上海交通大學

SHANGHAI JIAO TONG UNIVERSITY

INTRODUCTION OF BUSINESS COMPUTING

PROJECT REPORT

Project title： The application of Python on games

Team name: DionyStudio

Team member:

薛见悦	516030910095

周浩博	516051910029
宋姜康宏	516205910015
MAN HUA TAO	715120290014

INDEX

The application of Python on games

DionyStudio

ABSTRACT

We study the application of Python on games. Based on our interests, we decide to study the realization of 2048,Mine Sweeping, and Crazy Tetris on python.

1 Realization of 2048

1.1 Game rules

At the beginning, there are two “2” in two random squares at the interface; Click the buttons and merge the same numbers, noting that merging continuously is not allowed; At each move, “2” or “4” appears in a random blank square at the interface; The game is over when the player is unable to move; Support for undoing one step and resetting.

1.2 Key points of design

1.2.1 Application of knowledge

In the 2048 project, we use the knowledge such as random, class, graphics, button, list and so on.

1.2.2 Difficulties

In the 2048 project, we meet the difficulties such as judgement of losing; undoing one step; resetting; merging numbers; generation of random numbers; allowing new numbers not to appear when clicking the button not activated; removing the text noting victory of failure after clicking the “reset” button.

1.3 The whole idea

The whole idea is divided into three parts:

1. The main program including invocation of other modules at initialization, painting GUI, initializing matrix, binding button, drawing 2048 boxes, revocation, resetting and other functions.

2. The frame of the graphical interface including designing graphic interface.

3. The part that implements the functions of the up,down,left and right movements, the reset revocation, and the judgement of the end of the game including implementation of basic algorithm, initialization, block merging and block movement

The modules of button and graphics are downloaded from Internet.

1.4 Advantage and disadvantage

The advantage is writing 2048, frame, function as three files, and importing the latter

two in 2048, making the program clearer, and more convenient to debug. The disadvantage is the support for undoing only one step.

1.5 Implementation details

```
def interface(self, win):
    p = Point(10, 220)
    n = 4
    self.rt = [0 for row in range(n * n)]
    for i in range(n):
        for a in range(n):
            _p = Point(p.x + 60 * i, p.y + 60 * a)
            self.rt[i + 4 * a] = frame.rect(win, _p)
```

Use 4X4's list to draw the interface.

```
def move_left(matrix):
    left_list = []
    for item_list in matrix:
        left_list.append(_left(item_list))
    return left_list
```

```
def move_up(matrix):
    up_list = []
    l = [0, 0, 1, matrix]
    matrix =for_matrix(l)
    for item_list in matrix:
        up_list.append(_left(item_list))
    l = [0, 0, 1, up_list]
    up_list =for_matrix(l)
    return up_list
```

The above are left(right), up(down) movements.

```
def is_win(matrix):
    for item_list in matrix:
        if 2048 in item_list:
            return True
    return False
```

Judge winning according to finding whether 2048 is in the list.

```
def is_over(matrix):
    for item_list in matrix:
        if 0 in item_list:
            return False
    for i in range(n):
        for j in range(n):
            if i < n - 1:
                if matrix[i][j] == matrix[i+1][j]:
                    return False
            if j < n - 1:
                if matrix[i][j] == matrix[i][j+1]:
                    return False
    return True
```

Judge losing according to examining whether the current matrix is the same as the last matrix.

```
def add_same_number(my_list):
    global num
    for i in range(len(my_list)-1, -1, -1):
        if i >= 1:
            if my_list[i-1] == my_list[i]:
                my_list[i-1] = 0
                my_list[i] = 2*my_list[i]
                num += my_list[i]
                break
    return my_list
```

```
def del_item_0(my_list):
    list_0 = []
    for item in my_list:
        if item != 0:
            list_0.append(item)
    return list_0
```

Use list's “append” operation to realize a series of functions such as adding the same numbers and deleting numbers, so as to realize the mobile merging of numbers.

```
if (self.button1.clicked(p)):
  self.matrix = self.las_matrix
  self.num(self.matrix)
  self.button1.activate()
```

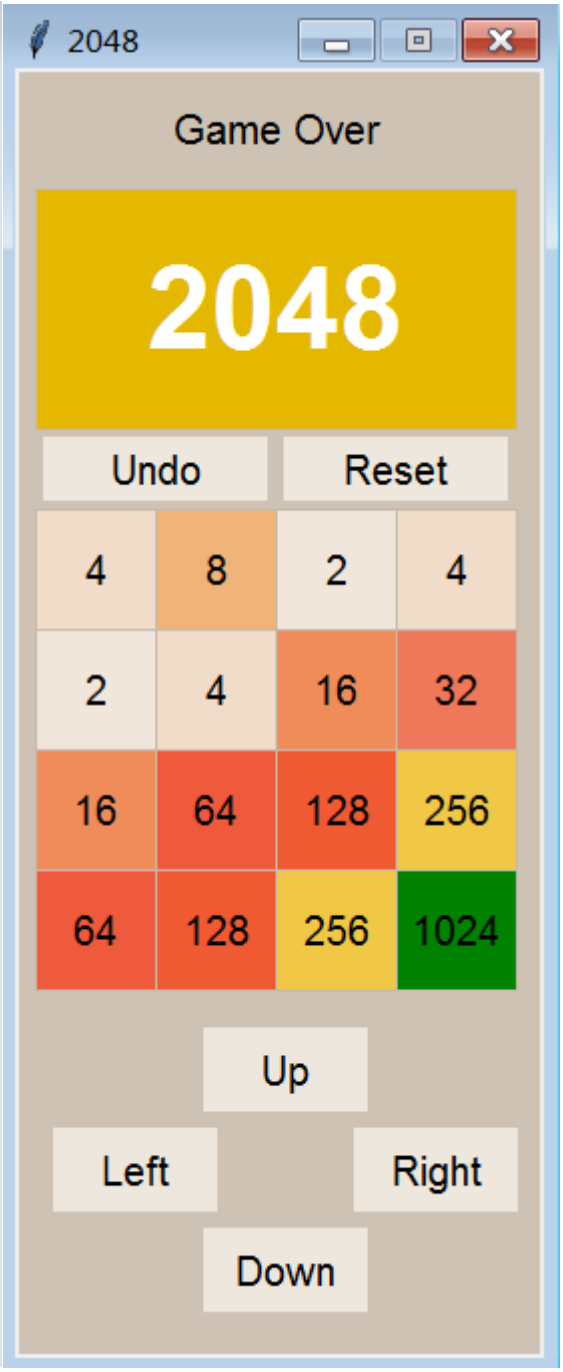
Assign the last matrix to the current matrix to realize the function of “undo”.

```
if(self.button2.clicked(p)):
  self.matrix = self._init()
  self.num(self.matrix)
  self.button1.deactivate()
  self.eliminate.undraw()
  self.eliminate_2.undraw()
  self.f=True
```

Assign the initial matrix to the current matrix and generate random numbers to realize the function of “reset”.

1.6 Test screenshot

Interface



Game over

2 Realization of Mine Sweeping

2.1 Game rules

A certain number of mines are randomly arranged in a rectangular minefield of a certain size (“easy” is 5*5 squares and 5 mines (1 per row), “normal” is 10*10 squares and 20 mines (2 per row) and “hard” is 15*15 squares and 45 mines (3 in each row). Players need to find out all the squares that are not mines and avoid stepping on the mines.

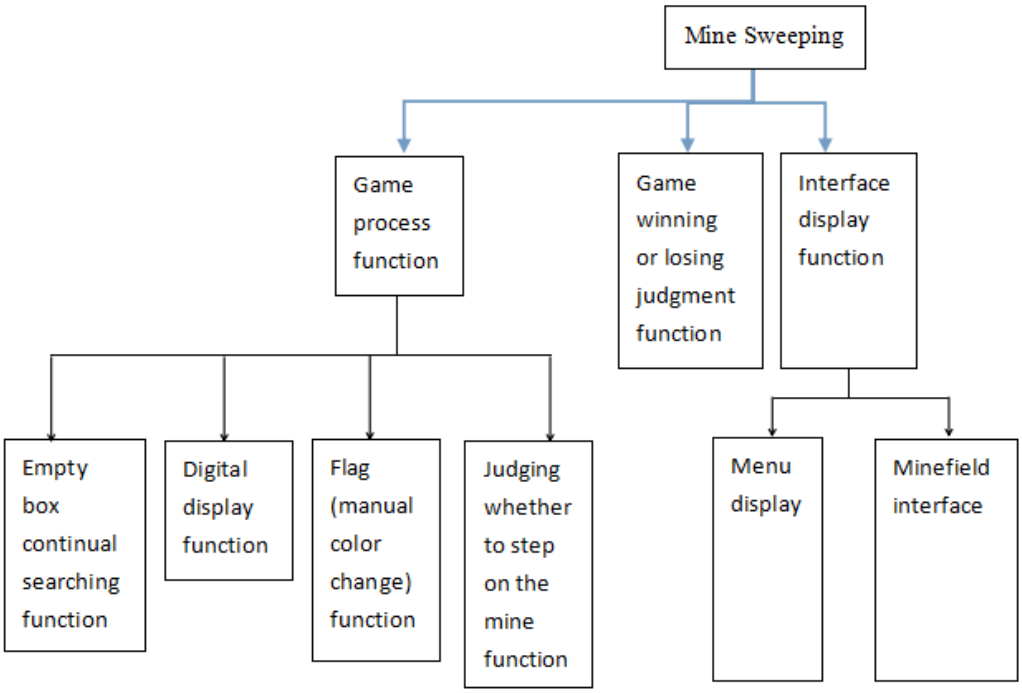
The basic operations of the game include Left Click and Right Click. The left button is used to open a safe grid to advance the progress of the game; the right button is used to mark mines to assist judgment

Left-click: Press the left button on a box that is not a mine, you can open the box. If a number appears on the block, the number indicates the number of mines in the surrounding 3x3 area (generally 8 grids, 5 grids for the edge blocks, and 3 grids for the corner blocks. So the biggest number in the Mine Sweeping is 8). If the box is empty (equivalent to 0), the boxes adjacent to the null can be recursively opened.if you unfortunately open the box of mine, the game is over.

Right-click: You can mark mines (shown as yellow box) by pressing the right button on the box that identifies mines. Repeat the operation to cancel the mark.

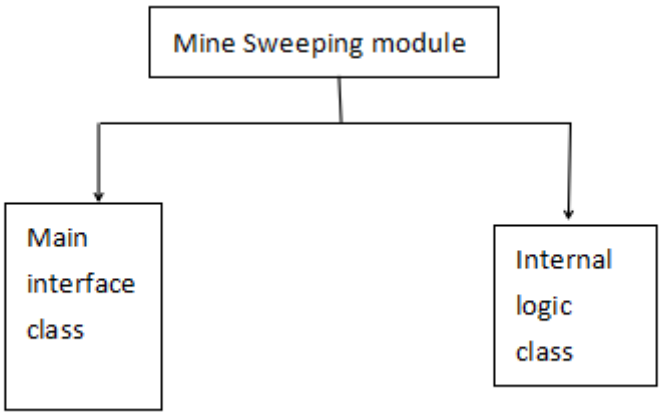
2.2 The implementation plan of Mine Sweeping

2.2.1 Overall functional design



2.2.2 Program Design Main Module

The game consists of one module and two classes.



2.3 Key points of design

2.3.1 The realization of minefield internal matrix

The key code:

```
class Model:#Internal game data matrix creation
    def __init__(self,r,c):
        self.height=r#Number of rows in the matrix
        self.width=c#Number of columns in the matrix
        self.items=[]
        for i in range(r):
            self.items.append([])
            for j in range(c):
                self.items[i].append(0)
```

Create a 0 matrix with r rows and c columns

```
def countValue(self, r, c, v):#Gets the number of elements v around the position of the Model matrix r row c column
    count=0
    if 1<=r<=self.height-2 and 1<=c<=self.width-2:
        for i in [r-1, r+1]:
            for j in [c-1, c, c+1]:
                if self.items[i][j]==v:
                    count+=1
        if self.items[r][c-1]==v:
            count+=1
        if self.items[r][c+1]==v:
            count+=1
    if r==0 and c==0:
        if self.items[r][c+1]==v:
            count+=1
        if self.items[r+1][c]==v:
            count+=1
        if self.items[r+1][c+1]==v:
            count+=1
    if r==0 and c==self.width-1:
        if self.items[r][c-1]==v:
            count+=1
        if self.items[r+1][c-1]==v:
            count+=1
        if self.items[r+1][c]==v:
            count+=1
    if r==self.height-1 and c==0:
        if self.items[r][c+1]==v:
            count+=1
        if self.items[r-1][c+1]==v:
            count+=1
        if self.items[r-1][c]==v:
            count+=1
    if r==self.height-1 and c==self.width-1:
        if self.items[r][c-1]==v:
            count+=1
        if self.items[r-1][c-1]==v:
            count+=1
        if self.items[r-1][c]==v:
            count+=1
    if r==0 and 1<=c<=self.width-2:
        for i in [c-1, c, c+1]:
            if self.items[r+1][i]==v:
                count+=1
        if self.items[r][c-1]==v:
            count+=1
        if self.items[r][c+1]==v:
            count+=1
    if r==self.height-1 and 1<=c<=self.width-2:
        for i in [c-1, c, c+1]:
            if self.items[r-1][i]==v:
                count+=1
        if self.items[r][c-1]==v:
            count+=1
        if self.items[r][c+1]==v:
            count+=1
    if c==0 and 1<=r<=self.height-2:
        for i in [r-1, r, r+1]:
            if self.items[i][c+1]==v:
                count+=1
        if self.items[r-1][c]==v:
            count+=1
        if self.items[r+1][c]==v:
            count+=1
    if c==self.width-1 and 1<=r<=self.height-2:
        for i in [r-1, r, r+1]:
            if self.items[i][c-1]==v:
                count+=1
        if self.items[r-1][c]==v:
            count+=1
        if self.items[r+1][c]==v:
            count+=1
    return count
```

Look at the number of v-values around the elements in row c, column c, and divide it

into corner, edge, and middle

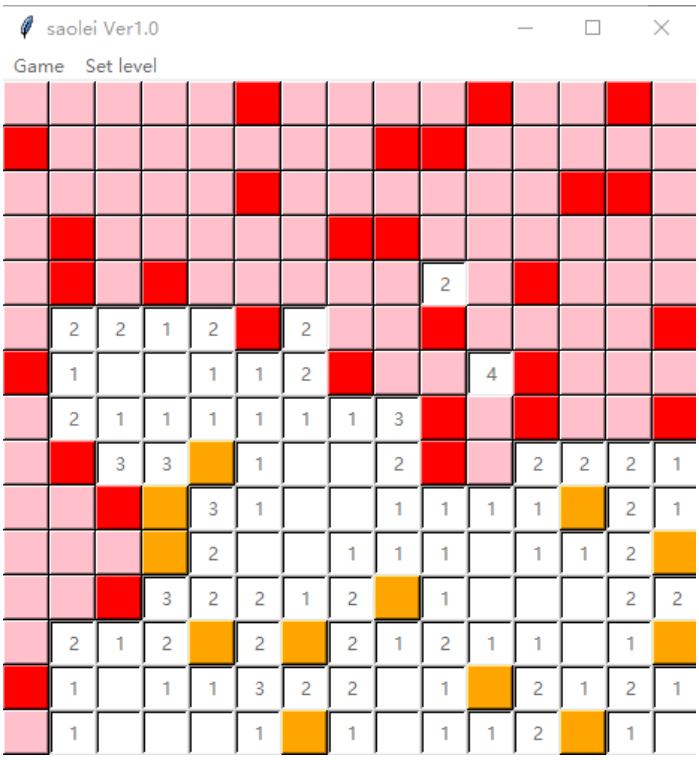
```
def checkValue(self, r, c, v):
    #Check whether the value of the element of the Model matrix r row c column is v, and return a Boolean variable
    if self.items[r][c]==v:
        return True
    else:
        return False
```

Check whether the value of the element of the Model matrix r row c column is v, and return a Boolean variable

```
def clear(self):
    #Empty the Model matrix with each position element set to 0
    for i in range(self.height):
        for j in range(self.width):
            self.setItemValue(i, j, 0)
            self.setItemValue(i, j, 0)
```

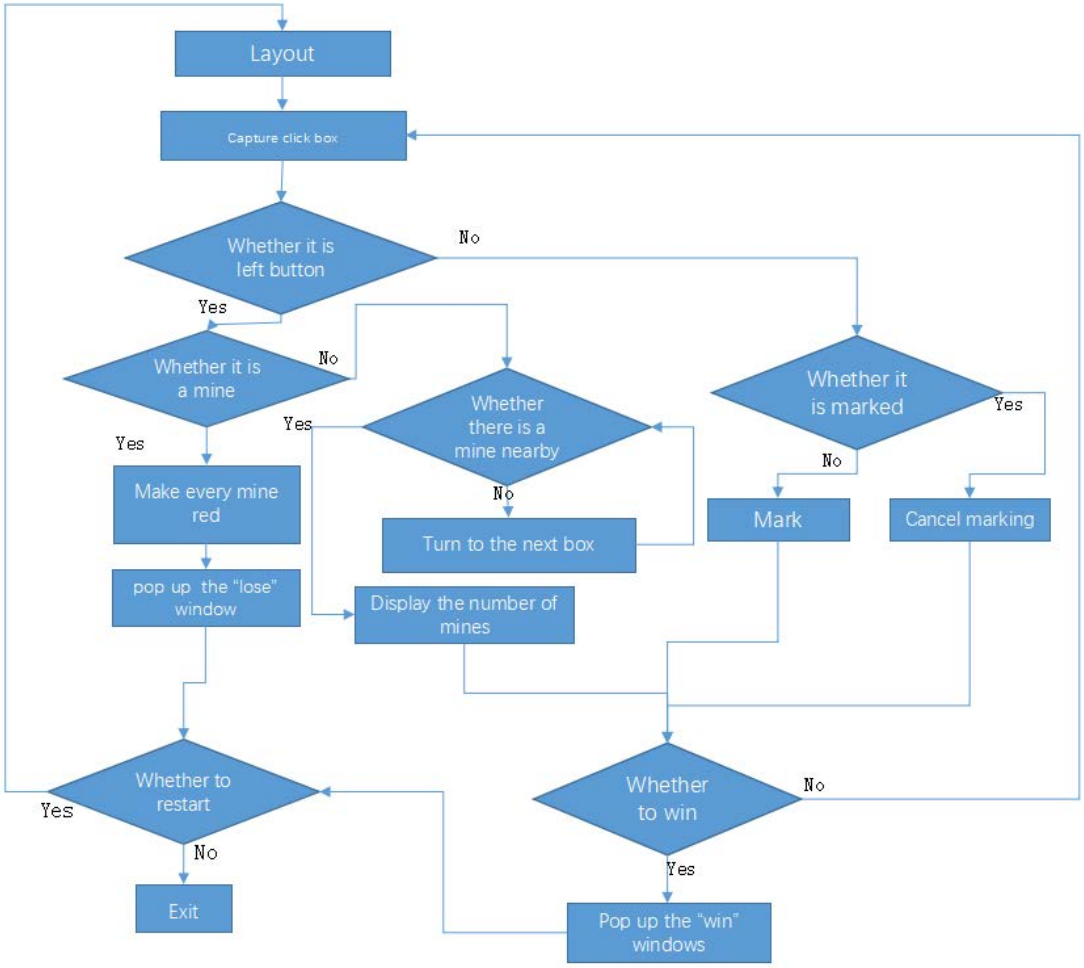
Clear the matrix and change it to 0 matrix

2.3.2 The realization of main interface and rules



The above picture shows the main interface of the Mine Sweeping game, namely the minefield. The minefield can be directly implemented by the Canvas class. The boxes can be directly implemented by the Button class.

Rules:



The key code:

```
def __init__(self,master,model,n):#Master, parent component; model, internal data matrix; n, number of rows per line
    n=int(n)
    self.master=master
    self.model=model
    self.setMines(n)
    self.createWidgets()
```

Take master as the canvas, model as internal data matrix, bury mines, and construct buttons

```
def setMines(self, n): #Randomly put n mines per line.
    for i in range(self.model.height):
        for j in range(n):
            a=random.randint(0, self.model.width-1)
            self.model.setItemValue(i, a, 1)
```

Randomly put n mines per row.

```
def clicked(self, event):
    #After the button is left-clicked, distinguish the clicked button the mine or not, and judge whether to win or not.If win, the wi
    widget = event.widget
    g = widget.grid_info()
    r = int(g["row"])
    c = int(g["column"])
    w = int(self.master['width'])
    h = int(self.master['height'])
    if self.model.checkValue(r, c, 1): # is mine
        self.showall()
    else: # not mine
        self.find(r, c)
    if self.isWin():
        top = Toplevel()
        txt = Label(top, text='You win!')
        txt.grid(row=0)
        button1 = Button(top, text='Restart', relief='raised', command=top.destroy)
        button1.grid(row=1, column=0)
        button2 = Button(top, text='Exit', relief='raised', command=root.destroy)
        button2.grid(row=2, column=0)
        button1.bind('<Button-1>', self.restart)
```

After the button is left-clicked, distinguish the clicked button the mine or not, and judge whether to win or not.If win, the window will pop up to obtain a restart or exit.

```
def setcolor(self, event):
    #Right-click on the button to make the button yellow when the player thinks it is mine, and cancel it when the second click occurs.
    widget=event.widget
    g=widget.grid_info()
    r=int(g["row"])
    c=int(g["column"])
    w=int(self.master['width'])
    h=int(self.master['height'])
    if widget['bg']=='yellow':
        widget.configure(bg='pink')
    else:
        widget.configure(bg='yellow')
```

Right-click on the button to make the button yellow when the player thinks it is mine.

```
def isWin(self):
    count=0
    for i in range(self.model.height):
        for j in range(self.model.width):
            if (self.buttonsgroup[i][j]['bg']=='white' and self.model.checkValue(i,j,-1)) or\
               (self.buttonsgroup[i][j]['bg']=='yellow' and self.model.checkValue(i,j,1)):
                count+=1
    if count ==self.model.width*self.model.height:
        return True
    else:
        return False
```

To determine whether to win, that is, if all mines are marked, all mineless boxes have been retrieved. Return boolean variable.

```
root = Tk()
root.title('saolei Ver1.0')
cv1=Canvas(root,width=200,height=400,bg='white')
m1=Model(5,5)
easy=Saolei(cv1,m1,1)
cv2 = Canvas(root,width=400,height=600,bg='white')
m2=Model(10,10)
normal=Saolei(cv2,m2,2)
m3=Model(15,15)
cv3=Canvas(root,width=600,height=800,bg='white')
hard=Saolei(cv3,m3,3)
cv1.grid()#Default open 'easy'
m=Menu()
root.config(menu=m)
filemenu = Menu(m)
m.add_cascade(label='Game',menu=filemenu)
filemenu.add_command(label='New',command=restart)
filemenu.add_command(label='Quit',command=root.destroy)
setmenu=Menu(m)
m.add_cascade(label='Set level',menu=setmenu)
setmenu.add_command(label='easy',command=changeeasy)
setmenu.add_command(label='normal',command=changenormal)
setmenu.add_command(label='hard',command=changehard)
root.mainloop()
```

Make three difficulty levels and open “easy” by default.

“Game” menu sets “New” and “Quit”.

“Set level” menu sets “easy”, “normal” and “hard”.

3 Realization of Crazy Tetris

Write Python language programs to implement Crazy Tetris games. Tetris is a puzzle casual game that is designed to exercise people's reaction ability, hand-brain coordination ability and space imagination ability. The player rotates seven differently shaped box assemblies and drops them to the most suitable position so that they are joined in a row or rows in the bottom of the screen. These rows will disappear and score; otherwise, if the boxes are stacked tp the top of the screen, the game is over. As a single-player stand-alone game, a Tetris game should have the characteristics of stable algorithm, moderate difficulty and friendly interface, which are also goals that our studio strives to achieve. Nevertheless, due to the limitaion of time and knowledge, we are not able to implement the function of game difficulty and mode setting, which was mentioned in the proposal submitted.

3.1 Crazy Tetris game rules introduction

3.1.1 Basic operation

The operation of this game is extremely simple: in the main interface of the game (Tetris-dropping interface), click the left mouse button to make the Tetris shift to the left, click the right mouse button to move the Tetris to the right, click the middle mouse button to make the Tetris Rotate 90 degrees counterclockwise.

3.1.2 Scoring mechanism

Each time the player drops a Tetris successfully, he scores 1 point; every time the player fills a row with Tetris and eliminates it, he scores 10 points.

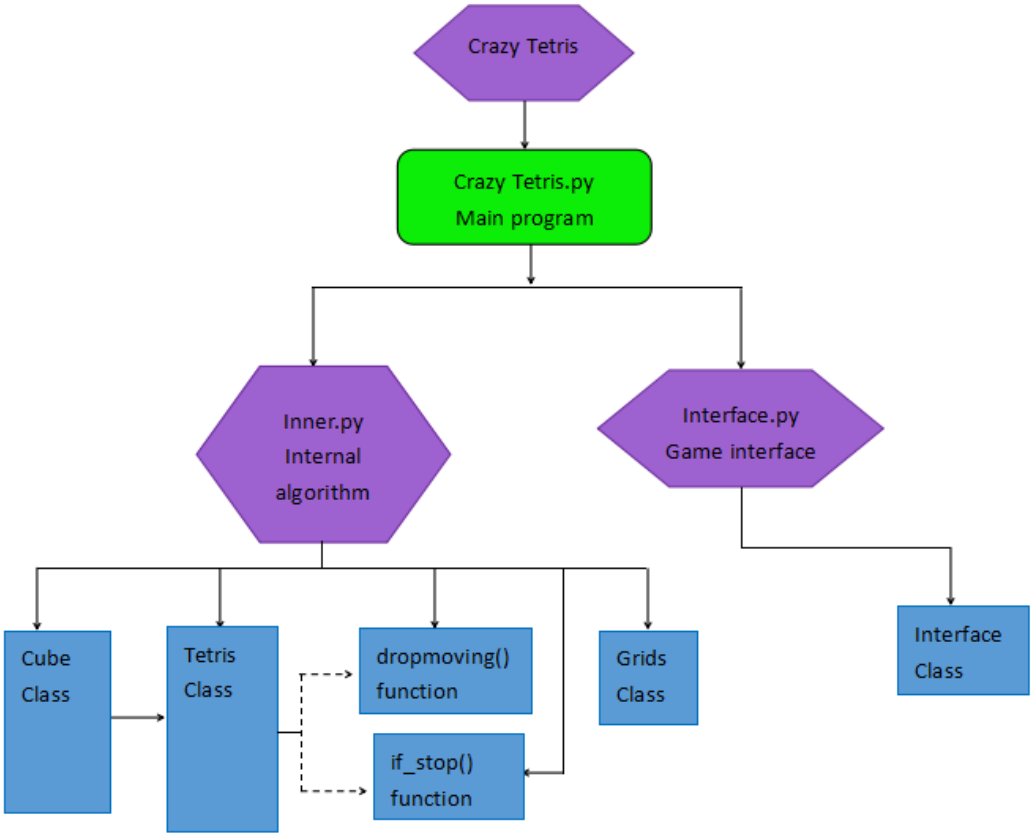
3.1.3 Game target

This game has no established victory conditions. Players eliminate as many rows as possible by judging and timely operations, and score as high as possible.

3.2 Modules of Crazy Tetris

3.2.1 Module Overview

The Crazy Tetris program is divided into three modules: Crazy Tetris.py, inner.py, and interface.py. The first one is the main program instantiated, and the last two are the internal logic and algorithm of the game, and the design module of the game interface.



3.2.2 Module list

(1) Crazy Tetris.py Module

Import module: from inner import *, from interface import *

Main () function: the main program, direct instantiation of the game interface, indirect instantiation of the game's various elements.

(2) Inner.py Module

Cube class: Classes of squares (ie, units of Tetris)

Tetris class: Classes of Tetris

Dropmoving() function: automatic drop of Tetris instance

If_stop() function: Determines if the Tetris instance falls to the bottom

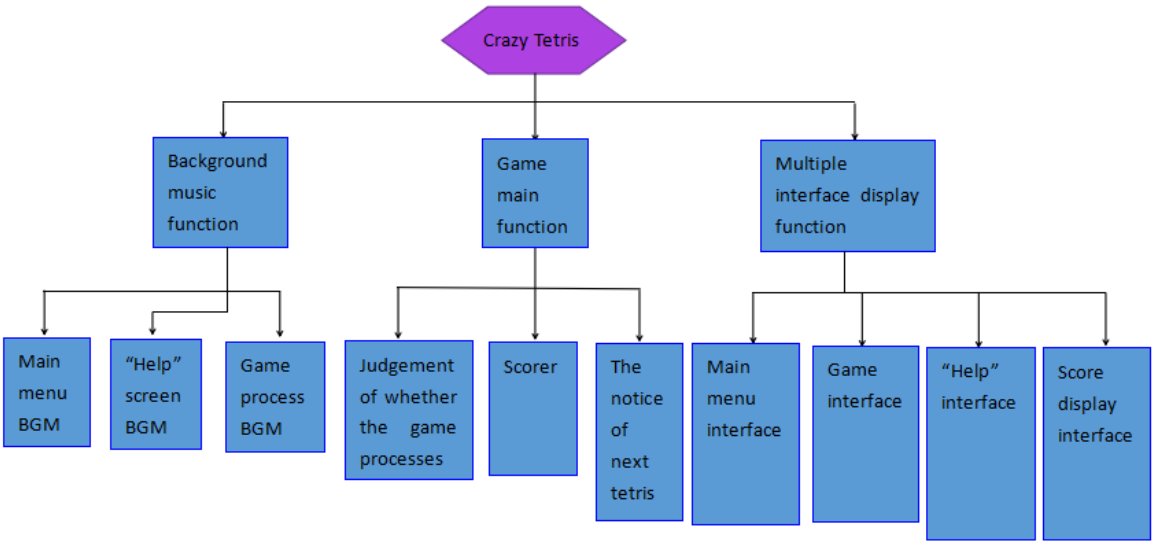
Grids class: used to record the color and coordinates of the blocks that have fallen to the bottom

(3) Interface.py Module

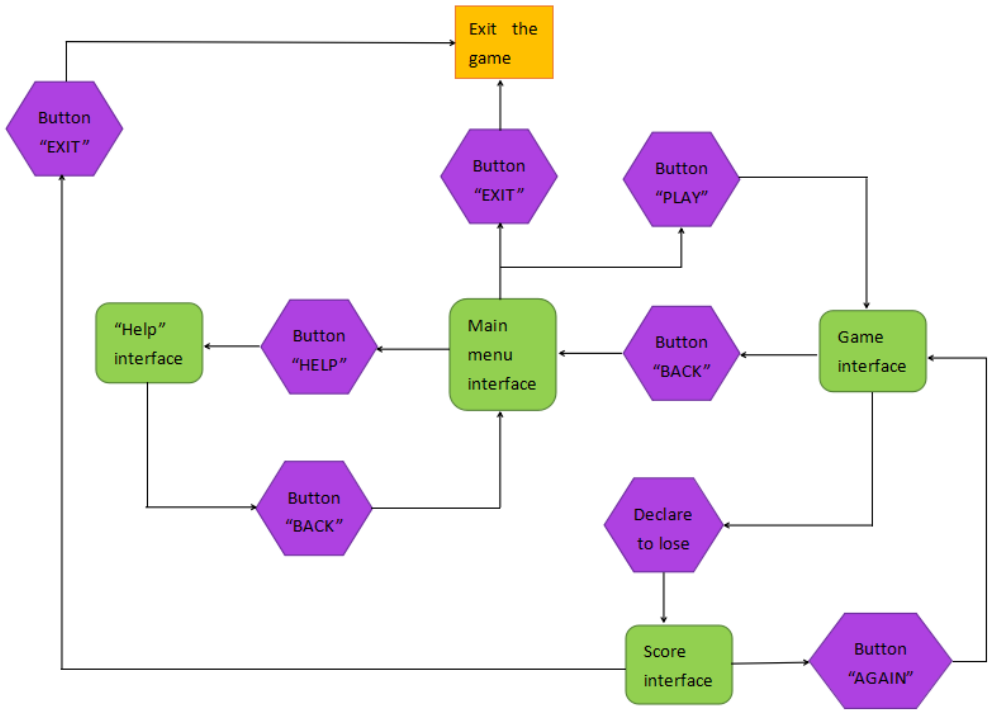
Interface class: Generates four interfaces (menu interface, game interface, scoring interface, and help interface) and the way to switch between them.

3.3 Main design of Crazy Tetris

3.3.1 Overall design of functions



3.3.2 Interface switching design



3.3.3 Algorithm design

Most functions of this game are implemented by classes, and a small number of function methods are used. Therefore, the algorithm design of this game is mainly

analyzed from the perspective of class design.

(1) The creation of each interface

Take the main interface as the starter:


```
def __init__(self):
    # create menu interface as root1 and display it
    self.root1 = Tk()
    self.root1.title('Crazy Teteris')
    self.base1 = Canvas(self.root1, width=500, height=700, bg='black')
    self.base1.pack()
    self.background1 = PhotoImage(file='background.gif')
    self.base1.create_image(250, 350, image=self.background1)
    self.menuImg = PhotoImage(file='menu.gif')
    self.base1.create_image(254, 280, image=self.menuImg)
    self.menuSound = winsound.PlaySound('sounds/menu', winsound.SND_ASYNC+winsound.SND_LOOP)
    self.root1.bind('<Button-1>', self.switch)
```

Create menu interface as root1 and display it.

The remaining three interfaces are root2: game interface; root3: help interface; root4: score display interface. The initialization mode of each interface is basically the same; the main differences are the following two points:

A. The other three interfaces finally call the withdraw() method to hide it and only show root1.

B. The background music of the other three interfaces is not initialized, and is created after switching the interface.

(2) The realization of the starting animation

The game startup animation includes the author's signature, game logo, and the gradient of the main menu. Here is an analysis of the first part as an example:

```
def launch(self):
    for i in range(1, 11):
        launchImg = PhotoImage(file='launching/'+str(i)+'.gif')
        launching = self.base1.create_image(254, 280, image=launchImg)
        self.base1.update()
        sleep(0.1)
        self.base1.delete(launching)
    launchImg = PhotoImage(file='launching/10.gif')
    launching = self.base1.create_image(254, 280, image=launchImg)
    self.base1.update()
    sleep(0.75)
    for i in range(10,0,-1):
        launchImg = PhotoImage(file='launching/'+str(i)+'.gif')
        launching = self.base1.create_image(254, 280, image=launchImg)
        self.base1.update()
        sleep(0.1)
        self.base1.delete(launching)
```

It is the method of animation at the start of the game.

Create continuous display of gradients using create, display, update(), sleep(), delete().

(3) Interface and corresponding background music switching method

Take the example of switching from root1 to other interfaces:

```
def switch(self, event):
# to switch between two certain interfaces
    if self.current == 'root1':
    # to see if the current interface is root1, namely the menu interface
    # similarly hereinafter
        if 194 <= event.x <= 332 and 289 <= event.y <= 335:
        # to see if the mouse clicked at a button
        # similarly hereinafter
            winsound.PlaySound(self.menuSound, winsound.SND_PURGE)
            self.root2.deiconify()
            self.root1.withdraw()
            self.gamingSound = winsound.PlaySound('sounds/gaming', winsound.SND_ASYNC+winsound.SND_LOOP)
            self.current = 'root2'
            self.grids = Grids(self.gamingboard)
            nextTetris = 0
            self.score = Label(self.root2, text='0', fg='white', bg='black',
                               font=('Kozuka Gothic Pr6N B', 16)).place(x=363, y=241)
```

This method first determines the current interface through the variable self.current, and then makes different designs for different interfaces.

For the game interface root2, we need to reset the relevant variables of the game interface after switching to other interfaces:

```
if self.current == 'root2':
    if 1 <= event.x <= 76 and 1 <= event.y <= 76:
        winsound.PlaySound(self.gamingSound, winsound.SND_PURGE)
        self.menuSound = winsound.PlaySound('sounds/menu', winsound.SND_ASYNC+winsound.SND_LOOP)
        self.root1.deiconify()
        self.root2.withdraw()
        self.current = 'root1'
        # redefine root2
        self.root2 = Tk()
        self.root2.title('Crazy Teteris')
        self.base2 = Canvas(self.root2, width=500, height=700, bg='black')
        self.base2.pack()
        self.background = PhotoImage(file ='background.gif', master=self.root2)
        self.base2.create_image(250, 350, image = self.background)
        self.scoreboard = Canvas(self.root2, width=370, height=420, bg='black')
        self.scoreboard.place(x=67, y=67)
        self.slogan = PhotoImage(file ='slogan.gif', master=self.root2)
        Label(self.root2, image=self.slogan).place(x=69, y=67)
        self.root2back = PhotoImage(file ='backarrow.gif', master=self.root2)
        Label(self.root2, image=self.root2back).place(x=360, y=410)
        self.gamingboard = Canvas(self.root2, width=210, height=420, bg='black')
        self.gamingboard.place(x=149, y=67)
        self.gbb = PhotoImage(file ='winbackground.gif', master=self.root2)
        self.gamingboard.create_image(105, 210, image = self.gbb)
        Label(self.root2, text='SCORE', fg='white', bg='black',
                anchor='center', font=('Kozuka Gothic Pr6N B', 16)).place(x=366, y=197)
        Label(self.root2, text='NEXT', fg='white', bg='black', width=5,
                anchor='center', font=('Kozuka Gothic Pr6N B', 18)).place(x=363, y=69)
        self.scoreboard.create_line(295, 126, 372, 126, fill='white', width=2)
        self.scoreboard.create_line(295, 225, 372, 225, fill='white', width=2)
        self.root2.withdraw()
```

This part of the design is similar to the root1 design described above

(4) Core program under the game interface

The following is a blueTetris example for creating Tetris in the interface.py module

```
tetrisList = [blueTetris, greenTetris, indigoTetris, orangeTetris, purpleTetris, redTetris, yellowTetris]
if nextTetris == 0:
    tetris = choice(tetrisList)
else:
    tetris = nextTetris
nextTetris = choice(tetrisList)
nextImg = PhotoImage(file = 'next/'+nextTetris.color+'Next.gif', master=self.root2)
nextDisplay = self.scoreboard.create_image(334, 87, image=nextImg)
self.gamingboard.bind('<Button-1>',tetris.moveleft)
self.gamingboard.bind('<Button-3>',tetris.moveright)
self.gamingboard.bind('<Button-2>',tetris.rotate)
while not if_stop(tetris):
    dropmoving(tetris, self.gamingboard)
if if_stop(tetris):
    self.grids.getCubes(tetris)
    self.grids.display()
    self.grids.fillRow()
    self.score = Label(self.root2, text=str(int(self.grids.score)), fg='white', bg='black',
                       font=('Kozuka Gothic Pr6N B', 16)).place(x=363, y=241)
    self.scoreboard.delete(nextDisplay)
    if self.grids.if_endgame():
        winsound.PlaySound(self.gamingSound,winsound.SND_PURGE)
        self.helpSound = winsound.PlaySound('sounds/help', winsound.SND_ASYNC+winsound.SND_LOOP)
        self.root4.deiconify()
        Label(self.base4, text=str(int(self.grids.score)), fg='yellow', bg='black',
              anchor='center', font=('Kozuka Gothic Pr6N B', 40)).place(x=220, y=245)
        self.current = 'root4'
        self.root2.withdraw()
        # redefine root2
        self.root2 = Tk()
        self.root2.title('Crazy Teteris')
        self.base2 = Canvas(self.root2, width=500, height=700, bg='black')
        self.base2.pack()
        self.background = PhotoImage(file = 'background.gif', master=self.root2)
        self.base2.create_image(250, 350, image = self.background)
        self.scoreboard = Canvas(self.root2, width=370, height=420, bg='black')
        self.scoreboard.place(x=67,y=67)
        self.slogan = PhotoImage(file = 'slogan.gif', master=self.root2)
        Label(self.root2, image=self.slogan).place(x=69, y=67)
        self.root2back = PhotoImage(file = 'backarrow.gif', master=self.root2)
        Label(self.root2, image=self.root2back).place(x=360, y=410)
        self.gamingboard = Canvas(self.root2, width=210, height=420, bg='black')
        self.gamingboard.place(x=149,y=67)
        self.gbb = PhotoImage(file = 'winbackground.gif', master=self.root2)
        self.gamingboard.create_image(105, 210, image = self.gbb)
        Label(self.root2, text='SCORE', fg='white', bg='black',
              anchor='center', font=('Kozuka Gothic Pr6N B', 16)).place(x=366, y=197)
        Label(self.root2, text='NEXT', fg='white', bg='black', width=5,
              anchor='center', font=('Kozuka Gothic Pr6N B', 18)).place(x=363, y=69)
        self.scoreboard.create_line(295, 126, 372, 126, fill='white', width=2)
        self.scoreboard.create_line(295, 225, 372, 225, fill='white', width=2)
        self.root2.withdraw()
```

(5) The realization of Class Cube

Cube is a class of square boxes (i.e. a basic unit of Tetris).

```
def __init__(self, x, y, color, win, grids):
    # definition
    self.x = x
    self.realX = (x-1)*21 + 11
    self.y = y
    self.realY = (y-1)*21 + 11
    self.win = win
    self.grids = grids
    self.color = color
    self.img = PhotoImage(file = 'cube/cube_'+self.color+'.gif',master=self.win)
def display(self):
    # to make a cube visible
    self.display = self.win.create_image(self.realX, self.realY, image=self.img)
```

Create class of cubes (unit of a Tetris) and display a Cube instance.

```
def if_movable(self, direction):
    # return a bool to determine whether a cube can move to it's 'direction'
    if direction == 'left':
        if self.grids.document[(self.x-1, self.y)][0]==0:
            return True
        else:
            return False
    if direction == 'right':
        if self.grids.document[(self.x+1, self.y)][0]==0:
            return True
        else:
            return False
    if direction == 'down':
        if self.grids.document[(self.x, self.y+1)][0]==0:
            return True
        else:
            return False
```

Determines whether the Cube instance is movable in a specified direction and returns a bool value

(6) The realization of Class Tetris

Class Tetris is the class of Tetris as a whole. The following describes the initialization method of this class, the method of moving to the left (the method of moving to the right is completely symmetrical), and the method of rotating it counterclockwise by

90 degrees (taking the blue Tetris as an example):

```
def __init__(self, color, centerX, centerY, origin, win, grids, rotateCount=0):
# definition
    self.color = color
    self.centerX = centerX
    self.centerY = centerY
    self.origin = origin
    self.cubes = {}
    self.grids = grids
    self.win = win
    self.rotateCount = rotateCount
    # count how many times have a tetris rotated
    for i in range(3, 8):
        #create a 5x5 map above the gaming board to temporarily record the position of the tetris
        for j in range(-3, 2):
            self.cubes[(i, j)] = 0
    for i in range(3, 8):
        for j in range(-3, 2):
            if self.origin[(i, j)] == 1:
                self.cubes[(i, j)] = Cube(i, j, color, win, self.grids)
                self.cubes[(i, j)].display()
```

Create class of Tetris

```
def moveleft(self, event):
    # to move the tetris towards left
    row1, row2, row3, row4, row5 = True, True, True, True, True
    # the tetris is movable towards left in every row by default
    r1, r2, r3, r4, r5 = [8], [8], [8], [8], [8]
    for i in range(3, 8):
        if self.cubes[(i,-3)] != 0:
            r1.append(i)
    r1min = min(r1)
    if r1min != 8:
        if not self.cubes[(r1min,-3)].if_movable('left'):
            row1 = False
            # if in row1 the tetris is blocked when moving left, return row1 as False.
            # similarly hereinafter
    for i in range(3, 8):
        if self.cubes[(i,-2)] != 0:
            r2.append(i)
    r2min = min(r2)
    if r2min != 8:
        if not self.cubes[(r2min,-2)].if_movable('left'):
            row2 = False
    for i in range(3, 8):
        if self.cubes[(i,-1)] != 0:
            r3.append(i)
    r3min = min(r3)
    if r3min != 8:
        if not self.cubes[(r3min,-1)].if_movable('left'):
            row3 = False
    for i in range(3, 8):
        if self.cubes[(i,0)] != 0:
            r4.append(i)
    r4min = min(r4)
    if r4min != 8:
        if not self.cubes[(r4min,0)].if_movable('left'):
            row4 = False
    for i in range(3, 8):
        if self.cubes[(i,1)] != 0:
            r5.append(i)
    r5min = max(r5)
    if r5min != 8:
        if not self.cubes[(r5min,0)].if_movable('left'):
            row5 = False
    if row1 and row2 and row3 and row4 and row5:
    # if the tetris is movable in evety row, conduct the action of moving
        self.centerX -= 1
        for i in range(3, 8):
            for j in range(-3, 2):
                if self.cubes[(i,j)] != 0:
                    self.cubes[(i,j)].x -= 1
                    self.cubes[(i,j)].realX -= 21
                    self.win.move(self.cubes[(i,j)].display,-21,0)
```

The method of moving left

```
def rotate(self, event):
    # to rotate the tetris anticlockwise
    coord = {}
    # create a dictionary to record the color and temporary coordinates of cubes of a tetris
    for i in range(3, 8):
        for j in range(-3, 2):
            coord[(i, j)] = [0, 0]
    for i in range(3, 8):
        for j in range(-3, 2):
            if self.cubes[(i, j)] != 0:
                self.win.delete(self.cubes[(i, j)].display)
                self.cubes[(i, j)] = 0
    if self.color == 'blue':
    # set method for rotating blue tetrises
    # similarly hereinafter except yellow
        if self.rotateCount % 4 == 0:
            coord[(4, 0)] = [4, 0]
            coord[(5, 0)] = [5, 0]
            coord[(6, 0)] = [6, 0]
            coord[(7, 0)] = [7, 0]
        if self.rotateCount % 4 == 1:
            coord[(5, 1)] = [5, 1]
            coord[(5, 0)] = [5, 0]
            coord[(5, -1)] = [5, -1]
            coord[(5, -2)] = [5, -2]
        if self.rotateCount % 4 == 2:
            coord[(4, 0)] = [4, 0]
            coord[(5, 0)] = [5, 0]
            coord[(6, 0)] = [6, 0]
            coord[(7, 0)] = [7, 0]
        if self.rotateCount % 4 == 3:
            coord[(5, 0)] = [5, 0]
            coord[(5, -1)] = [5, -1]
            coord[(5, -2)] = [5, -2]
            coord[(5, -3)] = [5, -3]
    for i in range(3, 8):
        for j in range(-3, 2):
            if coord[(i, j)] != [0, 0]:
                deltaX = i - 5
                deltaY = j + 1
                self.cubes[(i, j)] = Cube(self.centerX + deltaX, self.centerY + deltaY, self.color, self.win, self.grids)
                self.cubes[(i, j)].display()
    self.rotateCount += 1
    # add 1 to rotateCount to decide how to rotate next time
```

The method of Tetris to rotate 90 degrees counterclockwise (Take the bule Tetris as an example)

(7) The realization of Class Grids

An instance of the Grids class is used to store the fallen Tetris, and to eliminate rows, score, and determine whether the game is terminated.

```
def display(self):
# to make recorded cubes visible
    for i in range(1, 11):
        for j in range(1, 21):
            if self.document[(i, j)][0] == 1:
                self.cubes[(i, j)] = Cube(i, j, self.document[(i, j)][1], self.win, self)
                self.cubes[(i, j)].display()
```

The method of displaying the grid records of the Grids instance

```
def fillRow(self):
# check if any row is filled and delete filled one(s)
    for j in range(1, 21):
        t_or_f = True
        for i in range(1, 11):
            if self.document[(i, j)][0] == 0:
                t_or_f = False
        if t_or_f:
            self.score += 10
            for i in range(1, 11):
                if self.cubes[(i, j)] != 0:
                    for p in range(1, 11):
                        for q in range(1, 21):
                            if self.cubes[(p, q)] != 0:
                                self.win.delete(self.cubes[(p, q)].display)
                    self.cubes[(i, j)] = 0
            for k in range(j, 0, -1):
                for i in range(1, 11):
                    self.document[(i, k)] = self.document[(i, k-1)][:]
        for m in range(1, 11):
            for n in range(1, 21):
                if self.document[(m, n)][0] == 1:
                    self.cubes[(m, n)] = Cube(m, n, self.document[(m, n)][1], self.win, self)
                    self.cubes[(m, n)].display()
```

Check if any rows are filled and eliminate the filled rows

```
def if_endgame(self):
# check if the player lose the game (the last tetris reach the top of gaming board)
    t_or_f = False
    for i in range(1, 11):
        if self.document[(i, 0)][0] == 1:
            t_or_f = True
    return t_or_f
```

Determine whether the game is over, that is, whether the last Tetris reaches the top line of the game interface

Acknowledgements

There is no denying that it is a tremendous challenge for our team, DionyStudio, to do such a huge project, having been learning Python for only 4 months. Thanks to our proper distribution of jobs: the code by all members and mainly by Xue Jianyue, the report by Zhou Haobo, the slides by Song Jiangkanghong, the presentation by MAN HUA TAO, we finally overcame all difficulties and finished the whole project successfully. Every team member contributes a lot to the projects from diverse perspectives, such as ideas, design, and so on. At the same time, every team member learns a lot about Python through the project, as well as computing science.

Our first acknowledgements should go to our teacher Bao Yang, who taught us a great amount of Python knowledge through the 4 months' course. What's more, when we met problems, Bao could always help us in details. It is obvious that without his help and responsible teaching, we cannot make this huge project by ourselves.

We should also give our sincere thanks to some of our friends who majors in computer science and offered lots of help. They assisted us to write the code and gave us much advice. It is their help that propelled us to move forward step by step and overcome difficulties one by one.