名的实例变量。

2.类方法

下面是一个简单的类的代码，定义了一个类方法。

class Foo:

bar = 3

@classmethod

def c\_mthd(cls,v):

print(cls.a,v)

以下为实例化类后调用结果和直接用类名的调用结果：

>>> f = Foo()

>>> f.c\_mthd(0)

3 0

>>> Foo.c\_mthd(0)

3 0

类方法是被classmethod装饰器装饰的，其第一个参数必须是cls，用它来代表类自身。其它参数使用方法与一般方法相同。

注意：类方法中不能使用实例属性，因为调用时可能还没有实例化该类，否则会出错。

3.静态方法

静态方法在定义时就使用staticmethod装饰器，其参数既不用self，也不用cls。从定义形式来看，就像定义一个普通函数一样，如果不希望他被封装在类一起，完全可以在类的外部进行定义一个普通函数，然后在类内的需要的地方直接使用。

class Foo:

bar =3

@staticmethod

def s\_mthd(v):

print(Foo.bar,v)

以下是其调用方式：

>>> Foo.s\_mthd(0)

3 0

>>> f = Foo()

>>> f.s\_mthd(0)

3 0

注意：同类方法相同的是静态方法中也不能使用实例变量，但可以通过类名来使用类属性。但一般情况下，我们定义的静态方法中不应访问类及其实例的属性的。

类的继承

继承是面向对象编程的三大特征之一，也是面向对象编程中代码复用的重要方法。在前文中分析的图形化时钟程序时，仅定义了一个基础类MyLine，具有画布引用、线宽、颜色和部件ID四个属性，用它来作为时钟程序图形化界面的刻度还缺乏一些功能，如果用作时钟的指针也缺乏一些功能。

现在要补充一些功能，其实现方法可以是单独定义用于表示界面刻度的类和用于表示指针的类，但这样一来，公共属性必须多次定义，势必带来代码的重复，所以我们用面向对象中的继承特性来实现这两个类。

首先在基础类MyLine上添加一个方法：

class MyLine:

def \_\_init\_\_(self,canvas,width=1,color='black'):

self.canvas = canvas

self.width = width

self.color = color

self.widget\_id = None

def delete(self):

if self.widget\_id:

self.canvas.delete(self.widget\_id)

该类增加了一个实例方法，即delete()方法，用于从画布上清除自己。该类如果用来表示钟表盘上的刻度和指针，还是不够的，需要添加一些实例变量和实例方法。

现在要扩充以上类，就是通过继承来复用以上代码。

首先，继承基础类MyLine，实现用于表示时间刻度的类Marker，其代码如下：

class Marker(MyLine):

def \_\_init\_\_(self,start\_point,end\_point,canvas,width=2,color='black'):

super().\_\_init\_\_(canvas,width,color)

self.start\_point = start\_point

self.end\_point = end\_point

def draw(self):

self.widget\_id = self.canvas.create\_line(self.start\_point,self.end\_point,

width=self.width, fill=self.color)

在类Marker的定义中，类名后添加了一个(MyLine)表示该类继承了类MyLine，那么在MyLine类中定义的一些属性会被Marker类继承，即子类Marker拥有父类的实例属性。在Marker类的构造方法中，又添加了两个实例化参数：start\_point,end\_point，分别用来表示刻度线段的起点与终点。

注意super().\_\_init\_\_(canvas,width,color)语句的作用是调用其父类的构造方法。父类的构造方法只有在子类没有定义构造方法时才会在实例化时运行。

最后一个是draw()方法，其作用就是在指定的画面上完成刻度标记的显示，还将其引用赋值给了实例变量widget\_id。

下面再来看程序中要定义的指针类的实现，其代码如下：

class Pointer(MyLine):

def \_\_init\_\_(self,ptype,canvas,center\_point,plong=180,width=1,color='black'):

super().\_\_init\_\_(canvas,width,color)

self.center\_point = center\_point

self.plong = plong

self.ptype = ptype

self.end\_point = None

self.points = itertools.cycle(gen\_end\_points(self.center\_point,self.plong,self.ptype))

self.count,self.end\_point = self.points.\_\_next\_\_()

def draw(self):

self.widget\_id = self.canvas.create\_line(self.center\_point,self.end\_point,

width=self.width, fill=self.color)

self.count,self.end\_point = self.points.\_\_next\_\_()

def walk(self):

self.delete()

self.draw()

与Marker类定义类似，该类也继承了MyLine类，其中也增加了一些实例属性，其构造函数的参数也增加了ptype,center\_point,plong分别用来表示指针的类型（不同指针走动规律不同）、指针的中心点坐标（这里用的是钟表的中心点）和指针的长度（时针、分针和秒针长度不同）。

其次，Pointer类还增加了两个实例方法，draw()和walk()。由于指针类的draw方法与刻度类的draw方法不同，所以并没有在他们的共同父类中定义。指针绘制的起始点为表盘的中心点，而另一端点则由计算得到，每次绘制完指针时，并将下次需要绘制的指针端点座标进行赋值。

Pointer类的实例方法walk()实现的功能就是指针的移动，实际上是通过先将原位置的指针删除，然后在新的下一个位置绘制指针，就表现为指针的转动。代码仅表现为调用自身两个实例方法。

指针端点计算方法是通过一个生成器来完成的，其代码如下：

def get\_all\_points(center\_point,plong):

end\_points = []

for i in range(360):

x = center\_point[0] + plong \* math.cos(i \* math.pi / 180)

y = center\_point[1] + plong \* math.sin(i \* math.pi / 180)

end\_points.append((x,y))

return end\_points[270:] + end\_points[:270]

def gen\_end\_points(center\_point,plong,sep):

for i,p in enumerate(get\_all\_points(center\_point,plong)[::sep]):

yield i,p

其中get\_all\_points()函数根据传入的表盘的中心点坐标和指针长度每隔一度生成一个端点座标，根据tkinter库中的画面座标对起始位置做了相应调整，然后由生成器gen\_end\_points()将其提供给使用者。

而指针的转动又一直是循环往复的，所以这里使用了itertools.cycle()，用于循环产生所需要的座标点。

此外，Python语言中类的继承是允许多继承的，即同一个类可以同时有多个父类，其代码写法上类似单继承，形如：class A(B,C,D,…)。多个父类有相同方法的，在调用子类的方法时优先调用先继承的父类的方法，其他父类的同名方法不被调用。

类的多态

多态是面向对象的又一重要概念，字面意义就是“多种状态”。在程序中不需要区分引用对象，用相同的调用方法，完成引用对象的功能而表现出多种状态。有了多态，就不需要为每个具体的子类编写功能调用，是一种泛型编程。如果你是一个公司的后勤经理，需要打扫地面，不论是扫地机器人还是清洁工，你只要告诉他打扫的范围，他们就可以完成打扫的任务。

在一些语言中需要使用继承、接口、不同参数方法来实现多态。由于Python是动态语言，其实现的基本方式称为“鸭子类型”（duck typing）。即"当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子。"在Python的调用语义中，只要调用对象具有所调用的属性，那么不管调用对象是什么类型，就可以调用成功。

在维基百科的词条（https://en.wikipedia.org/wiki/Duck\_typing）中示例代码如下：

class Duck:

def fly(self):

print("Duck flying")

class Airplane:

def fly(self):

print("Airplane flying")

class Whale:

def swim(self):

print("Whale swimming")

def lift\_off(entity):

entity.fly()

duck = Duck()

airplane = Airplane()

whale = Whale()

lift\_off(duck) # prints `Duck flying`

lift\_off(airplane) # prints `Airplane flying`

lift\_off(whale) # Throws the error `'Whale' object has no attribute 'fly'`

在函数lift\_off()中对传入的entity调用了fly()方法，传入函数的是什么对象类型，只要entity引用的对象包括fly()方法，就可以成功调用，如lift\_off(duck)、lift\_off(airplane)。而第三次调用时，即lift\_off(whale)，传入的是Whale类的实例，其不具有fly()属性，因此调用就会失败。

正是因为Python的这种特性，降低了Python实现多态的难度，提高了多态实现的灵活性。所以在一些Python的面向对象程序资料中，对多态的介绍仅限于此。

对象的组合

在面向对象的编程中，经常用到的一种方式就是将更简单的对象组合形成复杂的对象，而复杂的对象还可以组合形成更复杂的对象。这也正是现实世界存在的一种方式，如一台电风扇，可以由底座、支架、控制器、扇头等组成，扇头又有网罩、扇叶、电机组成，电机又由轴承、转子、线圈等组成。编程实现的话，就可以把各种元件建模为对象，然后逐步组合即可。

下面以要实现的图形化时钟的组成，来看类的组合关系。时钟的组成部件可以简单看作为表盘（带有刻度和外周圆圈）、时针、分针和秒针组成，前面定义了刻度类，下面就是表盘外周圆圈类和表盘类的代码。

class PlateOuter:

def \_\_init\_\_(self,canvas,center\_point,radius):

self.canvas = canvas

self.center\_point = center\_point

self.radius = radius

self.widget\_id = None

def draw(self):

x0 = self.center\_point[0] - self.radius

y0 = self.center\_point[1] - self.radius

x1 = self.center\_point[0] + self.radius

y1 = self.center\_point[1] + self.radius

self.canvas.create\_oval(x0,y0,x1,y1)

def delete(self):

if self.widget\_id:

self.canvas.delete(self.widget\_id)

class Plate:

def \_\_init\_\_(self,canvas, center\_point, radius, plong):

self.canvas = canvas

self.center\_point = center\_point

self.radius = radius

self.plong = plong

self.markers = []

self.gen\_markers()

def draw(self):

for marker in self.markers:

marker.draw()

def gen\_markers(self):

for start,end in zip(gen\_end\_points(self.center\_point,self.radius,6),

gen\_end\_points(self.center\_point,self.radius + self.plong,6)):

self.markers.append(Marker(start[1],end[1],self.canvas))

self.markers.append(PlateOuter(self.canvas,self.center\_point,self.radius+20))

def delete(self):

for w in self.markers:

self.canvas.delete(w)

PlateOuter类就是用来代表个表盘外圆圈的，其基本结构也很简单，类似之前的Marker等类，包括四个实例变量和三个实例方法，四个实例变量分别用来表示画布、表盘的中心点座标、表盘的半径和表盘圆的部件id；三个实例方法，第一个是构造方法，draw()方法是依据表盘的中心座标和半径计算后在指定的位置绘制一个圆，delete()方法是用于清除自己的。

在Plate类中，组合了PlateOuter类和多个Marker类构成了表盘，在gen\_markers()方法中，生成了表示每个刻度的刻度类的实例和一个PlateOuter类的实例；在draw()方法中，只不过是调用每个组成部分的draw()方法来进行绘制就可以了，delete()方法中也是采取同样的方式来实现的。

下面来看看时钟代码：

class MyClocker:

def \_\_init\_\_(self,root,canvas, center\_point, radius, plong):

self.root = root

self.plate = Plate(canvas, center\_point, radius, plong)

self.s\_pointer = Pointer(6,canvas,center\_point,plong=180,width=1,color='red')

self.m\_pointer = Pointer(3,canvas,center\_point,plong=150,width=2,color='blue')

self.h\_pointer = Pointer(1,canvas,center\_point,plong=120,width=4,color='black')

self.display()

def display(self):

self.plate.draw()

self.s\_pointer.draw()

self.m\_pointer.draw()

self.h\_pointer.draw()

self.root.update()

def walk(self):

self.s\_pointer.walk()

if (self.s\_pointer.count + 1) % 30 == 0:

self.m\_pointer.walk()

if (self.m\_pointer.count + 1) % 4 == 0:

self.h\_pointer.walk()

def start(self):

while True:

self.walk()

self.root.update()

time.sleep(1)

在时钟代码类MyClocker中，第一个仍然是构造方法，其中的主要部分为实例化一个表示表盘类Plate，实例化三个Pointer类分别代表时针、分针和秒针，之后调用display()方法绘制各个组成部件；walk()方法中，秒针移动一步，之后检测分针是否应该移动一步，若分针移动一步，则之后还会检测时针是否应该移动一步。

start()方法是时钟启动的关键方法，在其中使用了一个while死循环，调用一次walk()方法后，就sleep一秒。这样就达到每秒移动秒针一步，其他指针会按规则移动的，整个时钟就可以正确无误的工作了。

如果你要知道时钟的图形界面是如何生成的，就必须要了解一下tkinter库。其主要调用代码如下：

root = tkinter.Tk()

cvns = tkinter.Canvas(root,width=530,height=530,bg='white')

cvns.pack()

mc = MyClocker(root,cvns,(260,260),200,10)

t = threading.Thread(target=mc.start)

t.setDaemon(True)

t.start()

root.resizable(False, False)

root.mainloop()

注意这里使用了一个线程来启动时钟的指针的转动，可以避免在进行其他图形化操作时，指针停止的问题。