
Pygame tutorial Documentation

Release 2019

Raphael Holzer

Jun 17, 2021

Contents:

1	Introduction to Pygame	3
1.1	Import the module	3
1.2	Show the event loop	4
1.3	Quit the event loop properly	4
1.4	Define colors	5
1.5	Switch the background color	6
1.6	Import pygame.locals	7
1.7	Use a dictionary to decode keys	7
1.8	Change the window caption	8
1.9	Explore a simple ball game	9
2	Drawing graphics primitives	13
2.1	Draw solid and outlined rectangles	13
2.2	Draw solid and outlined ellipses	14
2.3	Detect the mouse	15
2.4	Draw a rectangle with the mouse	16
2.5	Draw multiple shapes	17
2.6	Draw a polygon line with the mouse	19
3	Work with rectangles	23
3.1	Virtual attributes	23
3.2	Points of interest	25
3.3	Horizontal and vertical alignment	25
3.4	Move a rectangle with keys	27
3.5	Inflate a rectangle	28
3.6	Clip a rectangle	28
3.7	Move a rectangle with the mouse	29
3.8	A self-moving a rectangle	30
3.9	Colliding points	31
3.10	Colliding rectangles	32
3.11	Overlapping rectangles	33
3.12	The common code	35
4	Work with images	37
4.1	Load an image	37
4.2	Move the image with the mouse	38
4.3	Rotate and Scale the image	39

4.4	Reset the image to the original	40
4.5	Flip the image	40
4.6	Detect edges with the Laplacian	40
4.7	Transform the image with the mouse	41
5	Work with text	47
5.1	Initialize a font	47
5.2	Render the text	48
5.3	Edit text with the keyboard	50
5.4	Add a blinking cursor	51
6	Making apps with Pygame	53
6.1	Create the App class	53
6.2	Add the Text class	54
6.3	Shortcut keys	56
6.4	Fullscreen, resizable and noframe mode	57
6.5	Add the Scene class	58
6.6	Scenes with background images	60
6.7	Automatic node placement	62
7	Create a graphical user interface (GUI)	65
7.1	Text attributes	65
7.2	Horizontal and vertical alignment	67
7.3	Text attributes	68
7.4	Editable text	69
7.5	Buttons	71
7.6	ListBox	72
7.7	Detecting double-clicks	72
8	Playing sound	75
8.1	Making sounds	75
9	Board Games	77
9.1	Selecting cells with the mouse	78
9.2	Adding background color	78
9.3	Create a checkerboard pattern	79
10	Simple games	81
10.1	Pong	81
11	Breakout	85
11.1	Making an app	86
12	Tiles	91
12.1	Create a Tileset	93
12.2	Create a Tilemap	93
12.3	Navigation	95
12.4	Tiled : a tile-map editor	95
13	About Sphinx	97
13.1	Getting started	97
13.2	reStructuredText	98
13.3	Include from a file	99
13.4	Directives	100
13.5	Math formulas	102

13.6	The app module	102
13.7	Glossary	107
14	Indices and tables	109
	Python Module Index	111
	Index	113

This tutorial explains how to make interactive applications and games using Pygame. The first part is a general introduction to Pygame without defining classes and objects. The second part introduces classes and objects and teaches an **object-oriented programming** approach to making apps.

CHAPTER 1

Introduction to Pygame

Pygame is a multimedia library for Python for making games and multimedia applications.

It is a wrapper around the SDL (Simple DirectMedia Layer) library. In this section we introduce the basics of pygame functions without defining classes and objects.

1.1 Import the module

To use the methods in the Pygame library, the module must first be imported:

```
import pygame
```

The import statement writes the pygame version and a link to the Pygame website to the console (as a side effect):

```
pygame 1.9.6  
Hello from the pygame community.  
https://www.pygame.org/contribute.html
```

The Pygame import statement is always placed at the beginning of the program. It imports the pygame classes, methods and attributes into the current name space. Now this new methods can be called via `pygame.method()`.

For exemple we can now initialize or quit **pygame** with the following command:

```
pygame.init()  
pygame.quit()
```

The function `display.set_mode()` sets the screen size. It returns a Surface object wich we assign to the variable `screen`. This variable will be one of the most used variables. It represents the window we see:

```
screen = pygame.display.set_mode((640, 240))
```

You can now run this program and test it. At this moment it does very little. It opens a window and closes it immediately.

1.2 Show the event loop

The most essential part of any interactive application is the **event loop**. Reacting to events allows the user to interact with the application. Events are the things that can happen in a program, such as a

- mouse click,
- mouse movement,
- keyboard press,
- joystick action.

The following is an infinite loop which prints all events to the console:

```
while True:
    for event in pygame.event.get():
        print(event)
```

Try to move the mouse, click a mouse button, or type something on the keyboard. Every action you do produces an event which will be printed on the console. This will look something like this:

```
<Event(4-MouseMotion {'pos': (173, 192), 'rel': (173, 192), 'buttons': (0, 0, 0),
→ 'window': None})>
<Event(2-KeyDown {'unicode': 'a', 'key': 97, 'mod': 0, 'scancode': 0, 'window': None}
→)>
<Event(3-KeyUp {'key': 97, 'mod': 0, 'scancode': 0, 'window': None})>
<Event(12-Quit {})>
```

As we are in an infinite loop, it is impossible to quit this program from within the application. In order to quit the program, make the console the active window and type `ctrl-C`. This will write the following message to the console:

```
^CTraceback (most recent call last):
File "/Users/raphael/GitHub/pygame-tutorial/docs/tutorial1/introl.py", line 7, in
→ <module>
    for event in pygame.event.get():
KeyboardInterrupt
```

introl.py

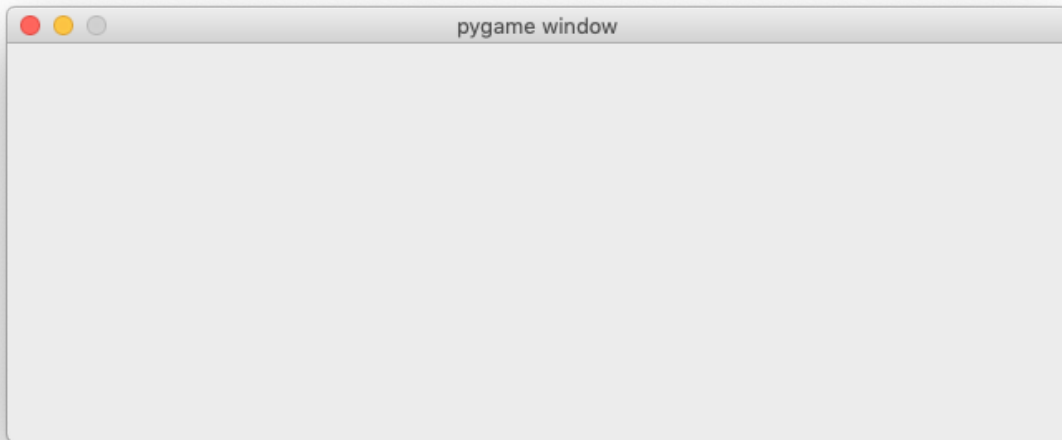
1.3 Quit the event loop properly

In order to quit the application properly, from within the application, by using the window close button (QUIT event), we modify the event loop. First we introduce the boolean variable `running` and set it to `True`. Within the event loop we check for the QUIT event. If it occurs, we set `running` to `False`:

```
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

pygame.quit()
```

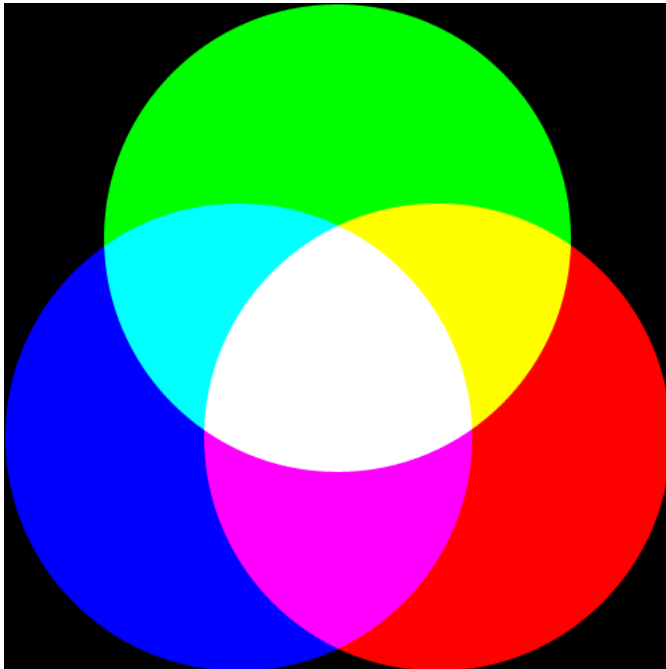
Once the event loop, we call the `pygame.quit()` function to end the application correctly.



```
intro2.py
```

1.4 Define colors

Colors are defined as tuples of the base colors **red**, **green** and **blue**. This is called the **RGB model**. Each base color is represented as a number between 0 (minimum) and 255 (maximum) which occupies 1 byte in memory. An RGB color is thus represented as a 3-byte value. Mixing two or more colors results in new colors. A total of 16 million different colors can be represented this way.



Let's define the base colors as tuples of the three base values. Since colors are constants, we will write them using capitals. The absence of all colors results in black. The maximum value for all three components results in white.

Three identical intermediate values result in gray:

```
BLACK = (0, 0, 0)
GRAY = (127, 127, 127)
WHITE = (255, 255, 255)
```

The tree base colors are defined as:

```
RED = (255, 0, 0)
GREEN = (0, 255, 0)
BLUE = (0, 0, 255)
```

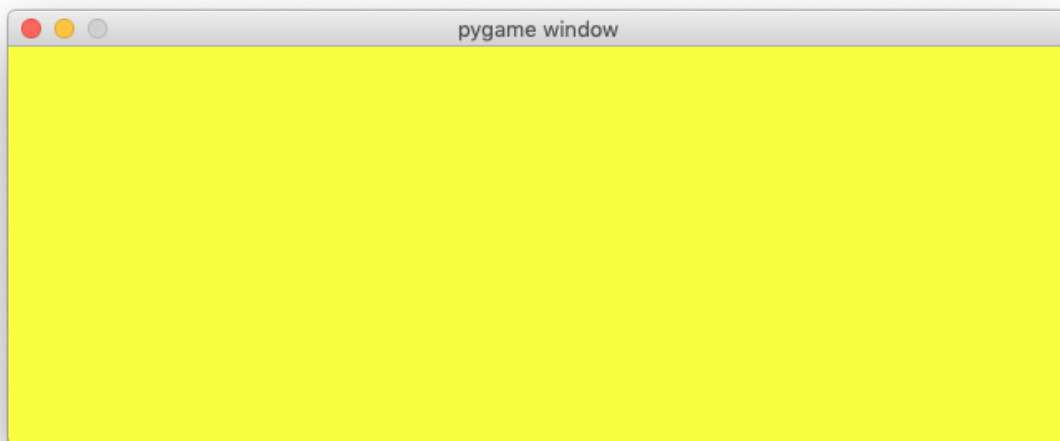
By mixing two base colors we obtained more colors:

```
YELLOW = (255, 255, 0)
CYAN = (0, 255, 255)
MAGENTA = (255, 0, 255)
```

At the end of the event loop, we add the following:

```
screen.fill(YELLOW)
pygame.display.update()
```

The method `fill(color)` fills the whole screen with the specified color. At this point nothing will be displayed. In order to show anything, the function `pygame.display.update()` must be called.



intro3.py

1.5 Switch the background color

At the beginning of the program we add a new variable `background` and initialize it to gray:

```
background = GRAY
```

Within the event loop we are looking now for `KEYDOWN` events. If found, we check if the R or G keys have been pressed and change the background color to red (R) and green (G). This is the code added in the event loop:

```
if event.type == pygame.KEYDOWN:
    if event.key == pygame.K_r:
        background = RED
    elif event.key == pygame.K_g:
        background = GREEN
```

In the drawing section we use now the variable `background` representing the background color:

```
screen.fill(background)
pygame.display.update()
```

Test the program. Pressing the R and G keys allows you to switch the background color.

intro4.py

1.6 Import `pygame.locals`

The `pygame.locals` module contains some 280 constants used and defined by `pygame`. Placing this statement at the beginning of your program imports them all:

```
import pygame
from pygame.locals import *
```

We find the key modifiers (alt, ctrl, cmd, etc.)

```
KMOD_ALT, KMOD_CAPS, KMOD_CTRL, KMOD_LALT,
KMOD_LCTRL, KMOD_LMETA, KMOD_LSHIFT, KMOD_META,
KMOD_MODE, KMOD_NONE, KMOD_NUM, KMOD_RALT, KMOD_RCTRL,
KMOD_RMETA, KMOD_RSHIFT, KMOD_SHIFT,
```

the number keys:

```
K_0, K_1, K_2, K_3, K_4, K_5, K_6, K_7, K_8, K_9,
```

the special character keys:

```
K_AMPERSAND, K_ASTERISK, K_AT, K_BACKQUOTE,
K_BACKSLASH, K_BACKSPACE, K_BREAK,
```

the letter keys of the alphabet:

```
K_a, K_b, K_c, K_d, K_e, K_f, K_g, K_h, K_i, K_j, K_k, K_l, K_m,
K_n, K_o, K_p, K_q, K_r, K_s, K_t, K_u, K_v, K_w, K_x, K_y, K_z,
```

Instead of writing `pygame.KEYDOWN` we can now just write `KEYDOWN`.

1.7 Use a dictionary to decode keys

The easiest way to decode many keys, is to use a dictionary. Instead of defining many if-else cases, we just create a dictionary with the keyboard key entries. In this example we want to associate 8 different keys with 8 different background colors. At the beginning of the program we define this key-color dictionary:

```
key_dict = {K_k:BLACK, K_r:RED, K_g:GREEN, K_b:BLUE,
            K_y:YELLOW, K_c:CYAN, K_m:MAGENTA, K_w:WHITE}

print(key_dict)
```

Printing the dictionary to the console gives this result:

```
{107: (0, 0, 0), 114: (255, 0, 0), 103: (0, 255, 0), 98: (0, 0, 255),
121: (255, 255, 0), 99: (0, 255, 255), 109: (255, 0, 255), 119: (255, 255, 255)}
```

The keys are presented here with their ASCII code. For exaple the ASCII code for k is 107. Colors are represented as tuples. The color black is represented as (0, 0, 0).

The event loop now becomes very simple. First we check if the event type is a KEYDOWN event. If yes, we check if the event key is in the dictionary. If yes, we look up the color which is associated with that key and set the background color to it:

```
if event.type == KEYDOWN:
    if event.key in key_dict:
        background = key_dict[event.key]
```

Try to press the 8 specified keys to change the background color.

1.8 Change the window caption

The fonction `pygame.display.set_caption(title)` allows to change the caption (title) of the application window. We can add this to the event loop:

```
if event.key in key_dict:
    background = key_dict[event.key]

    caption = 'background color = ' + str(background)
    pygame.display.set_caption(caption)
```

This will display the RGB value of the current background color in the window caption.



intro5.py

1.9 Explore a simple ball game

To show what Pygame can do, here is a simple program which demonstrates a bouncing ball animation. The program uses the Rect class to represent a rectangular region. An instance is created from the ball image:

```
rect = ball.get_rect()
```

A Rect object has 4 attributes:

```
rect.left
rect.top
rect.right
rect.bottom
```

A Rect object can be moved with the `move()` method:

```
rect = rect.move(speed)
```

After importing the pygame module, we define a few variables such as screen size and two colors:

```
import pygame
from pygame.locals import *

size = 640, 320
width, height = size
GREEN = (150, 255, 150)
RED = (255, 0, 0)
```

Then we initialize pygame and create the screen variable:

```
pygame.init()
screen = pygame.display.set_mode(size)
running = True
```

The ball position is represented with a Rect object:

```
ball = pygame.image.load("ball.gif")
rect = ball.get_rect()
speed = [2, 2]
```

Inside the event loop we only check for the QUIT event:

```
while running:
    for event in pygame.event.get():
        if event.type == QUIT:
            running = False
```

Then we move the rectangle and check the left/right and top/bottom borders:

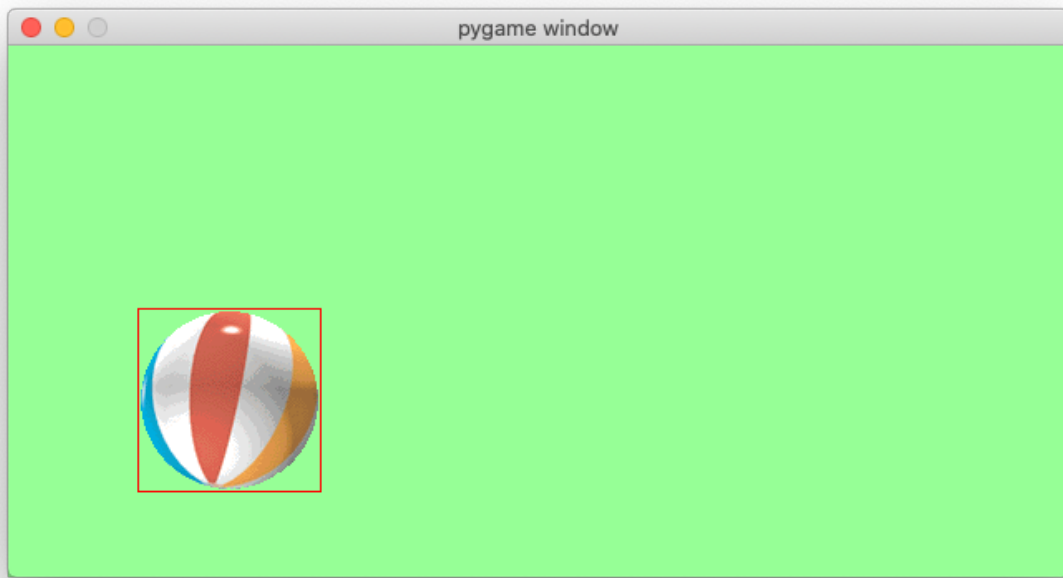
```
rect = rect.move(speed)
if rect.left < 0 or rect.right > width:
    speed[0] = -speed[0]
if rect.top < 0 or rect.bottom > height:
    speed[1] = -speed[1]
```

Finally we draw a green background, a red rectangle and the ball image:

```
screen.fill(GREEN)
pygame.draw.rect(screen, RED, rect, 1)
screen.blit(ball, rect)
pygame.display.update()

pygame.quit()
```

This is what the ball and the Rect outline looks:



ball.gif

Try to understand what the program does. Then try to modify it's parameters.

intro6.py

Drawing graphics primitives

The `pygame.draw` module allows to draw simple shapes to a surface. This can be the screen surface or any Surface object such as an image or drawing:

- `rectangle`
- `polygon`
- `circle`
- `ellipse`

The functions have in common that they:

- take a **Surface** object as first argument
- take a color as second argument
- take a width parameter as last argument
- return a **Rect** object which bounds the changed area

the following format:

```
rect(Surface, color, Rect, width) -> Rect
polygon(Surface, color, pointlist, width) -> Rect
circle(Surface, color, center, radius, width) -> Rect
```

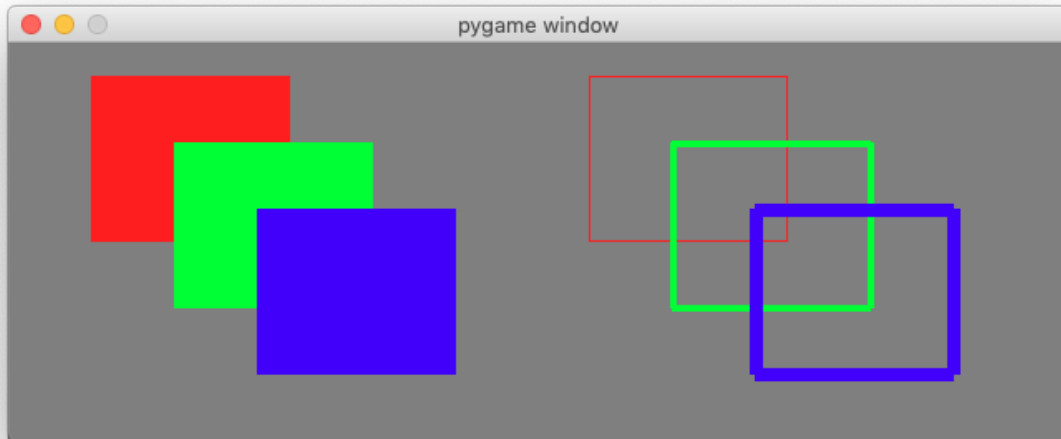
Most of the functions take a width argument. If the width is 0, the shape is filled.

2.1 Draw solid and outlined rectangles

The following draws first the background color and then adds three overlapping solid rectangles and next to it three outlined overlapping rectangles with increasing line width:

```
screen.fill(background)
pygame.draw.rect(screen, RED, (50, 20, 120, 100))
pygame.draw.rect(screen, GREEN, (100, 60, 120, 100))
pygame.draw.rect(screen, BLUE, (150, 100, 120, 100))

pygame.draw.rect(screen, RED, (350, 20, 120, 100), 1)
pygame.draw.rect(screen, GREEN, (400, 60, 120, 100), 4)
pygame.draw.rect(screen, BLUE, (450, 100, 120, 100), 8)
```



Try to modify the parameters and play with the drawing function.

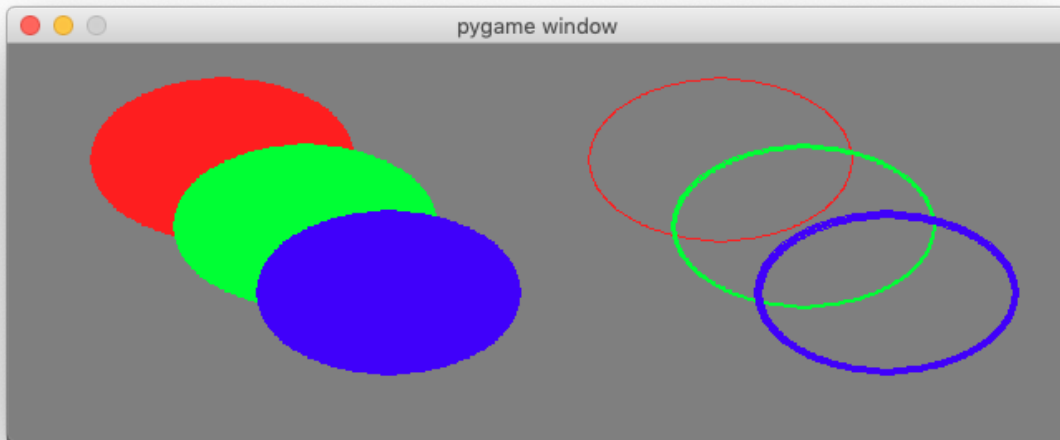
2.2 Draw solid and outlined ellipses

The following code draws first the background color and then adds three overlapping solid ellipses and next to it three outlined overlapping ellipses with increasing line width:

```
screen.fill(background)
pygame.draw.ellipse(screen, RED, (50, 20, 160, 100))
pygame.draw.ellipse(screen, GREEN, (100, 60, 160, 100))
pygame.draw.ellipse(screen, BLUE, (150, 100, 160, 100))

pygame.draw.ellipse(screen, RED, (350, 20, 160, 100), 1)
pygame.draw.ellipse(screen, GREEN, (400, 60, 160, 100), 4)
pygame.draw.ellipse(screen, BLUE, (450, 100, 160, 100), 8)

pygame.display.update()
```



draw2.py

2.3 Detect the mouse

Pressing the mouse buttons produces `MOUSEBUTTONDOWN` and `MOUSEBUTTONUP` events. The following code in the event loop detects them and writes the event to the console:

```
for event in pygame.event.get():
    if event.type == QUIT:
        running = False
    elif event.type == MOUSEBUTTONDOWN:
        print(event)
    elif event.type == MOUSEBUTTONUP:
        print(event)
```

Pressing the mouse buttons produces this kind of events:

```
<Event(5-MouseButtonDown {'pos': (123, 88), 'button': 1, 'window': None})>
<Event(6-MouseButtonUp {'pos': (402, 128), 'button': 1, 'window': None})>
<Event(5-MouseButtonDown {'pos': (402, 128), 'button': 3, 'window': None})>
<Event(6-MouseButtonUp {'pos': (189, 62), 'button': 3, 'window': None})>
```

Just moving the mouse produces a `MOUSEMOTION` event. The following code detects them and writes the event to the console:

```
elif event.type == MOUSEMOTION:
    print(event)
```

Moving the mouse produces this kind of event:

```
<Event(4-MouseMotion {'pos': (537, 195), 'rel': (-1, 0), 'buttons': (0, 0, 0), 'window': None})>
<Event(4-MouseMotion {'pos': (527, 189), 'rel': (-10, -6), 'buttons': (0, 0, 0), 'window': None})>
```

(continues on next page)

(continued from previous page)

```
<Event(4-MouseMotion {'pos': (508, 180), 'rel': (-19, -9), 'buttons': (0, 0, 0),  
↪ 'window': None})>
```

2.4 Draw a rectangle with the mouse

We can use this three events to draw a rectangle on the screen. We define the rectangle by its diagonal start and end point. We also need a flag which indicates if the mouse button is down and if we are drawing:

```
start = (0, 0)  
size = (0, 0)  
drawing = False
```

When the mouse button is pressed, we set start and end to the current mouse position and indicate with the flag that the drawing mode has started:

```
elif event.type == MOUSEBUTTONDOWN:  
    start = event.pos  
    size = 0, 0  
    drawing = True
```

When the mouse button is released, we set the end point and indicate with the flag that the drawing mode has ended:

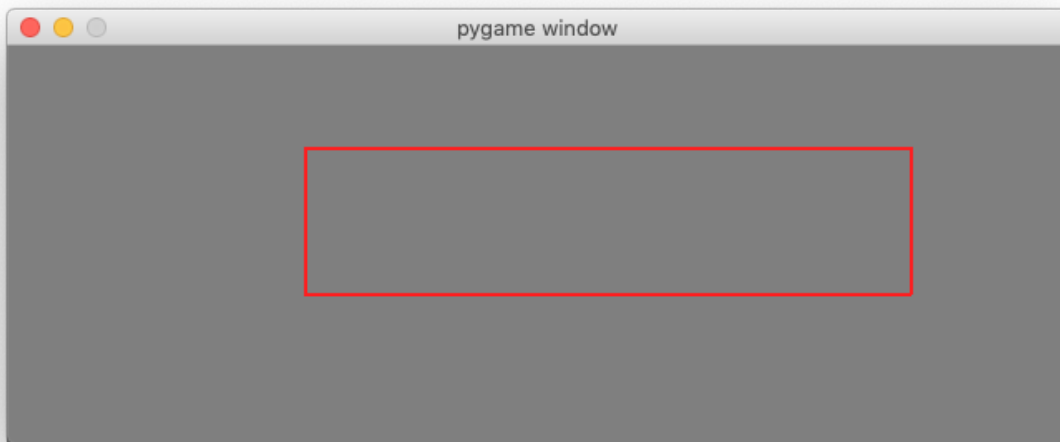
```
elif event.type == MOUSEBUTTONUP:  
    end = event.pos  
    size = end[0] - start[0], end[1] - start[1]  
    drawing = False
```

When the mouse is moving we have also have to check if we are in drawing mode. If yes, we set the end position to the current mouse position:

```
elif event.type == MOUSEMOTION and drawing:  
    end = event.pos  
    size = end[0] - start[0], end[1] - start[1]
```

Finally we draw the rectangle to the screen. First we fill in the background color. Then we calculate the size of the rectangle. Finally we draw it, and at the very last we update the screen:

```
screen.fill(GRAY)  
pygame.draw.rect(screen, RED, (start, size), 2)  
pygame.display.update()
```



mouse2.py

2.5 Draw multiple shapes

To draw multiple shapes, we need to place them into a list. Besides variables for start, end and drawing we add a rectangle list:

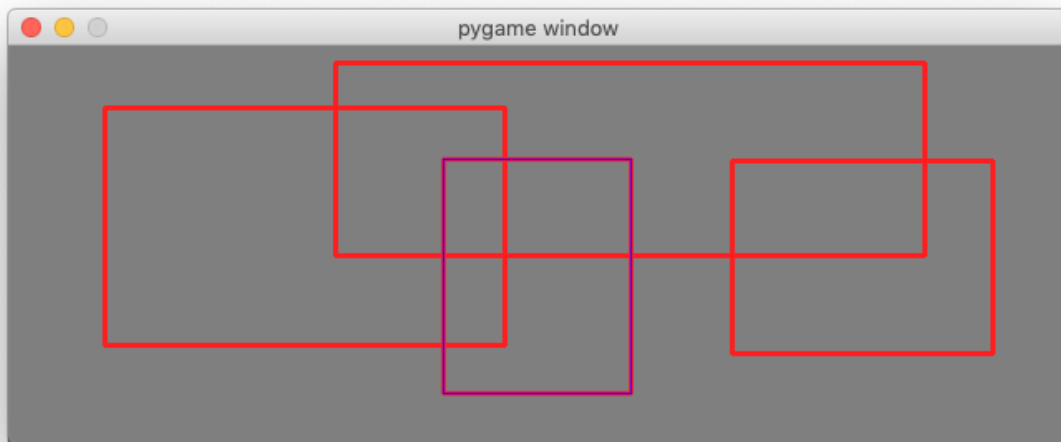
```
start = (0, 0)
size = (0, 0)
drawing = False
rect_list = []
```

When drawing of an object (rectangle, circle, etc.) is done, as indicated by a `MOUSEBUTTONUP` event, we create a rectangle and append it to the rectangle list:

```
elif event.type == MOUSEBUTTONUP:
    end = event.pos
    size = end[0]-start[0], end[1]-start[1]
    rect = pygame.Rect(start, size)
    rect_list.append(rect)
    drawing = False
```

In the drawing code, we first fill the background color, then iterate through the rectangle list to draw the objects (red, thickness=3), and finally we draw the current rectangle which is in the process of being drawn (blue, thickness=1):

```
screen.fill(GRAY)
for rect in rect_list:
    pygame.draw.rect(screen, RED, rect, 3)
pygame.draw.rect(screen, BLUE, (start, size), 1)
pygame.display.update()
```



Here is the complete file:

```
"""Place multiple rectangles with the mouse."""

import pygame
from pygame.locals import *

RED = (255, 0, 0)
BLUE = (0, 0, 255)
GRAY = (127, 127, 127)

pygame.init()
screen = pygame.display.set_mode((640, 240))

start = (0, 0)
size = (0, 0)
drawing = False
rect_list = []

running = True
while running:
    for event in pygame.event.get():
        if event.type == QUIT:
            running = False

        elif event.type == MOUSEBUTTONDOWN:
            start = event.pos
            size = 0, 0
            drawing = True

        elif event.type == MOUSEBUTTONUP:
            end = event.pos
            size = end[0]-start[0], end[1]-start[1]
            rect = pygame.Rect(start, size)
            rect_list.append(rect)
```

(continues on next page)

(continued from previous page)

```

        drawing = False

    elif event.type == MOUSEMOTION and drawing:
        end = event.pos
        size = end[0]-start[0], end[1]-start[1]

    screen.fill(GRAY)
    for rect in rect_list:
        pygame.draw.rect(screen, RED, rect, 3)
    pygame.draw.rect(screen, BLUE, (start, size), 1)
    pygame.display.update()

pygame.quit()

```

mouse3.py

2.6 Draw a polygon line with the mouse

To draw a polygon line we need to add the points to a list of points. First we define an empty point list and a drawing flag:

```

drawing = False
points = []

```

At the `MOUSEBUTTONDOWN` event we add the current point to the list and set the drawing flag to `True`:

```

elif event.type == MOUSEBUTTONDOWN:
    points.append(event.pos)
    drawing = True

```

At the `MOUSEBUTTONUP` event we deactivate the drawing flag:

```

elif event.type == MOUSEBUTTONUP:
    drawing = False

```

At the `MOUSEMOTION` event we move the last point in the polygon list if the drawing flag is set:

```

elif event.type == MOUSEMOTION and drawing:
    points[-1] = event.pos

```

If there are more than 2 points in the point list we draw a polygon line. Each `pygame.draw` function returns a `Rect` of the bounding rectangle. We display this bounding rectangle in green:

```

screen.fill(GRAY)
if len(points)>1:
    rect = pygame.draw.lines(screen, RED, True, points, 3)
    pygame.draw.rect(screen, GREEN, rect, 1)
pygame.display.update()

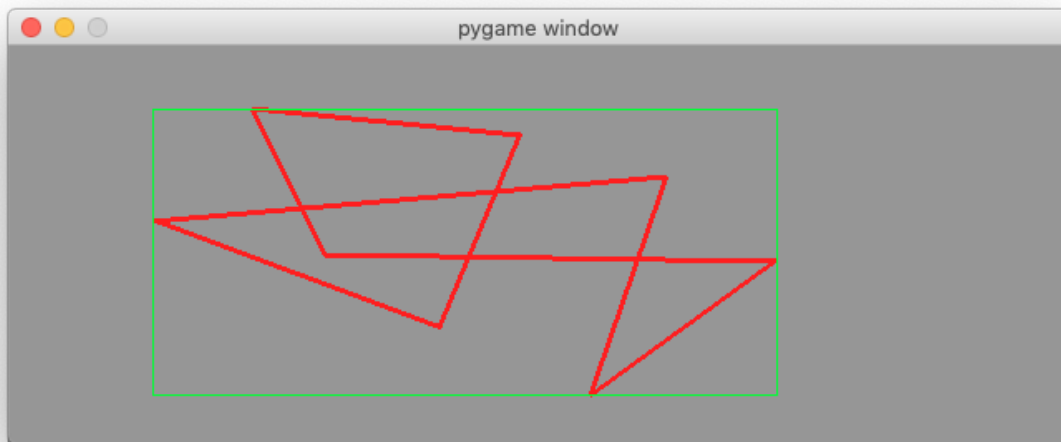
```

Pressing the `ESCAPE` key will remove the last point in the list:

```

elif event.type == KEYDOWN:
    if event.key == K_ESCAPE:
        if len(points) > 0:
            points.pop()

```



Here is the complete file:

```
"""Place a polygone line with the clicks of the mouse."""

import pygame
from pygame.locals import *

RED = (255, 0, 0)
GREEN = (0, 255, 0)
GRAY = (150, 150, 150)

pygame.init()
screen = pygame.display.set_mode((640, 240))

drawing = False
points = []
running = True

while running:
    for event in pygame.event.get():
        if event.type == QUIT:
            running = False

        elif event.type == KEYDOWN:
            if event.key == K_ESCAPE:
                if len(points) > 0:
                    points.pop()

        elif event.type == MOUSEBUTTONDOWN:
            points.append(event.pos)
            drawing = True

        elif event.type == MOUSEBUTTONUP:
            drawing = False

        elif event.type == MOUSEMOTION and drawing:
```

(continues on next page)

(continued from previous page)

```
        points[-1] = event.pos

    screen.fill(GRAY)
    if len(points)>1:
        rect = pygame.draw.lines(screen, RED, True, points, 3)
        pygame.draw.rect(screen, GREEN, rect, 1)
    pygame.display.update()

pygame.quit()
```

mouse4.py

Work with rectangles

The rectangle is a very useful object in graphics programming. It has its own `Rect` class in Pygame and is used to store and manipulate a rectangular area. A `Rect` object can be created by giving:

- the 4 parameters **left**, **top**, **width** and **height**
- the **position** and **size**
- an **object** which has a *rect* attribute

```
Rect(left, top, width, height)
Rect(pos, size)
Rect(obj)
```

A function which expects a `Rect` argument accepts equally one of the three above values. Methods which change the position or size, such as `move()` and `inflate()` leave the original `Rect` untouched and return a new `Rect`. They also have the *in place* version `move_ip` and `inflate_ip` which act upon the original `Rect`.

3.1 Virtual attributes

The `Rect` object has several virtual attributes which can be used to move and align the `Rect`. Assignment to these attributes just moves the rectangle without changing its size:

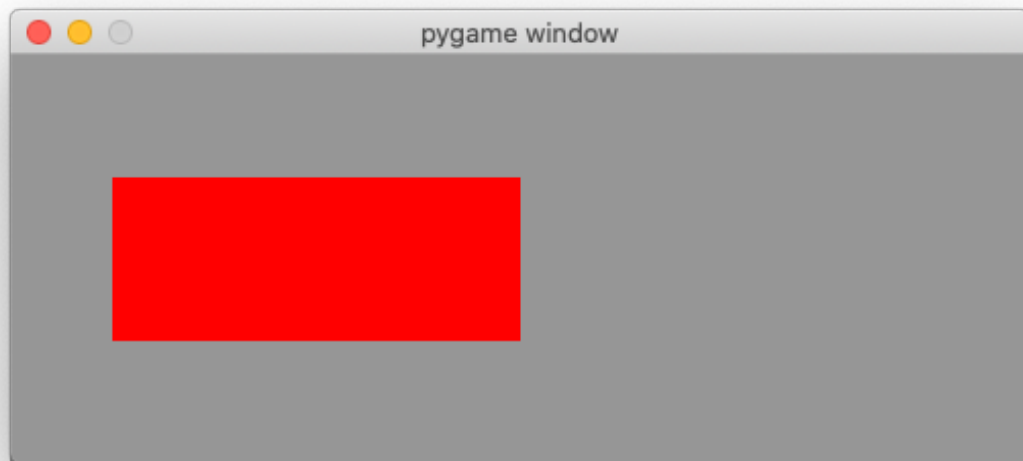
```
x, y
top, left, bottom, right
topleft, bottomleft, topright, bottomright
midtop, midleft, midbottom, midright
center, centerx, centery
```

The assignment of these 5 attributes changes the size of the rectangle, by keeping its top left position.

```
size, width, height, w, h
```

The following program prints these virtual attributes to the console:

```
x=50, y=60, w=200, h=80
left=50, top=60, right=250, bottom=140
center=(150, 100)
```



```
import pygame
from pygame.locals import *

SIZE = 500, 200
RED = (255, 0, 0)
GRAY = (150, 150, 150)

pygame.init()
screen = pygame.display.set_mode(SIZE)

rect = Rect(50, 60, 200, 80)
print(f'x={rect.x}, y={rect.y}, w={rect.w}, h={rect.h}')
print(f'left={rect.left}, top={rect.top}, right={rect.right}, bottom={rect.bottom}')
print(f'center={rect.center}')

running = True
while running:
    for event in pygame.event.get():
        if event.type == QUIT:
            running = False

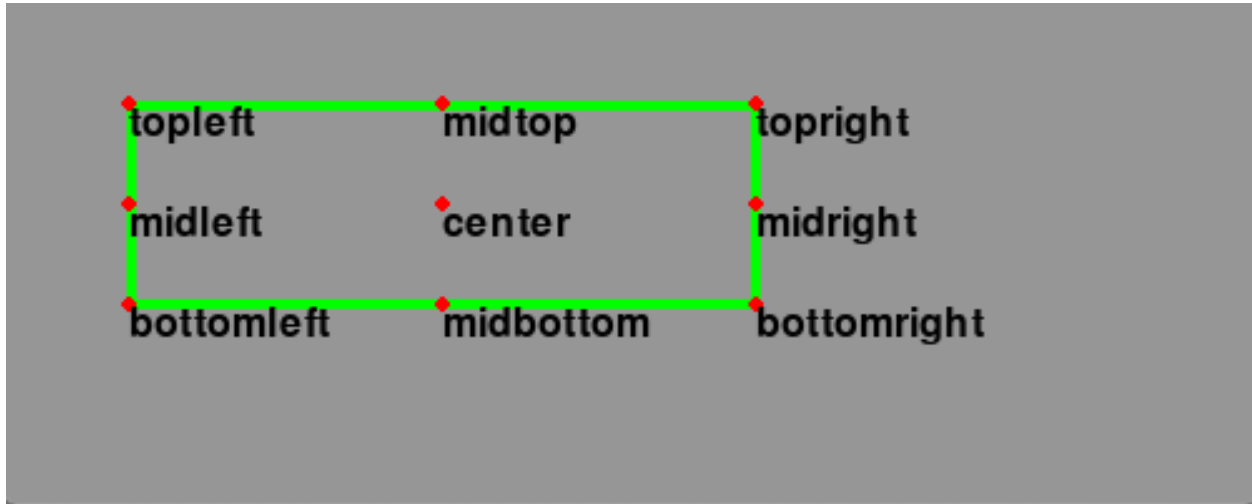
    screen.fill(GRAY)
    pygame.draw.rect(screen, RED, rect)
    pygame.display.flip()

pygame.quit()
```

rect1.py

3.2 Points of interest

The Rect class defines 4 cornerpoints, 4 mid points and 1 centerpoint.



```
from rect import *

pts = ('topleft', 'topright', 'bottomleft', 'bottomright',
       'midtop', 'midright', 'midbottom', 'midleft', 'center')

running = True
while running:
    for event in pygame.event.get():
        if event.type == QUIT:
            running = False

    screen.fill(GRAY)
    pygame.draw.rect(screen, GREEN, rect, 4)
    for pt in pts:
        pos = eval('rect.'+pt)
        draw_text(pt, pos)
        pygame.draw.circle(screen, RED, pos, 3)

    pygame.display.flip()

pygame.quit()
```

rect2.py

3.3 Horizontal and vertical alignment

In the following example we use 3 keys to align a rectangle horizontally:

- L - left
- C - center
- R - right

and 3 other keys to align the rectangle vertically:

- T - top
- M - middle
- B - bottom



```
from rect import *

rect = Rect(50, 60, 200, 80)

while running:
    for event in pygame.event.get():
        if event.type == QUIT:
            running = False

        if event.type == KEYDOWN:
            if event.key == K_l:
                rect.left = 0
            if event.key == K_c:
                rect.centerx = width//2
            if event.key == K_r:
                rect.right = width

            if event.key == K_t:
                rect.top = 0
            if event.key == K_m:
                rect.centery = height//2
            if event.key == K_b:
                rect.bottom = height

    screen.fill(GRAY)
    pygame.draw.rect(screen, BLUE, rect)
    pygame.display.flip()

pygame.quit()
```

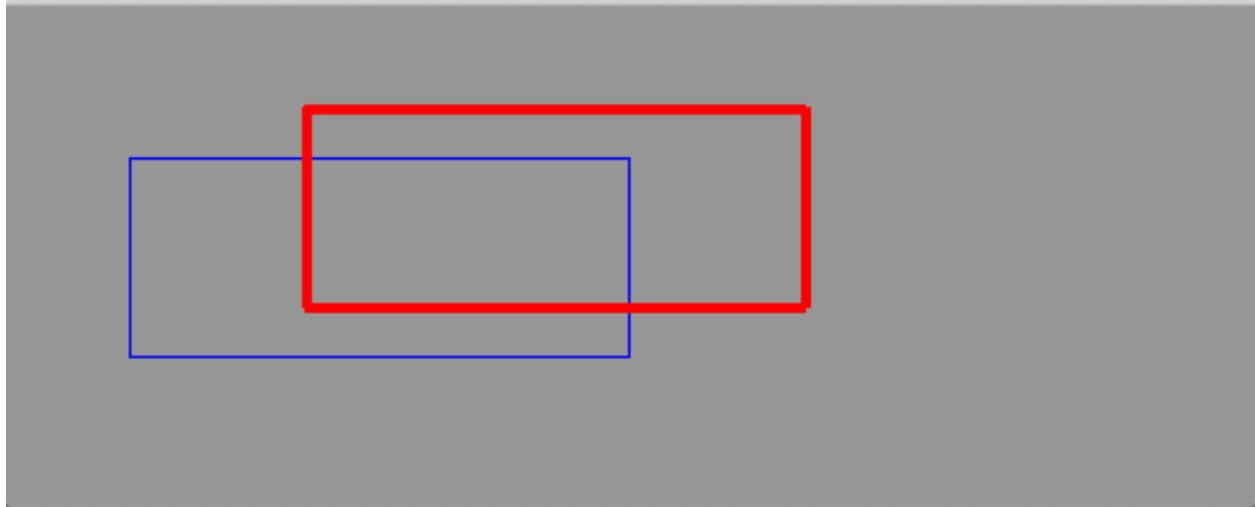
rect3.py

3.4 Move a rectangle with keys

The method `move(v)` creates a new `Rect` which has moved by a vector `v`. The method `move_ip(v)` moves a `Rect` **in place**. The following program uses the 4 arrow keys to move a rectangle around. The thin blue rectangle is the original one, the thick red rectangle is the moved one.

We use a dictionary to associate a motion vector to each of the 4 arrow keys. For each direction the movement is by 5 pixels:

```
dir = {K_LEFT: (-5, 0), K_RIGHT: (5, 0), K_UP: (0, -5), K_DOWN: (0, 5)}
```



```
from rect import *

rect0 = Rect(50, 60, 200, 80)
rect = rect0.copy()

while running:
    for event in pygame.event.get():
        if event.type == QUIT:
            running = False

        if event.type == KEYDOWN:
            if event.key in dir:
                v = dir[event.key]
                rect.move_ip(v)

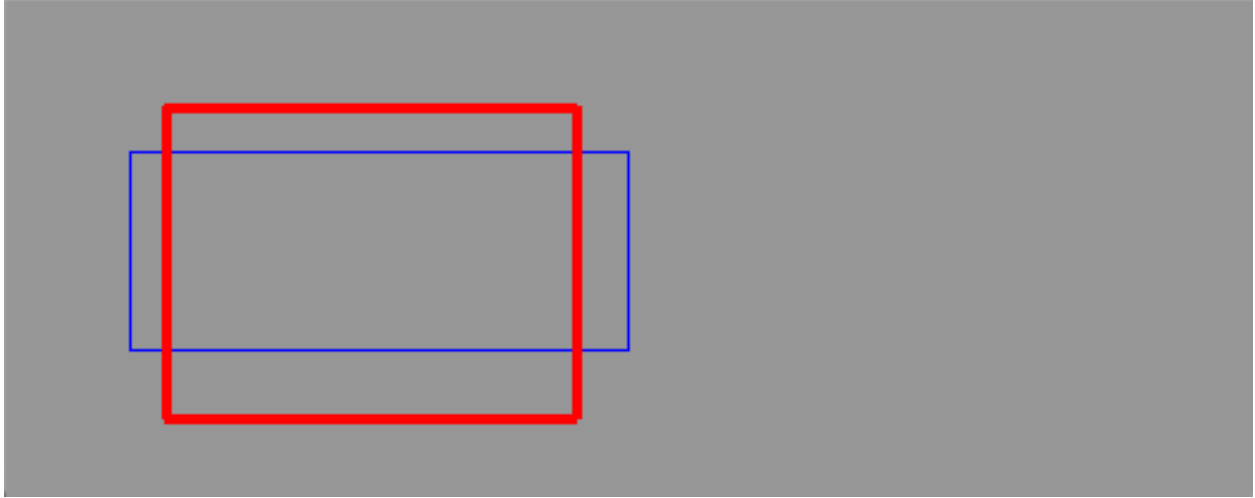
    screen.fill(GRAY)
    pygame.draw.rect(screen, BLUE, rect0, 1)
    pygame.draw.rect(screen, RED, rect, 4)
    pygame.display.flip()

pygame.quit()
```

rect4.py

3.5 Inflate a rectangle

The method `inflate(v)` grows or shrinks a rectangle by a vector `v` and creates a new `Rect`. The method `inflate_ip(v)` grows or shrinks a `Rect` **in place**. The following program uses the 4 arrow keys to change the size of a rectangle. The thin blue rectangle is the original one, the thick red rectangle is the changed one.



```
from rect import *

rect0 = rect.copy()

while running:
    for event in pygame.event.get():
        if event.type == QUIT:
            running = False

        if event.type == KEYDOWN:
            if event.key in dir:
                v = dir[event.key]
                rect.inflate_ip(v)

    screen.fill(GRAY)
    pygame.draw.rect(screen, BLUE, rect0, 1)
    pygame.draw.rect(screen, RED, rect, 4)
    pygame.display.flip()

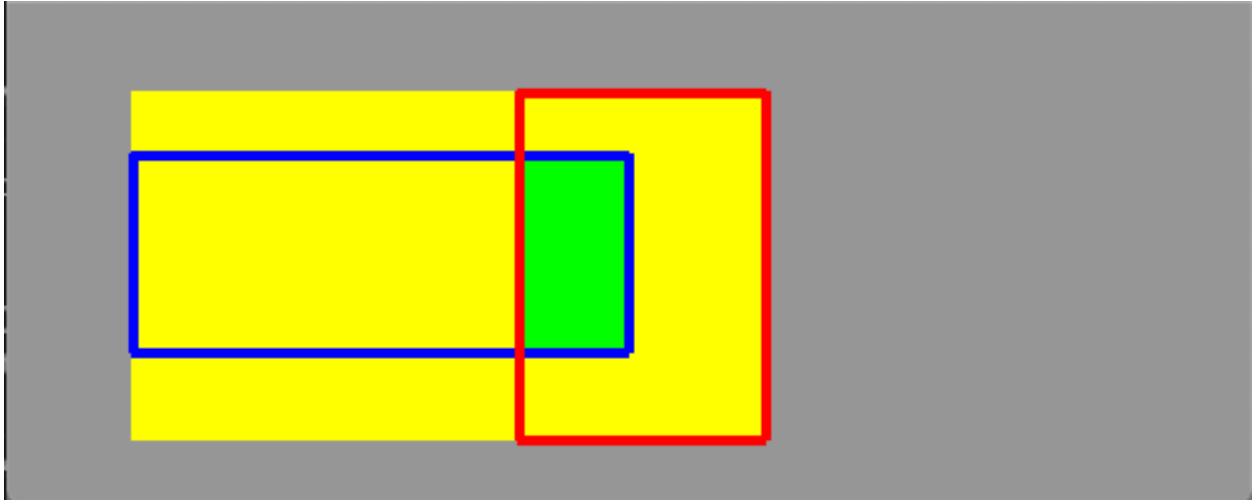
pygame.quit()
```

rect5.py

3.6 Clip a rectangle

The method `r0.clip(r1)` returns a new rectangle which is the intersection of the two rectangles. The method `r0.union(r1)` returns a new rectangle which is the union of the two rectangles.

The program belows shows two rectangles in red and blue outline. The green rectangle is the clipped area (intersection). The yellow rectangle is the union of the two rectangles.



```

from rect import *

r0 = Rect(50, 60, 200, 80)
r1 = Rect(100, 20, 100, 140)

while running:
    for event in pygame.event.get():
        if event.type == QUIT:
            running = False

        if event.type == KEYDOWN:
            if event.key in dir:
                r1.move_ip(dir[event.key])

    clip = r0.clip(r1)
    union = r0.union(r1)

    screen.fill(GRAY)
    pygame.draw.rect(screen, YELLOW, union, 0)
    pygame.draw.rect(screen, GREEN, clip, 0)
    pygame.draw.rect(screen, BLUE, r0, 4)
    pygame.draw.rect(screen, RED, r1, 4)
    pygame.display.flip()

pygame.quit()

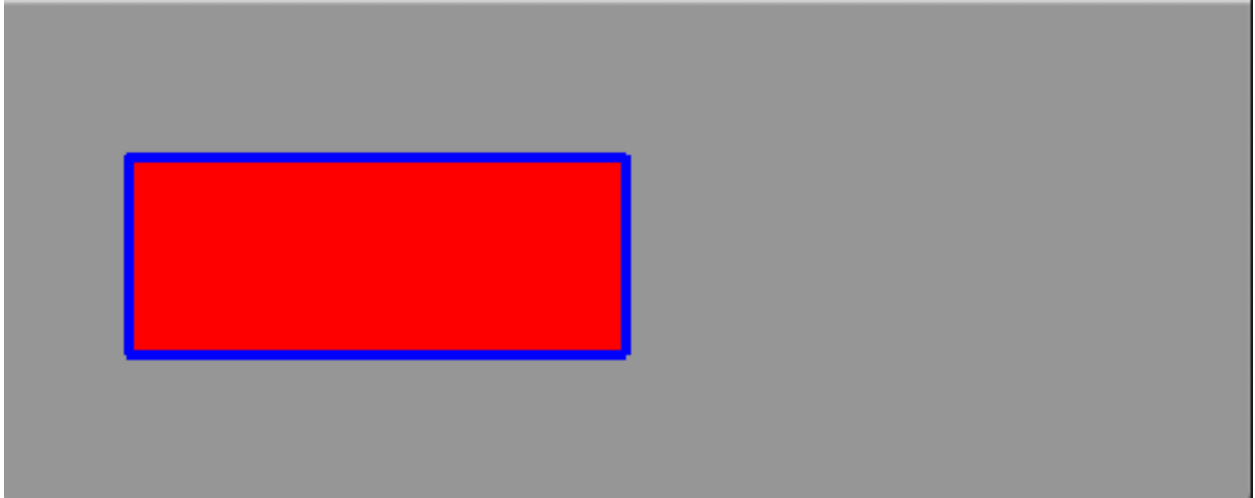
```

rect6.py

3.7 Move a rectangle with the mouse

The function `rect.collidepoint(pos)` returns True if the point collides with the rectangle. We use it with the mouse position to check if the mouse click has happened inside the rectangle. If that is the case, we move the rectangle by the relative motion of the mouse event `.rel`.

The boolean variable `moving` is set when the mouse button goes down inside the rectangle. It remains True until the button goes up again. The rectangle is only moved when the mouse click has happened inside the rectangle. While the rectangle is moving, we add a blue outline.



```
from rect import *

moving = False

while running:
    for event in pygame.event.get():
        if event.type == QUIT:
            running = False

        elif event.type == MOUSEBUTTONDOWN:
            if rect.collidepoint(event.pos):
                moving = True

        elif event.type == MOUSEBUTTONUP:
            moving = False

        elif event.type == MOUSEMOTION and moving:
            rect.move_ip(event.rel)

    screen.fill(GRAY)
    pygame.draw.rect(screen, RED, rect)
    if moving:
        pygame.draw.rect(screen, BLUE, rect, 4)
    pygame.display.flip()

pygame.quit()
```

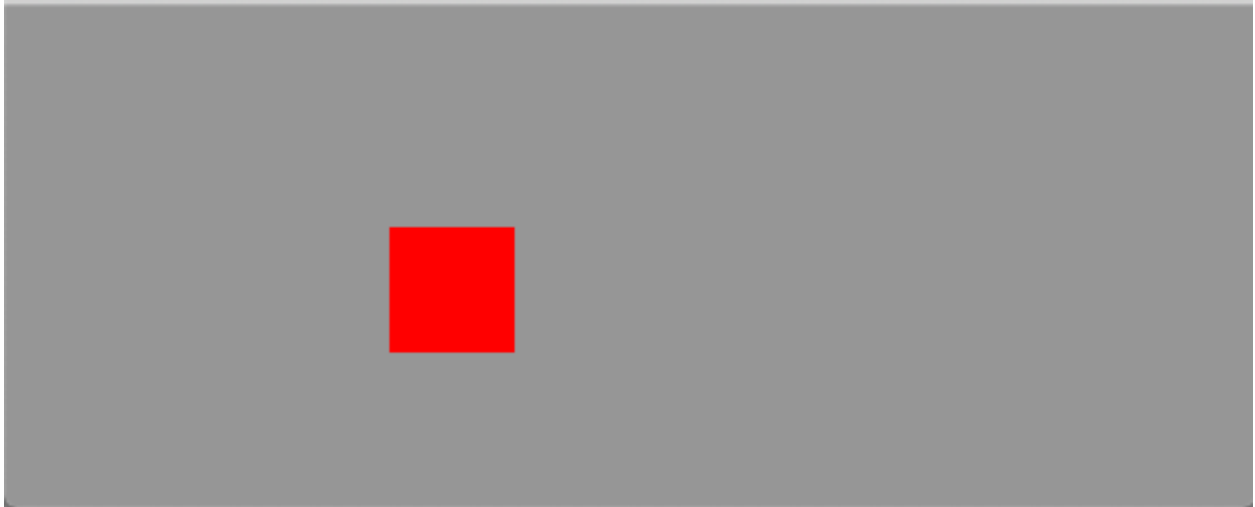
rect7.py

3.8 A self-moving a rectangle

The following code moves a rectangle by the amount `v`:

```
rect.move_ip(v)
```

It then checks the 4 borders and inverts the speed component if the rectangle is outside of the application window.



```
from rect import *

rect = Rect(100, 50, 50, 50)
v = [2, 2]

while running:
    for event in pygame.event.get():
        if event.type == QUIT:
            running = False

    rect.move_ip(v)

    if rect.left < 0:
        v[0] *= -1
    if rect.right > width:
        v[0] *= -1
    if rect.top < 0:
        v[1] *= -1
    if rect.bottom > height:
        v[1] *= -1

    screen.fill(GRAY)
    pygame.draw.rect(screen, RED, rect)
    pygame.display.flip()

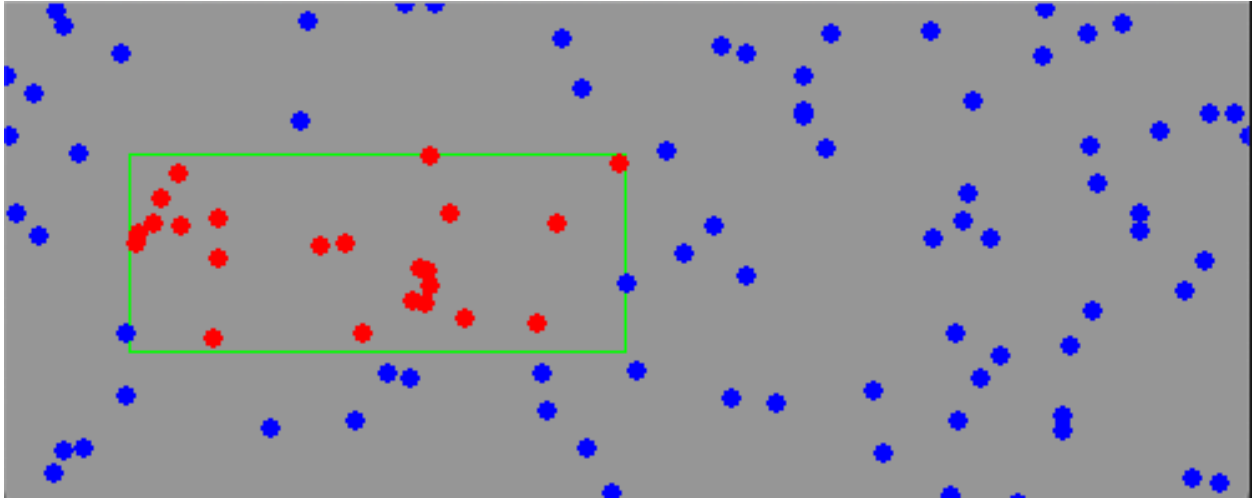
pygame.quit()
```

rect8.py

3.9 Colliding points

The method `rect.collidepoint(p)` checks if a rectangle `rect` collides with point `p`. In the following program we create 100 random points and color them red if they fall inside the rectangle.

Each time the R key is pressed 100 new random points are created.



```
from rect import *

points = random_points(100)

while running:
    for event in pygame.event.get():
        if event.type == QUIT:
            running = False

        if event.type == KEYDOWN:
            if event.key == K_r:
                points = random_points(100)

    screen.fill(GRAY)
    pygame.draw.rect(screen, GREEN, rect, 1)
    for p in points:
        if rect.collidepoint(p):
            pygame.draw.circle(screen, RED, p, 4, 0)
        else:
            pygame.draw.circle(screen, BLUE, p, 4, 0)

    pygame.display.flip()

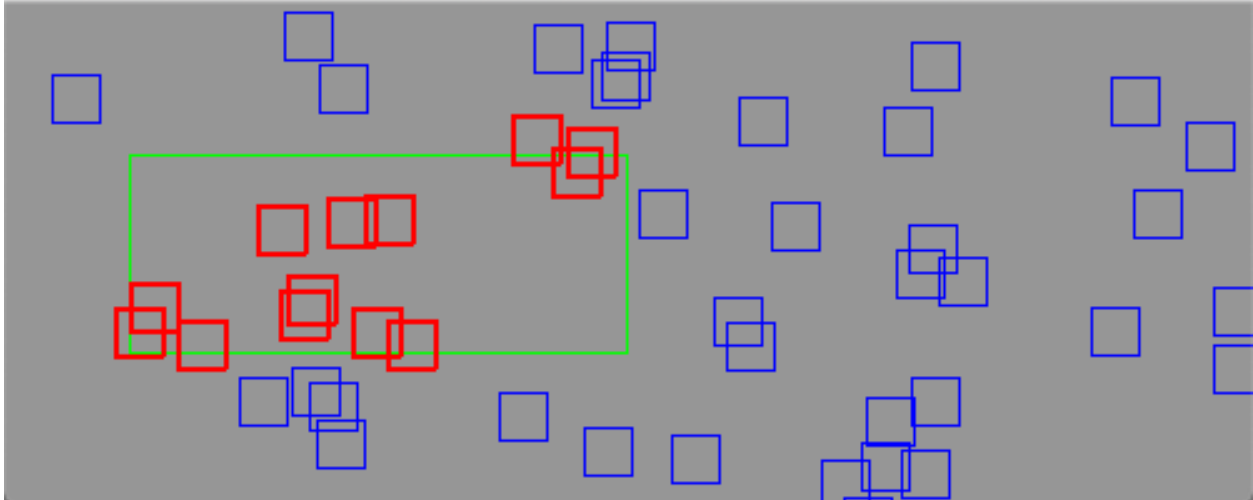
pygame.quit()
```

rect9.py

3.10 Colliding rectangles

The method `rect.colliderect(r)` checks if a rectangle `rect` collides with another rectangle `r`. In the following program we create 50 random rectangles and color them red if they collide with the green rectangle.

Each time the R key is pressed 50 new random rectangles are created.



```
from rect import *

n = 50
rects = random_rects(n)

while running:
    for event in pygame.event.get():
        if event.type == QUIT:
            running = False

        if event.type == KEYDOWN:
            if event.key == K_r:
                rects = random_rects(n)

    screen.fill(GRAY)
    pygame.draw.rect(screen, GREEN, rect, 1)

    for r in rects:
        if rect.colliderect(r):
            pygame.draw.rect(screen, RED, r, 2)
        else:
            pygame.draw.rect(screen, BLUE, r, 1)

    pygame.display.flip()

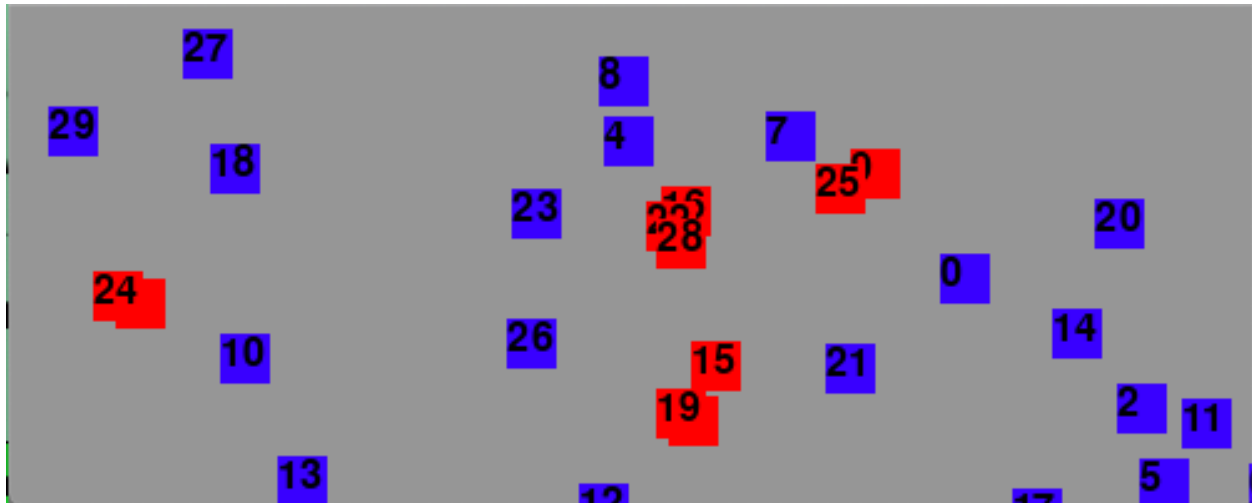
pygame.quit()
```

rect10.py

3.11 Overlapping rectangles

The method `rect.colliderect(r)` checks if a rectangle `rect` collides with another rectangle `r`. If we want to know if there are any two overlapping rectangles, then we have to compare each rectangle with each other one. The number of comparisons increases as power of 2.

Each time the R key is pressed 20 new random rectangles are created.



```

from rect import *

n = 30
rects = random_rects(n)

while running:
    for event in pygame.event.get():
        if event.type == QUIT:
            running = False

        if event.type == KEYDOWN:
            if event.key == K_r:
                rects = random_rects(n)

    screen.fill(GRAY)

    intersecting = []
    for i in range(n-1):
        r0 = rects[i]
        for j in range(i+1, n):
            r1 = rects[j]
            if r0.colliderect(r1):
                intersecting.append(r0)
                intersecting.append(r1)
                break

    for i, r in enumerate(rects):
        color = RED if r in intersecting else BLUE
        pygame.draw.rect(screen, color, r)
        draw_text(str(i), r.topleft)

    pygame.display.flip()

pygame.quit()

```

rect11.py

3.12 The common code

The common has been placed to a separate file:

```
import pygame
from pygame.locals import *
from random import randint

width = 500
height = 200

RED = (255, 0, 0)
GREEN = (0, 255, 0)
BLUE = (0, 0, 255)

YELLOW = (255, 255, 0)
MAGENTA = (255, 0, 255)
CYAN = (0, 255, 255)

BLACK = (0, 0, 0)
GRAY = (150, 150, 150)
WHITE = (255, 255, 255)

dir = {K_LEFT: (-5, 0), K_RIGHT: (5, 0), K_UP: (0, -5), K_DOWN: (0, 5)}
rect = Rect(50, 60, 200, 80)

def draw_text(text, pos):
    img = font.render(text, True, BLACK)
    screen.blit(img, pos)

def random_point():
    x = randint(0, width)
    y = randint(0, height)
    return (x, y)

def random_points(n):
    points = []
    for i in range(n):
        p = random_point()
        points.append(p)
    return points

def random_rects(n):
    rects = []
    for i in range(n):
        r = Rect(random_point(), (20, 20))
        rects.append(r)
    return rects

pygame.init()
screen = pygame.display.set_mode((width, height))
clock = pygame.time.Clock()
font = pygame.font.Font(None, 24)
running = True
```

rect.py

4.1 Load an image

The `pygame.image` module provides methods for loading and saving images. The method `load()` loads an image from the file system and returns a Surface object. The method `convert()` optimizes the image format and makes drawing faster:

```
img = pygame.image.load('bird.png')
img.convert()
```

Download the image `bird.png` to the same folder where your program resides:

`bird.png`

The method `get_rect()` returns a Rect object from an image. At this point only the size is set and position is placed at (0, 0). We set the center of the Rect to the center of the screen:

```
rect = img.get_rect()
rect.center = w//2, h//2
```

To recapitulate, we are working with 3 objects:

- **screen** is the Surface object representing the application window
- **img** is the Surface object of the image to display
- **rect** is the Rect object which is the bounding rectangle of the image

To display the image we fill the screen with a background color (GRAY). Then we blit the image, draw a red rectangle around it and finally update the screen:

```
screen.fill(GRAY)
screen.blit(img, rect)
pygame.draw.rect(screen, RED, rect, 1)
pygame.display.update()
```

4.2 Move the image with the mouse

At the beginning of the program we set a boolean variable `moving` to `False`. Only when the mouse button is pressed, and when the mouse position is within the image (`collidepoint`) we set it to `True`:

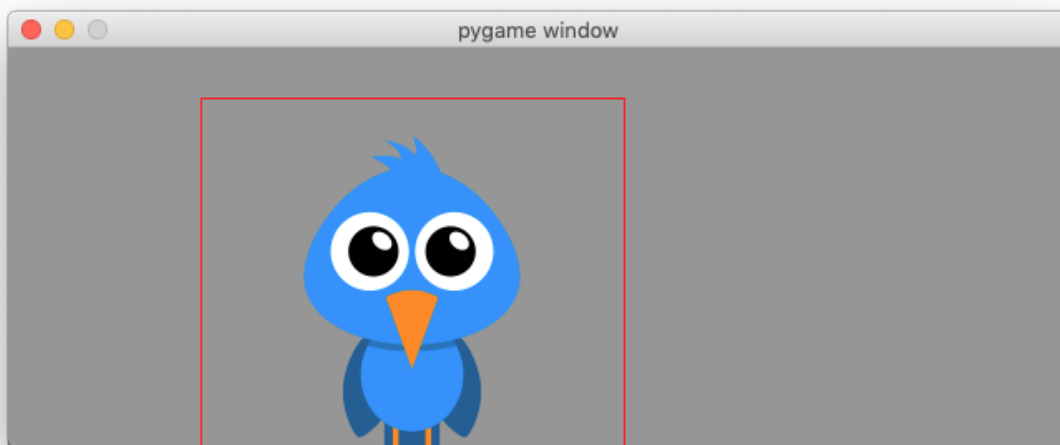
```
elif event.type == MOUSEBUTTONDOWN:
    if rect.collidepoint(event.pos):
        moving = True
```

When the mouse button is released, we set it to `False` again:

```
elif event.type == MOUSEBUTTONUP:
    moving = False
```

When the mouse moves, and the flag `moving` is `True`, then we move the image by the amount of relative movement (`event.rel`):

```
elif event.type == MOUSEMOTION and moving:
    rect.move_ip(event.rel)
```



This is the whole code:

```
"""Move an image with the mouse."""

import pygame
from pygame.locals import *

RED = (255, 0, 0)
GRAY = (150, 150, 150)

pygame.init()
w, h = 640, 240
screen = pygame.display.set_mode((w, h))
running = True
```

(continues on next page)

(continued from previous page)

```

img = pygame.image.load('bird.png')
img.convert()
rect = img.get_rect()
rect.center = w//2, h//2
moving = False

while running:
    for event in pygame.event.get():
        if event.type == QUIT:
            running = False

        elif event.type == MOUSEBUTTONDOWN:
            if rect.collidepoint(event.pos):
                moving = True

        elif event.type == MOUSEBUTTONUP:
            moving = False

        elif event.type == MOUSEMOTION and moving:
            rect.move_ip(event.rel)

    screen.fill(GRAY)
    screen.blit(img, rect)
    pygame.draw.rect(screen, RED, rect, 1)
    pygame.display.update()

pygame.quit()

```

image1.py

4.3 Rotate and Scale the image

The `pygame.transform` module provides methods for **scaling, rotating and flipping** images. As we are going to modify the image `img` we keep the original image in a variable called `img0`:

```

img0 = pygame.image.load(path)
img0.convert()

```

In order to show the image rectangle, we add a green border to the original image:

```

rect0 = img0.get_rect()
pygame.draw.rect(img0, GREEN, rect0, 1)

```

Then we place the image in the center of the screen:

```

center = w//2, h//2
img = img0
rect = img.get_rect()
rect.center = center

```

First we define the global variables `scale` and `angle`:

```

angle = 0
scale = 1

```

We use the R key to increment rotation by 10 degrees and (decrement if the SHIFT key is pressed). The function `rotozoom()` allows to combine rotation and scaling. We always transform the original image (`img0`). Repeated rotation or scaling of an image would degrade its quality:

```
if event.type == KEYDOWN:
    if event.key == K_r:
        if event.mod & KMOD_SHIFT:
            angle -= 10
        else:
            angle += 10
        img = pygame.transform.rotozoom(img0, angle, scale)
```

We use the S key to increment the scale by 10% (decrease if the SHIFT key is pressed):

```
elif event.key == K_s:
    if event.mod & KMOD_SHIFT:
        scale /= 1.1
    else:
        scale *= 1.1
    img = pygame.transform.rotozoom(img0, angle, scale)
```

As the image is transformed the bounding rectangle changes size. It must be recalculated and placed at the center again:

```
rect = img.get_rect()
rect.center = center
```

4.4 Reset the image to the original

We use the O key to reset the image to its original state:

```
elif event.key == K_o:
    img = img0
    angle = 0
    scale = 1
```

4.5 Flip the image

We use the H key to flip the image horizontally:

```
elif event.key == K_h:
    img = pygame.transform.flip(img, True, False)
```

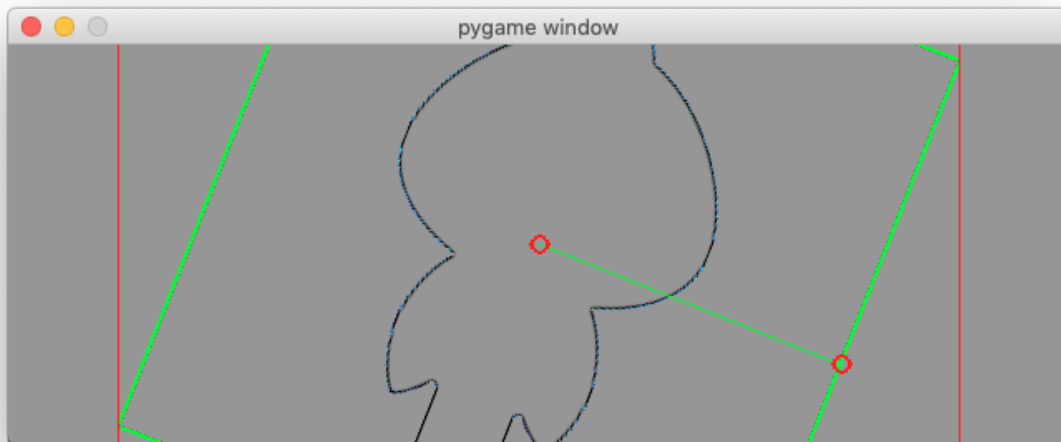
and the V key to flip the image vertically:

```
elif event.key == K_v:
    img = pygame.transform.flip(img, False, True)
```

4.6 Detect edges with the Laplacian

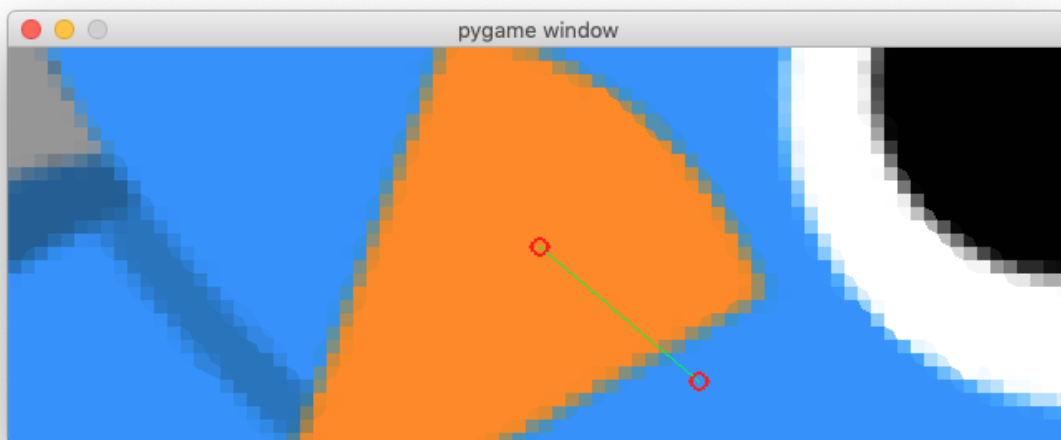
The fonction `laplacien(img)` allows to detect the outline of the image:

```
elif event.key == K_l:  
    img = pygame.transform.laplacian(img)
```



The function `scale2x(img)` doubles the size of a pixel:

```
elif event.key == K_2:  
    img = pygame.transform.scale2x(img)
```



4.7 Transform the image with the mouse

In this section we show how to use the mouse to scale and rotate an image. At the beginning we import the `math` module:

```
import math
```

At the beginning we store the initial mouse position:

```
mouse = pygame.mouse.get_pos()
```

When the mouse moves we update the mouse position `mouse` and calculate the `x`, `y` coordinates from the center of the image. We also calculate the center-mouse distance `d`

```
elif event.type == MOUSEMOTION:
    mouse = event.pos
    x = mouse[0] - center[0]
    y = mouse[1] - center[1]
    d = math.sqrt(x ** 2 + y ** 2)
```

The `atan2(y, x)` math function allows to find the rotation angle. We need to transform radians in degrees. From the distance mouse-center we calculate the scale argument:

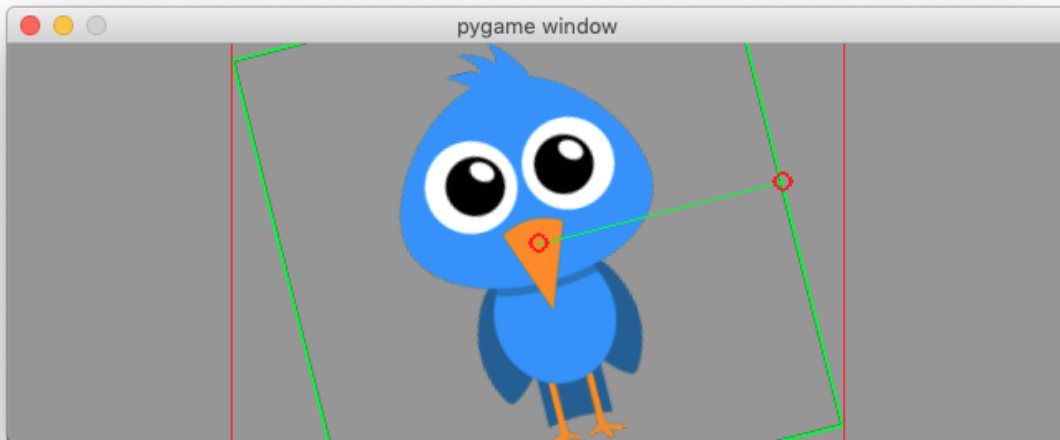
```
angle = math.degrees(-math.atan2(y, x))
scale = abs(5 * d / w)
img = pygame.transform.rotozoom(img0, angle, scale)
rect = img.get_rect()
rect.center = center
```

To finally draw the transformed image we first fill the whole screen background (GRAY), blit the transformed image, surround it with a red rectangle.

In order to give visual feedback for the mouse action when transforming an image, we

- draw a green line between the center of the image and the mouse position,
- place two circles on the center and on the mouse position:

```
screen.fill(GRAY)
screen.blit(img, rect)
pygame.draw.rect(screen, RED, rect, 1)
pygame.draw.line(screen, GREEN, center, mouse, 1)
pygame.draw.circle(screen, RED, center, 6, 1)
pygame.draw.circle(screen, RED, mouse, 6, 1)
pygame.display.update()
```

Here is the full code.

```
"""Rotate, scale and flip an image."""

import pygame
import math, sys, os
from pygame.locals import *

RED = (255, 0, 0)
GREEN = (0, 255, 0)
GRAY = (150, 150, 150)

pygame.init()
w, h = 640, 240
screen = pygame.display.set_mode((w, h))
running = True

module = sys.modules['__main__']
path, name = os.path.split(module.__file__)
path = os.path.join(path, 'bird.png')

img0 = pygame.image.load(path)
img0.convert()

# draw a green border around img0
rect0 = img0.get_rect()
pygame.draw.rect(img0, GREEN, rect0, 1)

center = w//2, h//2
img = img0
rect = img.get_rect()
rect.center = center

angle = 0
scale = 1
```

(continues on next page)

(continued from previous page)

```

mouse = pygame.mouse.get_pos()

while running:
    for event in pygame.event.get():
        if event.type == QUIT:
            running = False

        if event.type == KEYDOWN:
            if event.key == K_r:
                if event.mod & KMOD_SHIFT:
                    angle -= 10
                else:
                    angle += 10
            img = pygame.transform.rotozoom(img0, angle, scale)

            elif event.key == K_s:
                if event.mod & KMOD_SHIFT:
                    scale /= 1.1
                else:
                    scale *= 1.1
            img = pygame.transform.rotozoom(img0, angle, scale)

            elif event.key == K_o:
                img = img0
                angle = 0
                scale = 1

            elif event.key == K_h:
                img = pygame.transform.flip(img, True, False)

            elif event.key == K_v:
                img = pygame.transform.flip(img, False, True)

            elif event.key == K_l:
                img = pygame.transform.laplacian(img)

            elif event.key == K_2:
                img = pygame.transform.scale2x(img)

        rect = img.get_rect()
        rect.center = center

    elif event.type == MOUSEMOTION:
        mouse = event.pos
        x = mouse[0] - center[0]
        y = mouse[1] - center[1]
        d = math.sqrt(x ** 2 + y ** 2)

        angle = math.degrees(-math.atan2(y, x))
        scale = abs(5 * d / w)
        img = pygame.transform.rotozoom(img0, angle, scale)
        rect = img.get_rect()
        rect.center = center

    screen.fill(GRAY)
    screen.blit(img, rect)
    pygame.draw.rect(screen, RED, rect, 1)

```

(continues on next page)

(continued from previous page)

```
pygame.draw.line(screen, GREEN, center, mouse, 1)
pygame.draw.circle(screen, RED, center, 6, 1)
pygame.draw.circle(screen, RED, mouse, 6, 1)
pygame.display.update()

pygame.quit()
```

image2.py

CHAPTER 5

Work with text

In pygame, text cannot be written directly to the screen. The first step is to create a `Font` object with a given font size. The second step is to render the text into an image with a given color. The third step is to blit the image to the screen. These are the steps:

```
font = pygame.font.SysFont(None, 24)
img = font.render('hello', True, BLUE)
screen.blit(img, (20, 20))
```

Once the font is created, its size cannot be changed. A `Font` object is used to create a `Surface` object from a string. Pygame does not provide a direct way to write text onto a `Surface` object. The method `render()` must be used to create a `Surface` object from the text, which then can be blit to the screen. The method `render()` can only render single lines. A newline character is not rendered.

5.1 Initialize a font

Initializing the font can take a few seconds. On a MacBook Air the the creation of a system `Font` object:

```
t0 = time.time()
font = pygame.font.SysFont(None, 48)
print('time needed for Font creation :', time.time()-t0)
```

took more then 8 seconds:

```
time needed for Font creation : 8.230187892913818
```

The function `get_fonts()` returns a list of all installed fonts. The following code checks what fonts are on your system and how many, and prints them to the console:

```
fonts = pygame.font.get_fonts()
print(len(fonts))
for f in fonts:
    print(f)
```

You will get something like this:

```
344
bigcaslon.ttf
silom.ttf
sfnsdisplayblackitalic.otf
chalkduster.ttf
...
```

5.2 Render the text

The font object can render a given text into an image. In the example below, we place a blue bounding rectangle around that text image:

```
img = font.render(sysfont, True, RED)
rect = img.get_rect()
pygame.draw.rect(img, BLUE, rect, 1)
```

We then create two more fonts, *Chalkduster* and *Didot* at a size of 72 points. We render a text with both fonts:

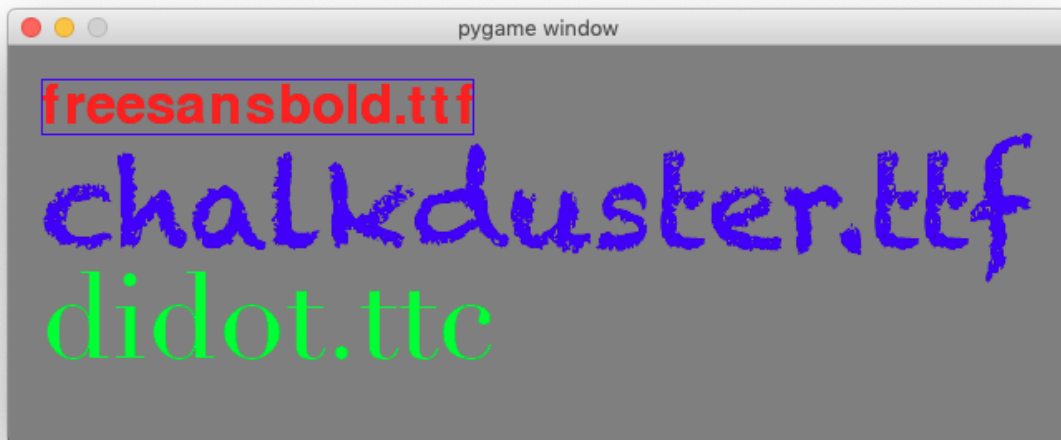
```
font1 = pygame.font.SysFont('chalkduster.ttf', 72)
img1 = font1.render('chalkduster.ttf', True, BLUE)

font2 = pygame.font.SysFont('didot.ttc', 72)
img2 = font2.render('didot.ttc', True, GREEN)
```

Finally the text images are blit to the screen like regular images:

```
screen.fill(background)
screen.blit(img, (20, 20))
screen.blit(img1, (20, 50))
screen.blit(img2, (20, 120))
pygame.display.update()
```

This is the result:



Here is the full code.

```
"""Draw text to the screen."""
import pygame
from pygame.locals import *
import time

BLACK = (0, 0, 0)
RED = (255, 0, 0)
GREEN = (0, 255, 0)
BLUE = (0, 0, 255)
GRAY = (200, 200, 200)

pygame.init()
screen = pygame.display.set_mode((640, 240))

sysfont = pygame.font.get_default_font()
print('system font:', sysfont)

t0 = time.time()
font = pygame.font.SysFont(None, 48)
print('time needed for Font creation:', time.time()-t0)

img = font.render(sysfont, True, RED)
rect = img.get_rect()
pygame.draw.rect(img, BLUE, rect, 1)

font1 = pygame.font.SysFont('chalkduster.ttf', 72)
img1 = font1.render('chalkduster.ttf', True, BLUE)

font2 = pygame.font.SysFont('didot.ttc', 72)
img2 = font2.render('didot.ttc', True, GREEN)

fonts = pygame.font.get_fonts()
print(len(fonts))
for i in range(7):
```

(continues on next page)

(continued from previous page)

```
print(fonts[i])

running = True
background = GRAY
while running:
    for event in pygame.event.get():
        if event.type == QUIT:
            running = False

    screen.fill(background)
    screen.blit(img, (20, 20))
    screen.blit(img1, (20, 50))
    screen.blit(img2, (20, 120))
    pygame.display.update()

pygame.quit()
```

5.3 Edit text with the keyboard

The keyboard event can be used to edit a text. First we create a text which we save in a string variable `text` and which we render to an image:

```
text = 'this text is editable'
font = pygame.font.SysFont(None, 48)
img = font.render(text, True, RED)
```

Then we define the bounding rectangle and furthermore a cursor rectangle which is juxtaposed to the text bounding rectangle:

```
rect = img.get_rect()
rect.topleft = (20, 20)
cursor = Rect(rect.topright, (3, rect.height))
```

Inside the event loop we watch out for `KEYDOWN` events. If the key press is a `BACKSPACE` and the length of the string is larger than 0, then we remove the last character, else we append the new character to the text variable:

```
if event.type == KEYDOWN:
    if event.key == K_BACKSPACE:
        if len(text)>0:
            text = text[:-1]
    else:
        text += event.unicode
```

Then we render the modified text, update the bounding rectangle, and place the cursor box at the end of the updated bounding rectangle:

```
img = font.render(text, True, RED)
rect.size=img.get_size()
cursor.topleft = rect.topright
```


5.4 Add a blinking cursor

In order to make the cursor more visible, we let it blink every 0.5 seconds. We do this using the `time.time()` floating point value:

```
screen.fill(background)
screen.blit(img, rect)
if time.time() % 1 > 0.5:
    pygame.draw.rect(screen, RED, cursor)
pygame.display.update()
```

This is the result:



Here is the full code.

```
"""Edit text with the keyboard."""
import pygame
from pygame.locals import *
import time

BLACK = (0, 0, 0)
RED = (255, 0, 0)
GRAY = (200, 200, 200)

pygame.init()
screen = pygame.display.set_mode((640, 240))

text = 'this text is editable'
font = pygame.font.SysFont(None, 48)
img = font.render(text, True, RED)

rect = img.get_rect()
rect.topleft = (20, 20)
cursor = Rect(rect.topright, (3, rect.height))

running = True
background = GRAY
```

(continues on next page)

(continued from previous page)

```
while running:
    for event in pygame.event.get():
        if event.type == QUIT:
            running = False

        if event.type == KEYDOWN:
            if event.key == K_BACKSPACE:
                if len(text)>0:
                    text = text[:-1]
            else:
                text += event.unicode
            img = font.render(text, True, RED)
            rect.size=img.get_size()
            cursor.topleft = rect.topright

    screen.fill(background)
    screen.blit(img, rect)
    if time.time() % 1 > 0.5:
        pygame.draw.rect(screen, RED, cursor)
    pygame.display.update()

pygame.quit()
```

Making apps with Pygame

In this section we are going to create applications and games with Pygame. From here on we will be using an object-oriented programming (OOP) approach.

Pygame only allows to create one single window. Different from other applications, those based on Pygame cannot have multiple windows. If for example dialog window is needed, it must be displayed within the main window.

Within an application we provide multiples scenes (environments, rooms, or levels). Each scene contains different objects such as:

- text
- sprites (images)
- GUI elements (buttons, menus)
- shapes (rectangles, circles)

6.1 Create the App class

The basis for a game or application is the App class. The first thing to do is to import the `pygame` module, as well as a series of useful constants:

```
import pygame
from pygame.locals import *
```

Then we create define the App class which initializes Pygame and opens a the app window:

```
class App:
    """Create a single-window app with multiple scenes."""

    def __init__(self):
        """Initialize pygame and the application."""
        pygame.init()
        flags = RESIZABLE
```

(continues on next page)

(continued from previous page)

```
App.screen = pygame.display.set_mode((640, 240), flags)

App.running = True
```

Further we have to define the main event loop:

```
def run(self):
    """Run the main event loop."""
    while App.running:
        for event in pygame.event.get():
            if event.type == QUIT:
                App.running = False
    pygame.quit()
```

At the end of the module we run a demo, if the program is run directly and not imported as a module:

```
if __name__ == '__main__':
    App().run()
```

6.2 Add the Text class

Now we add some text to the screen. We create a Text class from which we can instantiate text objects:

```
class Text:
    """Create a text object."""

    def __init__(self, text, pos, **options):
        self.text = text
        self.pos = pos

        self.fontname = None
        self.fontsize = 72
        self.fontcolor = Color('black')
        self.set_font()
        self.render()
```

The Font object needs to be created initially and everytime the font name or the font size changes:

```
def set_font(self):
    """Set the font from its name and size."""
    self.font = pygame.font.Font(self.fontname, self.fontsize)
```

The text needs to be rendered into a surface object, an image. This needs to be done only once, or whenever the text changes:

```
def render(self):
    """Render the text into an image."""
    self.img = self.font.render(self.text, True, self.fontcolor)
    self.rect = self.img.get_rect()
    self.rect.topleft = self.pos
```

Drawing the text means blitting it to the application screen:

```
def draw(self):
    """Draw the text image to the screen."""
    App.screen.blit(self.img, self.rect)
```

This is the result:



Here is the complete code:

```
import pygame
from pygame.locals import *

class Text:
    """Create a text object."""

    def __init__(self, text, pos, **options):
        self.text = text
        self.pos = pos

        self.fontname = None
        self.fontsize = 72
        self.fontcolor = Color('black')
        self.set_font()
        self.render()

    def set_font(self):
        """Set the Font object from name and size."""
        self.font = pygame.font.Font(self.fontname, self.fontsize)

    def render(self):
        """Render the text into an image."""
        self.img = self.font.render(self.text, True, self.fontcolor)
        self.rect = self.img.get_rect()
        self.rect.topleft = self.pos

    def draw(self):
        """Draw the text image to the screen."""
```

(continues on next page)

(continued from previous page)

```

        App.screen.blit(self.img, self.rect)

class App:
    """Create a single-window app with multiple scenes."""

    def __init__(self):
        """Initialize pygame and the application."""
        pygame.init()
        flags = RESIZABLE
        App.screen = pygame.display.set_mode((640, 240), flags)
        App.t = Text('Pygame App', pos=(20, 20))

        App.running = True

    def run(self):
        """Run the main event loop."""
        while App.running:
            for event in pygame.event.get():
                if event.type == QUIT:
                    App.running = False

            App.screen.fill(Color('gray'))
            App.t.draw()
            pygame.display.update()

        pygame.quit()

if __name__ == '__main__':
    App().run()

```

6.3 Shortcut keys

Key presses (called shortcuts) can be used to interact with the application and run commands. We can add the following code inside the event loop to intercept the S key and print a message:

```

if event.type == KEYDOWN:
    if event.key == K_s:
        print('Key press S')

```

If the application has many shortcuts, the keys alone may not be enough and modifier keys (cmd, ctrl, alt, shift) can be used to increase the number of combinations. The easiest way to represent these shortcuts is under the form of a dictionary, where the key/mod tuples are associated with a command strings. The dictionary has this shape:

```

self.shortcuts = {
    (K_x, KMOD_LMETA): 'print("cmd+X") ',
    (K_x, KMOD_LALT): 'print("alt+X") ',
    (K_x, KMOD_LCTRL): 'print("ctrl+X") ',
    (K_x, KMOD_LMETA + KMOD_LSHIFT): 'print("cmd+shift+X") ',
    (K_x, KMOD_LMETA + KMOD_LALT): 'print("cmd+alt+X") ',
    (K_x, KMOD_LMETA + KMOD_LALT + KMOD_LSHIFT): 'print("cmd+alt+shift+X") ',
}

```

Inside the event loop we detect keydown events and call the key handler:

```
if event.type == KEYDOWN:
    self.do_shortcut(event)
```

The `do_shortcut()` method looks up the shortcut and executes the command string:

```
def do_shortcut(self, event):
    """Find the the key/mod combination in the dictionary and execute the cmd."""
    k = event.key
    m = event.mod
    if (k, m) in self.shortcuts:
        exec(self.shortcuts[k, m])
```

This is the result on the console when pressing different key+modifier combinations:

```
cmd+X
alt+X
ctrl+X
cmd+shift+X
cmd+alt+X
cmd+alt+shift+X
```

6.4 Fullscreen, resizable and noframe mode

Pygame allows a window to be displayed in 3 different modes:

- fullscreen mode
- resizable (a resize edge is displayed)
- noframe mode (without a window title bar)

Inside the App class `__init__()` method we first define the screen size and the display mode flags, and then create the screen surface:

```
self.flags = RESIZABLE
self.rect = Rect(0, 0, 640, 240)
App.screen = pygame.display.set_mode(self.rect.size, self.flags)
```

In order to toggle (turn on and off) the three display modes we add these entries to the `shortcuts` dictionary:

```
(K_f, KMOD_LMETA): 'self.toggle_fullscreen()',
(K_r, KMOD_LMETA): 'self.toggle_resizable()',
(K_g, KMOD_LMETA): 'self.toggle_frame()',
```

Inside the App class we define three methods to toggle the corresponding mode flag, by using the bit-wise XOR operator (`^=`):

```
def toggle_fullscreen(self):
    """Toggle between full screen and windowed screen."""
    self.flags ^= FULLSCREEN
    pygame.display.set_mode((0, 0), self.flags)

def toggle_resizable(self):
    """Toggle between resizable and fixed-size window."""
    self.flags ^= RESIZABLE
    pygame.display.set_mode(self.rect.size, self.flags)
```

(continues on next page)

(continued from previous page)

```
def toggle_frame(self):  
    """Toggle between frame and noframe window."""  
    self.flags ^= NOFRAME  
    pygame.display.set_mode(self.rect.size, self.flags)
```

6.5 Add the Scene class

Most applications or games have different scenes, such as an introduction screen, an intro, and different game levels. So we are going to define the Scene class:

```
class Scene:  
    """Create a new scene (room, level, view)."""  
    id = 0  
    bg = Color('gray')
```

When creating a new scene, we append the scene to the applications scene list and make this scene the current scene:

```
def __init__(self, *args, **kwargs):  
    # Append the new scene and make it the current scene  
    App.scenes.append(self)  
    App.scene = self
```

Then we set a scene id, which is kept as class attribute of the Scene class. Then we set the nodes list to the empty list and set the background color:

```
# Set the instance id and increment the class id  
self.id = Scene.id  
Scene.id += 1  
self.nodes = []  
self.bg = Scene.bg
```

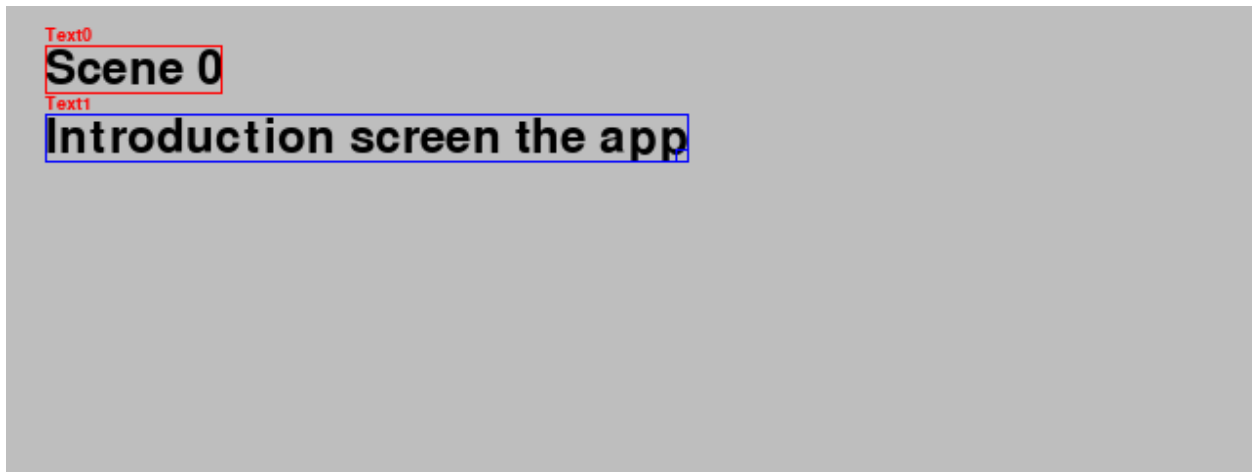
The scene object knows how to draw itself. It first fills the background with the background color, then draws each nodes and finally flips the display to update the screen:

```
def draw(self):  
    """Draw all objects in the scene."""  
    App.screen.fill(self.bg)  
    for node in self.nodes:  
        node.draw()  
    pygame.display.flip()
```

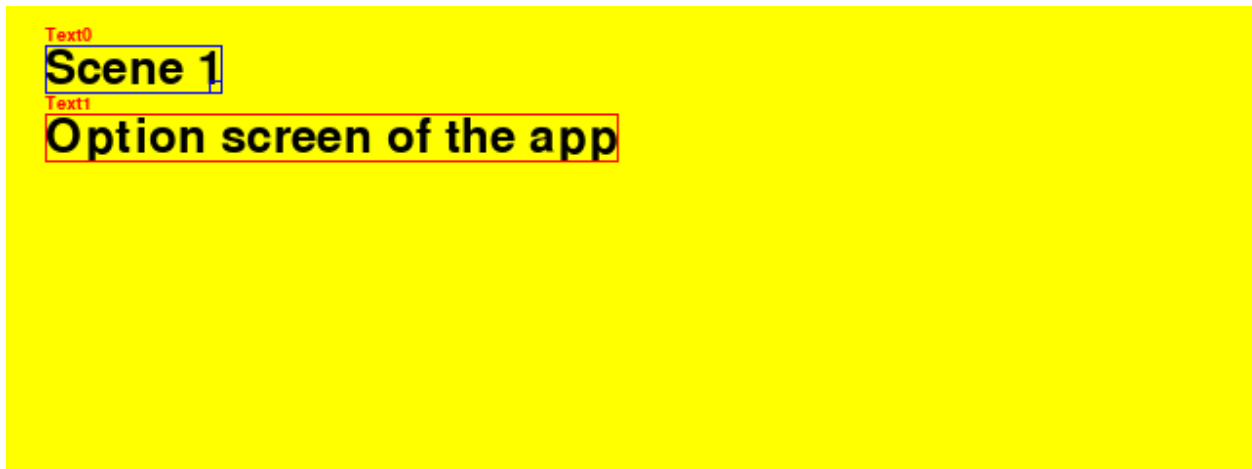
The string representation of the scene is *Scene* followed by its ID number:

```
def __str__(self):  
    return 'Scene {}'.format(self.id)
```

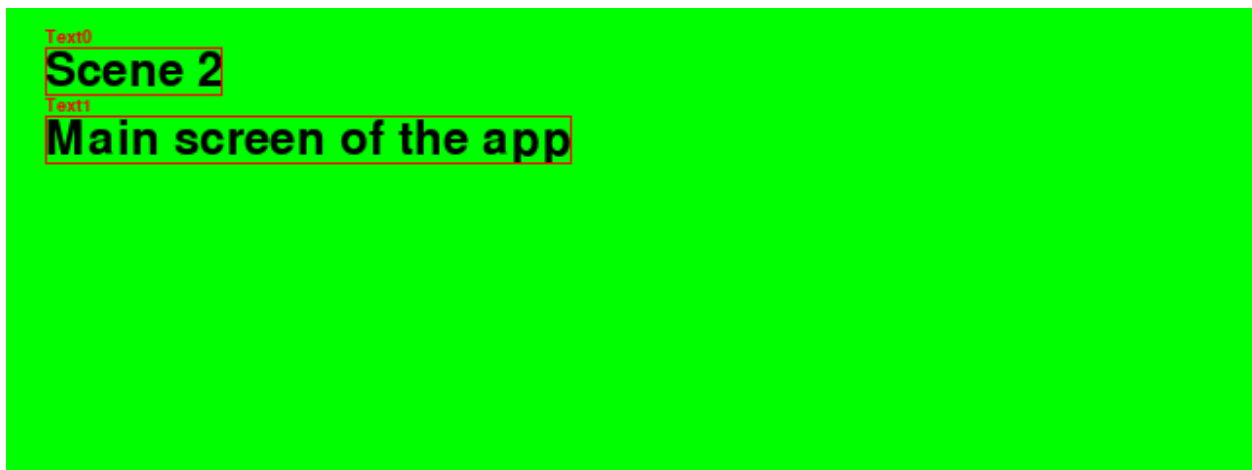
This is an image of scene 0 with two text objects and a default gray background color. The second text object has been selected.



This is an image of scene 1 with two text objects, the first one being selected and a yellow background color.



This is an image of scene 2 with two text objects, none being selected, and a green background color.



Here is the complete code:

```
from app import *
```

(continues on next page)

(continued from previous page)

```

class Demo(App):
    def __init__(self):
        super().__init__()

        Scene(caption='Intro')
        Text('Scene 0')
        Text('Introduction screen the app')

        Scene(bg=Color('yellow'), caption='Options')
        Text('Scene 1')
        Text('Option screen of the app')

        Scene(bg=Color('green'), caption='Main')
        Text('Scene 2')
        Text('Main screen of the app')

        App.scene = App.scenes[0]

if __name__ == '__main__':
    Demo().run()

```

6.6 Scenes with background images

We can add a background image to a scene:

```

self.file = Scene.options['file']

if self.file != '':
    self.img = pygame.image.load(self.file)
    size = App.screen.get_size()
    self.img = pygame.transform.smoothscale(self.img, size)
self.enter()

```

This is an image of scene 0 with a forest background image and a white Text object.



This is an image of scene 1 with a lake background image and a black Text object.



This is an image of scene 2 with a sunset background image and a white Text object.



Here is the complete code:

```
"""Display different scene background images."""
from app import *

class Demo(App):
    def __init__(self):
        super().__init__()
        Scene(img_folder='../background', file='forest.jpg', caption='Forest')
        Text('Forest scene', fontcolor=Color('white'))
        Scene(file='lake.jpg', caption='Lake')
        Text('Lake scene')
        Scene(file='sunset.jpg', caption='Sunset')
        Text('Sunset scene', fontcolor=Color('white'))
        Scene(file='', bg=Color('lightgreen'), caption='Green background')
        Text('Colored background scene')

if __name__ == '__main__':
    Demo().run()
```

6.7 Automatic node placement

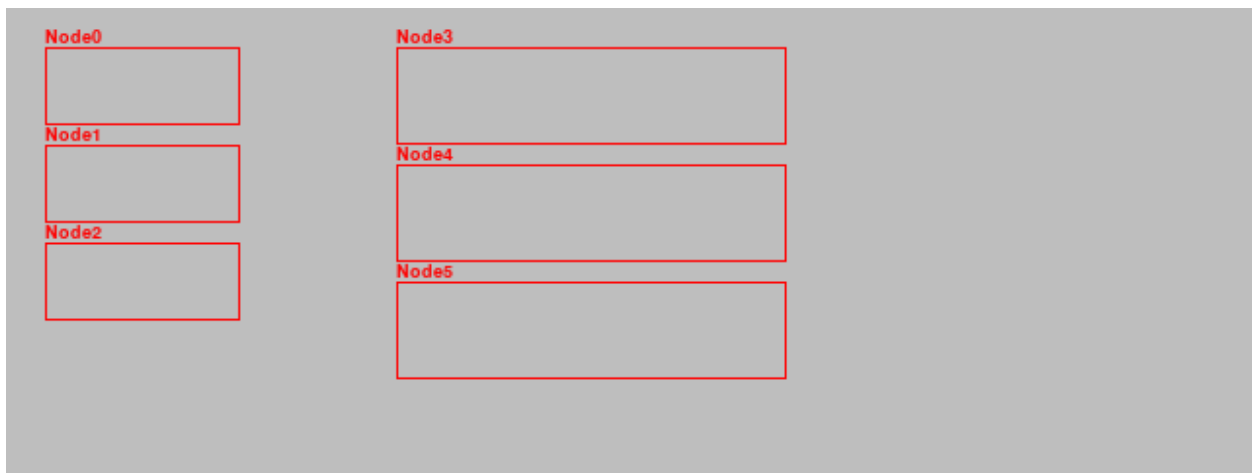
Nodes are containers for GUI elements. It is convenient if they can be placed automatically inside a scene.

- `pos` the current position
- `size` the current size
- `dir` the current direction: vertical (1, 0), horizontal (0, 1), diagonal (1, 1)
- `gap` the spacing

The default placement direction is vertical. Nodes placed in a scene stack up vertically. At any time the node position, node size, node gap or node direction can be changed:

```
Scene(caption='Nodes - vertical placement')
Node()
Node()
Node()

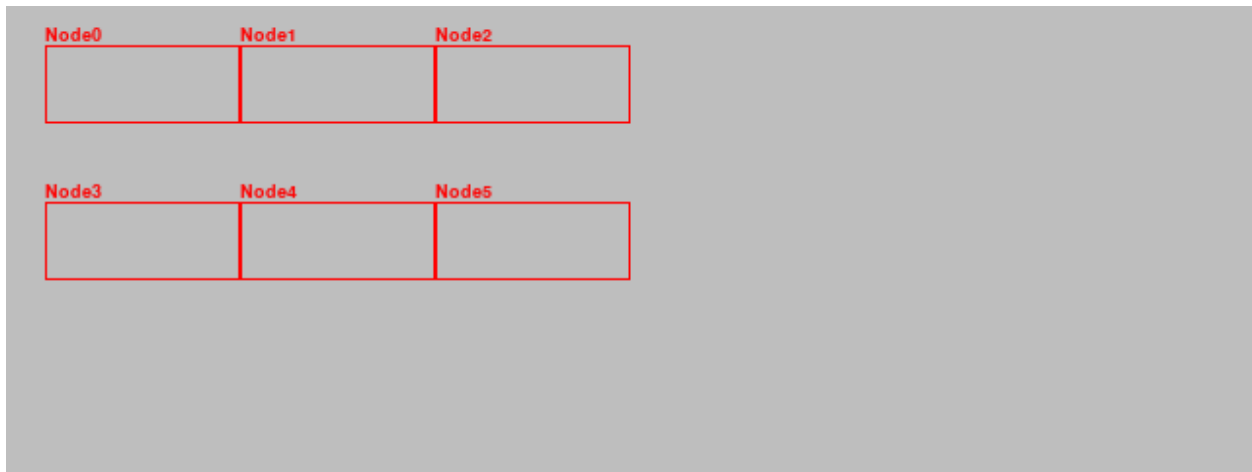
Node(pos=(200, 20))
Node()
Node()
```



Here we change the node placement direction to horizontal, `dir=(0, 1)`. At any time we can change the node position or gap. We can place the initial node position at (0, 0) and change the gap to (0, 0):

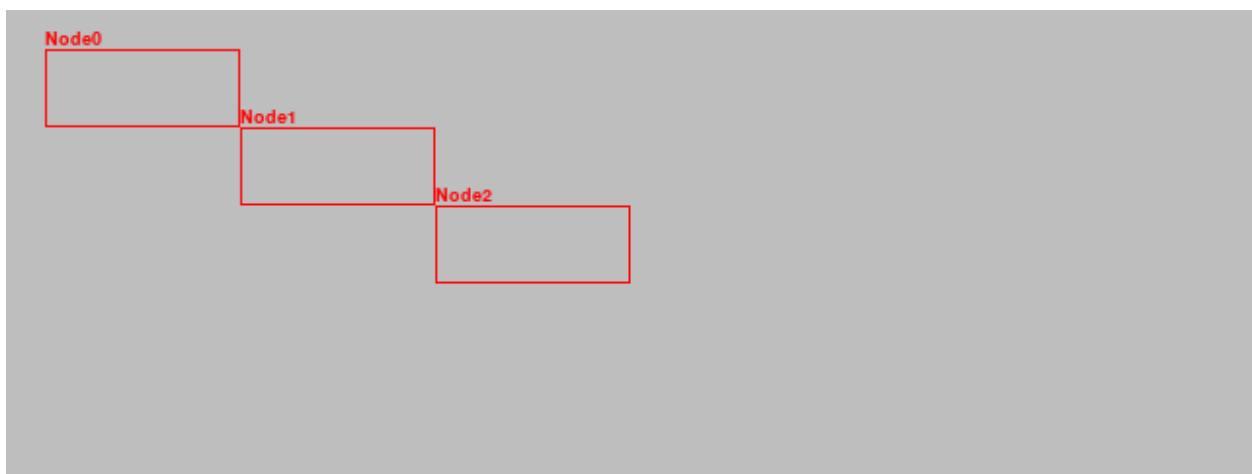
```
Scene(caption='Nodes - horizontal placement')
Node(dir=(1, 0), pos=(0, 0), gap=(0, 0))
Node()
Node()

Node(pos=(0, 100))
Node()
Node()
```



The placement can also be diagonal by choosing the direction vector `dir = (1, 1)`:

```
Scene(caption='Nodes - diagonale placement')
Node(dir=(1, 1), gap=(0, 0))
Node()
Node()
```



Here is the complete code:

```
from app import *

class Demo(App):
    def __init__(self):
        super().__init__()

        Scene(caption='Nodes - vertical placement')
        Node()
        Node()
        Node()

        Node(pos=(200, 20), size=(200, 50))
        Node()
        Node()

        Scene(caption='Nodes - horizontal placement')
```

(continues on next page)

(continued from previous page)

```
Node(dir=(1, 0), gap=(0, 0))
Node()
Node()

Node(pos=(20, 100))
Node()
Node()

Scene(caption='Nodes - diagonal placement')
Node(dir=(1, 1), gap=(0, 0))
Node()
Node()

if __name__ == '__main__':
    Demo().run()
```

Create a graphical user interface (GUI)

The graphical user interface (GUI) consists of all the elements the user can interact with (read, click, drag, resize, select, input):

- text
- button
- checkbutton
- radiobutton
- menu (pop-up, pull-down)
- listbox
- slider

7.1 Text attributes

We store all pygame text attributes as class variables:

```
class Text(Node):  
    """Create a text object which knows how to draw itself."""  
  
    fontname = None  
    fontsize = 36  
    fontcolor = Color('black')  
    background = None  
    italic = False  
    bold = False  
    underline = False
```

After initializing the Node, we update the instance variables from the Text class variables:

```
super().__init__(**options)
self.__dict__.update(Text.options)
```

The font size and the three styles (bold, italic, underline) are set at font creation:

```
def set_font(self):
    """Set the font and its properties."""
    self.font = pygame.font.Font(self.fontname, self.fontsize)
    self.font.set_bold(self.bold)
    self.font.set_italic(self.italic)
    self.font.set_underline(self.underline)
```

The font color and the background color are set when rendering the text:

```
def render(self):
    """Render the text into an image."""
    self.img = self.font.render(self.text, True, self.fontcolor, self.background)
    self.rect.size = self.img.get_size()
```

Here is a code example:

```
"""Display text with different size, color and font."""
from app import *

class Demo(App):
    def __init__(self):
        super().__init__()
        Scene(caption='Text')
        Text('Default text')
        Text('fontsize = 24', fontsize=24)
        Text('fontcolor = RED', fontcolor=Color('red'))
        Text('48 pts, blue', fontsize=48, fontcolor=Color('blue'))
        Text('fontbg = yellow', fontbg=Color('yellow'))

        Text('italic', pos=(400, 20), italic=True)
        Text('bold', bold=True)
        Text('underline', underline=True, font_bg=None)

if __name__ == '__main__':
    Demo().run()
```

Which produces this result:



7.2 Horizontal and vertical alignment

For a given box size, text can be aligned horizontally to the left, center, or right. The following code aligns the text image with these three positions:

```
w, h = self.rect.size
w0, h0 = self.text_img.get_size()

if self.h_align == 0:
    x = 0
elif self.h_align == 1:
    x = (w-w0)//2
else:
    x = w-w0
```

In the vertical direction the text image can be aligned at the top, middle or bottom:

```
if self.v_align == 0:
    y = 0
elif self.v_align == 1:
    y = (h-h0)//2
else:
    y = h-h0

self.img0.blit(self.text_img, (x, y))
self.img = self.img0.copy()
```

The image *img0* is the original, used for scaling. The *img* is the one used for drawing.

Here is a code example:

```
"""Horizontal and vertical text alignment."""
from app import *

class Demo(App):
    def __init__(self):
        super().__init__()

        Scene(caption='Text Alignment', bg=Color('pink'))
```

(continues on next page)

(continued from previous page)

```

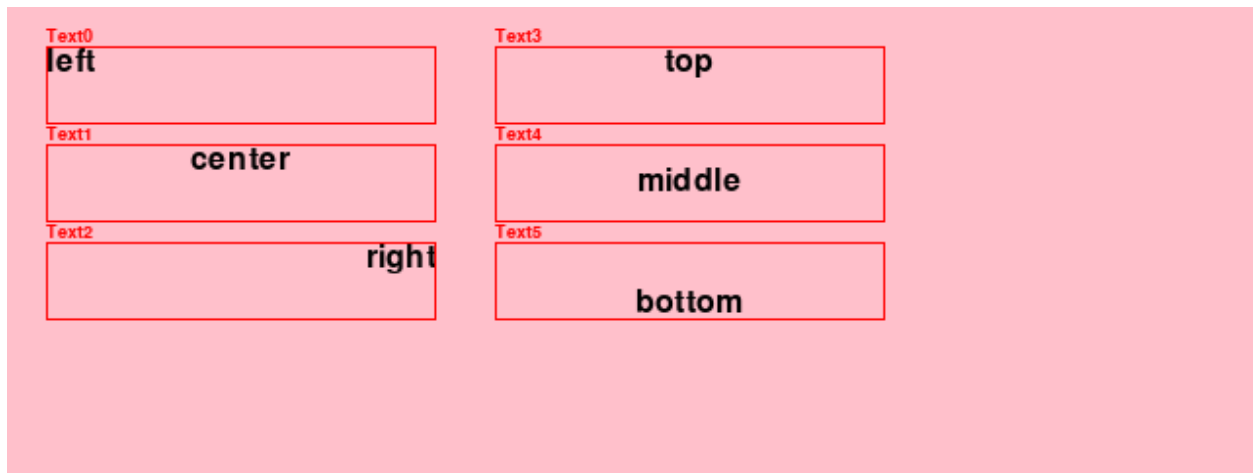
Text('left', size=(200, 40), fontsize=24)
Text('center', h_align=1)
Text('right', h_align=2)
Text(bg=Color('blue'), fontcolor=Color('white'))

Text('top', pos=(250, 20), h_align=1)
Text('middle', v_align=1)
Text('bottom', v_align=2)

if __name__ == '__main__':
    Demo().run()

```

Which produces the following result:



7.3 Text attributes

A Text object has various attributes which are remembered.

Here is a code example:

```

"""Text with size, alignment, fontcolor, font background..."""
from app import *

class Demo(App):
    def __init__(self):
        super().__init__()

        Scene(caption='Text', bg=Color('pink'))
        Text(size=(100, 40))
        Text(bg=Color('yellow'), h_align=1)
        Text(fontcolor=Color('red'))
        Text(fontbg=Color('green'), cmd='print(self.text)')

        Text(pos=(200, 20))
        Text(italic=True, v_align=1)
        Text(underline=True, fontsize=24)
        Text(bold=True)

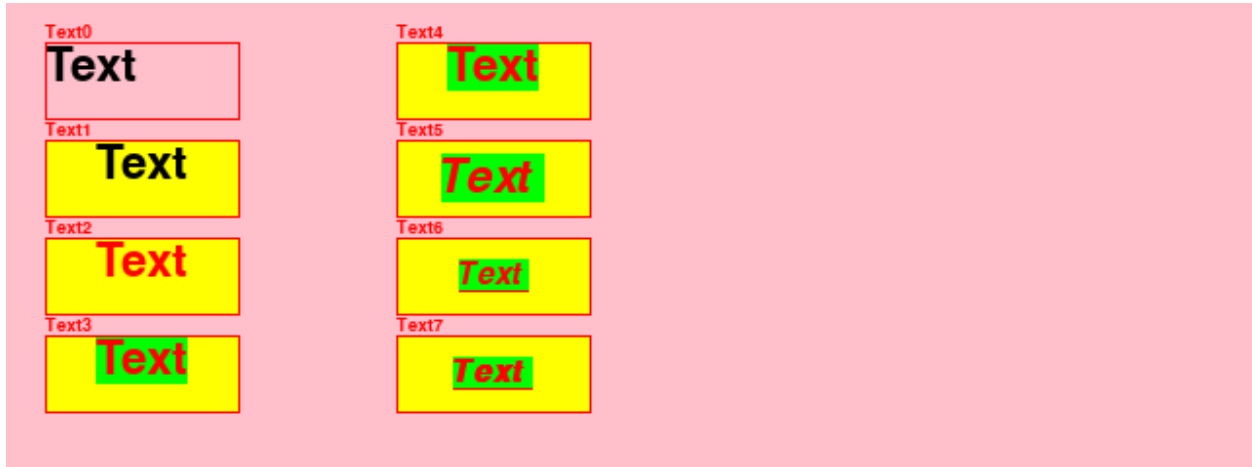
```

(continues on next page)

(continued from previous page)

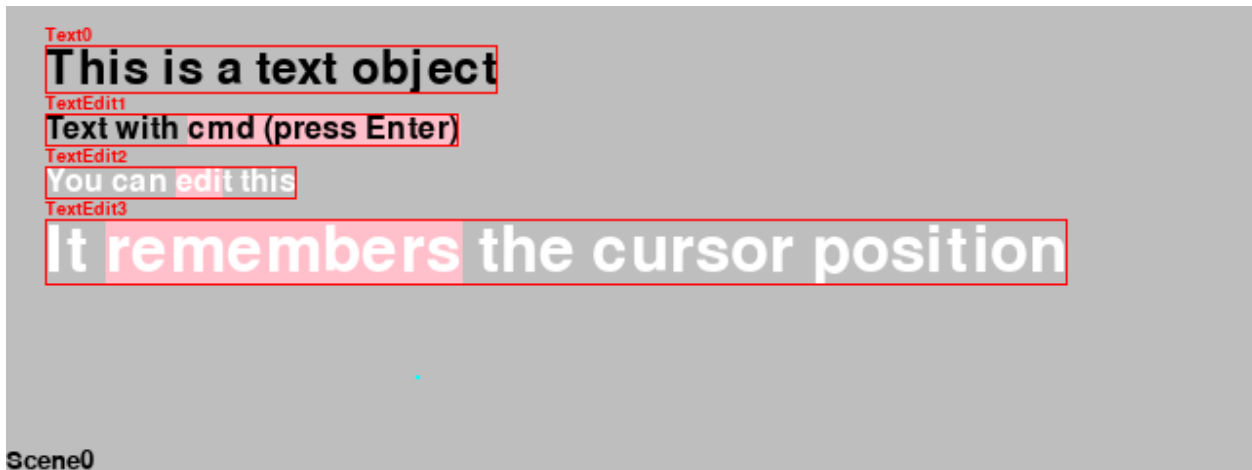
```
if __name__ == '__main__':
    Demo().run()
```

It produces the following result:



7.4 Editable text

The class `TextEdit` provides editable text with a movable cursor. The cursor is represented as a small rectangle which is rendered under the text. A selection is represented as a large rectangle under the selected letters.



7.4.1 Create the cursor

The class attribute `TextEdit.cursor` defines the cursor color and width:

```
cursor = Color('red'), 2 # cursor color and width
```

Inside the constructor, the cursor is placed at the end of the text. A cursor image is created and filled with the cursor color. The cursor rectangle is initially placed at the end of the text:

```
col, d = TextEdit.cursor
self.cursor = len(self.text)
self.cursor_img = pygame.Surface((d, self.rect.height))
self.cursor_img.fill(col)
self.cursor_rect = self.cursor_img.get_rect()
self.cursor_rect.topleft = self.rect.topright
```

7.4.2 Get the character index

The cursor is represented as an integer index in the range $[0 .. n]$ where n is the length of the text. Each letter has a different width. The list `self.char_positions` remembers the x position of each letter:

```
def set_char_positions(self):
    """Get a list of all character positions."""
    self.char_positions = [0]
    for i in range(len(self.text)):
        w, h = self.font.size(self.text[:i+1])
        self.char_positions.append(w)
```

When we click with the mouse anywhere in the text, we need to know the character index:

```
def get_char_index(self, position):
    """Return the character index for a given position."""
    for i, pos in enumerate(self.char_positions):
        if position <= pos:
            return i
    # if not found return the highest index
    return i
```

7.4.3 Move the cursor

The arrow keys allow to move the cursor to the left or to the right. The argument `d` is 1 or -1 and indicates the direction of movement. The cursor movement is limited to the interval $[0 .. n]$:

```
def move_cursor(self, d):
    """Move the cursor by d characters, and limit to text length."""
    mod = pygame.key.get_mods()
    n = len(self.text)
    i = min(max(0, self.cursor+d), n)
```

Pressing the CMD key, the cursor goes all the way to the beginning or the end of the line:

```
if mod & KMOD_META:
    if d == 1:
        i = n
    else:
        i = 0
```

Pressing the ALT key, the cursor goes to the end of the word:

```
if mod & KMOD_ALT:
    while (0 < i < n) and self.text[i] != ' ':
        i += d
```

Pressing the SHIFT key prevents cursor2 from moving, thus setting a selection:

```
if not mod & KMOD_SHIFT:
    self.cursor2 = i

self.cursor = i
```

7.4.4 Copy, cut and insert text

The two cursors can be inverted. The following method returns the two cursors (selection indices) in the right order:

```
def get_selection_indices(self):
    """Get ordered tuple of selection indices."""
    i = self.cursor
    i2 = self.cursor2

    if i < i2:
        return i, i2
    else:
        return i2, i
```

To copy text we save the selection in a Scene variable text:

```
def copy_text(self):
    """Copy text to Scene.text buffer."""
    i, i2 = self.get_selection_indices()
    text = self.text[i:i2]
    App.scene.text = text
```

To cut text we copy the text and replace the selection with an empty string:

```
def cut_text(self):
    """Cut text and place copy in Scene.text buffer."""
    self.copy_text()
    self.insert_text('')
```

To insert text we replace the current selection with the new text:

```
def insert_text(self, text):
    """Insert text at the cursor position or replace selection."""
    i, i2 = self.get_selection_indices()
    text1 = self.text[:i]
    text2 = self.text[i2:]
    self.text = text1 + text + text2
    self.cursor = i + len(text)
    self.cursor2 = self.cursor
```

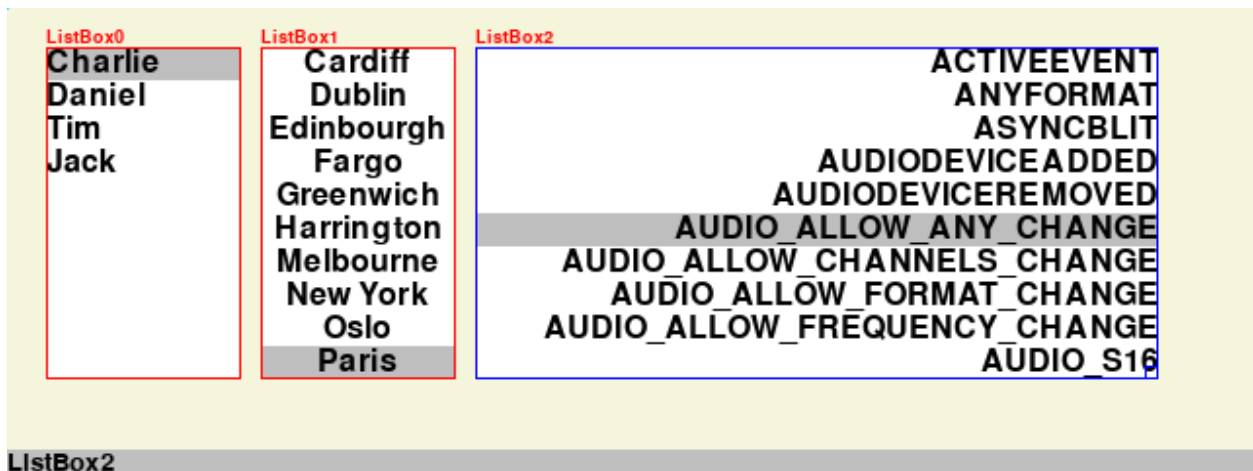
7.5 Buttons

The button class displays a text and executes a command upon a mouse-click



7.6 ListBox

The `ListBox` class displays a list of items. One item can be selected with a mouse-click or with the UP/DOWN arrow keys. Pressing the RETURN key executes the command.



7.7 Detecting double-clicks

In order to detect double-clicks or multiple clicks we need to use a timer event. The reason for using a timer is that we cannot know at the time of a mouse click if there are more clicks to follow. We only know for sure after a short timeout period. So we define a new event as the first `USEREVENT`:

```
DBL_CLICK_TIMER = pygame.USEREVENT
DBL_CLICK_TIMEOUT = 250
```

Inside the `Scene.do_event()` we look for a `MOUSEBUTTONDOWN` event and we set a timer and increment the clicks:

```
if event.type == MOUSEBUTTONDOWN:
    pygame.time.set_timer(DBL_CLICK_TIMER, DBL_CLICK_TIMEOUT)
    self.clicks += 1
```

Once the timeout occurs, we

- reset (disable) the timer
- print the number of clicks and
- reset the click count to zero:

```
elif event.type == DBL_CLICK_TIMER:
    pygame.time.set_time(DBL_CLICK_TIMER, 0)
    print(self.clicks, 'clicks in', self.focus)
    self.clicks = 0
```

The text printed to the console looks like this:

```
2 clicks in Text0
4 clicks in Text0
3 clicks in Ellipse1
1 clicks in Rectangle2
2 clicks in None
```


8.1 Making sounds

The `pygame.mixer` module allows to play compressed OGG files or uncompressed WAV files.

This checks the initialization parameters and prints the number of channels available. It opens a sound object and plays it:

```
print('init =', pygame.mixer.get_init())
print('channels =', pygame.mixer.get_num_channels())
App.snd = pygame.mixer.Sound('5_app/rpgaudio.ogg')
App.snd.play()
print('length =', App.snd.get_length())
```

Writes this to the console:

```
init = (22050, -16, 2)
channels = 8
length = 28.437868118286133
```

Here is a code example:

```
"""Play a sound."""
from app import *

class Demo(App):
    def __init__(self):
        super().__init__()

        print('init =', pygame.mixer.get_init())
        print('channels =', pygame.mixer.get_num_channels())
        App.snd = pygame.mixer.Sound('5_app/rpgaudio.ogg')
        App.snd.play()
        print('length =', App.snd.get_length())
```

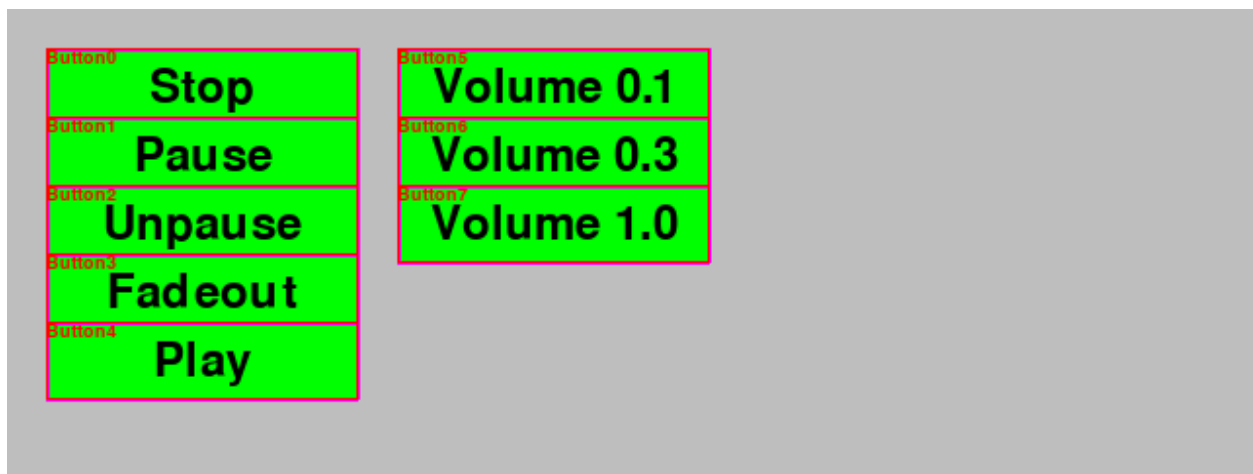
(continues on next page)

(continued from previous page)

```
Scene(caption='Sound mixer')
Button('Stop', cmd='pygame.mixer.stop()')
Button('Pause', cmd='pygame.mixer.pause()')
Button('Unpause', cmd='pygame.mixer.unpause()')
Button('Fadeout', cmd='pygame.mixer.fadeout(5000)')
Button('Play', cmd='App.snd.play()')
Button('Volume 0.1', cmd='App.snd.set_volume(0.1)', pos=(200, 20))
Button('Volume 0.3', cmd='App.snd.set_volume(0.3)')
Button('Volume 1.0', cmd='App.snd.set_volume(1.0)')

if __name__ == '__main__':
    Demo().run()
```

Which produces the following result.



In this section we create the framework for board games. These games are based on a $n \times m$ grid. Each cell can have

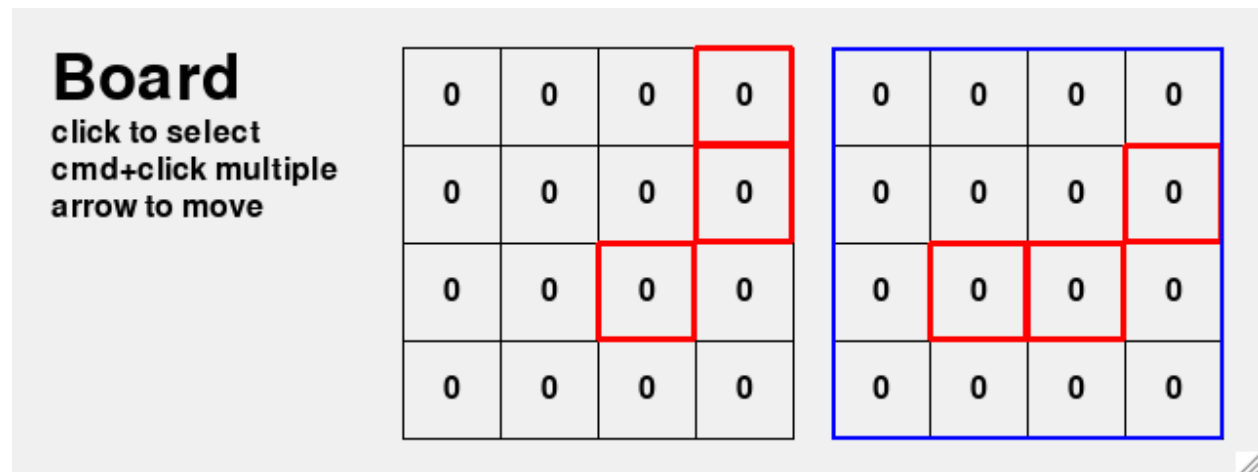
- text
- color
- image

Board

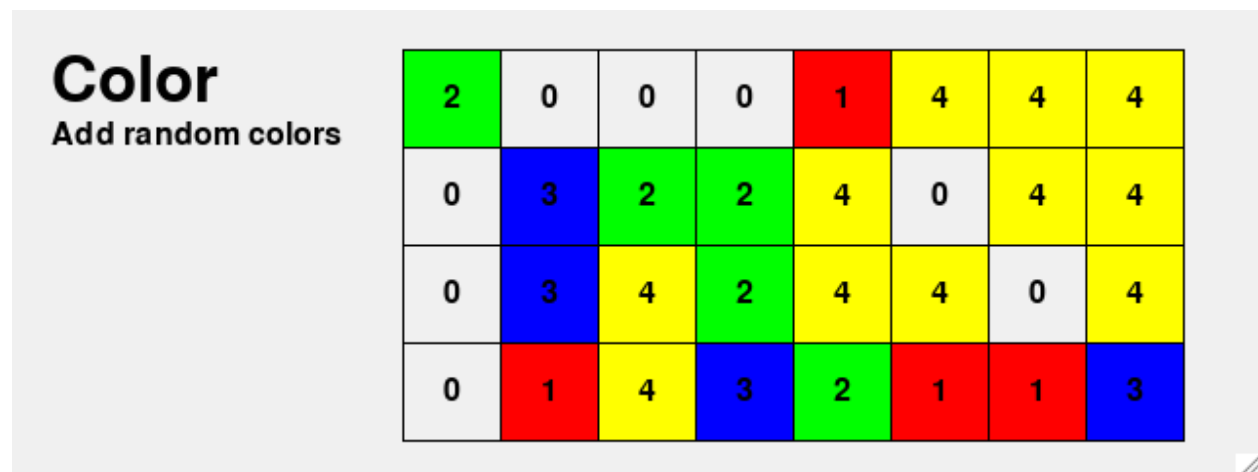
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0



9.1 Selecting cells with the mouse



9.2 Adding background color



9.3 Create a checkerboard pattern

Checker

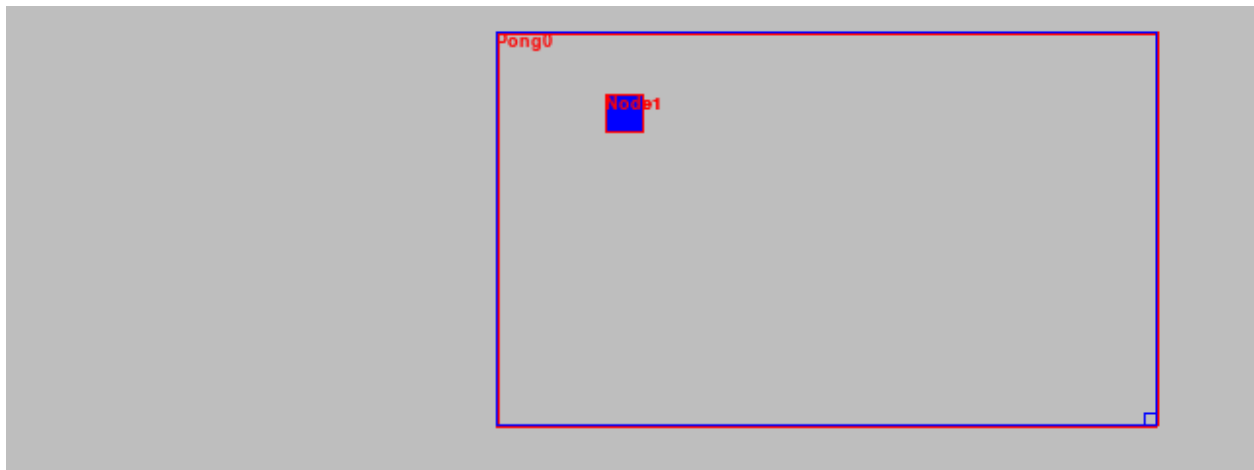
Create a pattern

0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0

CHAPTER 10

Simple games

10.1 Pong



```
import pygame
from pygame.locals import *
from pygamelib import *

class Ball:
    def __init__(self, pos, field, pad):
        self.pos = pos
        self.field = field
        self.pad = pad
        self.speed = [1, 1]
        self.color = RED
        self.rect = pygame.Rect(pos, (20, 20))

    def update(self):
```

(continues on next page)

(continued from previous page)

```

        self.rect.move_ip(self.speed)

        if self.rect.left < self.field.rect.left:
            self.speed[0] = abs(self.speed[0])
        if self.rect.right > self.field.rect.right:
            self.speed[0] = -abs(self.speed[0])

        if self.rect.top < self.field.rect.top:
            self.speed[1] = abs(self.speed[1])
        if self.rect.bottom > self.field.rect.bottom:
            self.speed[1] = -abs(self.speed[1])

        if self.rect.colliderect(self.pad.rect):
            self.speed[0] = abs(self.speed[0])

    def draw(self):
        pygame.draw.rect(App.screen, self.color, self.rect, 0)

class Pad:
    def __init__(self, keys, field):
        self.keys = keys
        self.field = field
        self.speed = [0, 0]
        self.v = 5
        self.color = GREEN
        self.rect = pygame.Rect(self.field.rect.topleft, (10, 50))
        self.rect.move_ip(10, 0)

    def do(self, event):
        if event.type == KEYDOWN:
            if event.key == self.keys[0]:
                self.speed[1] = -self.v
            if event.key == self.keys[1]:
                self.speed[1] = self.v

        elif event.type == KEYUP:
            self.speed[1] = 0

    def update(self):
        self.rect.move_ip(self.speed)

        if self.rect.top < self.field.rect.top:
            self.rect.top = self.field.rect.top
        if self.rect.bottom > self.field.rect.bottom:
            self.rect.bottom = self.field.rect.bottom

    def draw(self):
        pygame.draw.rect(App.screen, self.color, self.rect, 0)

class Field:
    def __init__(self, rect):
        self.color = WHITE
        self.bg_color = BLACK
        self.stroke = 10
        self.rect = pygame.Rect(rect)

```

(continues on next page)

(continued from previous page)

```

def draw(self):
    pygame.draw.rect(App.screen, self.color, self.rect, self.stroke)
    pygame.draw.rect(App.screen, self.bg_color, self.rect, 0)

class PongDemo(App):
    """Play the game of Pong."""
    def __init__(self):
        super().__init__()
        Text('Pong', size=48)

        self.field = Field((200, 10, 400, 200))
        self.pad = Pad((K_UP, K_DOWN), self.field)
        self.ball = Ball(self.field.rect.center, self.field, self.pad)
        self.bg_color = GRAY

    def run(self):
        while self.running:
            for event in pygame.event.get():
                if event.type == QUIT:
                    self.running = False
                    pygame.quit()

                self.pad.do(event)

            self.update()
            self.draw()

    def update(self):
        self.ball.update()
        self.pad.update()

    def draw(self):
        self.screen.fill(self.bg_color)
        self.field.draw()
        self.ball.draw()
        self.pad.draw()
        pygame.display.flip()

if __name__ == '__main__':
    PongDemo().run()

```

pong.py

CHAPTER 11

Breakout

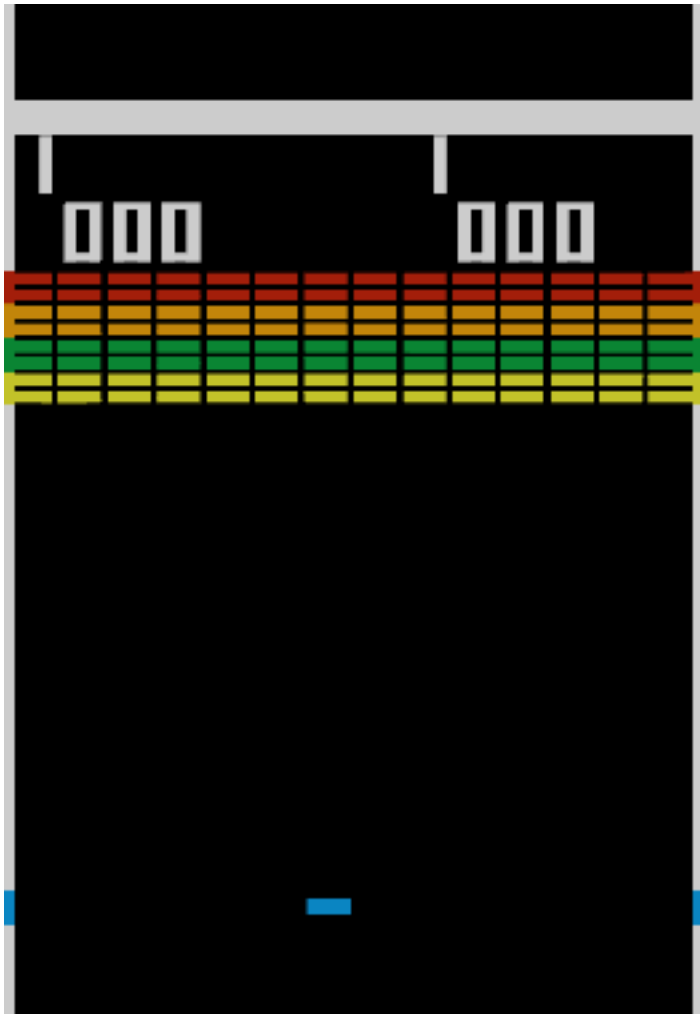
Breakout is an arcade game developed and published by Atari and released in 1976.

Breakout begins with eight rows of bricks, with each two rows a different color. The color order from the bottom up is - yellow, - green, - orange and - red.

Using a single ball, the player must knock down as many bricks as possible by using the walls and/or the paddle below to ricochet the ball against the bricks and eliminate them.

If the player's paddle misses the ball's rebound, they will lose a turn. The player has three turns to try to clear two screens of bricks. Yellow bricks earn one point each, green bricks earn three points, orange bricks earn five points and the top-level red bricks score seven points each. The paddle shrinks to one-half its size after the ball has broken through the red row and hit the upper wall.

Ball speed increases at specific intervals: after four hits, after twelve hits, and after making contact with the orange and red rows. The highest score achievable for one player is 896; this is done by eliminating two screens of bricks worth 448 points per screen.



11.1 Making an app

The first step is to create a frame work which creates a window

```
10_games/breakout/pong1.png
```

```
from pygame.locals import *
import numpy as np

class Sprite:
    def __init__(self, file=None, pos=(0, 0), size=None):
        self.parent = None
        self.size = size
        self.rect = Rect(pos, (20, 20))
        self.position = np.array(pos, dtype=float)
```

(continues on next page)

(continued from previous page)

```

        self.velocity = np.array([1.5, 0.5], dtype=float)

        self.angle = 0
        self.angular_velocity = 0

        self.color = 'red'
        self.speed = [0, 0]
        if file:
            self.image = pygame.image.load(file)
            if self.size:
                self.image = pygame.transform.scale(self.image, size)
                self.rect.size = self.image.get_size()
            else:
                self.image = pygame.Surface(self.rect.size)
                self.image.fill(self.color)
        self.image0 = self.image.copy()

    def set_pos(self, pos):
        self.position = np.array(pos, dtype=float)
        self.rect.center = pos

    def set_angle(self, angle):
        self.angle = angle
        self.image = pygame.transform.rotate(self.image0, self.angle)
        self.rect.size = self.image.get_size()

    def do(self, event):
        pass

    def update(self):
        self.move()

    def move(self):
        self.position += self.velocity

        if self.angular_velocity:
            self.angle += self.angular_velocity
            self.image = pygame.transform.rotate(self.image0, self.angle)
            self.rect.size = self.image.get_size()

        self.rect.center = self.position

    def draw(self, surf):
        surf.blit(self.image, self.rect)

    def distance(self, other):
        distance = self.position - other.position
        distance *= distance
        d = np.sqrt(np.sum(distance))
        return d

class App:
    def __init__(self, file=None, caption='Pygame'):
        pygame.init()
        pygame.display.set_caption(caption)
        self.flags = RESIZABLE

```

(continues on next page)

(continued from previous page)

```

self.size = (640, 240)
self.screen = pygame.display.set_mode(self.size, self.flags)
self.running = True
self.updating = True
self.objects = []
self.bg_color = 'gray'
if file:
    self.load_image(file)
else:
    self.image = pygame.Surface(self.size)
    self.image.fill(self.bg_color)
    self.rect = self.image.get_rect()
self.key_cmd = {}

def load_image(self, file):
    self.image = pygame.image.load(file).convert()
    self.rect = self.image.get_rect()
    self.screen = pygame.display.set_mode(self.rect.size, self.flags)

def run(self):
    while self.running:
        for event in pygame.event.get():
            self.do(event)
        self.update()
        self.draw()

def add_cmd(self, key, cmd):
    self.key_cmd[key] = cmd
    print(self.key_cmd)

def add(self, object):
    self.objects.append(object)
    object.parent = self

def do(self, event):
    if event.type == QUIT:
        self.running = False
        pygame.quit()
    elif event.type == KEYDOWN:
        if event.key == K_SPACE:
            self.updating = not self.updating

        if event.key in self.key_cmd:
            cmd = self.key_cmd[event.key]
            eval(cmd)

    for obj in self.objects:
        obj.do(event)

def update(self):
    if self.updating:
        for obj in self.objects:
            obj.update()

def draw(self):
    self.screen.blit(self.image, self.rect)
    for obj in self.objects:

```

(continues on next page)

(continued from previous page)

```
        obj.draw(self.screen)
    pygame.display.update()

if __name__ == '__main__':
    app = App('space.png', 'Asteroids')

    ship = Sprite('spaceship.png', size=(100, 50), pos=(300, 200))
    app.add(ship)
    app.add(Sprite('asteroid.png', size=(100, 100), pos=(100, 300)))
    app.add(Sprite('asteroid.png', size=(150, 150), pos=(400, 100)))

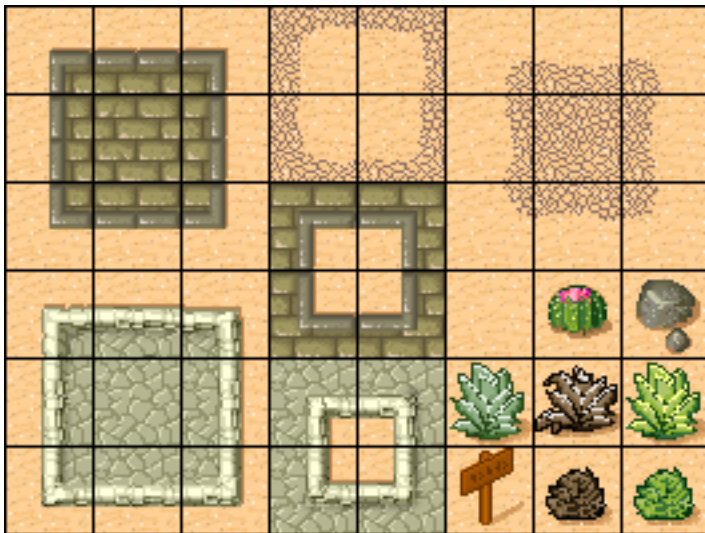
    app.add_cmd(K_a, 'print(123)')
    app.add_cmd(K_b, "self.load_image('space.png')")
    app.run()
```

app.py

CHAPTER 12

Tiles

Tiles are small squares to compose an image. Below is an image which has 6 x 8 tiles. Each tile has 32 x 32 pixels.



The tiles with a 1 pixel separation

```
import pygame

file = 'tmw_desert_spacing.png'
image = pygame.image.load(file)
rect = image.get_rect()
print(image)

pygame.init()
screen = pygame.display.set_mode(rect.size)

screen.blit(image, rect)
pygame.display.update()
```

(continues on next page)

(continued from previous page)

```
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
```

tiles0.py

The code displays the image and writes this to the output console:

```
<Surface(265x199x32 SW)>
```

The margin around the image and the separation line is 1 pixel.

- $8 \times 33 + 1 = 265$ pixel wide
- $6 \times 33 + 1 = 199$ pixel high

tiles1.py

```
import pygame
from pygame.locals import *

file = 'tmw_desert_spacing.png'

class Game:
    W = 640
    H = 240
    SIZE = W, H

    def __init__(self):
        pygame.init()
        self.screen = pygame.display.set_mode(Game.SIZE)
        pygame.display.set_caption("Pygame Tiled Demo")
        self.running = True

    def run(self):
        while self.running:
            for event in pygame.event.get():
                if event.type == QUIT:
                    self.running = False

                elif event.type == KEYDOWN:
                    if event.key == K_1:
                        self.load_image(file)

        pygame.quit()

    def load_image(self, file):
        self.file = file
        self.image = pygame.image.load(file)
        self.rect = self.image.get_rect()

        self.screen = pygame.display.set_mode(self.rect.size)
        pygame.display.set_caption(f'size:{self.rect.size}')
        self.screen.blit(self.image, self.rect)
        pygame.display.update()
```

(continues on next page)

(continued from previous page)

```
game = Game()
game.run()
```

12.1 Create a Tileset

```
class Tileset:
    def __init__(self, file, size=(32, 32), margin=1, spacing=1):
        self.file = file
        self.size = size
        self.margin = margin
        self.spacing = spacing
        self.image = pygame.image.load(file)
        self.rect = self.image.get_rect()
        self.tiles = []
        self.load()

    def load(self):

        self.tiles = []
        x0 = y0 = self.margin
        w, h = self.rect.size
        dx = self.size[0] + self.spacing
        dy = self.size[1] + self.spacing

        for x in range(x0, w, dx):
            for y in range(y0, h, dy):
                tile = pygame.Surface(self.size)
                tile.blit(self.image, (0, 0), (x, y, *self.size))
                self.tiles.append(tile)

    def __str__(self):
        return f'{self.__class__.__name__} file:{self.file} tile:{self.size}'
```

12.2 Create a Tilemap

```
class Tilemap:
    def __init__(self, tileset, size=(10, 20), rect=None):
        self.size = size
        self.tileset = tileset
        self.map = np.zeros(size, dtype=int)

        h, w = self.size
        self.image = pygame.Surface((32*w, 32*h))
        if rect:
            self.rect = pygame.Rect(rect)
        else:
            self.rect = self.image.get_rect()

    def render(self):
        m, n = self.map.shape
```

(continues on next page)

(continued from previous page)

```

for i in range(m):
    for j in range(n):
        tile = self.tiles[ self.map[i, j] ]
        self.image.blit(tile, (j*32, i*32))

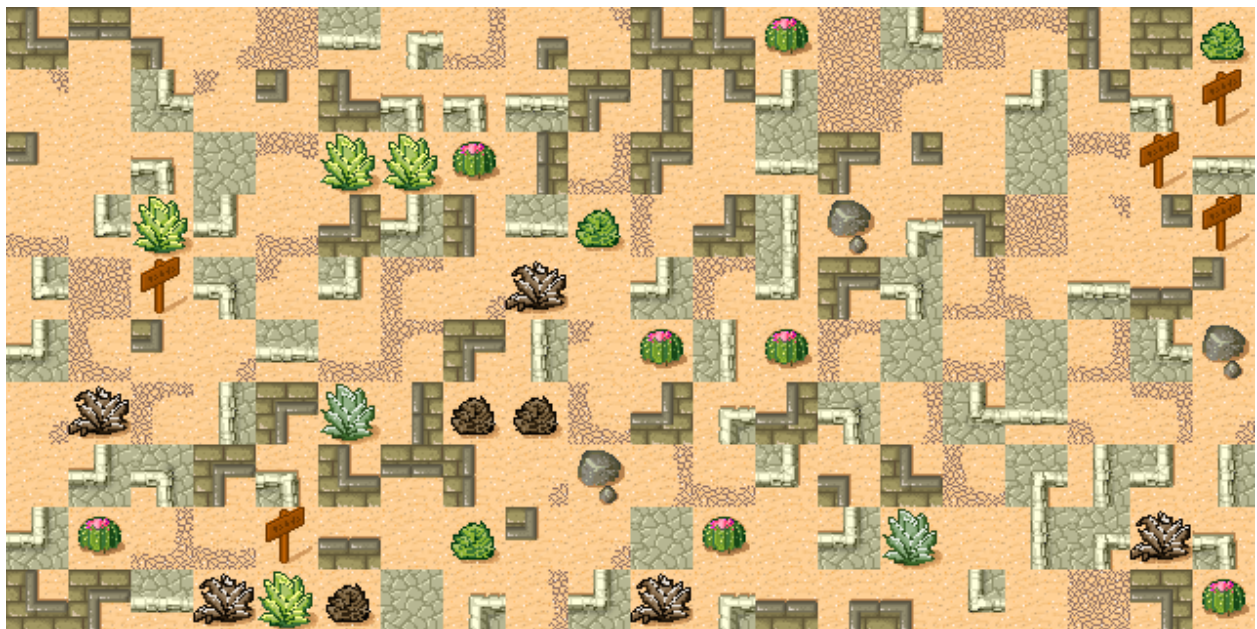
def set_zero(self):
    self.map = np.zeros(self.size, dtype=int)
    print(self.map)
    print(self.map.shape)
    self.render()

def set_random(self):
    n = len(self.tiles)
    self.map = np.random.randint(n, size=self.size)
    print(self.map)
    self.render()

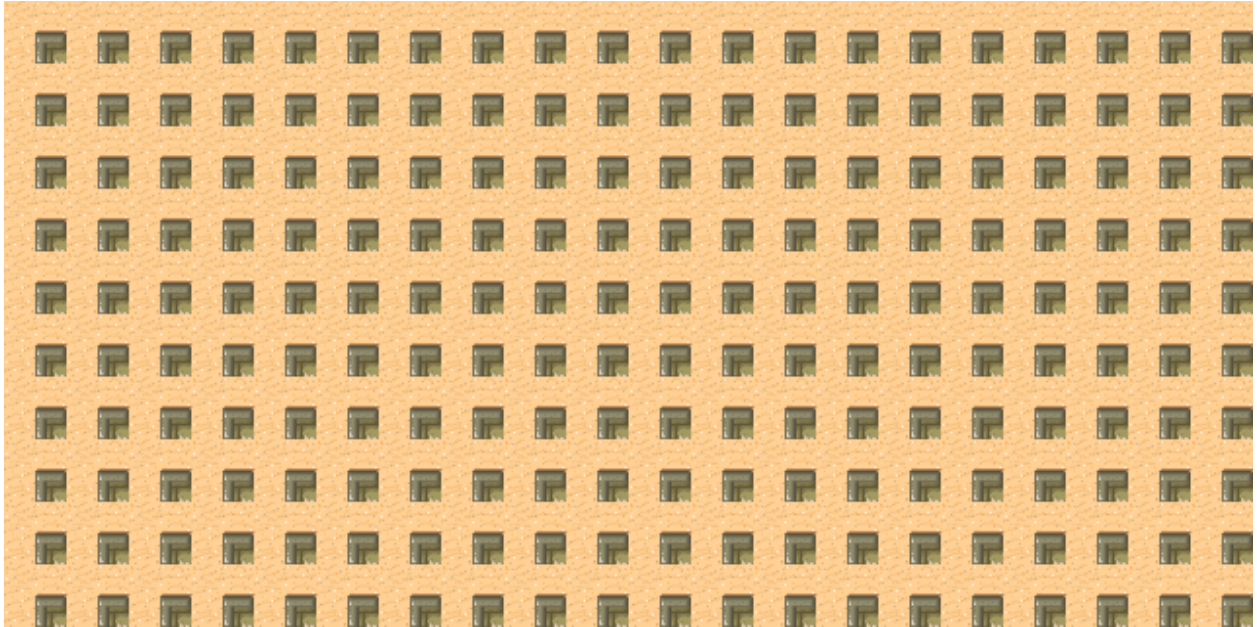
def __str__(self):
    return f'{self.__class__.__name__} {self.size}'

```

The R key fills the map with random tiles.



The Z key fills the map with zero-index tiles.



12.3 Navigation

12.4 Tiled : a tile-map editor

Tiled is a 2D level editor <https://www.mapeditor.org>

Download a tile set <https://opengameart.org/content/zelda-like-tilesets-and-sprites>

YouTube tutorials for Tiled <https://gamefromscratch.com/tiled-map-editor-tutorial-series/>

Importing a TMX map into Python <https://pytmx.readthedocs.io/en/latest/index.html>

Pygame Tiled demo <https://github.com/pricheal/pygame-tiled-demo>

CHAPTER 13

About Sphinx

Sphinx is a tool for making documentation. It was originally created for the [Python documentation](#), but is now used for many other software projects.

Sphinx uses *reStructuredText* as its markup language. It can produce HTML, LaTeX, ePub and PDF documents.

Source: <https://www.sphinx-doc.org>

13.1 Getting started

After installation, you can get started quickly with the tool **sphinx-quickstart**. Just enter:

```
sphinx-quickstart
```

Answer each customization question with yes or no. Be sure to say **yes** to the **autodoc** extension. The **sphinx-quickstart** creates a directory with several documents:

- `conf.py` file, the *default configuration file*
- `index.rst` file, the *master document*

The `conf.py` file let's you configure all aspects of Sphinx. The `index.rst` is the entry page for your documentation. It contains the `toctree` directive which determines the files to include. For this project it looks like this:

```
.. toctree::
   :maxdepth: 2
   :caption: Contents:

   1_intro/intro
   2_draw/draw
   3_image/image
   ...
```

To build the HTML pages just run:

```
make html
```

To make the PDF document run:

```
make pdf
```

13.2 reStructuredText

reStructuredText (.rst) is the default markup language used with Sphinx. It is important to know that:

- paragraphs are separated by one or more blank lines
- indentation is significant

13.2.1 Inline styles

Inside text you can use:

- one asterisk for *italics*
- two asterisks for **bold**
- backquotes for `code`

13.2.2 Lists

This code:

```
1 * This is a bulleted list.  
2 * It has two items, the second  
3   item uses two lines.  
4  
5 #. This is a numbered list.  
6 #. It has two items too.
```

produces this result:

- This is a bulleted list.
 - It has two items, the second item uses two lines.
1. This is a numbered list.
 2. It has two items too.

13.2.3 Hyperlinks

This code:

```
`Source <https://www.sphinx-doc.org>`_
```

produces [Source](#)

13.2.4 Admonitions

Danger: Be careful with this code!

Tip: Be careful with this code!

Warning: Be careful with this code!

13.2.5 Footnotes

This is a footnote¹ inside a text, this is another one².

13.2.6 Horizontal list

To add a horizontal list add this code:

```
.. hlist::
   :columns: 3

   * happy
   ...
```

- happy
- short
- intelligent
- thankful
- displayed
- horizontal

13.2.7 Download

To add a download link add this code:

```
:download:`requirements.txt<requirements.txt>`.
```

requirements.txt.

13.3 Include from a file

It is possible to include a Python object (class, method) from a file. For example you can include a **class** definition with:

¹ Text of the first footnote

² Text of the second footnote

```
.. literalinclude:: 5_app/app.py
:pyobject: Rectangle
:linenos:
:emphasize-lines: 5-7
```

resulting in

```
1 class Rectangle(Node):
2     """Draw a rectangle on the screen."""
3     options = { 'fg': Color('green'),
4                 'bg': Color('black'),
5                 'thickness': 2}
6
7     def __init__(self, **options):
8         super().__init__(**options)
9         self.set_options(Rectangle, options)
10        self.render()
11
12    def render(self):
13        self.img0 = pygame.Surface(self.rect.size, flags=SRCALPHA)
14        if self.fg != None:
15            pygame.draw.rect(self.img0, self.fg, Rect(0, 0, *self.rect.size), 0)
16        pygame.draw.rect(self.img0, self.bg, Rect(0, 0, *self.rect.size), self.
17↪thickness)
18        self.img = self.img0.copy()
```

Or you can include just a **method** definition with:

```
.. literalinclude:: 5_app/app.py
:pyobject: Rectangle.render
```

resulting in

```
def render(self):
    self.img0 = pygame.Surface(self.rect.size, flags=SRCALPHA)
    if self.fg != None:
        pygame.draw.rect(self.img0, self.fg, Rect(0, 0, *self.rect.size), 0)
    pygame.draw.rect(self.img0, self.bg, Rect(0, 0, *self.rect.size), self.
↪thickness)
    self.img = self.img0.copy()
```

13.4 Directives

A directive consists of

- name
- arguments
- options
- content

The structure is this:

```
.. name:: arguments
   :option: value

   content
```

13.4.1 Function

This directive defines a function:

```
.. function:: spam(eggs)
             ham(eggs)

   Spam ham ham the are made with a certain number of eggs.
```

spam(*eggs*)
ham(*eggs*)
 Spam and ham the are made with a certain number of eggs.

To cross-reference you can use:

- `method_name()` **with** `:meth:`method_name``
- `class_name` **with** `:class:`class_name``
- `function_name()` **with** `:func:`function_name``

For example with `:func:`spam`` one can reference the above functions `spam()` or `ham()` inside a sentence..

13.4.2 Data

To describe global data and constants in a module use this code:

```
.. data:: number=1000

   Describe data.
```

produces

number=1000
 Describe data.

13.4.3 Class

class App
 Describe class without parameters.

run()
 Describe the method.

class App(*parameters*)
 Describe class with parameters.

objects
 Global class attribute.

13.4.4 Functions with arguments

send_message (*sender*, *recipient*, *message_body* [, *priority*=1])

Send a message to a recipient

Parameters

- **sender** (*str*) – The person sending the message
- **recipient** (*str*) – The recipient of the message
- **message_body** (*str*) – The body of the message
- **priority** (*integer or None*) – The priority of the message, can be a number 1-5

Returns the message id

Return type int

Raises

- **ValueError** – if the message_body exceeds 160 characters
- **TypeError** – if the message_body is not a basestring

13.5 Math formulas

Since Pythagoras, we know that $a^2 + b^2 = c^2$.

$$e^{i\pi} + 1 = 0 \tag{13.1}$$

Euler's identity, equation (13.1), was elected one of the most beautiful mathematical formulas.

13.6 The app module

This code:

```
.. automodule:: app
   :members:
   :member-order: bysource
```

Prints the whole app documentation and lists members by source order.

App - there is only one App object - an app has multiple scenes (App.scenes) - an app has one current scene (App.scene)
- an app has one window to draw in (App.screen)

Scene

- a scene has multiple nodes (App.scene.nodes)
- nodes are ordered: the last in the list is displayed last
- the node which is clicked becomes active
- the active node becomes the top node
- the active node has focus (App.scene.focus)
- TAB and shift-TAB select the next node

Node (object)

- nodes have default position and size (pos, size)
- nodes are automatically placed at creation (dir, gap)
- nodes inherit options (color, size, ...) from the previous object

A Node object has the following properties

- clickable: mouse-click has effect
- movable: can be moved (mouse, arrow-keys)
- visible: is drawn
- has focus

Debug

- print events to console (cmd+E)
- display node label (cmd+L)
- display outline (cmd+O)

class app.App (size=(640, 240), shortcuts={})

Create a single-window app with multiple scenes having multiple objects.

run ()

Run the main event loop.

next_scene (d=1)

Switch to the next scene.

do_shortcut (event)

Find the key/mod combination in the dictionary and execute the cmd.

capture ()

Save a screen capture to the directory of the calling class, under the class name in PNG format.

toggle_fullscreen ()

Toggle between full screen and windowed screen.

toggle_resizable ()

Toggle between resizable and fixed-size window.

toggle_frame ()

Toggle between frame and noframe window.

class app.Scene (caption='Pygame', remember=True, **options)

Create a new scene and initialize the node options.

load_img (file)

Load the background image.

enter ()

Enter a scene.

update ()

Update the nodes in a scene.

set_status (txt)

Set status text and render it.

render_status ()

Render the status text.

```
draw ()
    Draw all objects in the scene.

do_event (event)
    Handle the events of the scene.

next_focus (d=1)
    Advance focus to next node.

cut ()
    Cuts the selected objects and places them in App.selection.

copy ()
    Copies the selected objects and places them in App.selection.

paste ()
    Pastes the objects from App.selection.

debug ()
    Print all scene/node options.

class app.Node (**options)
    Create a node object with automatic position and inherited size.

    set_options (cls, options)
        Set instance options from class options.

    create_img ()
        Create the image surface, and the original img0.

    color_img ()
        Add background color to the image.

    set_background (img)
        Set background color or transparency.

    load_img ()
        Load the image file.

    calculate_pos (options)
        Calculate the next node position.

    render_label ()
        Create and render the node label.

    do_event (event)
        React to events happening for focus node.

    draw ()
        Draw the node and optionally the outline, label and focus.

class app.TextObj (text='Text', **options)
    Create a text surface image.

    set_font ()
        Set the font and its properties.

    render_text ()
        Render the text into an image.

class app.Text (text='Text', **options)
    Create a text object horizontal and vertical alignment.

class app.TextLines (text, **options)
```

```
class app.EditableTextObj (text='Text', cmd=", **options)
    Create keyboard and mouse-editable text with cursor and selection.

    set_char_positions ()
        Make a list of all character positions.

    get_char_index (position)
        Return the character index for a given position.

    move_cursor (d)
        Move the cursor by d characters, and limit to text length.

    get_selection ()
        Get ordered tuple of selection indices.

    copy_text ()
        Copy text to Scene.text buffer.

    cut_text ()
        Cut text and place copy in Scene.text buffer.

    insert_text (text)
        Insert text at the cursor position or replace selection.

    select_word ()
        Select word at current position.

    select_all ()
        Select the whole text.

    do_event (event)
        Move cursor, handle selection, add/backspace text, copy/paste.

    render ()
        Render cursor, selection and text to an image.

class app.EditableText (text='Text', **options)
    Create an editable text node.

    do_event (event)
        React to events happening for focus node.

    draw ()
        Draw the node and optionally the outline, label and focus.

    double_click ()
        Select the current word.

class app.Button (text='Button', cmd=", **options)
    Create a button object with command.

    do_event (event)
        React to events happening for focus node.

class app.Toggle
    Add toggle button behavior.

class app.Checkbox (**options)

class app.Radiobutton (**options)

class app.ListBox (items, i=0, **options)
    Show a list of text items.
```

```
set_list (items)
    Set items and selection list.

scroll (d)
    Scroll listbox up and down.

move_cursor (d)
    Move the active cell up or down.

do_event (event)
    React to events happening for focus node.

class app.ListMenu (items, **options)
    Display a drop-down menu.

class app.SliderObj (**options)
    Define a slider object.

class app.Slider (**options)

    do_event (event)
        React to events happening for focus node.

class app.NumInput (**options)

    do_event (event)
        React to events happening for focus node.

class app.Spinbox (**options)
    Input a number.

    do_event (event)
        React to events happening for focus node.

class app.Rectangle (**options)
    Draw a rectangle on the screen.

class app.Ellipse (**options)
    Draw an ellipse on the screen.

class app.Board (**options)
    Draw a mxn board grid with m lines and n columns. m, n number of cells (row, col) i, j index of cell (row, col)
    dx, dy size of cell Num numeric matrix Num0 initial numeric matrix Col color matrix

    set_Num (s)
        Load Num table from a string.

    render_colors ()
        Render the background colors.

    render_grid ()
        Render the grid lines.

    render_num ()
        Draw number.

    render_tile ()
        Draw number.

    render ()
        Render the whole board.
```



```
get_index (x, y)
    Get index (i, j) from mouse position (x, y).

do_event (event)
    React to events.

class app.Sudoku (**options)
    Create a sudoko game board.

    render_grid ()
        Override the grid lines.

class app.Chess (**options)
    Create a sudoko game board.

class app.Go (**options)

    render ()
        Render the Go board and add extra dots on certain intersections.

class app.Puzzle (div=(3, 3), **options)
    Take an image and create a puzzle.
```

13.7 Glossary

reStructuredText reStructedText is ligh-weight markup langage.

CHAPTER 14

Indices and tables

- `genindex`
- `modindex`
- `search`

a

app, [102](#)

A

App (*built-in class*), 101
App (*class in app*), 103
app (*module*), 102

B

Board (*class in app*), 106
Button (*class in app*), 105

C

calculate_pos() (*app.Node method*), 104
capture() (*app.App method*), 103
Checkbox (*class in app*), 105
Chess (*class in app*), 107
color_img() (*app.Node method*), 104
copy() (*app.Scene method*), 104
copy_text() (*app.EditableTextObj method*), 105
create_img() (*app.Node method*), 104
cut() (*app.Scene method*), 104
cut_text() (*app.EditableTextObj method*), 105

D

debug() (*app.Scene method*), 104
do_event() (*app.Board method*), 107
do_event() (*app.Button method*), 105
do_event() (*app.EditableText method*), 105
do_event() (*app.EditableTextObj method*), 105
do_event() (*app.ListBox method*), 106
do_event() (*app.Node method*), 104
do_event() (*app.NumInput method*), 106
do_event() (*app.Scene method*), 104
do_event() (*app.Slider method*), 106
do_event() (*app.Spinbox method*), 106
do_shortcut() (*app.App method*), 103
double_click() (*app.EditableText method*), 105
draw() (*app.EditableText method*), 105
draw() (*app.Node method*), 104
draw() (*app.Scene method*), 103

E

EditableText (*class in app*), 105
EditableTextObj (*class in app*), 104
Ellipse (*class in app*), 106
enter() (*app.Scene method*), 103

G

get_char_index() (*app.EditableTextObj method*), 105
get_index() (*app.Board method*), 106
get_selection() (*app.EditableTextObj method*), 105
Go (*class in app*), 107

H

ham() (*built-in function*), 101

I

insert_text() (*app.EditableTextObj method*), 105

L

ListBox (*class in app*), 105
ListMenu (*class in app*), 106
load_img() (*app.Node method*), 104
load_img() (*app.Scene method*), 103

M

move_cursor() (*app.EditableTextObj method*), 105
move_cursor() (*app.ListBox method*), 106

N

next_focus() (*app.Scene method*), 104
next_scene() (*app.App method*), 103
Node (*class in app*), 104
NumInput (*class in app*), 106

O

objects (*App attribute*), 101

P

`paste()` (*app.Scene method*), 104

`Puzzle` (*class in app*), 107

R

`Radiobutton` (*class in app*), 105

`Rectangle` (*class in app*), 106

`render()` (*app.Board method*), 106

`render()` (*app.EditableTextObj method*), 105

`render()` (*app.Go method*), 107

`render_colors()` (*app.Board method*), 106

`render_grid()` (*app.Board method*), 106

`render_grid()` (*app.Sudoku method*), 107

`render_label()` (*app.Node method*), 104

`render_num()` (*app.Board method*), 106

`render_status()` (*app.Scene method*), 103

`render_text()` (*app.TextObj method*), 104

`render_tile()` (*app.Board method*), 106

`reStructuredText`, 107

`run()` (*App method*), 101

`run()` (*app.App method*), 103

S

`Scene` (*class in app*), 103

`scroll()` (*app.ListBox method*), 106

`select_all()` (*app.EditableTextObj method*), 105

`select_word()` (*app.EditableTextObj method*), 105

`send_message()` (*built-in function*), 102

`set_background()` (*app.Node method*), 104

`set_char_positions()` (*app.EditableTextObj method*), 105

`set_font()` (*app.TextObj method*), 104

`set_list()` (*app.ListBox method*), 105

`set_Num()` (*app.Board method*), 106

`set_options()` (*app.Node method*), 104

`set_status()` (*app.Scene method*), 103

`Slider` (*class in app*), 106

`SliderObj` (*class in app*), 106

`spam()` (*built-in function*), 101

`Spinbox` (*class in app*), 106

`Sudoku` (*class in app*), 107

T

`Text` (*class in app*), 104

`TextLines` (*class in app*), 104

`TextObj` (*class in app*), 104

`Toggle` (*class in app*), 105

`toggle_frame()` (*app.App method*), 103

`toggle_fullscreen()` (*app.App method*), 103

`toggle_resizable()` (*app.App method*), 103

U

`update()` (*app.Scene method*), 103