



A function to evaluate the quadratic function $ax^2 + bx + c$:

```
def quadratic(a, b, c, x):  
    quad_term = a * x ** 2  
    lin_term = b * x  
    return quad_term + lin_term + c
```



A function to evaluate the quadratic function $ax^2 + bx + c$:

```
def quadratic(a, b, c, x):  
    quad_term = a * x ** 2  
    lin_term = b * x  
    return quad_term + lin_term + c
```

The names `quad_term` and `lin_term` exist only during the execution of the function `quadratic`. They are called **local variables**.



A function to evaluate the quadratic function $ax^2 + bx + c$:

```
def quadratic(a, b, c, x):  
    quad_term = a * x ** 2  
    lin_term = b * x  
    return quad_term + lin_term + c
```

The names `quad_term` and `lin_term` exist only during the execution of the function `quadratic`. They are called **local variables**.

A function's **parameters** are also **local variables**. When the function is called, the arguments in the function call are assigned to them.



```
def quadratic(a, b, c, x):  
    quad_term = a * x ** 2  
    lin_term = b * x  
    return quad_term + lin_term + c  
  
result = quadratic(2, 4, 5, 3)
```



```
def quadratic(a, b, c, x):  
    quad_term = a * x ** 2  
    lin_term = b * x  
    return quad_term + lin_term + c  
  
result = quadratic(2, 4, 5, 3)
```

Local variables are names that only exist during the execution of the function:

$a \rightarrow 2$

$b \rightarrow 4$

$c \rightarrow 5$

$x \rightarrow 3$

$\text{quad_term} \rightarrow 18$

$\text{lin_term} \rightarrow 12$



```
a = "Letter a"
```

```
def f(a):  
    print "A = ", a
```

```
def g():  
    a = 7  
    f(a + 1)  
    print "A = ", a
```

```
print "A = ", a  
f(3.14)  
print "A = ", a  
g()  
print "A = ", a
```



```
a = "Letter a"
```

```
def f(a):  
    print "A = ", a
```

```
def g():  
    a = 7  
    f(a + 1)  
    print "A = ", a
```

```
print "A = ", a  
f(3.14)  
print "A = ", a  
g()  
print "A = ", a
```

```
A = Letter a  
A = 3.14  
A = Letter a  
A = 8  
A = 7  
A = Letter a
```



Why local variables?

Humans are not good at remembering too many things at the same time. We can only understand software if we can use each part without needing to remember how it works internally.



Why local variables?

Humans are not good at remembering too many things at the same time. We can only understand software if we can use each part without needing to remember how it works internally.

To use the function `quadratic`, we only want to remember this:

```
def quadratic(a, b, c, x):  
    # implemented somehow
```



Humans are not good at remembering too many things at the same time. We can only understand software if we can use each part without needing to remember how it works internally.

To use the function `quadratic`, we only want to remember this:

```
def quadratic(a, b, c, x):  
    # implemented somehow
```

Modularization means that software consists of parts that are developed and tested separately. To use a part, you do not need to understand how it is implemented.

Humans are not good at remembering too many things at the same time. We can only understand software if we can use each part without needing to remember how it works internally.

To use the function `quadratic`, we only want to remember this:

```
def quadratic(a, b, c, x):  
    # implemented somehow
```

Modularization means that software consists of parts that are developed and tested separately. To use a part, you do not need to understand how it is implemented.

`cs1robots` is a module that implements the `object` type `Robot`. You can use `Robot` easily without understanding how it is implemented. → **object-oriented programming**



Variables defined outside of a function are called **global variables**.



Variables defined outside of a function are called **global variables**.

Global variables can be used inside a function:

```
hubo = Robot()
```

← global variable **hubo**

```
def turn_right():  
    for i in range(3):  
        hubo.turn_left()
```

← using global variable



Variables defined outside of a function are called **global variables**.

Global variables can be used inside a function:

```
hubo = Robot()
```

← global variable **hubo**

```
def turn_right():  
    for i in range(3):  
        hubo.turn_left()
```

← using global variable

In large programs, using global variables is dangerous, as they can be accessed (by mistake) by all functions of the program.



If a name is only used inside a function, it is **global**:

```
def f1():  
    return 3 * a + 5
```



If a name is only used inside a function, it is **global**:

```
def f1():  
    return 3 * a + 5
```

If a name is assigned to in a function, it is **local**:

```
def f2(x):  
    a = 3 * x + 17  
    return a * 3 + 5 * a
```




If a name is only used inside a function, it is **global**:

```
def f1():  
    return 3 * a + 5
```

If a name is assigned to in a function, it is **local**:

```
def f2(x):  
    a = 3 * x + 17  
    return a * 3 + 5 * a
```

What does this **test** function print?

```
a = 17  
def test():  
    print a  
    a = 13  
    print a
```



If a name is only used inside a function, it is **global**:

```
def f1():  
    return 3 * a + 5
```

If a name is assigned to in a function, it is **local**:

```
def f2(x):  
    a = 3 * x + 17  
    return a * 3 + 5 * a
```

What does this **test** function print?

```
a = 17  
def test():  
    print(a)  
    a = 13  
    print a
```

Error!

a is a **local** variable in **test** because of the assignment, but has no value inside the first print statement.



Sometimes we want to **change** the value of a **global** variable inside a function.

```
hubo = Robot
```

```
hubo_direction = 0
```

```
def turn_left():  
    hubo.turn_left()  
    global hubo_direction  
    hubo_direction += 90
```

```
def turn_right():  
    for i in range(3):  
        hubo.turn_left()  
    global hubo_direction  
    hubo_direction -= 90
```



What does this code print?

```
def swap(a, b):  
    a, b = b, a
```

```
x, y = 123, 456  
swap(x, y)  
print x, y
```



What does this code print?

```
def swap(a, b):  
    a, b = b, a
```

```
x, y = 123, 456  
swap(x, y)  
print x, y
```

x → 123

y → 456



What does this code print?

```
def swap(a, b):  
    a, b = b, a
```

```
x, y = 123, 456  
swap(x, y) ←  
print x, y
```

x → 123

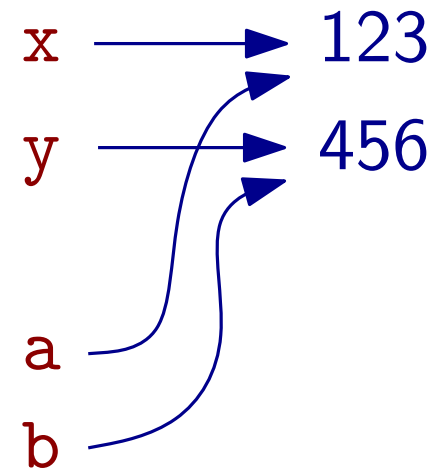
y → 456

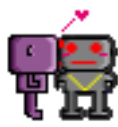


What does this code print?

```
def swap(a, b):  
    a, b = b, a
```

```
x, y = 123, 456  
swap(x, y)  
print x, y
```

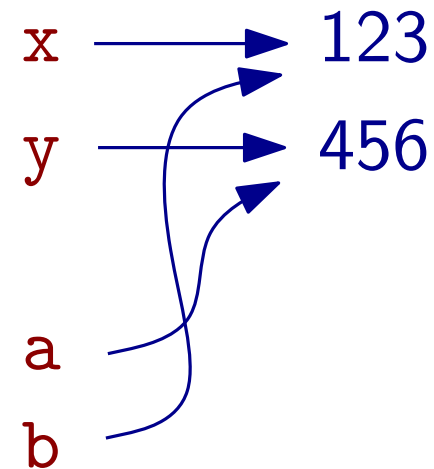




What does this code print?

```
def swap(a, b):  
    a, b = b, a ←
```

```
x, y = 123, 456  
swap(x, y) ←  
print x, y
```





What does this code print?

```
def swap(a, b):  
    a, b = b, a
```

```
x, y = 123, 456  
swap(x, y)  
print x, y ←
```

x → 123

y → 456



What does this code print?

```
def swap(a, b):  
    a, b = b, a
```

```
x, y = 123, 456  
swap(x, y)  
print x, y ←
```

x → 123

y → 456

a is a new name for the object **123**, not for the name **x**!



A Python module is a collection of functions that are grouped together in a file. Python comes with a large number of useful modules. We can also create our own modules.

- `math` for mathematical functions
- `random` for random numbers and shuffling
- `sys` and `os` for accessing the operating system
- `urllib` to download files from the web
- `cs1robots` for playing with Hubo
- `cs1graphics` for graphics
- `cs1media` for processing photos



A Python module is a collection of functions that are grouped together in a file. Python comes with a large number of useful modules. We can also create our own modules.

- `math` for mathematical functions
- `random` for random numbers and shuffling
- `sys` and `os` for accessing the operating system
- `urllib` to download files from the web
- `cs1robots` for playing with Hubo
- `cs1graphics` for graphics
- `cs1media` for processing photos

You can get information about a module using the `help` function:

```
>>> help("cs1media")  
>>> help("cs1media.picture_tool")
```



Before you can use a module you have to **import** it:

```
import math  
print math.sin(math.pi / 4)
```



Before you can use a module you have to **import** it:

```
import math  
print math.sin(math.pi / 4)
```

Sometimes it is useful to be able to use the functions from a module without the module name:

```
from math import *  
print sin(pi / 4)          # OK
```



Before you can use a module you have to **import** it:

```
import math
print math.sin(math.pi / 4)
```

Sometimes it is useful to be able to use the functions from a module without the module name:

```
from math import *
print sin(pi / 4)           # OK
print math.pi               # NameError: name 'math'
```



Before you can use a module you have to **import** it:

```
import math
print math.sin(math.pi / 4)
```

Sometimes it is useful to be able to use the functions from a module without the module name:

```
from math import *
print sin(pi / 4)           # OK
print math.pi               # NameError: name 'math'
```

Or only import the functions you need:

```
from math import sin, pi
print sin(pi / 4)          # OK
```


Before you can use a module you have to **import** it:

```
import math
print math.sin(math.pi / 4)
```

Sometimes it is useful to be able to use the functions from a module without the module name:

```
from math import *
print sin(pi / 4)          # OK
print math.pi              # NameError: name 'math'
```

Or only import the functions you need:

```
from math import sin, pi
print sin(pi / 4)          # OK

print cos(pi / 4)          # NameError: name 'cos'
print math.cos(pi/4)       # NameError: name 'math'
```



We used this:

```
from cs1robots import *  
create_world()  
hubo = Robot()  
hubo.move()  
hubo.turn_left()
```



We used this:

```
from cs1robots import *  
create_world()  
hubo = Robot()  
hubo.move()  
hubo.turn_left()
```

Instead we could use this:

```
import cs1robots  
cs1robots.create_world()  
hubo = cs1robots.Robot()  
hubo.move()  
hubo.turn_left()
```



We used this:

```
from cs1robots import *  
create_world()  
hubo = Robot()  
hubo.move()  
hubo.turn_left()
```

Instead we could use this:

```
import cs1robots  
cs1robots.create_world()  
hubo = cs1robots.Robot()  
hubo.move()  
hubo.turn_left()
```

In general, it is considered better not to use `import *`.



We first need to create a canvas to draw on:

```
from cs1graphics import *  
  
canvas = Canvas(400, 300)  
canvas.setBackgroundColor("light blue")  
canvas.setTitle("CS101 Drawing exercise")
```



We first need to create a canvas to draw on:

```
from cs1graphics import *  
  
canvas = Canvas(400, 300)  
canvas.setBackgroundColor("light blue")  
canvas.setTitle("CS101 Drawing exercise")
```

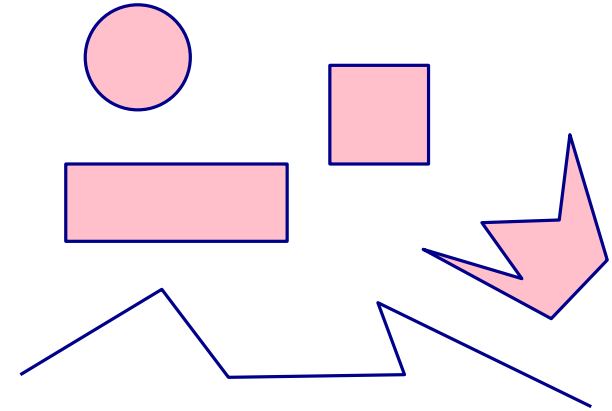
The coordinate system: x goes from 0 to 399 left-to-right, y from 0 to 299 top-to-bottom.





To create a drawing, we **add** drawable objects to the canvas:

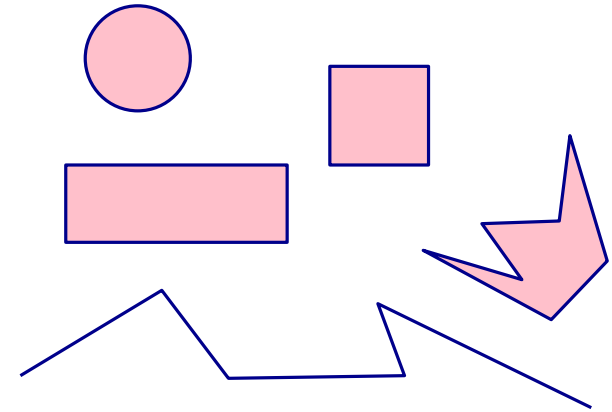
1. `Circle(radius)`
2. `Square(side)`
3. `Rectangle(width, height)`
4. `Polygon`
5. `Path`
6. `Text(message, font_size)`
7. `Image(image_filename)`





To create a drawing, we **add** drawable objects to the canvas:

1. `Circle(radius)`
2. `Square(side)`
3. `Rectangle(width, height)`
4. `Polygon`
5. `Path`
6. `Text(message, font_size)`
7. `Image(image_filename)`



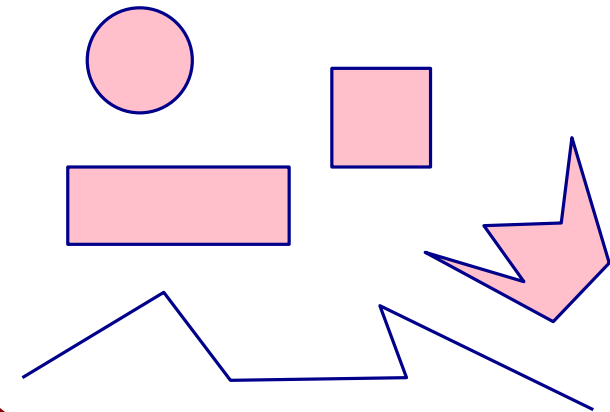
Border color (color is a string or an (r, g, b) -tuple):

```
obj.setBorderColor(color)  
obj.getBorderColor()
```




To create a drawing, we **add** drawable objects to the canvas:

1. `Circle(radius)`
2. `Square(side)`
3. `Rectangle(width, height)`
4. `Polygon`
5. `Path`
6. `Text(message, font_size)`
7. `Image(image_filename)`



fillable objects

Border color (color is a string or an (r, g, b) -tuple):

```
obj.setBorderColor(color)  
obj.getBorderColor()
```

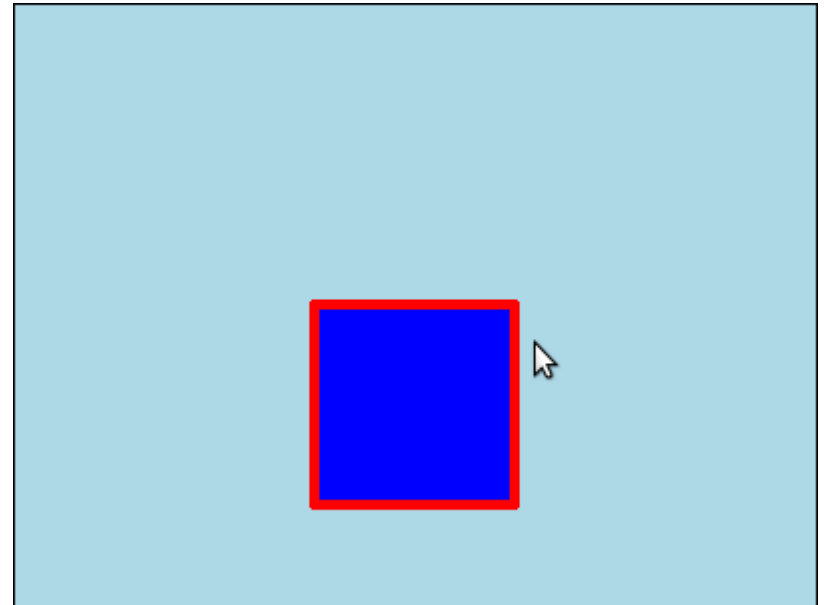
Fill color (color is a string or an (r, g, b) -tuple):

```
obj.setFillColor(color)  
obj.getFillColor()
```



Every object has a reference point. The location of the reference point on the canvas is set using `move(dx, dy)` and `moveTo(x, y)`.

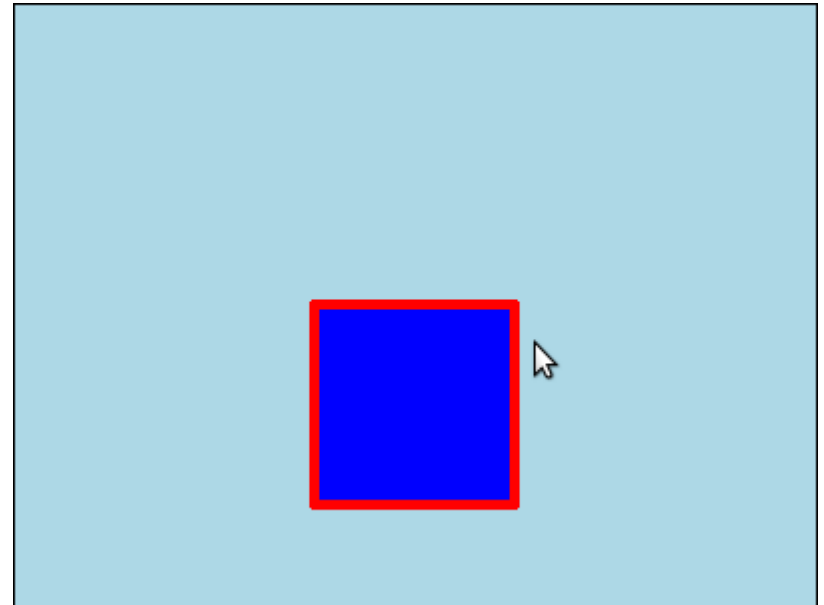
```
sq = Square(100)
canvas.add(sq)
sq.setFillColor("blue")
sq.setBorderColor("red")
sq.setBorderWidth(5)
sq.moveTo(200, 200)
```





Every object has a reference point. The location of the reference point on the canvas is set using `move(dx, dy)` and `moveTo(x, y)`.

```
sq = Square(100)
canvas.add(sq)
sq.setFillColor("blue")
sq.setBorderColor("red")
sq.setBorderWidth(5)
sq.moveTo(200, 200)
```



Animation:

```
for i in range(100):
    sq.move(1, 0)
```

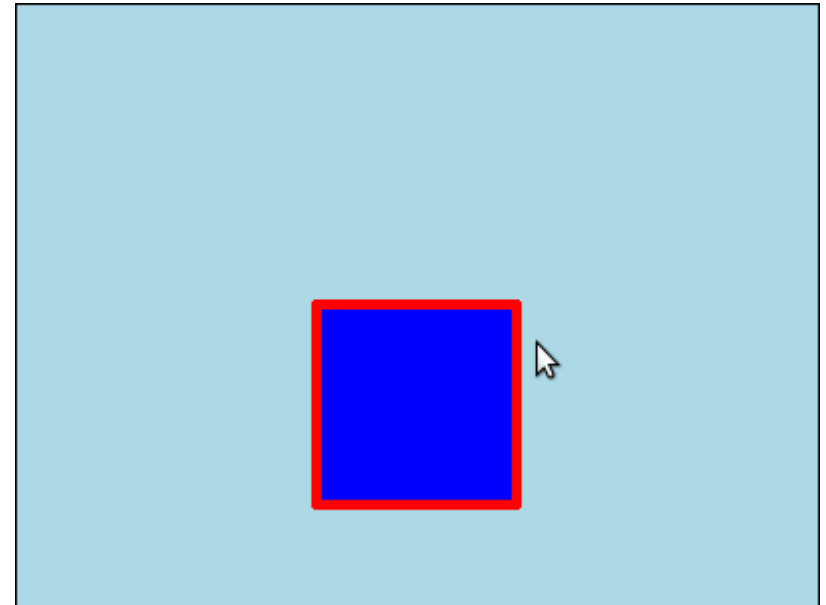


Every object has a reference point. The location of the reference point on the canvas is set using `move(dx, dy)` and `moveTo(x, y)`.

```
sq = Square(100)
canvas.add(sq)
sq.setFillColor("blue")
sq.setBorderColor("red")
sq.setBorderWidth(5)
sq.moveTo(200, 200)
```

Animation:

```
for i in range(100):
    sq.move(1, 0)
```



absolute coordinates

relative coordinates



```
def animate_sunrise(sun):  
    w = canvas.getWidth()  
    h = canvas.getHeight()  
    r = sun.getRadius()  
    x0 = w / 2.0  
    y0 = h + r  
    xradius = w / 2.0 - r  
    yradius = h  
    for angle in range(181):  
        rad = (angle/180.0) * math.pi  
        x = x0 - xradius * math.cos(rad)  
        y = y0 - yradius * math.sin(rad)  
        sun.moveTo(x, y)
```



```
def interpolate_colors(t, color1, color2):
    """Interpolate between color1 (for t == 0.0)
    and color2 (for t == 1.0)."""
    r1, g1, b1 = color1
    r2, g2, b2 = color2
    return (int((1-t) * r1 + t * r2),
            int((1-t) * g1 + t * g2),
            int((1-t) * b1 + t * b2))

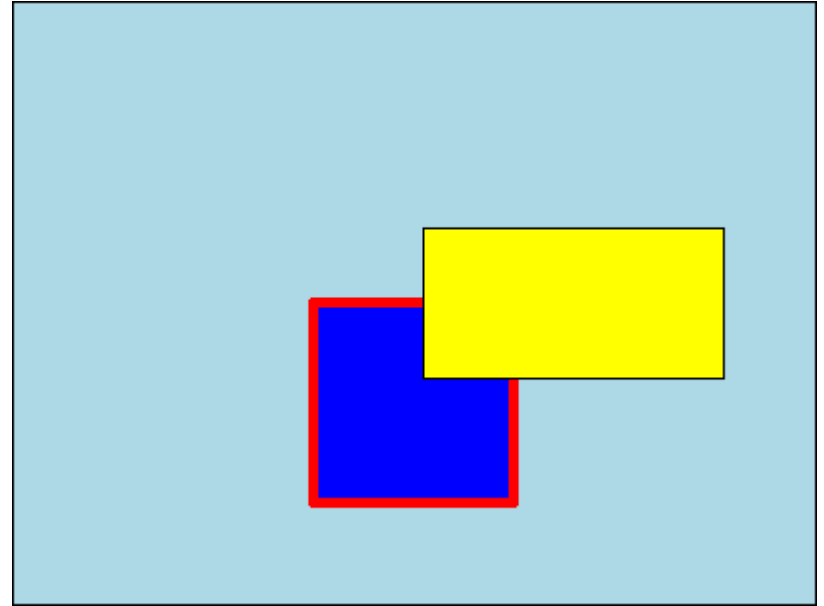
def color_value(color):
    """Convert a color name to an (r,g,b) tuple."""
    return Color(color).getColorValue()
```



```
def animate_sunrise(sun, morning_sun, noon_sun,
                    morning_sky, noon_sky):
    morning_color = color_value(morning_sun)
    noon_color = color_value(noon_sun)
    dark_sky = color_value(morning_sky)
    bright_sky = color_value(noon_sky)
    w = canvas.getWidth()
    # as before ...
    for angle in range(181):
        rad = (angle/180.0) * math.pi
        t = math.sin(rad)
        col = interpolate_colors(t, morning_color, noon_color)
        sun.setFillColor(col)
        col = interpolate_colors(t, dark_sky, bright_sky)
        canvas.setBackgroundColor(col)
        x = x0 - xradius * math.cos(rad)
        y = y0 - yradius * math.sin(rad)
        sun.moveTo(x, y)
```



```
r = Rectangle(150, 75)
canvas.add(r)
r.setFillColor("yellow")
r.moveTo(280, 150)
```



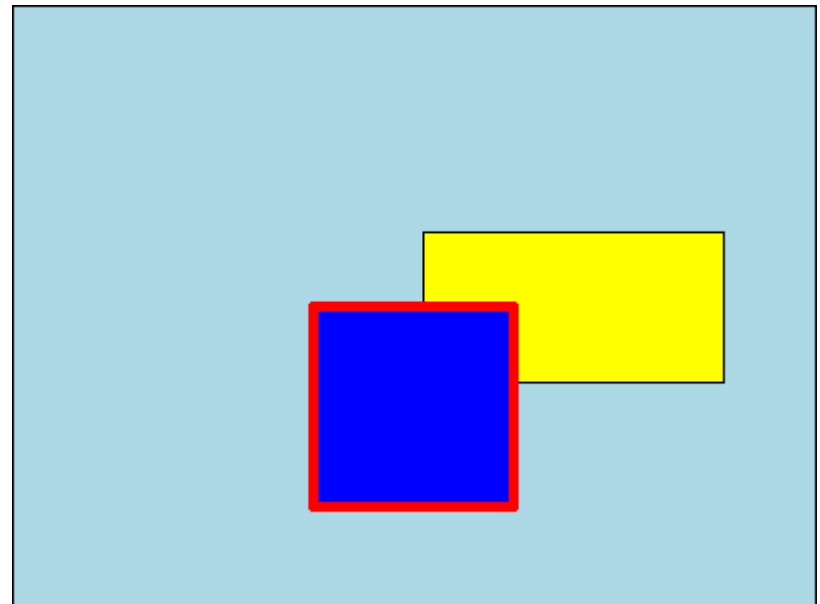
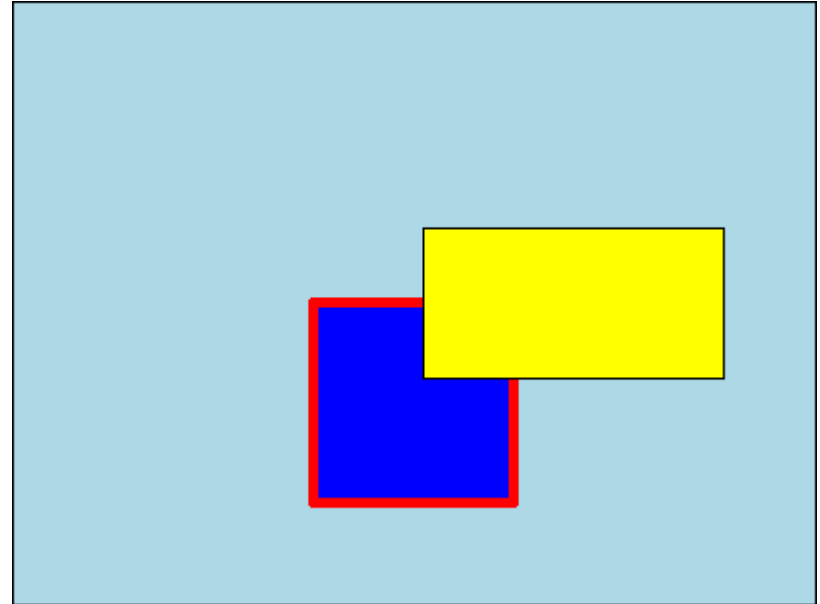


```
r = Rectangle(150, 75)
canvas.add(r)
r.setFillColor("yellow")
r.moveTo(280, 150)
```

Changing the depth:

```
sq.setDepth(10)
r.setDepth(20)
```

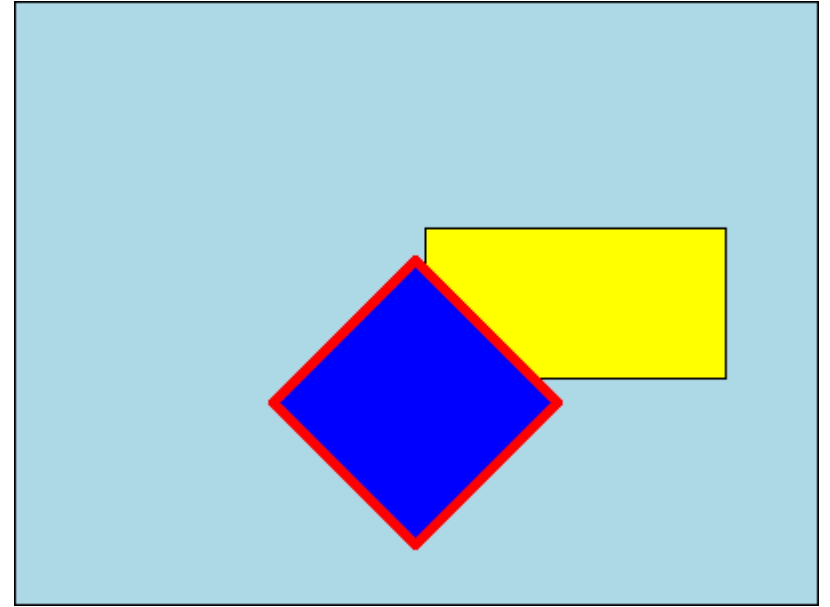
Objects with smaller depth
appear in foreground.

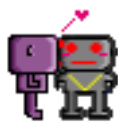




We can rotate an object around its reference point:

```
sq.rotate(45)
```





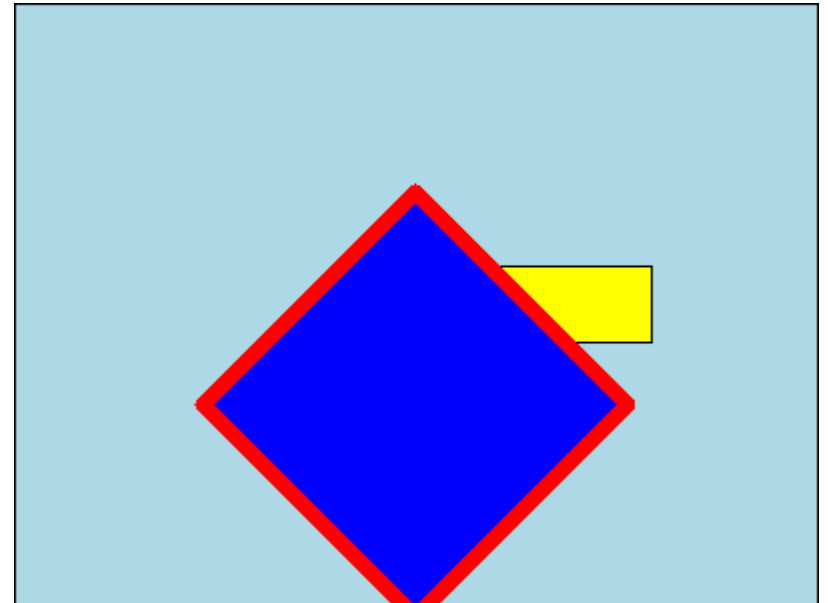
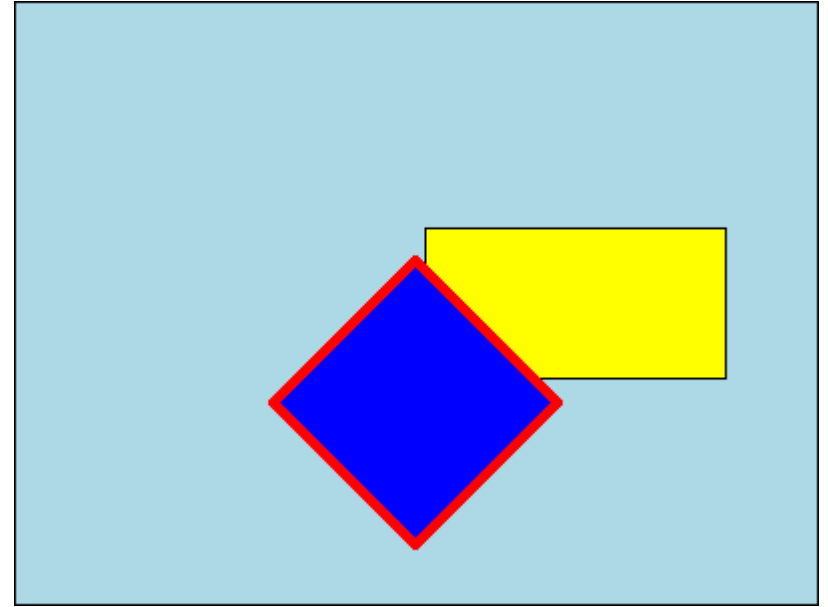
We can rotate an object around its reference point:

```
sq.rotate(45)
```

Scaling makes an object smaller or larger:

```
sq.scale(1.5)
```

```
r.scale(0.5)
```





We can rotate an object around its reference point:

```
sq.rotate(45)
```

Scaling makes an object smaller or larger:

```
sq.scale(1.5)
```

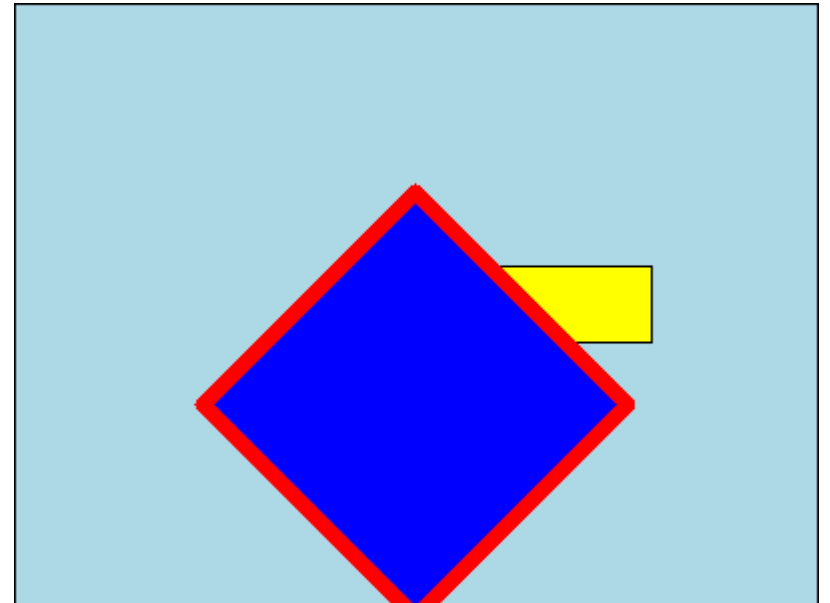
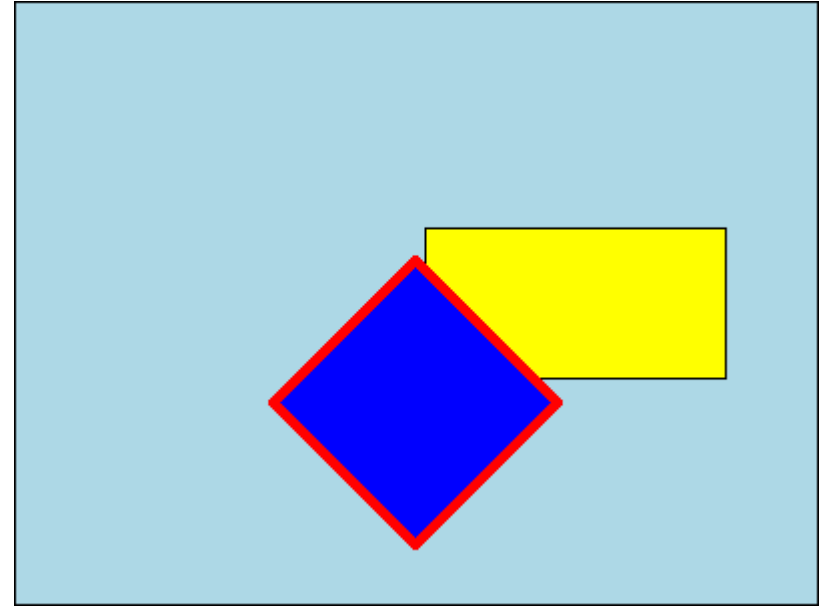
```
r.scale(0.5)
```

Fade-out:

```
for i in range(80):
```

```
    sq.scale(0.95)
```

```
canvas.remove(sq)
```





We can rotate an object around its reference point:

```
sq.rotate(45)
```

Scaling makes an object smaller or larger:

```
sq.scale(1.5)
```

```
r.scale(0.5)
```

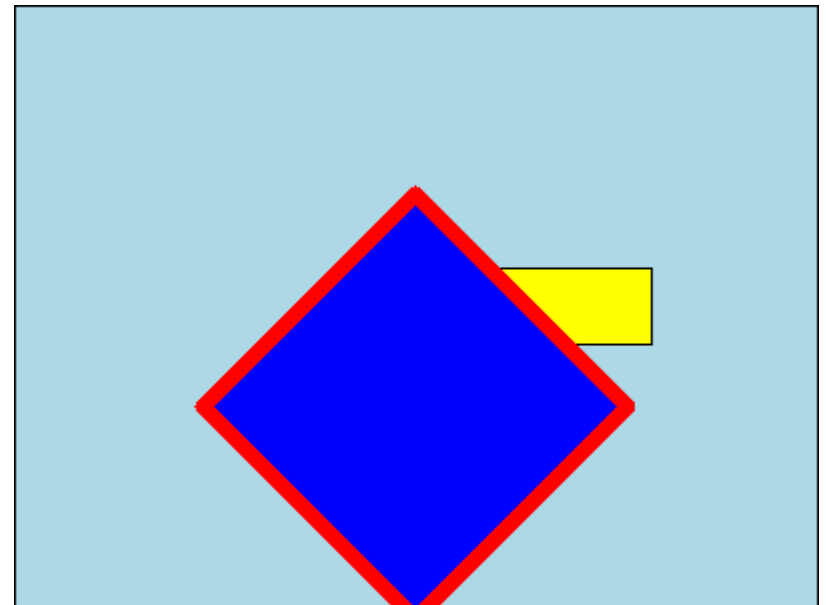
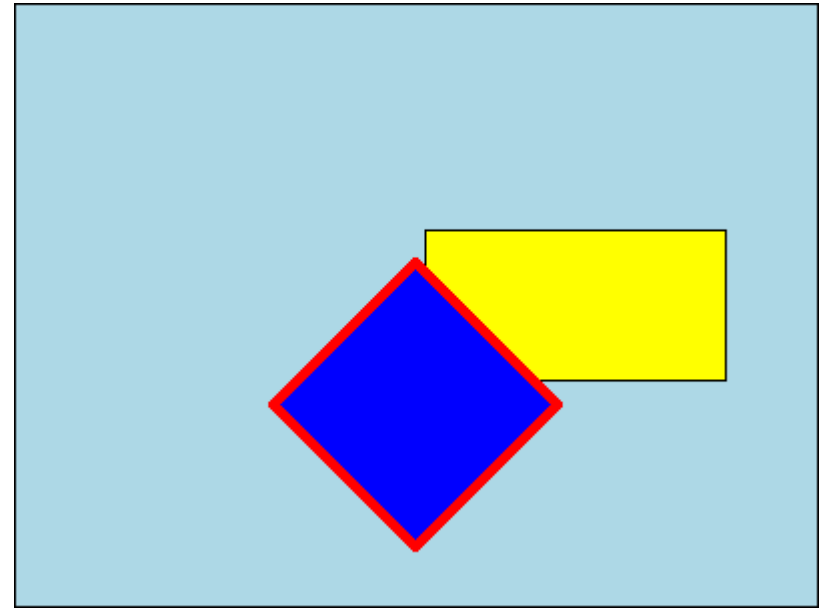
Fade-out:

```
for i in range(80):
```

```
    sq.scale(0.95)
```

```
canvas.remove(sq)
```

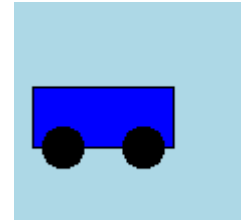
Flipping mirrors around an axis.





A layer groups together several graphics object so that they can be moved and transformed as a whole:

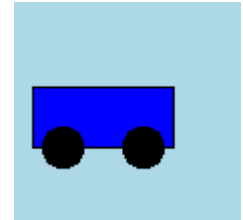
```
car = Layer()
tire1 = Circle(10, Point(-20,-10))
tire1.setFillColor('black')
car.add(tire1)
tire2 = Circle(10, Point(20,-10))
tire2.setFillColor('black')
car.add(tire2)
body = Rectangle(70, 30, Point(0, -25))
body.setFillColor('blue')
body.setDepth(60)
car.add(body)
```





A layer groups together several graphics object so that they can be moved and transformed as a whole:

```
car = Layer()
tire1 = Circle(10, Point(-20,-10))
tire1.setFillColor('black')
car.add(tire1)
tire2 = Circle(10, Point(20,-10))
tire2.setFillColor('black')
car.add(tire2)
body = Rectangle(70, 30, Point(0, -25))
body.setFillColor('blue')
body.setDepth(60)
car.add(body)
```



Animate car:

```
for i in range(250):
    car.move(2, 0)
```



The whole layer can be transformed as a single object:

```
for i in range(50):  
    car.move(2, 0)  
for i in range(22):  
    car.rotate(-1)  
for i in range(50):  
    car.move(2,-1)  
for i in range(22):  
    car.rotate(1)  
for i in range(50):  
    car.move(2,0)  
for i in range(10):  
    car.scale(1.05)  
car.flip(90)
```




Objects: state and actions

We have met some interesting types of objects: tuples, strings, robots, photos, and graphic objects like circles and squares.



Objects: state and actions

We have met some interesting types of objects: tuples, strings, robots, photos, and graphic objects like circles and squares.

An object has **state** and can perform **actions**.



Objects: state and actions

We have met some interesting types of objects: tuples, strings, robots, photos, and graphic objects like circles and squares.

An object has **state** and can perform **actions**.

Robot: The robot's state includes its position, orientation, and number of beepers carried.

It supports actions to move, turn, drop and pick beepers, and to test various conditions.



Objects: state and actions

We have met some interesting types of objects: tuples, strings, robots, photos, and graphic objects like circles and squares.

An object has **state** and can perform **actions**.

Robot: The robot's state includes its position, orientation, and number of beepers carried.

It supports actions to move, turn, drop and pick beepers, and to test various conditions.

Circle: Its state consists of its radius, position, depth, border and fill color.

It supports various actions to change its color, size, and position, and to perform transformations.



Objects: state and actions

We have met some interesting types of objects: tuples, strings, robots, photos, and graphic objects like circles and squares.

An object has **state** and can perform **actions**.

Robot: The robot's state includes its position, orientation, and number of beepers carried.

It supports actions to move, turn, drop and pick beepers, and to test various conditions.

Circle: Its state consists of its radius, position, depth, border and fill color.

It supports various actions to change its color, size, and position, and to perform transformations.

Picture: Its state consists of the photo's width and height, and a color value for every pixel.

It supports actions to look at or modify the color of each pixel.



Mutable and immutable objects

Objects whose state can never change are called **immutable**. In Python, string and tuple objects are immutable.



Mutable and immutable objects

Objects whose state can never change are called **immutable**. In Python, string and tuple objects are immutable.

Objects whose state can change are called **mutable**. Robots, photos, and graphics objects are mutable.



Mutable and immutable objects

Objects whose state can never change are called **immutable**. In Python, string and tuple objects are immutable.

Objects whose state can change are called **mutable**. Robots, photos, and graphics objects are mutable.

Remember that we can have more than one name for the same object. Be careful if this is a mutable object!

```
sun = Circle(30)
sun.setFillColor("dark orange")
moon = sun
moon.setFillColor("wheat")
print sun.setFillColor()
```




A function is an object:

```
def f(x):  
    return math.sin(x / 3.0 + math.pi/4.0)  
  
print f  
print type(f)
```



A function is an object:

```
def f(x):  
    return math.sin(x / 3.0 + math.pi/4.0)
```

```
print f           —————> <function f at 0xb7539a3c>  
print type(f)    —————> <type 'function'>
```



A function is an object:

```
def f(x):  
    return math.sin(x / 3.0 + math.pi/4.0)
```

```
print f           —————> <function f at 0xb7539a3c>  
print type(f)    —————> <type 'function'>
```

We can use a function as an argument:

```
def print_table(func, x0, x1, step):  
    x = x0  
    while x <= x1:  
        print x, func(x)  
        x += step
```

```
print_table(f, -math.pi, 3 * math.pi, math.pi/8)
```