

---

CS101, Spring 2015

# Introduction to Programming

## Lecture #1

---

## CS101:

- 11 lectures and 10 lab sections; 5 lecturers
- 6 lead TAs; 22 helper TAs

All sections have one 2-hour lecture per week (Tuesday 10:30AM–12:30PM).

Each section has one 3-hour lab per week.

This is **the most important part of the course!**

- Sections A and B: Monday (9AM) - Sue-Bok Moon
- Sections C and D: Tuesday (1PM) - Moonzoo Kim
- Sections E and F: Wednesday (9AM) - Ho-Jin Choi
- Sections G and H: Thursday (1PM) - Min H. Kim
- Sections I and J: Friday (9AM) - Sung-Eui Yoon

You must regularly check the course announcements on

<http://cs101.kaist.ac.kr>



## Practice points:

- 100 points for lecture attendance;
- 100 points for lab work;
- 200 points for homework.

Students need to collect at least **320** practice points.

## Theory points:

- 100 points for midterm exam;
- 100 points for final exam.

The grade is determined **by the theory points only**.

The practice points over 320 are **ONLY** qualification for grading.



### Extra homework:

- You can choose to work on an extra homework if your practice points are less than 320;
- Maximum points that you can obtain by doing the extra homework is **50**.

### Penalty of taking an extra homework:

- The final grade will be downgraded by 1 letter grade;
- For example, although you receive the maximum score for both exams, you will get B+ rather than A+ as the final grade.

## Only by pass exam

- No credit transfer possible from other universities

## Pass Exam

- Registration: 2015.3.2.– 3.4.
- When: 2015.3.9.(Monday) 8pm–11pm
- Where: Creative Learning Building Rm. # 308
- Language: Python
- Pass if above B-



## Freshmen

- No section change is allowed.

## New Registration

- Through online: Portal
- First Come First Served
- Class size is limited to 45 students.
- 2015.2.23. – 3.6.



---

No attendance check in the first lecture.

When you come to the 2nd lecture

- Pick a seat and it will be your seat for the rest of the semester



Cheating is strongly forbidden.

Cheating on homework or exams will give an F.

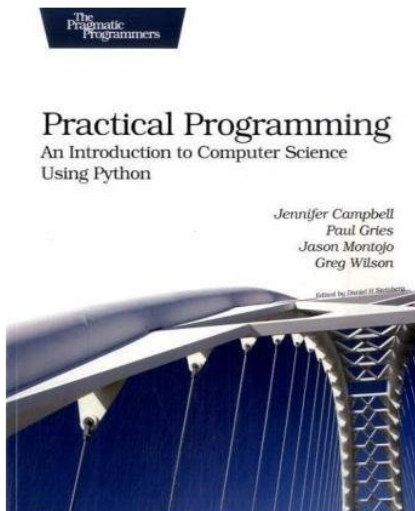




All course related materials will be made available on the course website.

- Lecture slides and sample codes
- Lecture notes on robot programming and photo processing
- Lab materials
- <http://cs101.kaist.ac.kr/>

## Main reference



Practical Programming: An Introduction to Computer Science Using Python by Jennifer Campbell, Paul Gries, Jason Montojo, and Greg Wilson. Pragmatic Programmers, 2009.



This course is about **programming** and **computational thinking**, not about learning a programming language.

Python is a programming language that is easy to learn and very powerful.

- Used in many universities for introductory courses.
- Main language used for web programming at Google.
- Widely used in scientific computation, for instance at NASA, by mathematicians and physicists.
- Available on embedded platforms, for instance Nokia mobile phones.
- Large portions of games (such as Civilization IV) are written in Python.

Once you learnt programming in one language, it is relatively easy to learn another language, such as C++ or Java.



## Why are you here?

Every scientist and engineer must know some programming. It is part of basic education, like calculus, linear algebra, introductory physics and chemistry, or English.

*Alan Perlis 1961*

Computer science is not computer programming. We teach programming to teach **computational thinking**:

- Solving problems (with a computer).
- Thinking on multiple levels of abstraction.  
Decompose a problem into smaller problems.
- A way of human thinking (**not** “thinking like a computer”)
- Thinking about recipes (**algorithms**).

30 years ago the solution to a problem in science or engineering was usually a formula. Today it is usually an algorithm (DNA, proteins, chemical reactions, factory planning, logistics).



# What is a program?

A program is a **sequence of instructions** that solves some problem (or achieves some effect).

For instance: Call your friend on the phone and give her instructions to find your favorite café. Or explain how to bake a cake.

**Instructions** are operations that the computer can already perform.

But we can define **new instructions** and raise the **level of abstraction**!

A program implements an **algorithm** (a recipe for solving a problem).



# What is debugging?

A **bug** is a mistake in a program. **Debugging** means to find the mistake and to fix it.

Computer programs are very complex systems. Debugging is similar to an experimental science: You experiment, form hypotheses, and verify them by modifying your program.

Kinds of errors:

- **Syntax error.** Python cannot understand your program, and refuses to execute it.
- **Runtime error.** When executing your program (**at runtime**), your program suddenly terminates with an error message.
- **Semantic error.** Your program runs without error messages, but does not do what it is supposed to do.



# Why is programming useful?

- 20 years ago, **electrical engineering students** learnt about circuits. Today they learn about embedded systems.
- You can build **a radio** in software.
- **Industrial engineers** program industrial robots. Moreover, today's industrial engineers work on logistics—problems that can only be solved by computer.
- **Modern automobiles** contain thousands of lines of code, and would not run without microprocessors.
- **Mathematicians** gain insight and intuition by experimenting with mathematical structures, even for discrete objects such as groups and graphs.
- **Experimental data** often needs to be reformatted to be analyzed or reused in different software. Python is fantastic for this purpose.
- **Experimental data sets** are nowadays too large to be handled manually.



## Why learn programming?

- If you can only use software that **someone else** made for you, you limit your ability to achieve what you want.
- For instance, digital media is manipulated by software. If you can only use Photoshop, you limit your ability to express yourself.
- Programming gives you freedom.



# Why is programming fun?

---

Programming is a creative process.

A single person can actually build a software system of the complexity of the space shuttle hardware. Nothing similar is true in any other discipline.

There is a large and active **open-source community**: people who write software in their free time for fun, and distribute it for free on the internet. For virtually any application there is code available that you can download and modify freely.





---

But now let me show you some Python code...

- interactive Python
- Python programs (scripts)
- comments
- your own instructions: functions
- keywords
- for loops
- indentation



A **function definition** specifies the name of a new function and the sequence of statements that execute when the function is called.

keyword                      colon

```
def print_message():  
    print "CS101 is fantastic!"  
    print "Programming is fun!"
```

indentation

You can call a function inside another function:

```
def repeat_message():  
    print_message()  
    print_message()
```



```
def print_message():  
    print "CS101 is fantastic!"  
    print "Programming is so much fun!"  
  
def repeat_message():  
    print_message()  
    print_message()  
  
repeat_message()
```

function definitions

function calls

Execution begins at the first statement. Statements are executed one-by-one, top to bottom.

Function **definitions** do not change the flow of execution—but only **define** a function.

Function **calls** are like **detours** in the flow of execution.



```
# create a robot with one beeper  
hubo = Robot(beepers = 1)
```

```
# move one step forward  
hubo.move()
```

```
# turn left 90 degrees  
hubo.turn_left()
```

object with default parameters

method: dot notation

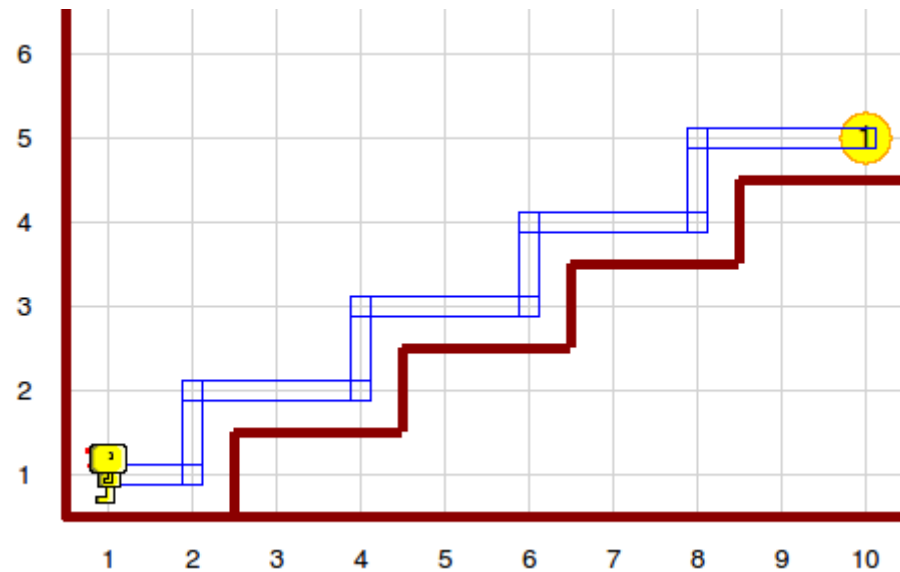
How can **hubo** turn right?

```
def turn_right():  
    hubo.turn_left()  
    hubo.turn_left()  
    hubo.turn_left()
```

Define a function!



Hubo should climb the stairs to the front door, drop a newspaper there, and return to his starting point.



Problem outline:

- Climb up four stairs
- Drop the newspaper
- Turn around
- Climb down four stairs

Python version:

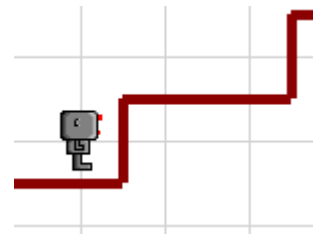
```
climb_up_four_stairs()  
hubo.drop_beeper()  
turn_around()  
climb_down_four_stairs()
```



```
def turn_around():  
    hubo.turn_left()  
    hubo.turn_left()
```

```
def climb_up_four_stairs():  
    climb_up_one_stair()  
    climb_up_one_stair()  
    climb_up_one_stair()  
    climb_up_one_stair()
```

```
def climb_up_one_stair():  
    hubo.turn_left()  
    hubo.move()  
    turn_right()  
    hubo.move()  
    hubo.move()
```





---

Start at the top of the problem, and make an outline of a solution.

For each step of this solution, either write code directly, or outline a solution for this step.

When all the partial problems have become so small that we can solve them directly, we are done and the program is finished.



To repeat the same instruction 4 times:

```
for i in range(4):  
    print "CS101 is fantastic!"
```

← for-loop

↑ Don't forget the indentation!

What is the difference:

```
for i in range(4):  
    print "CS101 is great!"  
    print "I love programming!"
```

and

```
for i in range(4):  
    print "CS101 is great!"  
print "I love programming!"
```

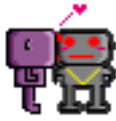




```
def climb_up_four_stairs():  
    climb_up_one_stair()  
    climb_up_one_stair()  
    climb_up_one_stair()  
    climb_up_one_stair()
```

We should avoid writing the same code repeatedly. A **for**-loop allows us to write this more elegantly:

```
def climb_up_four_stairs():  
    for i in range(4):  
        climb_up_one_stair()
```



---

CS101, Spring 2015

# Conditionals and `while` Loops

## Lecture #2



Last week we covered

- Functions and for loops

This week we will learn

- Conditionals
- if statements
- while loops



Cheating is strongly forbidden.

Cheating on homework or exams will give an **F**.



So far, our programs performed exactly the same steps every time the program is run.

Often, what the robot does must depend on the environment:

```
if it rains:
    listen_to_cs101_lecture()
else:
    eat_strawberries_in_the_sun()
```

condition

if condition is true, do this

if condition is false, do that

Note the indentation!

A **condition** is something that is either **True** or **False**.



```
if True:
```

```
    print "CS101 is my favorite course"
```

```
if False:
```

```
    print "Every CS101 student will receive an A+"
```

```
if 3 < 5:
```

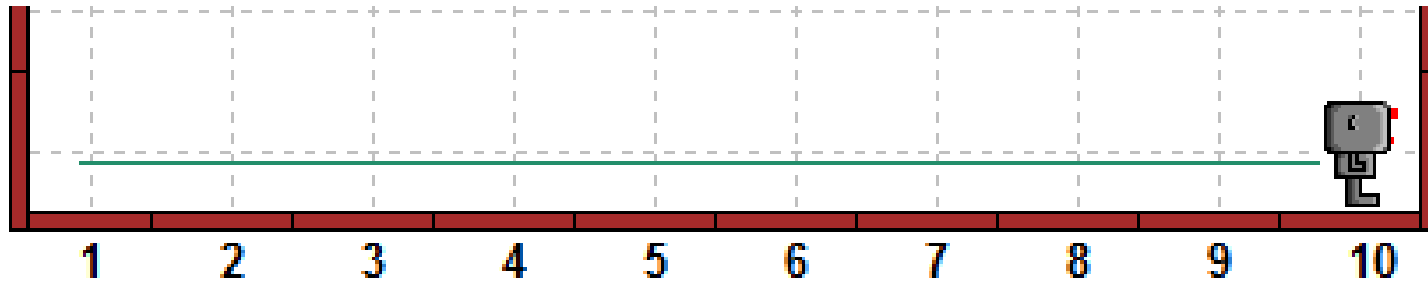
```
    print "3 is less than 5"
```

```
else:
```

```
    print "3 is larger than 5"
```



We want the robot to make 9 steps and pick up all beepers on the way.



`hubo.pick_beeper()` causes an error if there is no beeper.

Repeat the following 9 times:

- Take a step forward.
- Check if there is a beeper.
- If yes, pick it up.

```
def move_and_pick():  
    hubo.move()  
    if hubo.on_beeper():  
        hubo.pick_beeper()
```

```
for i in range(9):  
    move_and_pick()
```



---

Let's do the opposite: we want to drop a beeper, but only if there is **no** beeper at the current location.

```
if not hubo.on_beeper():  
    hubo.drop_beeper()
```

The keyword **not** inverts the sense of the condition: **not** true is false, and **not** false is true.

What is the output?

```
print not 3 < 5
```





---

Let's try to follow the boundary of the world: We move forward if there is no wall, otherwise turn to the left.

```
def move_or_turn():  
    if hubo.front_is_clear():  
        hubo.move()  
    else:  
        hubo.turn_left()  
  
for i in range(20):  
    move_or_turn()
```



```
def dance():
    for i in range(4):
        hubo.turn_left()

def move_or_turn():
    if hubo.front_is_clear():
        dance()
        hubo.move()
    else:
        hubo.turn_left()
hubo.drop_beeper()

for i in range(18):
    move_or_turn()
```

...and now?



```
if hubo.on_beeper():
    hubo.pick_beeper()
elif hubo.front_is_clear():
    hubo.move()
elif hubo.left_is_clear():
    hubo.turn_left()
elif hubo.right_is_clear():
    turn_right()
else:
    turn_around()
```

`elif` combines `else` and `if` to express many alternatives without complicated indentation.



A **for**-loop repeats some instructions a fixed number of times.

A **while**-loop repeats instructions as long as some condition is true.

Go forward until we reach a beeper:

```
while not hubo.on_beeper():  
    hubo.move()
```



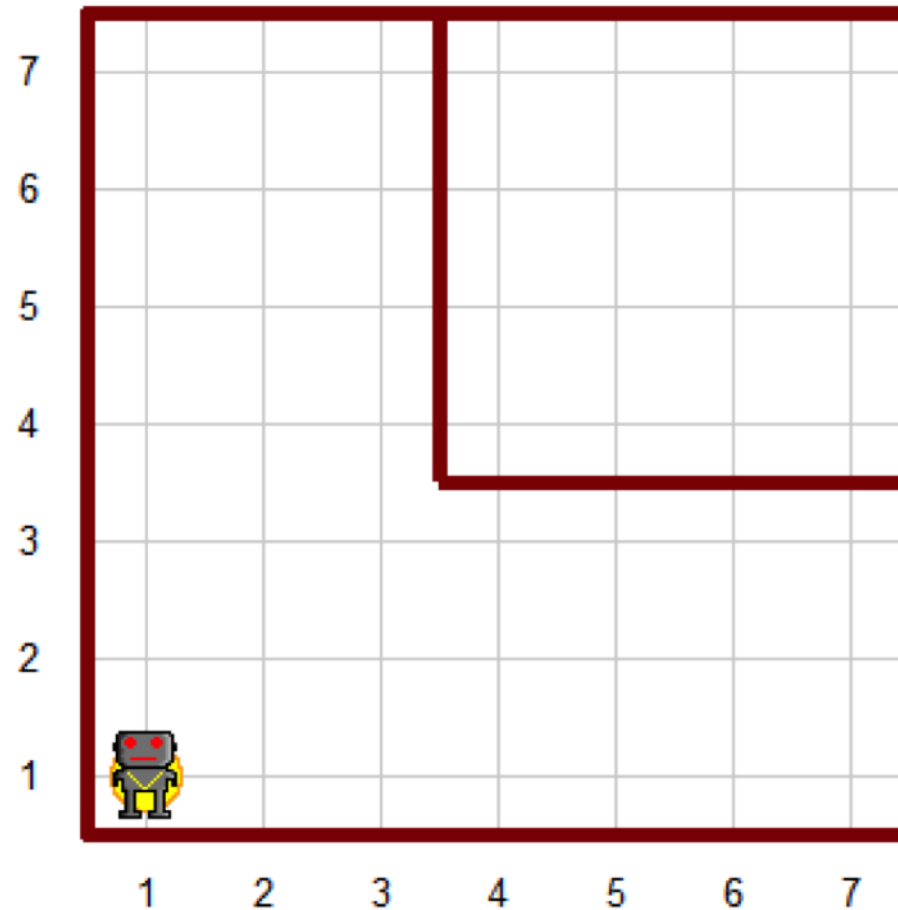
Let's write a program to let the robot walk around the boundary of the world until he comes back to the starting point.

Solution outline:

1. Put down a beeper to mark starting point
2. Move forward until facing wall
3. Turn left
4. Repeat steps 2 and 3 until we find the beeper
5. Finish when we found the beeper

```
hubo.drop_beeper()  
while not hubo.on_beeper():  
    if hubo.front_is_clear():  
        hubo.move()  
    else:  
        hubo.turn_left()
```

A red arrow points from the `hubo.move()` call in the `while` loop back to the `hubo.drop_beeper()` call, indicating a loop that continues until the robot finds the beeper it dropped.



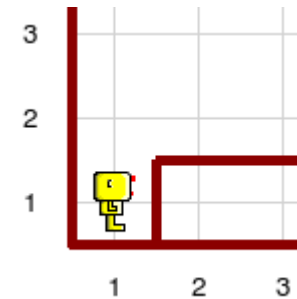
Try the code in the previous page with "amazing2.wld" and see if the previous code works.



# Sometimes we need right turns

```
hubo.drop_beeper()
hubo.move()
while not hubo.on_beeper():
    if hubo.right_is_clear():
        turn_right()
    elif hubo.front_is_clear():
        hubo.move()
    else:
        hubo.turn_left()
```

*hubo.move()*

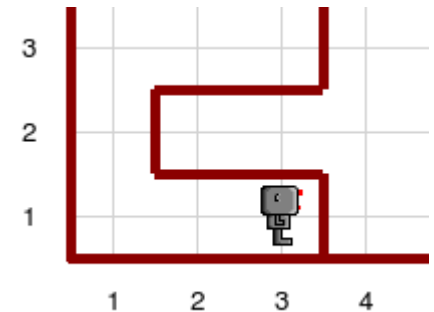


This can go into an **infinite loop**!

Still does not work when there is a wall in front of the starting position!



```
hubo.drop_beeper()
while not hubo.front_is_clear():
    hubo.turn_left()
hubo.move()
while not hubo.on_beeper():
    if hubo.right_is_clear():
        turn_right()
        hubo.move()
    elif hubo.front_is_clear():
        hubo.move()
    else:
        hubo.turn_left()
```



Still has a problem if not starting at position (1,1).





One of the secrets of writing good, correct, elegant programs is to write them as if you wrote them for a **human** reader, not a computer. Let's clean up our program:

```
# This program lets the robot go around his world counter-  
# clockwise, stopping when he returns to the starting point.  
from cs1robots import *  
load_world()  
hubo = Robot(b beepers = 1)  
  
def turn_right():  
    for i in range(3):  
        hubo.turn_left()  
  
def mark_starting_point_and_move():  
    hubo.drop_beeper()  
    while not hubo.front_is_clear():  
        hubo.turn_left()  
    hubo.move()
```

module import



```
def follow_right_wall():
    if hubo.right_is_clear():
        # Keep to the right
        turn_right()
        hubo.move()
    elif hubo.front_is_clear():
        # move following the right wall
        hubo.move()
    else:
        # follow the wall
        hubo.turn_left()

# end of definitions, begin solution

mark_starting_point_and_move()

while not hubo.on_beeper():
    follow_right_wall()
```



---

Steps to follow when writing a program:

- start simple;
- introduce small changes, one at a time;
- **make sure** that each change does not invalidate the work you have done before;
- add appropriate comments (not just repeating what the instruction does);
- choose descriptive names.



---

CS101, Spring 2015

# Variables and Basic Data Types

Lecture #3



Last week we covered

- Conditionals and `while` loops

This week we will learn

- Objects
- Types
- Variables
- Methods
- Tuples



Programs work with data. Every piece of data in a Python program is called an **object**.

Objects can be very small (the number **3**) or very large (a digital photograph).

Every object has a **type**. The type determines what you can do with an object.

The **Python Zoo**:

Imagine there is a zoo inside your Python interpreter.

Every time you create an object, an animal is born.

What an animal can do depends on the type (kind) of animal: birds can fly, fish can swim, elephants can lift weights, etc.

When an animal is no longer used, it dies (disappears).



You can create objects as follows:

**Numbers:** Simply write them:

13

3.14159265

-5

3 + 6j      ← complex number

**Strings:** (a piece of text)

Write text between quotation marks (" and ' are both okay):

"CS101 is wonderful"

'The instructor said: "Well done!" and smiled'

**Booleans:** (truth values)

Write True or False.



Complicated objects are made by calling functions that create them:

```
from cs1robots import *  
Robot()
```

```
from cs1media import *  
load_picture("photos/geowi.jpg")
```

A **tuple** object is an object that contains other objects.

To create a tuple, write objects separated by commas (usually in parenthesis):

```
(3, 2.5, 7)
```

```
("red", "yellow", "green")
```

```
(20100001, "Hong Gildong")
```





Every object has a **type**. The type determines what the object can do, and what you can do with the object. For instance, you can add two numbers, but you cannot add two robots.

The Python interpreter can tell you the type of an object:

```
>>> type(3)
```

```
<type 'int'>
```

Integer number: **int**

```
>>> type(3.1415)
```

```
<type 'float'>
```

Floating point number: **float**

```
>>> type("CS101 is fantastic")
```

```
<type 'str'>
```

String: **str**

```
>>> type(3 + 7j)
```

```
<type 'complex'>
```

Complex number: **complex**

```
>>> type(True)
```

```
<type 'bool'>
```

Boolean: **bool**



Types of more complicated objects:

```
>>> type(Robot())  
<class 'cs1robots.Robot'>  
>>> type( (3, -1.5, 7) )  
<type 'tuple'>  
>>> type( load_picture("geowi.jpg") )  
<class 'cs1media.Picture'>
```

Some object types are built into the Python language:

```
<type 'xxx'>
```

Other object types are defined by Python modules:

```
<class 'xxx'>
```

Objects can be given a **name**:

```
message = "CS101 is fantastic"  
n = 17  
hubo = Robot()  
pi = 3.1415926535897931  
finished = True  
img = load_picture("geowi.jpg")
```

We call a statement like **n = 17** an **assignment**, because the **name n** is **assigned** to the object **17**.

In the Python zoo, the name is a sign board on the animal's cage.





The rules for variable and function names:

- A name consists of letters, digits, and the underscore `_`.
- The first character of a name should not be a digit.
- The name should not be a keyword such as `def`, `if`, `else`, or `while`.
- Upper case and lower case are different: `Pi` is not the same as `pi`.

Good:

```
my_message = "CS101 is fantastic"  
a13 = 13.0
```

Bad:

```
more@ = "illegal character"  
13a = 13.0  
def = "Definition 1"
```



Names are often called **variables**, because the meaning of a name is variable: the same name can be assigned to different objects during a program:

```
n = 17
```

```
n = "Seventeen"
```

```
n = 17.0
```

In the Python zoo, this means that the sign board is moved from one animal to a different animal.

The object assigned to a name is called the **value** of the variable. The value can change over time.

To indicate that a variable is **empty**, we use the special object **None** (of type **NoneType**):

```
n = None
```



What objects can do depends on the type of object: a bird can fly, a fish can swim.

Objects provide **methods** to perform these actions.

The methods of an object are used through **dot-syntax**:

```
>>> hubo = Robot()
```

```
>>> hubo.move()
```

```
>>> hubo.turn_left()
```

```
>>> img = load_picture()
```

```
>>> print img.size()
```

```
(58, 50)
```

← width and height in pixels

```
>>> img.show()
```

← display the image

```
>>> b = "banana"
```

```
>>> print b.upper()
```

```
BANANA
```



For numbers, we use the operators  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $//$ ,  $\%$ , and  $**$ .

```
>>> 2**16
```

```
65536
```

$$a ** b = a^b$$

```
>>> 7 % 3
```

```
1
```

Remainder after division

$//$  is integer division (division without fractional part):

```
>>> 13.0 // 4.0
```

```
3.0
```

**Warning:** In Python 2, the division operator  $/$  works like  $//$  if both objects are `int` objects. This has been fixed in Python 3:

```
>>> 9 / 7
```

```
1
```

```
>>> from __future__ import division
```

```
>>> 9 / 7
```

```
1.2857142857142858
```



An **expression** is a combination of objects, variables, operators, and function calls:

`3.0 * (2 ** 15 - 12 / 4) + 4 ** 3`

The operators have precedence as in mathematics:

1. exponentiation **\*\***
2. multiplication and division **\***, **/**, **//**, **%**
3. addition and subtraction **+**, **-**

When in doubt, use parentheses!

$\frac{a}{2\pi}$  is **not** `a/2*pi`.

Use `a/(2*pi)` or `a/2/pi`.

All operators also work for complex numbers.





The operators `+` and `*` can be used for strings:

```
>>> "Hello" + "CS101"
```

```
'HelloCS101'
```

```
>>> "CS101 " * 8
```

```
'CS101 CS101 CS101 CS101 CS101 CS101 CS101 CS101 '
```



A **boolean expression** is an expression whose value has type **bool**. They are used in **if** and **while** statements.

The operators **==**, **!=**, **>**, **<**, **<=**, and **>=** return boolean values.

```
>>> 3 < 5
```

```
True
```

```
>>> 27 == 14
```

```
False
```

```
>>> 3.14 != 3.14
```

```
False
```

```
>>> 3.14 >= 3.14
```

```
True
```

```
>>> "Cheong" < "Choe"
```

```
True
```

```
>>> "3" == 3
```

```
False
```

Equality—don't confuse with **=**



The keywords **not**, **and**, and **or** are logical operators:

```
not True == False
```

```
not False == True
```

```
False and False == False
```

```
False and True == False
```

```
True and False == False
```

```
True and True == True
```

```
False or False == False
```

```
False or True == True
```

```
True or False == True
```

```
True or True == True
```

**Careful:** if the second operand is not needed, Python does not even compute its value.



A tuple is an object that contains other objects:

```
position = (3.14, -5, 7.5)
```

```
profs = ("Key-Sun Choi", "Gee-Hyuk Lee",  
         "In-Young Ko", "Min-Hyuk Kim", "John Kim")
```

A tuple is a single object of type `tuple`:

```
>>> print position, type(position)  
(3.14, -5, 7.5) <type 'tuple'>
```

We can “unpack” tuples:

```
x, y, z = position
```

Packing and unpacking in one line:

```
a, b = b, a
```



Colors are often represented as a tuple with three elements that specify the intensity of red, green, and blue light:

```
red = (255, 0, 0)
```

```
blue = (0, 0, 255)
```

```
white = (255, 255, 255)
```

```
black = (0, 0, 0)
```

```
yellow = (255, 255, 0)
```

```
purple = (128, 0, 128)
```

```
from cs1media import *
```

```
img = create_picture(100, 100, purple)
```

```
img.show()
```

```
img.set_pixels(yellow)
```

```
img.show()
```



A digital image of width  $w$  and height  $h$  is a rectangular matrix with  $h$  rows and  $w$  columns:

0, 0	1, 0	2, 0	3, 0	4, 0
0, 1	1, 1	2, 1	3, 1	4, 1
0, 2	1, 2	2, 2	3, 2	4, 2

We access pixels using their  $x$  and  $y$  coordinates.  
 $x$  is between 0 and  $w-1$ ,  $y$  is between 0 and  $h-1$ .

```
>>> img.get(250, 188)
(101, 104, 51)
```

red, green, blue triple

```
>>> img.set(250, 188, (255, 0, 0))
```



A for-loop assigns integer values to a variable:

```
for i in range(4):
```

```
    print i
```

prints 0, 1, 2, 3.

```
>>> for i in range(7):
```

```
>>>     print "*" * i
```

```
*
```

```
**
```

```
***
```

```
****
```

```
*****
```

```
*****
```



```
from cs1media import *  
  
img = load_picture("../photos/geowi.jpg")  
w, h = img.size()  
for y in range(h):  
    for x in range(w):  
        r, g, b = img.get(x, y)  
        r, g, b = 255 - r, 255 - g, 255 - b  
        img.set(x, y, (r, g, b))  
img.show()
```







```
from cs1media import *
threshold = 100
white = (255, 255, 255)
black = (0, 0, 0)

img = load_picture("../photos/yuna1.jpg")
w, h = img.size()
for y in range(h):
    for x in range(w):
        r, g, b = img.get(x, y)
        v = (r + g + b) // 3          # average of r,g,b
        if v > threshold:
            img.set(x, y, white)
        else:
            img.set(x, y, black)
img.show()
```



The same object can have more than one name:

```
hubo = Robot("yellow")
```

```
hubo.move()
```

```
ami = hubo
```

```
ami.turn_left()
```

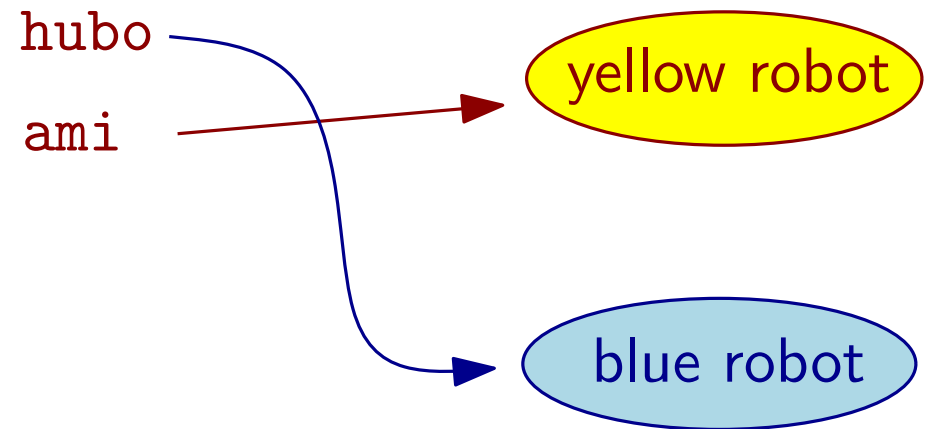
```
hubo.move()
```

```
hubo = Robot("blue")
```

```
hubo.move()
```

```
ami.turn_left()
```

```
ami.move()
```





---

CS101, Spring 2015

# Functions with Parameters and Return Values

Lecture #4



Last week we covered

- Objects and Types
- Variables
- Methods
- Tuples

This week we will learn

- Functions with parameters and return values



The name **function** comes from mathematics. A function is a mapping from one set to another set:

$$f : \mathbb{R} \rightarrow \mathbb{R}$$

$$x \mapsto \pi \times \frac{x}{180.0}$$

Here,  $x$  is the **argument** of the function,  $f(x)$  is the **result** of the function.

In Python, functions also take **arguments** and return a **result**:

```
def to_radians(deg):  
    return (deg / 180.0) * math.pi
```

```
>>> a = to_radians(90)
```

```
>>> print a
```

```
1.5707963267948966
```

Python comes with many built-in functions.

Type conversion functions convert from one type to another type:

```
>>> int("32")
```

```
32
```

```
>>> int(17.3)
```

```
17
```

```
>>> float(17)
```

```
17.0
```

```
>>> float("3.1415")
```

```
3.1415
```

```
>>> str(17) + " " + str(3.1415)
```

```
'17 3.1415'
```

```
>>> complex(17)
```

```
(17 + 0j)
```



To use math functions, we need to tell Python that we want to use the `math` module:

```
import math
degrees = 45
radians = degrees / 360.0 * 2 * math.pi
print math.sin(radians)
print math.sqrt(2) / 2
```

When using math functions often, we can use shorter names:

```
import math
sin = math.sin
pi = math.pi
radians = degrees / 360.0 * 2 * pi
print sin(radians)
```



The function definition uses **names** for the arguments of the function. These names are called **parameters**:

```
def compute_interest(amount, rate, years):
```

Inside the function, the parameter is just a name:

```
    value = amount * (1 + rate/100.0) ** years
```

When we have computed the result of the function, we **return** it from the function. The function ends at this point, and the result object is given back:

```
    return value
```

We can now call the function with different argument values:

```
>>> s1 = compute_interest(200, 7, 1)
```

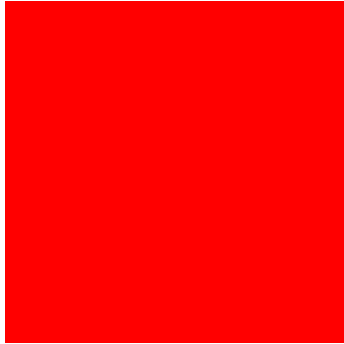
```
>>> s2 = compute_interest(500, 1, 20)
```



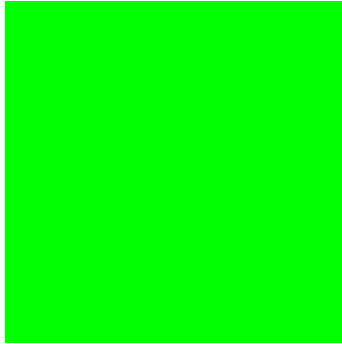


What is the light intensity (**luma**) of pixel  $(r, g, b)$ ?

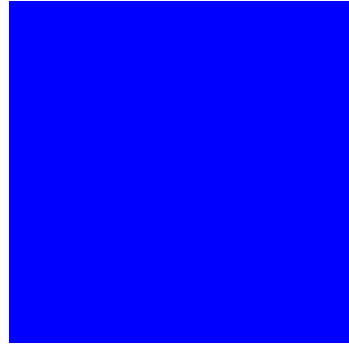
$(255, 0, 0)$



$(0, 255, 0)$



$(0, 0, 255)$



A good formula is:

```
def luma(p):  
    r, g, b = p  
    return int(0.213 * r + 0.715 * g + 0.072 * b)
```



Compute the absolute value (like builtin function `abs`):

```
def absolute(x):  
    if x < 0:  
        return -x  
    else:  
        return x
```

The same function can be written like this:

```
def absolute(x):  
    if x < 0:  
        return -x  
    return x
```

But not like this:

```
def absolute(x):  
    if x < 0:  
        return -x  
    if x > 0:  
        return x
```



A function that tests a condition and returns either **True** or **False** is often called a **predicate**:

# is integer a divisible by b?

```
def is_divisible(a, b):
```

```
    if a % b == 0:
```

```
        return True
```

```
    else:
```

```
        return False
```

Easier:

```
def is_divisible(a, b):
```

```
    return a % b == 0
```

A predicate (function) can be used directly in an **if** or **while** statement:

```
if is_divisible(x, y):
```

```
    print 'x is divisible by y'
```



We have seen many functions that do not use **return**:

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

In fact, a function that does not call **return** automatically returns **None**:

```
>>> s = turn_right()  
>>> print s  
None
```



When a function is called, the **arguments** of the function call are assigned to the **parameters**:

```
def print_twice(text):  
    print text  
    print text
```

Parameter

The number of arguments in the function call must be the same as the number of parameters.

```
>>> print_twice("I love CS101")  
I love CS101  
I love CS101  
>>> print_twice(math.pi)  
3.14159265359  
3.14159265359
```

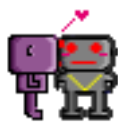


We can now write a `turn_right` function that will work for any robot, not just for Hubo:

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

```
ami = Robot("yellow")  
hubo = Robot("blue")  
turn_right(ami)  
turn_right(hubo)
```

Remember: A `parameter` is a `name` for an object. The name can only be used `inside` the function.

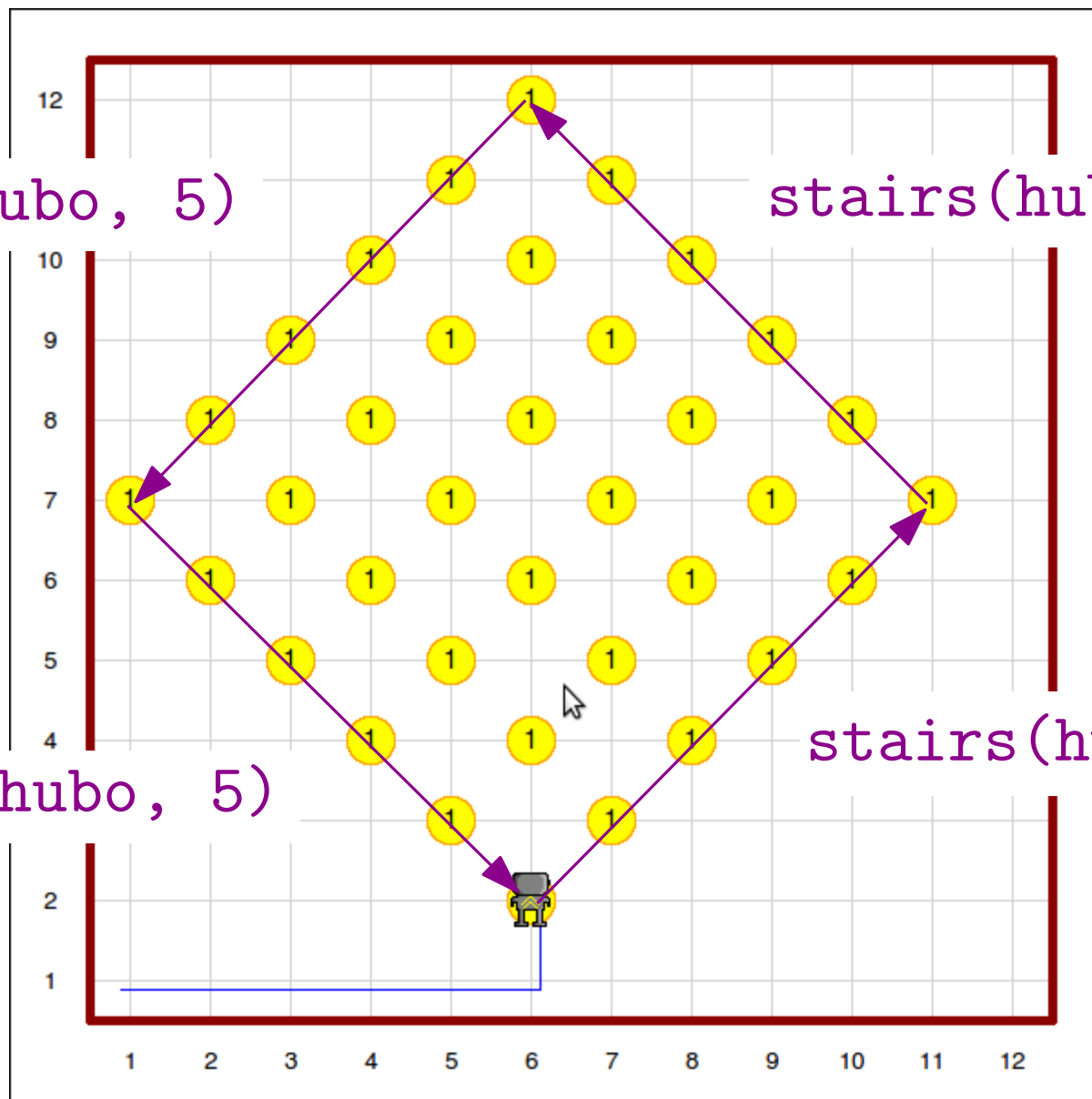


stairs(hubo, 5)

stairs(hubo, 5)

stairs(hubo, 5)

stairs(hubo, 5)



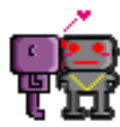


```
def stairs(robot, n):  
    for i in range(n):  
        robot.pick_beeper()  
        robot.move()  
        turn_right(robot)  
        robot.move()  
        robot.turn_left()
```

```
def diamond(robot, n):  
    for i in range(4):  
        stairs(robot, n)  
        robot.turn_left()
```

```
def harvest_all(robot):  
    for i in range(3):  
        n = 5 - 2 * i  
        diamond(robot, n)  
        robot.move()  
        robot.move()
```





```
white = (255, 255, 255)
```

```
black = (0, 0, 0)
```

```
def blackwhite(img, threshold):
```

```
    w, h = img.size()
```

```
    for y in range(h):
```

```
        for x in range(w):
```

```
            v = luma(img.get(x, y))
```

```
            if v > threshold:
```

```
                img.set(x, y, white)
```

```
            else:
```

```
                img.set(x, y, black)
```

```
pict = load_picture("../photos/yuna1.jpg")
```

```
blackwhite(pict, 100)
```

```
pict.show()
```



---

A function can only return one value.

But this value can be a tuple, and functions can return arbitrarily many values by returning them as a tuple:

```
def student():  
    name = "Hong, Gildong"  
    id = 20101234  
    return name, id
```

Often function results are unpacked immediately:

```
name, id = student()
```

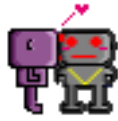


The `raw_input` function prints a message and waits for the user to enter a string on the keyboard. When the user presses the Enter key, the whole string is returned:

```
name = raw_input("What is your name? ")  
print "Welcome to CS101, " + name
```

If we need a number, we should convert the string:

```
raw_n = raw_input("Enter a positive integer> ")  
n = int(raw_n)  
for i in range(n):  
    print "*" * i
```



CS101, Spring 2015

# Local/Global Variables and Graphical Objects

Lecture #5



---

Last week we covered

- Functions with parameters and return values

This week we will learn

- Local and global variables
- Modules
- Graphics
  - Drawable objects
  - Reference points
  - Color interpolation
  - Depth
  - Transformation
- Mutability



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)
```

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

$a \rightarrow 2$

$b \rightarrow 4$

$c \rightarrow 5$

$x \rightarrow 3$

$\text{quad\_term} \rightarrow 18$

$\text{lin\_term} \rightarrow 12$



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**.

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 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
```



```
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
```



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

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)
```

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()
```

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")
```

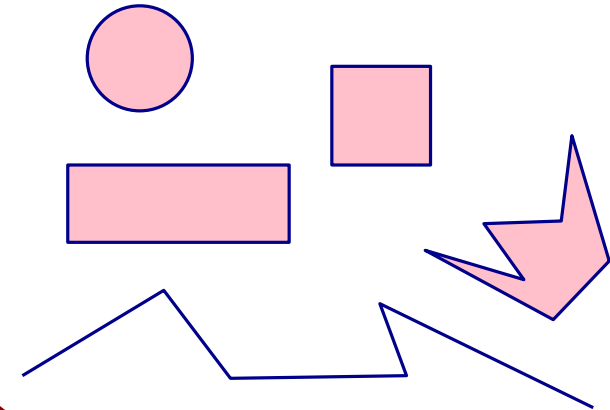
The coordinate system:  $x$  goes from 0 to 399 left-to-right,  $y$  from 0 to 299 top-to-bottom.





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)`



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()
```

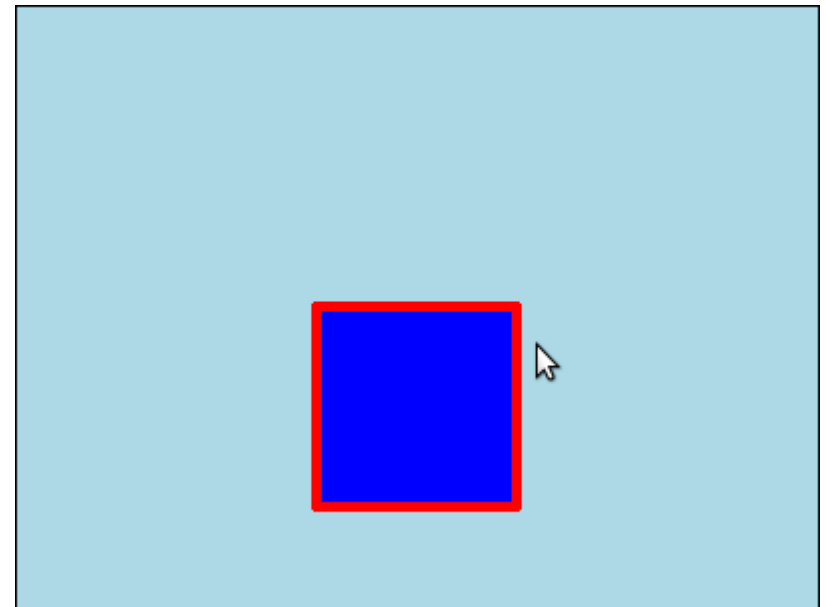


Every object has a reference point. The location of the reference point on the canvas is set using `move(dx, dy)` and `moveTo(x, y)`.

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

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)
```

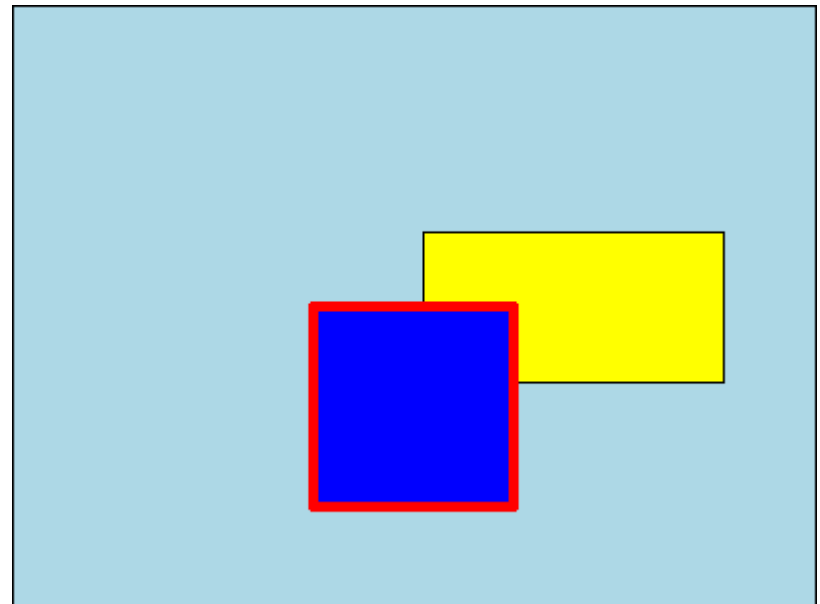
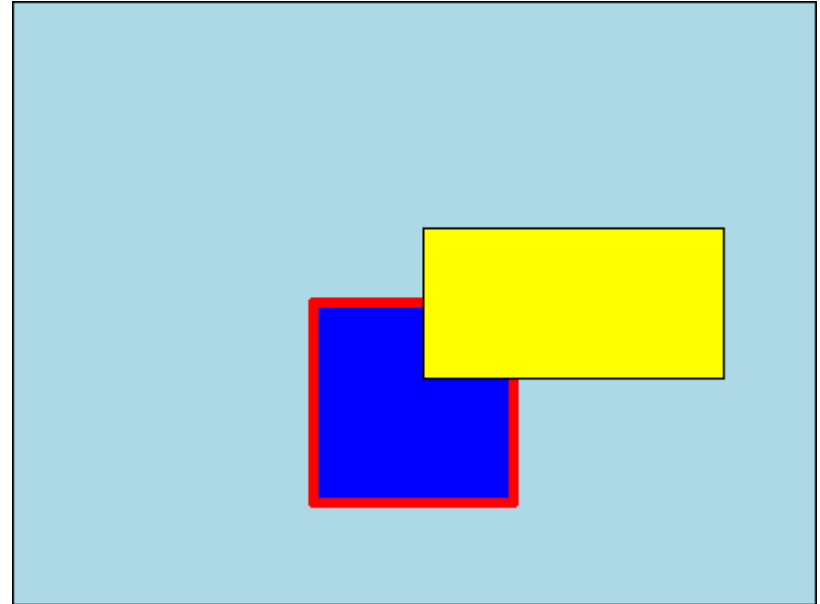


```
r = Rectangle(150, 75)
canvas.add(r)
r.setFillColor("yellow")
r.moveTo(280, 150)
```

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)
```

Scaling makes an object smaller or larger:

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

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

Fade-out:

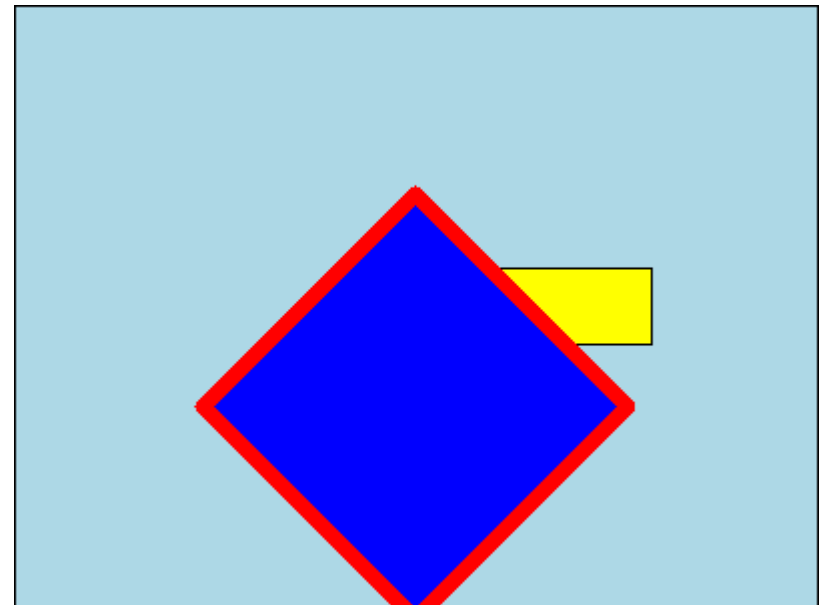
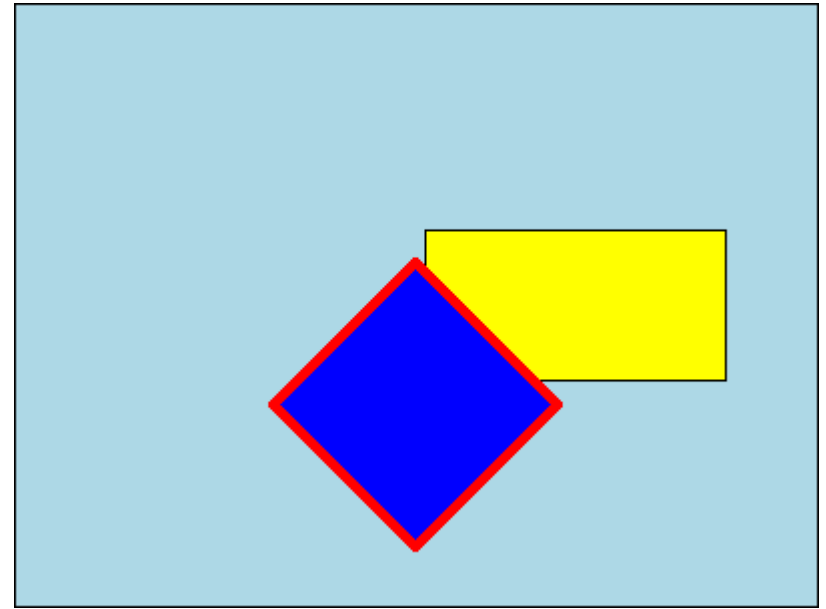
```
for i in range(80):
```

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

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

Flipping mirrors around an axis.

```
r.flip(45)
```

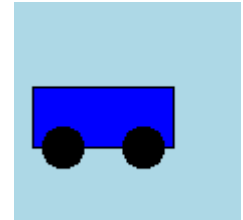






A layer groups together several graphic objects so that they can be moved and transformed as a whole:

```
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.

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.

Objects whose state can change are called **mutable**. Robots, photos, and graphic 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:

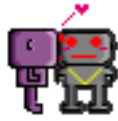
```
def f(x):  
    return math.sin(x / 3.0 + math.pi/4.0)
```

```
print f           —————> <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)
```



CS101, Spring 2015

# Sequences: Lists, Strings, and Tuples

Lecture #6

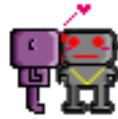


Last week we learned

- Local and global variables
- Modules
- Graphics

This week we will learn

- Sequences
  - Lists
  - Strings
  - Tuples



Here is a table of olympic medals from the 2010 Vancouver winter games:

Source: [www.vancouver2010.com](http://www.vancouver2010.com)

Australia	2	1	0
Austria	4	6	6
Belarus	1	1	1
Canada	14	7	5
China	5	2	4
Croatia	0	2	1
Czech Republic	2	0	4
Estonia	0	1	0
Finland	0	1	4
France	2	3	6
Germany	10	13	7
Great Britain	1	0	0
Italy	1	1	3
Japan	0	3	2
Kazakhstan	0	1	0
Korea	6	6	2
Latvia	0	2	0
Netherlands	4	1	3
Norway	9	8	6
Poland	1	3	2
Russian Federation	3	5	7
Slovakia	1	1	1
Slovenia	0	2	1
Sweden	5	2	4
Switzerland	6	0	3
United States	9	15	13

How can we store this much data in Python? We would need  $4 \times 26$  variables...

The solution is to store all values together in a **list**.





To create a list, enclose the values in square brackets:

```
countries = [ "Australia", ... , "United States" ]  
gold = [2, 4, 1, 14, 5, 0, 2, 0, 0, 2, 10, 1, 1, 0,  
        0, 6, 0, 4, 9, 1, 3, 1, 0, 5, 6, 9]
```

A list is an object of type `list`.

We can access the elements of a list using an integer index.  
The first element is at index `0`, the second at index `1`, and so on:

```
>>> countries[0]  
'Australia'  
>>> countries[15]  
'Korea'  
>>> gold[15]  
6
```

Negative indices start at the end of the list:

```
>>> countries[-1]  
'United States'  
>>> countries[-11]  
'Korea'
```



The length of a list is given by `len`:

```
>>> len(countries)
26
```

The empty list is written `[]` and has length zero.

Lists can contain a mixture of objects of any type:

```
>>> korea = [ 'Korea', 'KR', 6, 6, 2 ]
>>> korea[1]
'KR'
>>> korea[2]
6
```

Or even:

```
>>> korea = [ "Korea", 'KR', (6, 6, 2) ]
```



A list of noble gases:

```
>>> nobles = [ 'helium', 'none', 'argon', 'krypton',  
                'xenon' ]
```

Oops. Correct the typo:

```
>>> nobles[1] = "neon"
```

```
>>> nobles
```

```
['helium', 'neon', 'argon', 'krypton', 'xenon']
```

Oops oops. I forgot radon!

```
>>> nobles.append('radon')
```

```
>>> nobles
```

```
['helium', 'neon', 'argon', 'krypton', 'xenon', 'radon']
```



Reminder: An object can have more than one name. This is called **aliasing**. We have to be careful when working with mutable objects:

```
>>> list1 = ["A","B","C"]
>>> list2 = list1
>>> len(list1)
3
>>> list2.append("D")
>>> len(list1)
4
>>> list1[1] = "X"
>>> list2
['A', 'X', 'C', 'D']
>>> list1 is list2
True
```

```
>>> list1 = ["A","B","C"]
>>> list2 = ["A","B","C"]
>>> len(list1)
3
>>> list2.append("D")
>>> len(list1)
3
>>> list1[1] = "X"
>>> list2
['A', 'B', 'C', 'D']
>>> list1 is list2
False
```



`len` returns length of a list, `sum` the sum of the elements, `max` the largest element, `min` the smallest element:

```
>>> len(gold), sum(gold), max(gold), min(gold)
(26, 86, 14, 0)
```

```
>>> len(silver), sum(silver), max(silver)
(26, 87, 15)
```

```
>>> len(bronze), sum(bronze), max(bronze)
(26, 85, 13)
```



A **for** loop looks at every element of a list:

```
for country in countries:  
    print country
```

The **range** function returns a list:

```
>>> range(10)  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
>>> range(10, 15)  
[10, 11, 12, 13, 14]
```

If we want to modify elements, we need the index:

```
>>> l = range(1, 11)  
>>> for i in range(len(l)):  
...     l[i] = l[i] ** 2  
>>> l  
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```



Let's print out the total number of medals for each country:

```
for i in range(len(countries)):
    print countries[i], gold[i]+silver[i]+bronze[i]
```

We can create a new list:

```
totals = []
for i in range(len(countries)):
    medals = gold[i]+silver[i]+bronze[i]
    totals.append( (medals, countries[i]) )
```

The list **totals** is now a list of tuples (**medals, country**).

```
[(3, 'Australia'), (16, 'Austria'), (3, 'Belarus'), ...,
(14, 'Korea'), (2, 'Latvia'), (8, 'Netherlands'), ...,
(11, 'Sweden'), (9, 'Switzerland'), (37, 'United States')]
```



We can sort a list using its `sort` method:

```
>>> ta = [ "Jongbin", "Sungeun", "Dongyoung",  
...       "Suwon", "YeongJae", "Jinki" ]  
>>> ta.sort()  
>>> ta  
['Dongyoung', 'Jinki', 'Jongbin', 'Sungeun',  
'Suwon', 'YeongJae']
```

Let's sort the medal totals: `totals.sort()`.

```
[(1, 'Estonia'), (1, 'Great Britain'), (1, 'Kazakhstan'),  
 (2, 'Latvia'), (3, 'Australia'), (3, 'Belarus'), ...,  
 (14, 'Korea'), ..., (26, 'Canada'), (30, 'Germany'),  
 (37, 'United States')]
```



We rather want the countries with the largest number of medals at the top:

```
totals.reverse()
```

```
[(37, 'United States'), (30, 'Germany'), (26, 'Canada'),  
 (23, 'Norway'), (16, 'Austria'), ..., (14, 'Korea'),  
 (11, 'Sweden'), ... (1, 'Estonia')]
```

Actually we only care about the top 10:

```
top_ten = totals[:10]
```

```
for p in top_ten:
```

```
    medals, country = p
```

```
    print medals, country
```

Slicing

Unpack immediately

```
for medals, country in top_ten:
```

```
    print medals, country
```



Slicing creates a **new list** with elements of the given list:

```
sublist = mylist[i:j]
```

Then **sublist** contains elements  $i, i + 1, \dots, j - 1$  of **mylist**.

If **i** is omitted, the sublist starts with the first element.

If **j** is omitted, then the sublist ends with the last element.

Special case: We can create a copy of a list with

```
list2 = list1[:]
```



Let's create the top-10 lexicographical ranking:

```
table = []
for i in range(len(countries)):
    table.append( (gold[i], silver[i],
                  bronze[i], countries[i]) )
table.sort()
top_ten = table[-10:]
top_ten.reverse()
for g,s,b,country in top_ten:
    print country, g, s, b
```

Canada	14	7	5
Germany	10	13	7
United States	9	15	13
Norway	9	8	6
Korea	6	6	2
Switzerland	6	0	3
Sweden	5	2	4
China	5	2	4
Austria	4	6	6
Netherlands	4	1	3



Let's find all countries that have only one kind of medal:

```
def no_medals(countries, a1, b1):  
    result = []  
    for i in range(len(countries)):  
        if a1[i] == 0 and b1[i] == 0:  
            result.append(countries[i])  
    return result
```

```
only_gold = no_medals(countries, silver, bronze)  
only_silver = no_medals(countries, gold, bronze)  
only_bronze = no_medals(countries, gold, silver)
```

```
only_one = only_gold ⊕ only_silver ⊕ only_bronze
```

list concatenation



List objects `L` have the following methods:

- `L.append(v)` add object `v` at the end
- `L.insert(i, v)` insert element at position `i`
- `L.pop()` remove and return last element
- `L.pop(i)` remove and return element at position `i`
- `L.remove(v)` remove first element equal to `v`
- `L.index(v)` return index of first element equal to `v`
- `L.count(v)` return number of elements equal to `v`
- `L.extend(K)` append all elements of sequence `K` to `L`
- `L.reverse()` reverse the list
- `L.sort()` sort the list

What is the difference?

```
L.append(13)
```

```
L + [ 13 ]
```



Lists are a kind of **sequence**. We already met other kinds of sequences: strings, and tuples:

### Strings:

```
>>> a = "CS101"
>>> a[0]
'C'
>>> a[-1]
'1'
>>> a[2:]
'101'
>>> for i in a:
...     print i,
C S 1 0 1
```

### Tuples:

```
>>> t = ("CS101", "A+", 13)
>>> t[0]
'CS101'
>>> t[-1]
13
>>> t[1:]
('A+', 13)
>>> for i in t:
...     print i,
CS101 A+ 13
```



Lists and tuples are very similar, but lists are **mutable**, while tuples (and strings) are **immutable**:

```
>>> t[0] = "CS206"
```

```
TypeError: 'tuple' object does not support  
item assignment
```

We can convert a sequence into a list or tuple using the **list** and **tuple** functions:

```
>>> list(t)
```

```
['CS101', 'A+', 13]
```

```
>>> tuple(gold)
```

```
(2, 4, 1, 14, 5, 0, 2, 0, 0, ..., 0, 5, 6, 9)
```

```
>>> list("CS101")
```

```
['C', 'S', '1', '0', '1']
```



Using four lists to store the medal information is not typical for Python. We would normally make a single list of tuples:

```
medals = [ ( 'Australia', 2, 1, 0 ),  
            ( 'Austria', 4, 6, 6 ),  
            ...  
            ( 'United States', 9, 15, 13 ) ]
```

Print total number of medals for each country:

```
def print_totals1():  
    for country, g, s, b in medals:  
        print country + ":", g + s + b  
  
def print_totals2():  
    for item in medals:  
        print item[0] + ":", sum(item[1:])
```





Instead of creating a new list, let's sort the original list by total number of medals:

```
def compare(item1, item2):  
    medals1 = sum(item1[1:])  
    medals2 = sum(item2[1:])  
    return cmp(medals2, medals1)
```

```
def top_ten():  
    medals.sort(compare)  
    top_ten = medals[:10]  
    for item in top_ten:  
        print item[0] + ":", sum(item[1:])
```

United States: 37  
Germany: 30  
Canada: 26  
Norway: 23  
Austria: 16  
Russian Federation: 15  
Korea: 14  
China: 11  
France: 11  
Sweden: 11

`cmp(a,b)` returns `-1` if  $a < b$ , `0` if  $a = b$ , and `+1` if  $a > b$ .



We want to create a histogram of medals:

```
0~2:    ****
3~5:    ****
6~8:    ***
9~11:   ****
12~14:  *
15~17: **
18~20:
21~23: *
24~26: *
27~29:
30~32: *
33~35:
36~38: *
```

```
def histogram():
    t = [0] * 13
    for item in medals:
        total = sum(item[1:])
        t[total / 3] += 1
    for i in range(13):
        print str(3*i) + "~" + str(3*i+2)
              + ":\t" + ("*" * t[i])
```



## Sieve of Eratosthenes

```
def sieve(n):
    t = range(3, n, 2)
    sqrtn = int(math.sqrt(n))
    i = 0
    while t[i] <= sqrtn:
        # remove all multiples of t[i]
        p = t[i]
        for j in range(len(t)-1, i, -1):
            if t[j] % p == 0:
                t.pop(j)
        i += 1
    return t
```

3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61,  
67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137,  
139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199,  
211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277,  
281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359,  
367, 373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439,  
443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503, 509, 521,  
523, 541, 547, 557, 563, 569, 571, 577, 587, 593, 599, 601, 607,  
613, 617, 619, 631, 641, 643, 647, 653, 659, 661, 673, 677, 683,  
691, 701, 709, 719, 727, 733, 739, 743, 751, 757, 761, 769, 773,  
787, 797, 809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863,  
877, 881, 883, 887, 907, 911, 919, 929, 937, 941, 947, 953, 967,  
971, 977, 983, 991, 997



---

CS101, Spring 2015

# Image Processing

## Lecture #7



Last time we learned

- Sequences
  - Lists
  - Strings
  - Tuples

This week we will learn

- Default parameters
- Named parameters
- Formatting
- String methods
- Image processing



We have learnt about **parameters** and function **arguments**:

```
def create_sun(radius, color):  
    sun = Circle(radius)  
    sun.setFillColor(color)  
    sun.setBorderColor(color)  
    sun.moveTo(100, 100)  
    return sun
```

```
sun = create_sun(30, "yellow")
```

Arguments are mapped to parameters one-by-one, left-to-right.



We can provide default parameters:

```
def create_sun(radius = 30, color = "yellow"):
    # as before
```

Now we can call it like this:

```
sun = create_sun()
star = create_sun(2)
moon = create_sun(28, "silver")
```

But not like this:

```
moon = create_sun("silver")
```





Default parameters have to follow normal parameters:

```
def avg(data, start = 0, end = None):  
    if not end:  
        end = len(data)  
    return sum(data[start:end]) / float(end-start)
```

```
>>> d = [ 1, 2, 3, 4, 5 ]
```

```
>>> avg(d)
```

```
3.0
```

```
>>> avg(d, 2)
```

```
4.0
```

```
>>> avg(d, 1, 4)
```

```
3.0
```





We can include the name of the parameter in the function call to make the code clearer. Then the order of arguments does not matter:

```
moon = create_sun(color = "silver")
```

```
moon = create_sun(color = "silver", radius = 28)
```

```
>>> avg(d, end=3)
```

```
2.0
```

```
>>> avg(data=d, end=3)
```

```
2.0
```

```
>>> avg(end=3, data=d)
```

```
2.0
```

```
>>> avg(end=3, d)
```

```
SyntaxError: non-keyword arg after keyword arg
```



We often want to produce nicely formatted output:

```
print "Max between " + str(x0) + " and " +  
      str(x1) + " is " + str(val)
```

The string formatting operator `%` makes this much easier:

```
print "Max between %d and %d is %g" % (x0, x1, val)
```

Formatting operator:

```
format_string % (arg0, arg1, .... )
```

Tuple has one element for each **place holder** in the **format\_string**. Place holders are:

- `%d` for integers in decimal,
- `%g` for `float`,
- `%.2f` for `float` with fixed precision (2 digits after period),
- `%s` for anything (like `str(x)`).



If there is only one place holder, tuple is not necessary:

```
print "Maximum is %g" % val
```

We can align tables by using field width:

```
print "%3d ~ %3d : %10g" % (x0, x1, val)
```

A value can be left-aligned in its field:

```
print "%3d ~ %-3d : %-12g" % (x0, x1, val)
```



Strings are sequences:

```
def is_palindrome(s):  
    for i in range(len(s) / 2):  
        if s[i] != s[len(s) - i - 1]:  
            return False  
    return True
```

Strings are **immutable**.

The **in** operator for strings:

```
>>> "abc" in "01234abcdefg"  
True  
>>> "abce" in "01234abcdefg"  
False
```

Different from the **in** operator for lists and tuples, which tests whether something is equal to an element of the list or tuple.



String objects have many useful methods:

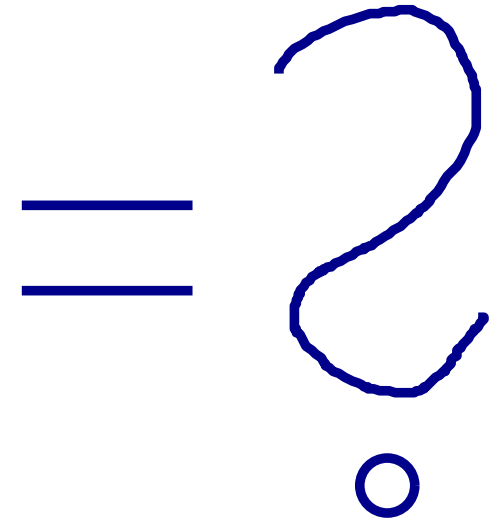
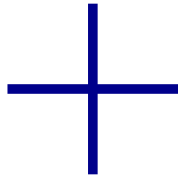
- `upper()`, `lower()`, and `capitalize()`
- `isalpha()` and `isdigit()`
- `startswith(prefix)` and `endswith(suffix)`
- `find(str)`, `find(str, start)`, and `find(str, start, end)`  
(return -1 if `str` is not in the string)
- `replace(str1, str2)`
- `rstrip()`, `lstrip()` and `strip()` to remove white space on the right, left, or both ends.

String methods for converting between lists and strings:

- `split()` splits with white space as separator
- `split(sep)` splits with given separator `sep`
- `join(l)` concatenates strings from a list `l`



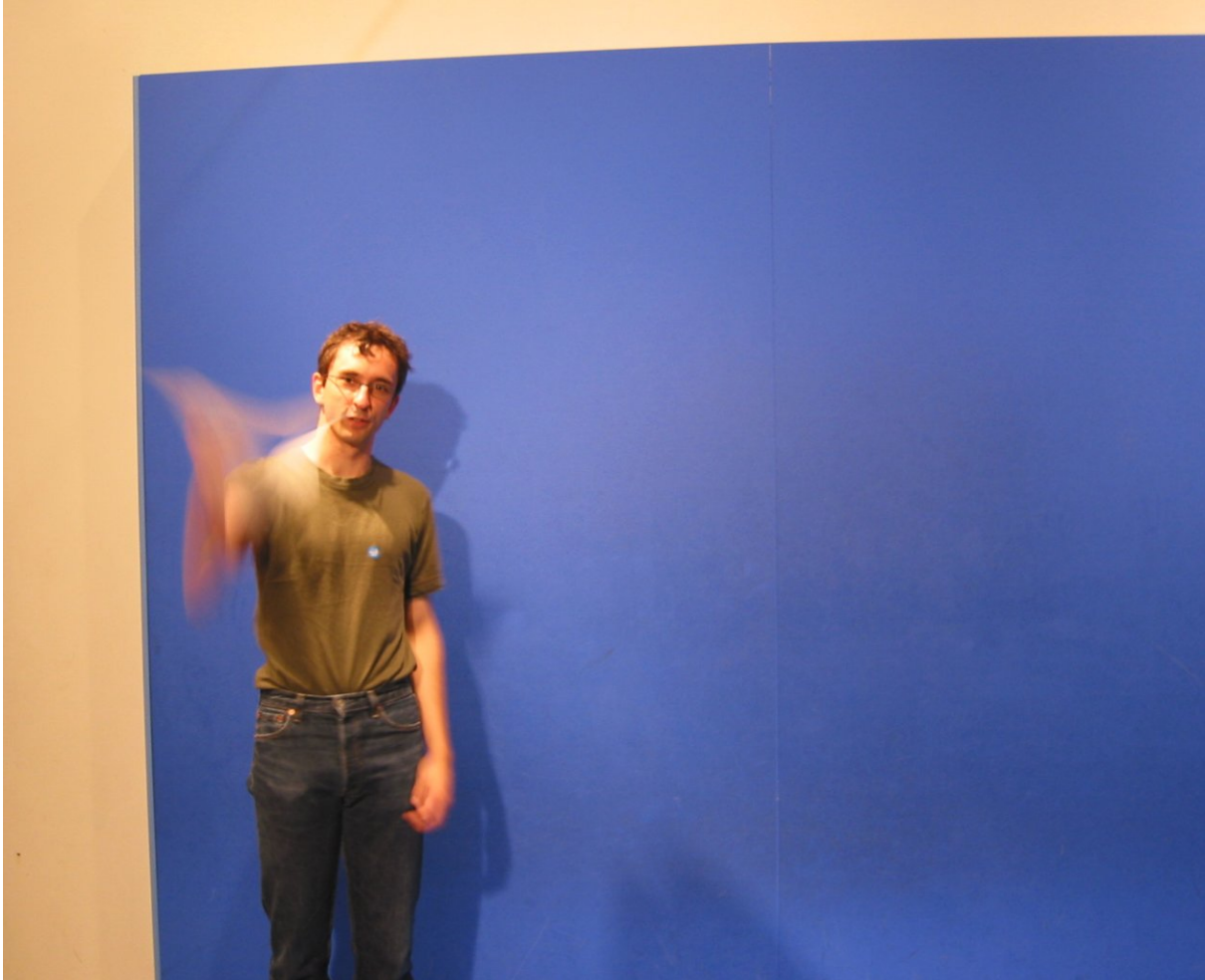
Let's put the KAIST statue on a nice background:



```
def paste(canvas, img, x1, y1):  
    w, h = img.size()  
    for y in range(h):  
        for x in range(w):  
            canvas.set(x1 + x, y1 + y, img.get(x, y))
```



**Chromakey** is a technique to overlay one scene on top of another one. It is commonly used for weather maps.





Actually, the background is not exactly blue - just blueish.  
We need a function to decide how similar two colors are:

```
def dist(c1, c2):  
    r1, g1, b1 = c1  
    r2, g2, b2 = c2  
    return math.sqrt((r1-r2)**2 + (g1-g2)**2 +  
                      (b1-b2)**2)
```

This is just the Euclidean distance in  $\mathbb{R}^3$ .





```
def chroma(img, key, threshold):  
    w, h = img.size()  
    for y in range(h):  
        for x in range(w):  
            p = img.get(x, y)  
            if dist(p, key) < threshold:  
                img.set(x, y, Color.yellow)
```





Now all we need is a paste function that skips the color-coded background:

```
def chroma_paste(canvas, img, x1, y1, key):  
    w, h = img.size()  
    for y in range(h):  
        for x in range(w):  
            p = img.get(x, y)  
            if p != key:  
                canvas.set(x1 + x, y1 + y, p)
```





Humans cannot perceive a small change in light intensity or color value. We can use this to hide information inside images.

Here is an algorithm to hide a black/white image `secret` in an image `img`:

- For all pixels  $(r, g, b)$  of `img`, if `r` is odd then subtract one from `r`;
- For each black pixel of `secret`, add one to the red value of the same pixel in `img`.

To decode the secret, we look at all pixels  $(r, g, b)$  of the image, and turn it black if `r` is odd, and white otherwise.



---

CS101, Spring 2015

# Objects: Creation and Attributes

Lecture #8



Last week we learned

- Default parameters
- Named parameters
- Formatting
- String methods
- Image processing

This week we will learn

- Objects
  - object creation
  - object attributes



There are 52 cards. Each card has a **face** and a **suit**. The suits are **clubs**, **spades**, **hearts**, and **diamonds**. The faces are 2, 3, ..., 10, Jack, Queen, King, Ace.

The **value** of a card is the number for a number card, 11 for an Ace, and 10 for Jack, Queen, and King.

We can represent cards as a triple (**face**, **suit**, **value**).

If **card** is a card, then **card[0]** is the face, **card[1]** is the suit, and **card[2]** is the value.



Computing the value of a hand:

```
def hand_value(hand):  
    total = 0  
    for card in hand:  
        total += card[2]  
    return total
```

Printing a card nicely:

```
def card_string(card):  
    article = "a "  
    if card[0] in [8, "Ace"]: article = "an "  
    return article + str(card[0]) + " of " + card[1]
```

Easy to make mistakes: What does `card[2]` mean? What if somebody creates a card `("Ace", "Spades", 5)`?



Let us define a new object type with **attributes** for face, suit, and value:

```
class Card(object):  
    """A Blackjack card."""  
    pass
```

```
card = Card()  ← create Card object  
card.face = "Ace"  
card.suit = "Spades"  ← set attributes of card  
card.value = 11
```

**card** has a user-defined type:

```
>>> type(card)  
<class '__main__.Card'>
```





Computing the value of a hand:

```
def hand_value(hand):  
    total = 0  
    for card in hand:  
        total += card.value  
    return total
```

Printing a card nicely:

```
def card_string(card):  
    article = "a "  
    if card.face in [8, "Ace"]: article = "an "  
    return (article + str(card.face) +  
            " of " + card.suit)
```



There is one big difference between tuples and **Card** objects:  
Card objects are mutable:

```
>>> card = Card()
>>> card.face = "Ace"
>>> card.suit = "Spades"
>>> card.value = 11
... and later ...
>>> card.suit = "Hearts"
```



An animation by Jeong-eun Yu and Geum-hyeon Song (2010 Freshmen).



Three **Layer** objects: mother hen, chick1, chick2.

Each chicken has **body**, **wing**, **eye**, and **beak**. Mother hen also has two red dots on the head.

The two chickens are exactly the same. Mother hen is larger and white.



The simplest method to make similar objects is to write the code once, and copy & paste it (with the necessary modifications).

**Disadvantage:** When you find a bug, you have to debug **all** copies of the code. It is not easy to change the appearance of all the chickens at once.

Let's try to implement the chicken as an object:

```
class Chicken(object):  
    """Graphic representation of a chicken."""  
    pass
```

Our chicken will have attributes **layer**, **body**, **wing**, **eye**, and **beak**.



The function `make_chicken` creates a chicken object, with positioned at (0,0).

```
def make_chicken(hen = False):  
    layer = Layer()  
    if hen:  
        body = Ellipse(70,80)  
        body.setFillColor("white")  
    else:  
        body = Ellipse(40,50)  
        body.setFillColor("yellow")  
        body.move(0, 10)  
    body.setBorderColor("yellow")  
    body.setDepth(20)  
    layer.add(body)  
    # similar for wing, eye, beak, dots
```



Finally we create and return the **Chicken** object:

```
def make_chicken(hen = False):
```

```
    # ... see previous page
```

```
    ch = Chicken()
```

```
    ch.layer = layer
```

```
    ch.body = body
```

```
    ch.wing = wing
```

```
    ch.eye = eye
```

```
    # return the Chicken object
```

```
    return ch
```



We use **Chicken** objects by accessing their attributes:

```
hen = make_chicken(True)
chick1 = make_chicken()
chick1.layer.move(120,0)
```

```
herd = Layer()
herd.add(hen.layer)
herd.add(chick1.layer)
herd.move(600, 200)
```

```
chick2 = make_chicken()
chick2.layer.move(800,200)
```



---

CS101, Spring 2015

# Text Processing

## Lecture #9





---

Last week we learned

- Objects
  - object creation
  - object attributes

This week we will learn

- Files
  - reading from a file
  - writing to a file
- break
- continue



We have a file "C:/CS101/planets.txt" on our hard disk with this contents:

Mercury  
Venus  
Earth  
Mars  
Jupiter  
Saturn  
Uranus  
Neptune

`f` is a file object (of type `<type 'file'>`), not the contents of the file.

mode: "r" for reading

```
>>> f = open("C:/CS101/planets.txt", "r")
>>> s = f.readline()
>>> s, len(s)
('Mercury\n', 8)
```

line separator in file



We typically use `strip()` or at least `rstrip()` for the lines we read from a file to get rid of white space.

```
>>> for l in f:  
...     s = l.strip()  
...     print s,
```

Venus Earth Mars Jupiter Saturn Uranus Neptune

`for`-loop with a file object calls `readline()` automatically for each element, and stops after reading the last line.

Call `f.close()` when finished with the file object.



A typical program for reading the contents of an entire file and storing it in a list:

```
planets = []  
  
f = open("C:/CS101/planets.txt", "r")  
for line in f:  
    planets.append(line.strip())  
f.close()  
  
print planets
```

In fact, file objects provide a method to do this (but then you get all the white space):

```
planets = f.readlines()
```



We want to find the line in the file containing earth:

```
f = open("C:/CS101/planets.txt", "r")
current = 0
earth = 0
for line in f:
    current += 1
    planet = line.strip().lower()
    if planet == "earth":
        earth = current
print "Earth is planet #%d" % earth
```

The program reads the entire file, even if earth is right at the beginning. After having found earth, there is no need to continue the loop.



The keyword **break** terminates the current loop:

```
f = open("C:/CS101/planets.txt", "r")
earth = 0
for line in f:
    earth += 1
    planet = line.strip().lower()
    if planet == "earth":
        break
print "Earth is planet #%d" % earth
```

**break** breaks out of the innermost loop only:

```
>>> for x in range(10):
...     for y in range(10):
...         print y,
...         if y == 5: break
```



Some data files contain useful comments, let's say starting with a # sign.

```
f = open("C:/CS101/planet.sc.txt", "r")
earth = 0
for line in f:
    planet = line.strip().lower()
    if planet[0] == "#":
        continue
    earth += 1
    if planet == "earth":
        break
print "Earth is planet #%d" % earth
```

**continue** makes the loop move to the next element immediately



Let's do some word games. We use a file `words.txt` with 113809 English words (<http://icon.shef.ac.uk/Moby/>).

Let's print all English words longer than 18 letters:

```
f = open("C:/CS101/words.txt", "r")
```

```
for line in f:
    word = line.strip()
    if len(word) > 18:
        print word
```

```
f.close()
```





Count all the words without the letter 'e':

```
f = open("C:/CS101/words.txt", "r")

count = 0
for line in f:
    word = line.strip()
    if not "e" in word:
        count += 1

print "%d words have no 'e'" % count
f.close()
```



Let's find all words whose letters are sorted:

```
def is_abecedarian(word):  
    for i in range(1, len(word)):  
        if word[i-1] > word[i]:  
            return False  
    return True  
  
f = open("C:/CS101/words.txt", "r")  
  
for line in f:  
    word = line.strip()  
    if is_abecedarian(word):  
        print word  
  
f.close()
```



## Three double letters in a row?

Is there a word that has three double letters in a row?

Committee and Mississippi are close...

```
def three_doubles(word):  
    s = ""  
    for i in range(1, len(word)):  
        if word[i-1] == word[i]:  
            s = s + "*"  
        else:  
            s = s + " "  
    return "* * *" in s
```



We can also create and write to files:

```
f = open("C:/CS101/test.txt", "w")  
f.write("CS101 is fantastic\n")  
f.close()
```

Use mode `"w"` to open a file for writing.

The file object has a method `write(text)` to write to the file. Unlike `print`, this does not start a new line after the `text`, not even a single space. Use `\n` to include a line break.

Do not forget to `close()` the file—otherwise, the file contents may be incomplete.

Let's exercise with a currency exchange rate data set. We use files `1994.txt ... 2009.txt` with the KRW-USD exchange rate for every day. ([www.oanda.com](http://www.oanda.com))

```
2009/05/11          0.00080110
```

We first read the entire data set (16 files) into a long list of pairs:

```
[... (20091227, 1154), (20091228, 1154),  
(20091229, 1167), (20091230, 1167),  
(20091231, 1163)]
```

Let's find the maximum, minimum, and average for each year.

```
Minimum: (19950705, 755)
```

```
Maximum: (19971223, 1960)
```

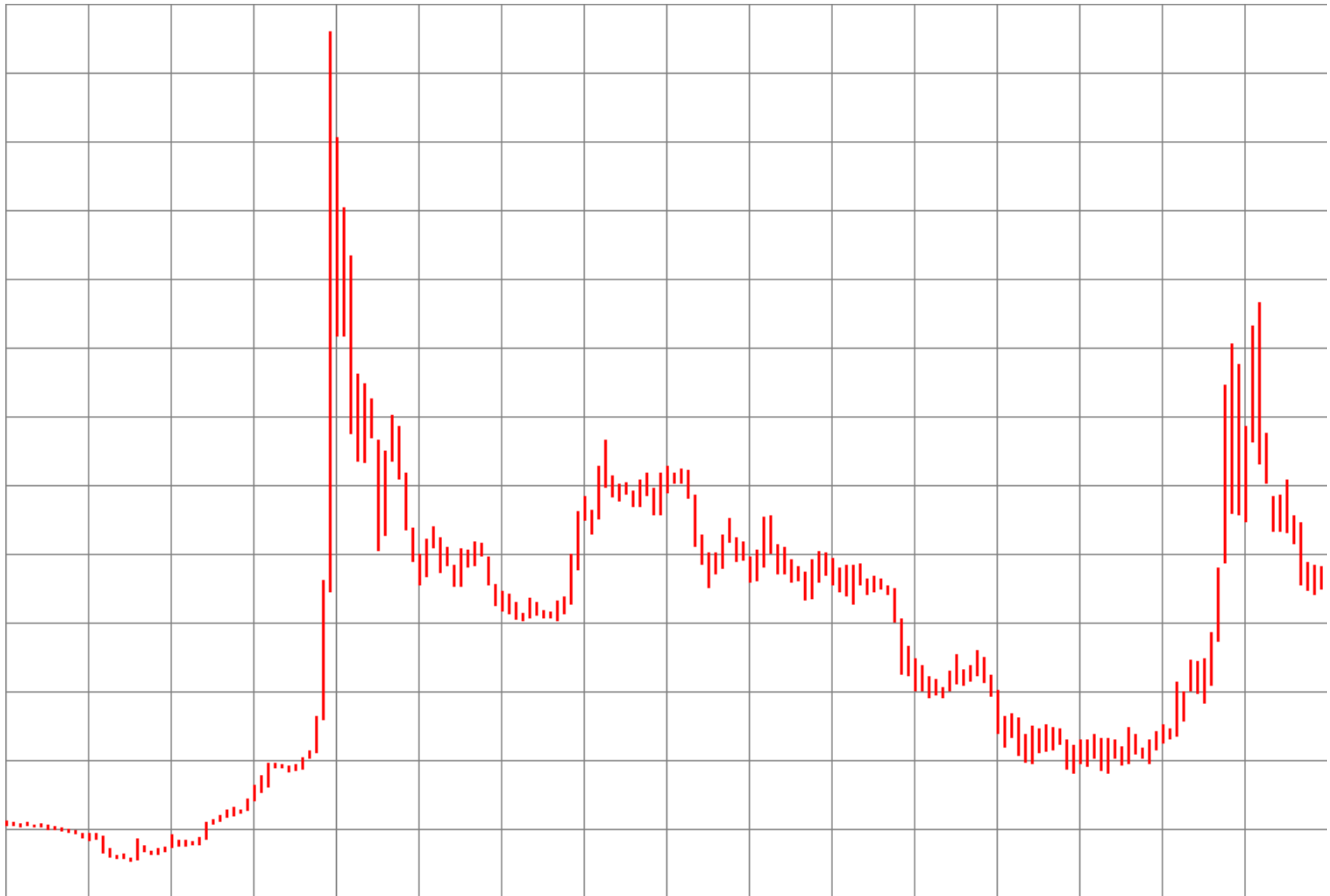


Minimum and maximum for every month of a year:

```
def find_minmax(yr):  
    minmax = [ (9999, 0) ] * 12  
    data = read_year(yr)  
    for d, v in data:  
        # make month 0 .. 11  
        month = (d / 100) % 100 - 1  
        minr, maxr = minmax[month]  
        if v < minr:  
            minr = v  
        if v > maxr:  
            maxr = v  
        minmax[month] = minr, maxr  
    return minmax
```



Let's use `cs1media` to create a nice plot of the exchange rate.





## Case study: maintaining stock

Let's write software for a vegetable and fruit shop. We keep a file with the current stock and prices. We should be able to sell items, list the current stock, and report all sales done during the day.

File format `stock.txt`:

```
190,apple,900
```

```
30,orange,1300
```

Main menu:

What would you like to do:

S) Sell item

P) Print stock

R) Report sales

E) Exit

Enter your choice>





If possible, we test each function after writing it. In the Wing IDE, pressing **Run** executes your current file. You can then use the functions inside the file from the Python shell. (From the command line, use `python -i script.py`)

Testing `load_stock("stock.txt")`:

```
>>> load_stock("stock.txt")
[(190, 'apple', 900), (30, 'orange', 1300),
(13, 'pineapple', 5500), (60, 'carrot', 600),
(30, 'cucumber', 900), (20, 'egg plant', 1100),
(10, 'zucchini', 1300), (70, 'garlic', 300)]
```

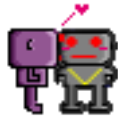


We can test code before having written all the functions called:

```
def main():  
    stock = load_stock(stock_file_name)  
    sales = []  
    while True:  
        s = show_menu()  
        if s == 'e':  
            break  
        elif s == 's':  
            sell(stock, sales)  
        elif s == 'p':  
            print_stock(stock)  
        elif s == 'r':  
            print_sales(sales)
```

need these functions to test

needed only when selected



---

CS101, Spring 2015

# Object Construction and User Interface Programming

## Lecture #10



---

Last week we learned

- Files
  - reading from a file
  - writing to a file
- break
- continue

This week we will learn

- Objects
  - Constructors
  - String conversion
- User interface programming



There are 52 cards. Each card has a **face** and a **suit**. The suits are **clubs**, **spades**, **hearts**, and **diamonds**. The faces are **2**, **3**, ..., **10**, **Jack**, **Queen**, **King**, and **Ace**. The **value** of a card is the number for a number card, 11 for an Ace, and 10 for Jack, Queen, and King.

```
class Card(object):  
    """A Blackjack card."""  
    pass
```

```
card = Card()  
card.face = "Ace"  
card.suit = "Spades"  
card.value = 11
```



We do not really need the `value` attribute, since the value can be computed from the `face` attribute. We add a `value()` method to our `Card` class:

```
class Card(object):  
    """A Blackjack card."""  
    def value(self):  
        if type(self.face) == int:  
            return self.face  
        elif self.face == "Ace":  
            return 11  
        else:  
            return 10
```

← method of `Card`

`self` refers to the object itself inside the method.



We can create and use **Card** objects:

```
>>> card = Card()
>>> card.face = "Ace"
>>> card.suit = "Spades"
>>> card_string(card)
'an Ace of Spades'
>>> card.value()
11
```

We need nicer syntax to create **Card** objects:

**Card(8, "Clubs").**

And **card\_string** should be a method of **Card**.



Objects can have a special method `__init__`, called a **constructor**. Whenever an object of this type is created, the constructor is called.

```
FACES = range(2,11) + ['Jack', 'Queen', 'King', 'Ace']  
SUITS = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
```

```
class Card(object):  
    """A Blackjack card."""  
  
    def __init__(self, face, suit):  
        assert face in FACES and suit in SUITS  
        self.face = face  
        self.suit = suit
```





Now creating cards is elegant:

```
hand = [ Card("Ace", "Spades"),  
         Card(8, "Diamonds"),  
         Card("Jack", "Hearts"),  
         Card(10, "Clubs") ]
```

Let's change `card_string(card)` into `card.string()`:

```
def string(self):  
    article = "a "  
    if self.face in [8, "Ace"]: article = "an "  
    return (article + str(self.face) +  
           " of " + self.suit)
```

```
for card in hand:  
    print card.string(), "has value", card.value()
```



We can make conversion to strings even nicer: `str(card)` calls the special method `__str__`:

```
def __str__(self):  
    article = "a "  
    if self.face in [8, "Ace"]: article = "an "  
    return (article + str(self.face) +  
            " of " + self.suit)
```

Now we can write:

```
for card in hand:  
    print card, "has value", card.value()
```

 `print` automatically converts its arguments to `str`



Let's improve our **Chicken** object by adding a constructor and a **move** method.

```
class Chicken(object):  
    """Graphic representation of a chicken."""  
    def __init__(self, hen = False):  
        layer = Layer()  
        # make all the parts  
        self.layer = layer  
        self.body = body  
        self.wing = wing  
        self.eye = eye  
  
    def move(self, dx, dy):  
        self.layer.move(dx, dy)
```



Let's create another object that represents a shuffled deck of 52 cards. We only need one method: drawing a card from the top of the deck:

```
class Deck(object):
    """A deck of cards."""
    def __init__(self):
        "Create a deck of 52 cards and shuffle them."
        self.cards = []
        for suit in SUITS:
            for face in FACES:
                self.cards.append(Card(face, suit))
        random.shuffle(self.cards)

    def draw(self):
        """Draw the top card from the deck."""
        return self.cards.pop()
```



```
num_players = 3
num_cards = 5
deck = Deck()
hands = []
for j in range(num_players):
    hands.append([])

for i in range(num_cards):
    for j in range(num_players):
        card = deck.draw()
        hands[j].append(card)
        print "Player", j+1, "draws", card

for j in range(num_players):
    print ("Player %d's hand (value %d):" %
          (j+1, hand_value(hands[j])))
    for card in hands[j]:
        print " ", card
```

A list of lists (one for each player)



---

## Time to play Blackjack:

You are dealt a 6 of Hearts

Dealer is dealt a hidden card

You are dealt a 3 of Spades

Dealer is dealt a 9 of Hearts

Your total is 9

Would you like another card? (y/n) y

You are dealt an Ace of Clubs

Your total is 20

Would you like another card? (y/n) n

The dealer's hidden card was a 10 of Spades

The dealer's total is 19

Your total is 20

The dealer's total is 19

You win!



The comparison operators `==`, `!=`, `<` etc. do not work automatically for objects:

```
>>> Card(8, "Diamonds") == Card(8, "Diamonds")
```

```
False
```

```
>>> Card(8, "Diamonds") == Card(9, "Diamonds")
```

```
False
```

We can define **equality** through the special method `__eq__`:

```
def __eq__(self, rhs):  
    return (self.face == rhs.face and  
            self.suit == rhs.suit)  
  
def __ne__(self, rhs):  
    return not self == rhs
```



Player hand

Dealer hand

Player score

Dealer score

You have a tie!

Another round?

Message

Question (with y/n answer)





A **Table** represents the Blackjack table. It provides the following methods:

- `clear()` clear everything
- `close()` close window and end game
- `set_score(which, text)` where `which` in `[ 0, 1 ]`
- `set_message(text)`
- `ask(prompt)` waits for `y` or `n` and returns `True` or `False`

**Table** has two attributes `dealer` and `player`. These are **Hand** objects that represent the hand on the table.

Methods of **Hand** objects:

- `clear()`
- `add(card, hidden = False)`
- `show()` shows all hidden cards
- `value()` return value of hand



The `Table.ask(prompt)` method must wait for the user to press a key:

```
def ask(self, prompt):
    self.question.setMessage(prompt)
    while True:
        e = self.canvas.wait() ← e is an event object
        d = e.getDescription()
        if d == "canvas close":
            sys.exit(1)
        if d == "keyboard":
            key = e.getKey()
            if key == 'y':
                return True
            if key == 'n':
                return False
```



Programs with a **graphical user interface (GUI)** are structured around **events**. Most of the time, the program just waits for an event to happen.

Events are for instance:

- key presses
- window is minimized, maximized, or closed
- mouse is moved
- mouse button is pressed
- cursor enters window or leaves window

**Event-based** programming means that a program doesn't have a sequential flow of control, but consists of functions that are called by events.

---

# Object-Oriented Programming in Python

---

**Michael H. Goldwasser**

*Saint Louis University*

**David Letscher**

*Saint Louis University*



*Upper Saddle River, New Jersey 07458*

## CHAPTER 3

# Getting Started with Graphics

- 
- 3.1 The Canvas
  - 3.2 Drawable Objects
  - 3.3 Rotating, Scaling, and Flipping
  - 3.4 Cloning
  - 3.5 Case Study: Smiley Face
  - 3.6 Layers
  - 3.7 Animation
  - 3.8 Graphical User Interfaces
  - 3.9 Case Study: Flying Arrows
  - 3.10 Chapter Review
- 

In Section 1.4.4 we designed a hierarchy for a hypothetical set of drawable objects. Those classes are not actually built into Python, but we have implemented precisely such a package for use with this book. Our software is available as a module named `cs1graphics`, which can be downloaded and installed on your computer system. This chapter provides an introductory tour of the package. Since these graphics classes are not a standard part of Python, they must first be loaded with the command

```
>>> from cs1graphics import *
```

The package can be used in one of two ways. To begin, we suggest experimenting in an interactive Python session. In this way, you will see the immediate effect of each command as you type it. The problem with working interactively is that you start from scratch each time. As you progress, you will want to save the series of commands in a separate file and then use that source code as a script for controlling the interpreter (as introduced in Section 2.9).

Our tour begins with the `Canvas` class, which provides the basic windows for displaying graphics. Next we introduce the various `Drawable` objects that can be added to a canvas. We continue by discussing techniques to control the relative depths of objects that appear on a canvas and to perform rotations, scaling, and cloning of those objects. Near the end of the chapter we discuss more advanced topics, including the use of a `Layer` class that provides a convenient way to group shapes into a composite object, techniques to create dynamic animations rather than still images, and preliminary support for monitoring a user's interactions with the mouse and keyboard.

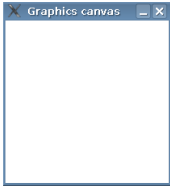


FIGURE 3.1: A new Canvas on the desktop; the exact appearance depends upon the computing environment.

3.1 The Canvas

We begin by introducing the Canvas class. A canvas represents a window upon which we draw. We can create a canvas by calling the constructor as follows:

```
>>> Canvas()
>>>
```

After entering this command, a new window should appear on the screen, such as the one shown in Figure 3.1. However, we have an immediate problem; we have no further way to interact with the canvas because we did not assign an identifier to the newly created object. A more useful beginning is the following:

```
>>> paper = Canvas()
>>>
```

This creates an instance of the Canvas class that appears on the screen, and it also assigns the identifier `paper` to that object. This is similar to the syntax `groceries = list()` which we used when instantiating a `list` in Section 2.2.1.

By default, a newly created canvas is 200 pixels wide and 200 pixels tall, has a white background color, and is titled “Graphics Canvas.” But a canvas is mutable, so we can change several aspects of its state by calling appropriate methods. For example, the following commands modify the canvas that was earlier identified as `paper`:

```
>>> paper.setBackgroundColor('skyBlue')
>>> paper.setWidth(300)
>>> paper.setTitle('My World')
```

As each individual command is entered into the interpreter, the change to the Canvas is immediately visible. When the desired characteristics of a canvas are known in advance, they can be specified as *optional parameters* to the constructor (see page 43 for further

Canvas	
Canvas(w, h, bgColor, title, autoRefresh)	add(drawable)
getWidth()	remove(drawable)
setWidth(w)	clear()
getHeight()	open()
setHeight(h)	close()
getBackgroundColor()	saveToFile(filename)
setBackgroundColor(color)	setAutoRefresh(trueOrFalse)
getTitle()	refresh()
setTitle(title)	wait()

FIGURE 3.2: Overview of the Canvas class.

discussion of optional parameters). In the case of the Canvas class, the constructor accepts optional parameters respectively specifying the initial width, height, background color, and title. So we could have created and configured the previous Canvas succinctly as follows:

```
>>> paper = Canvas(300, 200, 'skyBlue', 'My World')
>>>
```

The caller may elect to fill in only some of the parameter values, although only when starting from the leftmost. So the syntax `Canvas(500, 270)` creates a new canvas with the specified width and height, yet using the default color of white and the default window title. The simple syntax `Canvas()` relies upon all of the default values. On the other hand, the syntax `Canvas('black')` fails because the first parameter sent must be the width. The interpreter reports the problem with an error message similar to the following:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <toplevel>-
  File "cslgraphics.py", line 862, in __init__
    raise TypeError('numeric value expected for width')
TypeError: numeric value expected for width
```

As we continue, we will explore many other important behaviors of the Canvas class. A more complete reference is given in the form of a class diagram in Figure 3.2. There are methods that allow a user to query and to alter aspects such as the width, height, background color, and title. In the next section, we discuss how drawable objects can be added to or removed from a canvas. The canvas can be explicitly closed (i.e., iconified) and reopened, and the canvas’s image can even be saved to a file. Finally, there are methods that involve what we term the “refresh” mode of the canvas; we delay discussion of this issue until page 113.

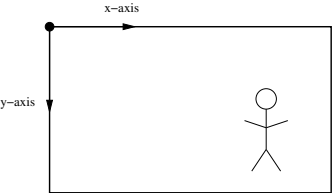


FIGURE 3.3: The canvas coordinate system, with the origin at the top left corner.

The coordinate system

Before introducing the individual shapes, we discuss the coordinate system used to describe the positions of those objects relative to a canvas. We have already seen that each canvas has a width and a height, measured in pixels. To describe the locations of shapes, we consider a coordinate system with the x-axis running horizontally and the y-axis vertically. The standard coordinate system for computer graphics uses the *top left* corner of a canvas as the origin, as shown in Figure 3.3. A typical position in the canvas is then specified as a pair of coordinates, with the x-coordinate measuring the number of pixels to the *right* of that corner, and the y-coordinate measuring the number of pixels *below* that corner.

When we want to place one of our graphical objects upon a canvas, we specify the coordinates for a key *reference point* of that shape, for example, the center of a rectangle. As a physical analogy, assume that our canvas is a bulletin board and that our shapes are made of paper. Each shape is attached to the canvas by an imaginary thumbtack that pokes through the shape at its reference point. We specify the location of the overall shape relative to the canvas by designating the placement of that thumbtack in the canvas’s coordinate system. Later we will discuss how we can even rotate or scale an object about its reference point.



A WORD OF WARNING

Do not confuse the coordinate system used for computer graphics with the traditional mathematical convention. The computer graphics system uses the top left corner as the origin, with the positive y-axis oriented *downward* from the origin; the usual mathematics convention uses the bottom left corner as the origin, with the positive y-axis oriented *upward* from the origin.

3.2 Drawable Objects

The `cs1graphics` module supports a variety of objects that can be drawn onto a canvas. These objects are organized in a class hierarchy originally modeled in Section 1.4.4. A more detailed summary of the various classes is given in Figure 3.4. This diagram is packed with information and may seem overwhelming at first, but it provides a nice summary that can be used as a reference. The use of inheritance emphasizes the similarities, making it easier to learn how to use each class.

When examining this diagram, remember that classes in a hierarchy inherit methods from their parent class. For example, each type of `FillableShape` supports a `setFillColor` method. So not only does a `Circle` support the `setRadius` method, but also the `setFillColor` method due to its parent class, and the `setBorderWidth` method due to its grandparent `Shape` class. Many of the names should be self-explanatory and formal documentation for each class can be viewed directly in the Python interpreter as you work. For example, complete documentation for the `Circle` class can be viewed by typing `help(Circle)` at the Python prompt.

Once you know how to create and manipulate circles, it will not take much effort to learn to create and manipulate squares. Circles and squares are not identical; there are a few ways in which they differ. But there are significantly more ways in which they are similar. To demonstrate the basic use of the individual classes, we spend the rest of this section composing a simple picture of a house with scenery. This example does not show every single feature of the graphics library, but it should provide a good introduction.

Circle

As our first concrete example of a drawable class, we examine the `Circle` class. A new circle can be instantiated as

```
>>> sun = Circle()
```

However, the `Circle` is *not* automatically added to our `Canvas`. In general, a programmer may wish to have multiple canvases and shapes, choosing which shape is added to which canvas. So a drawable object is not placed onto any canvas until we explicitly add it. This is done by using the `add` method of the `Canvas` class. Assuming that we are still working with the original canvas that we created in Section 3.1, the syntax we use is

```
>>> paper.add(sun)
```

Having typed the command to add the sun to the paper, we might wonder where it is. By default, this circle has a radius of 10 pixels, a black border, transparent interior, and a center position of (0,0). So it has indeed been added; you just need to look carefully at the top left corner of the screen (shown in Figure 3.5). We see only the portion of the circle that is within the canvas view. The rest of the circle is there in spirit, but not currently visible. Since this is not exactly how we envisioned the sun, let’s go ahead and change the settings by calling several methods supported by the `Circle` class.

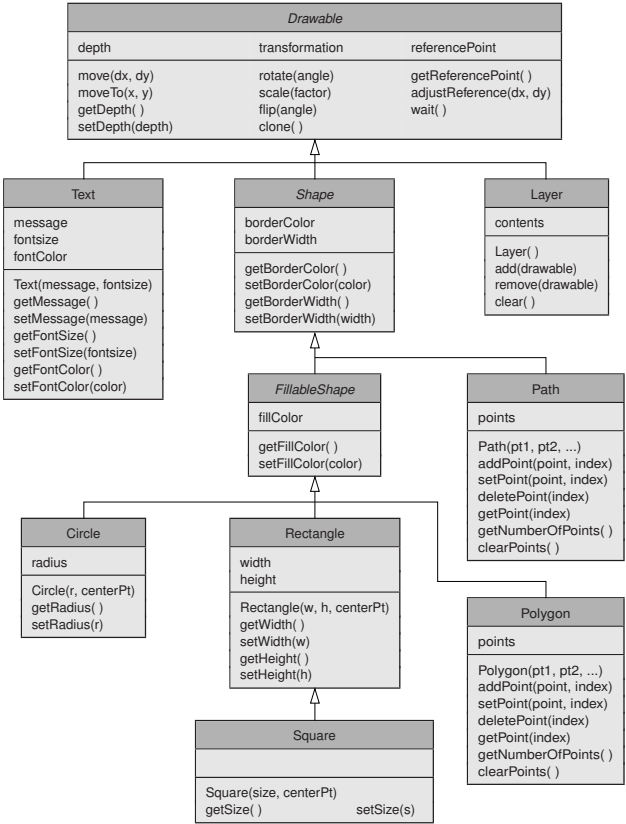


FIGURE 3.4: An overview of the Drawable objects.



FIGURE 3.5: A Canvas with a brand new Circle added (centered at the top left).

```
>>> sun.setRadius(30)
>>> sun.setFillColor('yellow')
>>> sun.move(250, 50)
```

Having made these changes to our sun, our scene now appears as shown in Figure 3.6. The first of these commands changes the radius of the circle. The second command changes the interior color of the shape to yellow (rather than the default, which is transparent). The third command repositions the shape relative to the canvas. The two parameters sent to `move` represent a change along the x-axis and y-axis respectively. Since our circle was originally centered at position (0,0), it will be centered as a result at position (250,50). However, each move is relative to the previous position. So if we were to call `sun.move(0, 20)` at this point, the sun would not move to position (0, 20) but would instead move 0 pixels to the right and 20 pixels down, and therefore be positioned at (250,70).

Although it is fine to create a default circle and then to mutate it, some (but not all) of its settings can be specified as optional parameters to the constructor. The `Circle` constructor accepts up to two additional parameters. The first parameter specifies the initial radius. Thus a call to `Circle(30)` could be used to create a circle with radius 30, although still centered at the origin with a black border and transparent interior. A second optional parameter can be used to choose a different initial center position. This parameter should be an instance of another class from our graphics package, named `Point`. A `Point` is a rather simple class that allows us to represent the two coordinates of a position as a single object, rather than as two distinct integers (in Chapter 6 we will even implement such a class). A new `Point` instance can be created using a syntax such as

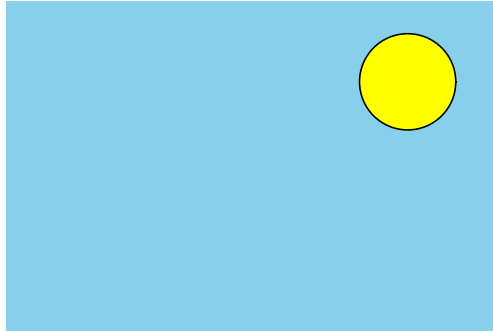


FIGURE 3.6: Our scene after reconfiguring the sun.

`Point(250,50)`. Therefore, we could have partially configured our original sun using the syntax `sun = Circle(30, Point(250,50))`. That circle would still have a default border of black and a transparent interior. Alternative colors can only be selected *after* constructing the circle. (As the designers of the `cs1graphics` package, we could have included further optional parameters for the constructor, but we opted for the simpler design.)

### Square

As the next piece of our scene, we use a white square as the front of the house. The `Square` class is another example of a `FillableShape`. In fact, it supports almost the identical behaviors of a circle, except that its size is described based upon the length of a side instead of the radius of the circle. The constructor for the class accepts two parameters, the first being the width (and thus height) of the square and the second being a `Point` that designates the initial placement of the square's center. If these parameters are not specified, a default square is  $10 \times 10$  and centered at  $(0,0)$ . We create the front of our house and add it to our scene as follows:

```
>>> facade = Square(60, Point(140,130))
>>> facade.setFillColor('white')
>>> paper.add(facade)
```

It is also possible to change the size of a square after it has been created, using a syntax such as `facade.setSize(60)`.

### Rectangle

Another available shape is a `Rectangle`, which is similar to a `Square` except that we can set its width and height independently. By default, a rectangle has a width of 20 pixels, a height of 10 pixels, and is centered at the origin. However, the initial geometry can be specified using three optional parameters to the constructor, respectively specifying the width, the height, and the center point. Adding to our ongoing picture, we place a chimney on top of our house with the following commands:

```
>>> chimney = Rectangle(15, 28, Point(155,85))
>>> chimney.setFillColor('red')
>>> paper.add(chimney)
```

We carefully design the geometry so that the chimney appears to rest on top of the right side of the house. Recall that the facade is a 60-pixel square centered with a y-coordinate of 130. So the top edge of that facade has a y-coordinate of 100. To rest above, our chimney is centered vertically at 85 but with a height of 28. So it extends along the y-axis from 71 to 99. Alternatively, we could have waited until after the rectangle had been constructed to adjust those settings, using the `setWidth` and `setHeight` methods.

We take this opportunity to distinguish between the interior of a shape and its border. The chimney as described above has a red interior but a thin black outline. For all of the shapes we have seen thus far, we can separately control the color of the interior, via `setFillColor`, and the color of the border, via `setBorderColor`. If we want to get rid of the border for our chimney, we can accomplish this in one of three ways. The first is `chimney.setBorderColor('red')`. This does not really get rid of the border but colors it the same as the interior. It is also possible to set the border color to a special color `'Transparent'`, in which case it is not even drawn. Finally, we can adjust the *width* of the border itself. A call to `chimney.setBorderWidth(0)` should make the border unseen, even if it were still designated as black.

### Polygon

The `Polygon` class provides a much more general shape, allowing for filled polygons with arbitrarily many sides. The geometry of a polygon is specified by identifying the position of each corner, as ordered along the boundary. As was the case with other shapes, the initial geometry of a polygon can be specified using optional parameters to the constructor. Because polygons may have many points, the constructor accepts an arbitrary number of points as optional parameters. For example, we might add an evergreen tree to our scene by creating a green triangle as follows:

```
>>> tree = Polygon(Point(50,80),Point(30,140),Point(70,140))
>>> tree.setFillColor('darkGreen')
>>> paper.add(tree)
```

The resulting scene is shown in Figure 3.7. When specifying the points, they must come in order as seen on the border of the polygon, but it does not matter whether that order is clockwise or counterclockwise. By default, the polygon's reference point is its first point.



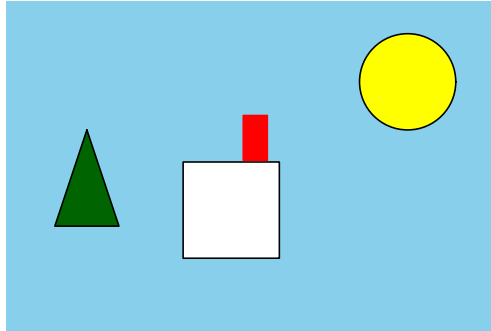


FIGURE 3.7: Further progress on our scene.

It is also possible to add additional points to a polygon after it is constructed. This is done by calling the `addPoint` method. By default the new point is appended to the end of the current sequence of points. For example, we could have created our tree as follows:

```
>>> tree = Polygon()
>>> tree.addPoint(Point(50,80))
>>> tree.addPoint(Point(30,140))
>>> tree.addPoint(Point(70,140))
>>> tree.setFillColors('darkGreen')
>>> paper.add(tree)
```

The `addPoint` method can also be used to add a new point into the middle of the existing sequence of points. In this case, an optional second parameter is used to designate the index of the new point within the sequence of points. Just as with Python's lists in Section 2.2, the first point is considered to have index 0, the next point index 1, and so on. For example, a concave angle could be added at the bottom of the tree by inserting a new point as

```
>>> tree.addPoint(Point(50,120), 2)
```

An existing point can be replaced with a new value using the `setPoint` method. The first parameter is the new point value and the second is an integer index. For example, we could raise the tip of our tree using the syntax `tree.setPoint(Point(50,70), 0)`. An existing point can be removed altogether using the syntax `deletePoint(i)`, where `i` is the index.

### Path

A Path is a shape that connects a series of points; in this respect, it is very similar to a Polygon. However, there are two key differences: the ends of a path are not explicitly connected to each other, and a path does not have an interior. A Path qualifies in our hierarchy as a Shape but not as a FillableShape. We can change the color by calling `setBorderColor`, and the thickness by calling `setBorderWidth`. Its behaviors are otherwise similar to those described for the Polygon (e.g., `addPoint`, `setPoint`, `deletePoint`). As a simple example of a path, we add a bit of smoke coming out of the top of our chimney.

```
>>> smoke = Path(Point(155,70), Point(150,65),
>>>               Point(160,55), Point(155,50))
>>> paper.add(smoke)
```

A path between two points is simply a line segment (akin to the hypothetical Segment class modeled in Figure 1.13). Let's add a few rays around our sun using four simple paths. We place one ray to the left of and below the sun; we will call this object `sunraySW` to designate this as the “southwest” of the four rays. The biggest challenge in this case is to determine the proper coordinates. We create it as follows:

```
>>> sunraySW = Path(Point(225,75), Point(210,90))
>>> sunraySW.setBorderColor('yellow')
>>> sunraySW.setBorderWidth(6)
>>> paper.add(sunraySW)
```

Sun rays emanating in other directions can be declared in similar fashion. Our updated drawing to this point is shown in Figure 3.8.

### Text and Image classes

Two additional classes are not used in the current drawing of a house but are very helpful in general. The first of these is the `Text` class, used for rendering character strings within the drawing area of the canvas. The constructor takes two parameters: the first is a string designating the actual message to be displayed, and the second is a font size (12 by default). This class fits into our hierarchy somewhat separately; it does not even qualify as being a Shape (as it does not support a concept of a border). Instead, it has dedicated methods to support the manipulation of the message, the font size, and the font color. Once created, the text can be repositioned to a desired location. By default, the reference point for a text object is aligned with the center of the displayed message. The summary of its methods is included in Figure 3.4 on page 94; we provide an example usage in a later case study in this chapter.

Although not shown in Figure 3.4, `cs1graphics` includes an `Image` class that provides support for using a raw image loaded from a file. An image object is constructed by specifying the underlying file name, as in `Image('lightbulb.gif')`. However, the precise set of supported file formats (e.g., gif, jpg, bmp, tiff) will depend upon your computer system. When controlling the placement of an image on a canvas, the reference point is aligned with the top left corner of the image.

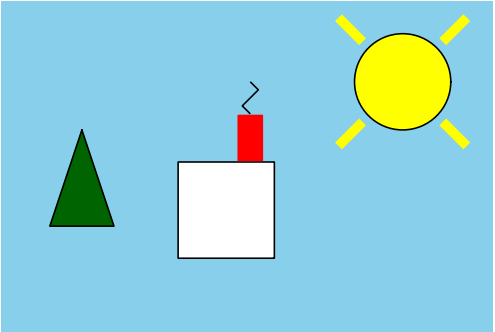


FIGURE 3.8: Sun rays and smoke have been added to our scene.

3.2.1 Colors

We have seen the use of various colors for the borders and interiors of our shapes. Thus far we have relied upon names like `'skyBlue'` when specifying a desired color. A color is represented behind the scene by what is known as its *RGB value*. This is a **tuple** of three numbers that represent the intensity of red, green, and blue respectively (hence the acronym, RGB), using a scale from 0 (no intensity) to 255 (full intensity). Names like `'skyBlue'` are predefined for convenience by the `cs1graphics` package and mapped to an appropriate RGB value, for example, `(136, 206, 235)`.

The set of predefined color names is somewhat arbitrary; you may type `help(Color)` in the interpreter for more information. However, you may define new colors by directly specifying the underlying RGB tuple in place of a name. As an example, if we wanted our sky to be slightly brighter blue than the official `'skyBlue'`, we could use the command `paper.setBackgroundColor( (136, 206, 244) )`. Note the use of parentheses here. The caller is sending a tuple as a single parameter; see Section 2.3.2 for a discussion of tuples.

3.2.2 Depths

To further improve our scene we wish to add grass to the picture. We do so by placing one very large green rectangle to cover the bottom portion of the sky blue background. We also add a window, and roof to our house. The updated image we have in mind is shown in Figure 3.9, however, achieving this effect brings up a new issue. While the shapes for the grass, window and roof are simple, we have to consider the apparent overlap among the objects. The tree and house must appear to be in front of the grass, the window in front of the house, and the roof behind the chimney yet in front of the facade of the house.

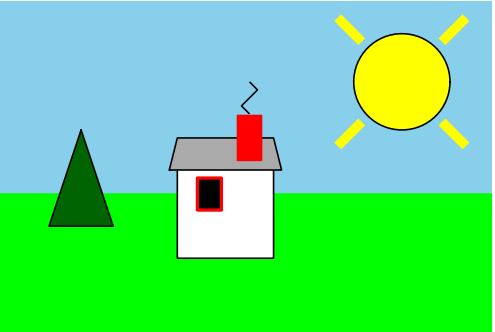


FIGURE 3.9: Demonstrating the use of depths in placing the grass, window, and roof.

The relative ordering of conflicting shapes is controlled by an underlying numeric attribute representing the “depth” of each drawable object. By default, all objects are assigned a depth value of 50 and their relative ordering is arbitrary. However, those depths can be changed to control the image. When two or more objects overlap, the one with the smallest depth will be drawn “nearer” to the viewer. For example, by giving the grass a greater depth than the default, say `grass.setDepth(75)`, it will be drawn behind both the tree and the house. This is not a three-dimensional rendering, so the picture will look the same no matter whether we set this depth to 75 or 100 or 1000. In similar spirit, we might give the window a depth of 30 so that it is drawn in front of the house facade. We set the depth of the roof to 30 so that it is in front of the facade, yet the chimney a depth of 20 so that it appears even nearer than the roof. Although we have given both the window and the roof a depth of 30, those shapes do not overlap so the conflict is irrelevant. The complete source code used to generate our latest image is given in Figure 3.10.



A WORD OF WARNING

When objects have overlapping geometries, their relative appearance on a canvas is based upon specified depths. An object with greater depth will appear to be obscured by one with lesser depth. Items with equal depths are ordered arbitrarily.

```

1 from cs1graphics import *
2 paper = Canvas(300, 200, 'skyBlue', 'My World')
3
4 sun = Circle(30, Point(250,50))
5 sun.setFillColor('yellow')
6 paper.add(sun)
7
8 facade = Square(60, Point(140,130))
9 facade.setFillColor('white')
10 paper.add(facade)
11
12 chimney = Rectangle(15, 28, Point(155,85))
13 chimney.setFillColor('red')
14 chimney.setBorderColor('red')
15 paper.add(chimney)
16
17 tree = Polygon(Point(50,80), Point(30,140), Point(70,140))
18 tree.setFillColor('darkGreen')
19 paper.add(tree)
20
21 smoke = Path(Point(155,70), Point(150,65), Point(160,55), Point(155,50))
22 paper.add(smoke)
23
24 sunraySW = Path(Point(225,75), Point(210,90))
25 sunraySW.setBorderColor('yellow')
26 sunraySW.setBorderWidth(6)
27 paper.add(sunraySW)
28 sunraySE = Path(Point(275,75), Point(290,90))
29 sunraySE.setBorderColor('yellow')
30 sunraySE.setBorderWidth(6)
31 paper.add(sunraySE)
32 sunrayNE = Path(Point(275,25), Point(290,10))
33 sunrayNE.setBorderColor('yellow')
34 sunrayNE.setBorderWidth(6)
35 paper.add(sunrayNE)
36 sunrayNW = Path(Point(225,25), Point(210,10))
37 sunrayNW.setBorderColor('yellow')
38 sunrayNW.setBorderWidth(6)
39 paper.add(sunrayNW)

```

FIGURE 3.10: Complete source code for drawing our house (continued on next page).

```

40 grass = Rectangle(300, 80, Point(150,160))
41 grass.setFillColor('green')
42 grass.setBorderColor('green')
43 grass.setDepth(75) # must be behind house and tree
44 paper.add(grass)
45
46 window = Rectangle(15, 20, Point(130,120))
47 paper.add(window)
48 window.setFillColor('black')
49 window.setBorderColor('red')
50 window.setBorderWidth(2)
51 window.setDepth(30)
52
53 roof = Polygon(Point(105, 105), Point(175, 105), Point(170,85), Point(110,85))
54 roof.setFillColor('darkgray')
55 roof.setDepth(30) # in front of facade
56 chimney.setDepth(20) # in front of roof
57 paper.add(roof)

```

FIGURE 3.10 (continuation): Complete source code for drawing our house.

### 3.3 Rotating, Scaling, and Flipping

By design, all of the `Drawable` objects can be rotated, scaled, or flipped. For these operations, the object's reference point has greater significance. It serves as a point of the shape that stays fixed during the transformation. Earlier, we used the analogy of a shape's reference point as the place where a thumbtack attaches it to a canvas. When an object is moved, the reference point moves relative to the canvas but stays in the same place within the context of the shape. The thumbtack and shape move together as they are stuck elsewhere on the canvas. The effect of rotating a square that is tacked through its center is quite different than if rotating that square when tacked near a corner.

In Section 3.2, we noted that each shape has a natural location for its initial reference point. For example, the reference point of a `Square`, `Circle`, `Rectangle`, or `Text` is initially at its center; for a `Polygon` or `Path`, it is initially the location of the first point; for an `Image` it is the top left corner. However, we can change the relative location of that reference point if we wish. All drawable objects support a method with the signature `adjustReference(dx, dy)`. A call to this method behaves as follows. The shape is not moved at all relative to the canvas. Instead, the thumbtack holding it is taken out and moved by the specified amount in each direction, and then stuck back through the shape to hold it in its place. This call has no immediately visible effect on the screen. But it can be used as a precursor to a later rotation or scaling to achieve desired results.

We should note that the reference point is not actually required to lie within the boundary of the shape; in this way, our thumbtack analogy breaks down somewhat. For example, we could choose to move the reference point for a circle to be 100 pixels to its left, making it easier to animate the moon orbiting around the earth.

Rotating

With the preceding discussion of the reference point, we are ready to discuss rotation. Formally, drawable objects support a `rotate` method that takes a parameter specifying the clockwise rotation measured in degrees. That value can be an integer or floating-point number. As an example, we can rotate a `Square` 20° about its center point as follows:

```
>>> diamond = Square(40, Point(100,100))
>>> diamond.rotate(20)
```

This rotates the square, leaving its center point fixed. If we want to rotate the square about its bottom left corner, we need to first adjust the reference point. For example,

```
>>> block = Square(40, Point(100,100))
>>> block.adjustReference(-20, 20)
>>> block.rotate(20)
```

Both of these scenarios are diagrammed in Figure 3.11, with the original and new positions superimposed and the reference point highlighted.

Scaling

For the most basic shapes, we already have specialized methods to accomplish scaling. For example, we can directly modify the radius of a circle, or the width and height of a rectangle. Yet there is a more general approach that applies to all drawable objects. Each supports a method `scale` that takes a single parameter specifying a (positive) multiplicative factor by which the object is to be scaled. If that factor is greater than one, it causes the size to increase; when the factor is less than one it causes the size to decrease. For intricate shapes, the use of the `scale` method can help simplify otherwise complicated geometry.

When performing a scale, the reference point remains fixed. All other points in the shape are scaled relative to that reference point. To demonstrate the effect, Figure 3.12 shows two examples of a pentagon being scaled. In each example, the pentagon is scaled by a factor of 0.5. Yet in the first case, it is scaled with the center as the reference point; in the second case, the rightmost corner serves as the reference point.

Flipping

Another convenient transformation is to take a mirror image of an object. For this reason, drawable objects support a `flip()` method. By default, this causes a flip to take place across a *vertical axis of symmetry* passing through the reference point. In effect, this causes a left-to-right, right-to-left flip as shown in the first part of Figure 3.13.

To allow a flip in an arbitrary direction, the method accepts an optional parameter that specifies the clockwise rotation of the axis of symmetry away from vertical. For example the call `flip(10)` produces the result shown on the right side of Figure 3.13, with a slightly askew axis of symmetry. As is the case with the other transformation, notice that the reference point always remains fixed by this operation.

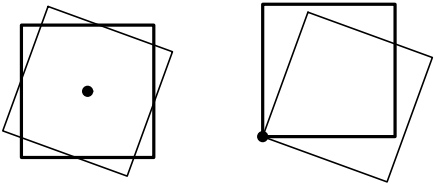


FIGURE 3.11: Rotating a square 20° in the clockwise direction. In both diagrams the original position is drawn with darker border and the resulting position with lighter border. In the case on the left, the square is rotated about its center point. On the right, the same square is rotated about its lower left corner.

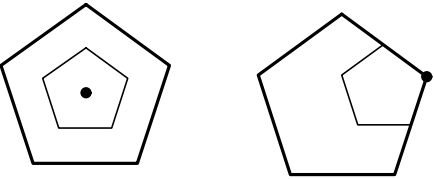


FIGURE 3.12: Scaling a pentagon about two different reference points. In both diagrams the original position is drawn with darker border and the resulting position with lighter border. On the left, the pentagon is scaled relative to its center point. On the right, the pentagon is scaled about the rightmost corner.

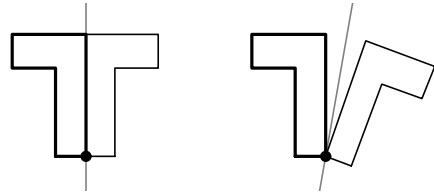


FIGURE 3.13: Flipping a flag about two different axes of symmetry. In both diagrams the original position is drawn with darker border and the resulting position with lighter border. On the left, the flag is flipped about the vertical axes by default with `flip()`. The right shows the result of `flip(10)`.

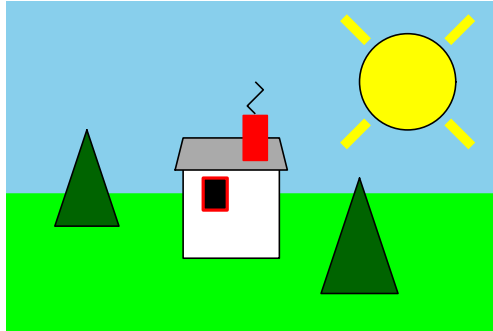


FIGURE 3.14: A scene with two trees; the right tree is drawn larger than the left.

### 3.4 Cloning

Although an exact copy of an existing shape could presumably be constructed and configured from scratch, the `Drawable` types support a convenient clone method that returns a brand new copy. The clone has precisely the same settings as the original element, but is not automatically added to any canvases. Once created, the clone can be manipulated independently of the original. Figure 3.14 shows an updated version of our scene with a cloned tree that has the same shape but a different location and size from the original.

```
otherTree = tree.clone()
otherTree.move(170,30)
otherTree.scale(1.2)
paper.add(otherTree)
```

### 3.5 Case Study: Smiley Face

Many interesting pictures can be drawn using the techniques we have learned thus far; we encourage you to have some fun. As an example, we draw the “smiley face” shown in Figure 3.15 using the code in Figure 3.16. We start with a yellow circle with black border for the overall head. The crescent-shaped mouth is a clever illusion. There is no shape in the package for representing a crescent directly. Instead, the visual effect is created with depths. A black circle is drawn on top of the original face (see lines 9–13), then a well-placed yellow circle on top of that black circle (see lines 15–19). The remaining facial features are drawn with the default depth, and thus appear in front of the mouth.



FIGURE 3.15: A smiley face drawn using the graphics package.

```
1 from cs1graphics import *
2 paper = Canvas()
3
4 head = Circle(75, Point(100,100))
5 head.setFillColor('yellow')
6 head.setDepth(60)
7 paper.add(head)
8
9 mouth = Circle(40, Point(100,110))
10 mouth.setFillColor('black')
11 mouth.setBorderWidth(0)
12 mouth.setDepth(52)
13 paper.add(mouth)
14
15 mouthCover = Circle(40, Point(100,100))
16 mouthCover.setFillColor('yellow')
17 mouthCover.setBorderWidth(0)
18 mouthCover.setDepth(51)
19 paper.add(mouthCover)
20
21 nose = Polygon(Point(100,90), Point(92,110), Point(108,110))
22 nose.setFillColor('black')
23 paper.add(nose)
24
25 leftEye = Circle(10, Point(70,80))
26 leftEye.setFillColor('black')
27 rightEye = Circle(10, Point(130,80))
28 rightEye.setFillColor('black')
29 paper.add(leftEye)
30 paper.add(rightEye)
```

FIGURE 3.16: Code to draw a smiley face (continued on next page).

```
31 leftEyebrow = Path(Point(60,65), Point(70,60), Point(80,65))
32 leftEyebrow.setBorderWidth(3)
33 leftEyebrow.adjustReference(10,15) # set to center of left eyeball
34 leftEyebrow.rotate(-15)
35 paper.add(leftEyebrow)
36
37 rightEyebrow = leftEyebrow.clone( )
38 rightEyebrow.flip( ) # still relative to eyeball center
39 rightEyebrow.move(60,0) # distance between eyeball centers
40 paper.add(rightEyebrow)
```

FIGURE 3.16 (continuation): Code to draw a smiley face.

The eyebrows are created using a combination of techniques. We start at lines 31 and 32, creating the left eyebrow as a thickened path which is centered directly above the left eye. To achieve a crooked appearance, we then rotate the eyebrow counterclockwise about the center of the eye itself. This requires careful use of the eyebrow’s reference point. Upon construction, the path’s reference point is aligned with the first declared point of that path: (60,65) in this case. Since the left eye is centered at (70,80) we adjust the eyebrow’s reference point by (10,15) at line 33. Once that has been done, we rotate 15° in the counterclockwise direction at line 34.

Though we could create the right eyebrow using a similar technique, we instead make a clone of the left eyebrow and then flip that clone horizontally. Because the reference point for the brow was already re-aligned with the center of the eyeball, the flip at line 38 causes the new brow to be cocked to the right rather than the left. However the new brow is still located near the left eye. Line 39 translates the new brow rightward, precisely the distance separating the two eyes. This causes the right eyebrow to be aligned above the right eye rather than the left eye.

3.6 Layers

Our next class is an extremely valuable tool (and one of our favorites). It allows us to treat a collection of other elements as a single composite object that we term a *Layer*. For motivation, let’s add a car to our earlier scene. Visually, we could achieve the image of a car perhaps by using three separate shapes: the car body and two tires that appear nearer than the body. However, if we want to move the “car” elsewhere we have to move the body and each tire. Separately moving each piece the same amount is time consuming and error prone. We might accidentally forget to move a tire, or move it by the wrong amount. Things get even more complicated when working out the correct geometry for scaling or rotating the car or when working with a more intricate design using more elements. A much better programming technique is to group those related shapes into a single composite object. In this section, we introduce a *Layer* class for this purpose. A *Layer* is sort of a hybrid of a *Canvas* and a *Drawable*. It serves as a container for other shapes while also serving as a *Drawable* object that can itself be added to a *Canvas* (or even to another *Layer*).

Earlier, we explained our drawing package through the analogy of physical shapes being attached to a canvas. We wish to extend this analogy to layers. We consider a layer to be a thin clear film. We can attach many shapes directly to this film rather than to the canvas. Of course the film itself can be positioned over the canvas and combined with other shapes to make a complete image. In fact, this is precisely the technology that advanced the creation of animated movies in the early 1900s. Rather than drawing the artwork on a single piece of paper, different layers of the background and characters were separated on clear sheets of celluloid. These were commonly called animation cels. The advantages were great. The individual cels could still be layered, with appropriate depths, to produce the desired result. Yet each cel could be moved and rotated independently of the others to adjust the overall image. For example, a character could be on a cel of its own and then moved across a background in successive frames.

We demonstrate the use of the *Layer* class by adding a car to our earlier house scene. Before concerning ourselves with the overall scene, we compose the layer itself. A layer is somewhat similar to a canvas in that it has its own relative coordinate system. Shapes can be added to the layer, in which case the shape’s reference point determines the placement of that shape relative to the *layer’s* coordinate system (i.e., the shape is tacked to a position on the layer). We use the origin of the layer as a landmark when placing individual elements. A sketch of our suggested geometry for the components of a car is given in Figure 3.17. We place the car so that the tires rest on the x-axis and so that the car is centered horizontally about the origin. We build this model using the code in Figure 3.18. Three individual parts are added to a new layer with coordinates based upon our sketch. When placed on a layer, a shape’s depth determines how that shape appears relative to other shapes on the same layer. We set the body’s depth to 60 so that it appears behind the two tires, which have default depth of 50.

At this point, we have created our layer but not yet displayed it. Yet the layer is itself a *Drawable* object, and so it supports all of the familiar behaviors. If we add the layer to a canvas, the reference point of the layer (its origin, by default) represents where that layer is tacked upon the canvas. A layer has its own depth attribute, used to determine whether

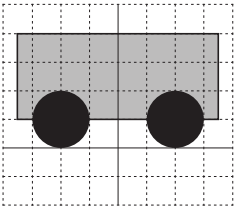


FIGURE 3.17: Geometric sketch of a car, within the coordinate system of a layer. The solid grid lines represent the x-axis and y-axis. The dashed grid lines designate a separation of 10 pixels each.

```

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)          # behind the tires
car.add(body)

```

FIGURE 3.18: Code to create a Layer that represents a car.

the layer is considered to be above or below other objects on the canvas. So to add the car to our house scene, we position it properly relative to the canvas's coordinate system. This is accomplished with the following:

```

car.moveTo(110,180)
car.setDepth(20)
paper.add(car)

```

The resulting image is portrayed in Figure 3.19. It is important to keep straight the distinction between the canvas's frame of reference and the layer's frame of reference. The car's placement on the canvas was made according to the coordinate (110, 180) of the canvas. More specifically it is the reference point of the layer (the hypothetical thumbtack through the layer's origin) that is tacked to the canvas at that coordinate. This example also demonstrates an important subtlety in the use of depths. Recall that when building the layer we left the tires with the default depth of 50 and gave the car body a depth of 60. That is why the body appears behind the tires. However, you will notice that the car body appears in front of the house facade, even though that facade has depth 50. The reason is that the depths of individual components are only relevant when compared to other elements in the same context. There are three objects added to the layer and their depths distinguish how those objects are interleaved. But the placement of the layer as a whole onto the canvas is based upon the depth attribute of the layer object. In this case we intentionally set the layer's depth to 30 so that the entire car appears nearer than the house.

As a `Drawable` object, a layer can be rotated, flipped, cloned, or scaled about its reference point. For example, if we were to scale our car, that imaginary thumbtack through the origin of the layer remains fixed, and so the bottom of the tires remain at the same y-coordinate. So if our car appears to be resting on top of a street or driveway, it will remain so when scaled. Similarly, a rotated car could be placed properly on a hill by placing its origin on the edge of the hill.

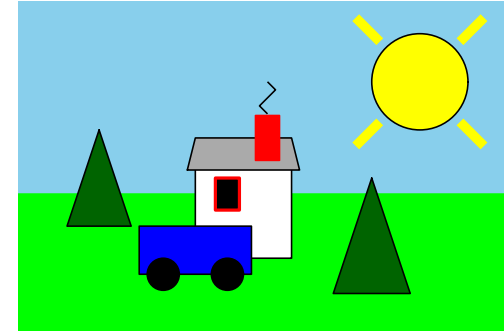


FIGURE 3.19: Adding a car in front of our house.

### 3.7 Animation

Most of our efforts thus far have been used to design a single static image, albeit with increasing complexity. Yet graphics can be used to generate intricate animations, involving multiple scenes and moving objects. Creating an animation can be straightforward. In this section we address several new issues that arise in the context of animation.

#### Controlling timing

When working within an interactive interpreter session, we see the immediate effect of each command displayed on the screen. However, if we type a long series of graphical commands into a file and execute that file, chances are that all of the intermediate stages will go by in a blink of an eye. For example, we might wish to move a car across the screen using a series of calls to the `move` method. Yet the computer is rather quick and may display the images too fast for the eye to see.

To better control the images seen by a viewer, we can use a `sleep` function that is part of Python's standard `time` library. We can load this module using the command `from time import sleep`, as originally described in Section 2.7. Once the library is loaded, a call to the `sleep` function causes the program to wait a given number of seconds before proceeding. For example, `sleep(1.5)` will wait one and a half seconds before proceeding. This command can be used not just in graphical programs, but as part of any software. Within the context of graphics, a pause can be used to give the impression of motion. Consider the following code fragment to move our car.

```

from time import sleep
timeDelay = .25          # one-quarter second
car.move(-10, 0)
sleep(timeDelay)
car.move(-30, 0)
sleep(timeDelay)
car.move(-60, 0)
sleep(timeDelay)
car.move(-100, 0)
sleep(timeDelay)

```

The speed of the animation can be adjusted by altering the time delay or by having a longer series of motion. Another use for `sleep` is to produce an animation with multiple frames. We can develop an intricate scene, and then intentionally pause so that the user can take in the scene for a certain amount of time. After the pause, additional code can alter portions of the scene or even compose an entirely new scene for the viewer.

#### Controlling flicker

There is a drawback in trying to create multiframe animations as we have described. When there are a large number of shapes, or a significant number of separate changes that take place from one frame to the next, there is the possibility that the user may see some partially composed images that we may not intend to have viewed. For example, if one command moves a character’s head and the next moves the body, the viewer might glimpse the fraction of a second in which the head is detached. Although the viewer may not know exactly what happened, this sort of “flicker” can be distracting.

The use of a layer was one approach to have such motion coordinated, but if several different changes in a scene are to be made, there may still be unwanted artifacts. A common technique for trying to minimize flicker in computer graphics is called **double buffering**. If we need to make ten separate changes when moving from one frame to the next, the idea is the following: don’t change the displayed image at all in the interim. While the original frame is still in view, we would like to perform all ten of the hypothetical changes behind the scene. Only after the next frame is computed internally, should it be displayed on the screen for the viewer.



## FOR THE GURU

In this chapter, we create motion and frames in an ad hoc way, manually repeating commands to produce the effect. This is not really a good programming style. In Chapter 4, we will introduce a technique known as a *loop* for more elegantly expressing the repetition of commands. Loops can be extremely useful when creating animations.

The `cs1graphics` package supports a form of double buffering as follows. Until now, we have relied upon the default behavior of a Canvas, which is to immediately display each change to the screen. We refer to this behavior as “auto-refresh” mode. Although it is the default, we can change it by using the `setAutoRefresh` method. This takes a single boolean parameter. If we call `paper.setAutoRefresh(False)`, this turns off the automatic refresh mode and instead reverts to a manual refresh mode. When in this mode, no changes are rendered to the screen unless an explicit call to `paper.refresh()` is made. In the context of an animation, we can enact all of the changes that we want to make behind the scene and then refresh the canvas (and perhaps pause) each time we want to show a new frame to the viewer. To demonstrate this mode, try the following in an interactive session.

```

>>> paper = Canvas(200, 200, 'skyBlue')
>>> ball = Circle(25, Point(100,75))
>>> paper.add(ball)          # ball immediately appears
>>> ball.setRadius(50)       # ball immediately changes
>>> paper.setAutoRefresh(False)
>>> ball.setFillColor('red')  # no visible change
>>> ball.move(50,0)           # no visible change
>>> paper.refresh()          # image is updated

```

When there are relatively few objects in the scene, or relatively few changes being made at a time, there is really not much difference between the quality of automatic refresh mode and manual refresh. However, if you ever start to notice artifacts like flicker in complicated animations, you might try switching to manual refresh. If you want, you can always turn automatic refresh back on using a syntax such as `paper.setAutoRefresh(True)`.

## 3.8 Graphical User Interfaces

When using purely text-based software, the only way a user can interact with a program is by entering text into the console, and only when prompted to do so. For most modern software, users interact with programs through a combination of keyboard strokes and mouse movements and clicks. Users select from menus, click on buttons, and enter text within boxes. Such an interface is known as a **graphical user interface**, or GUI for short (pronounced “gooey”). In this section, we outline some of the basic support for building simple GUIs using the `cs1graphics` package.

One of the most important issues for a graphical interface is being able to detect a user *event*, such as a mouse click or a keystroke. We will explore this concept of **event-driven programming** as an advanced topic in Chapter 15, but for now we wish to introduce a few simple features that you can use right away. All of our `Drawable` objects support a method, named `wait()`, that causes the program to pause indefinitely until an event is detected upon the given object. To demonstrate the technique, we provide the following very simple example.



```
1 paper = Canvas( )
2 light = Circle(20, Point(100,100))
3 light.setFill( 'red' )
4 paper.add(light)
5 light.wait( )
6 light.setFill( 'green' )
```

The key to this example is line 5. When this command is executed, the flow of the program waits indefinitely. You will not immediately see the light change to green. The program is waiting for the user to trigger an event, such as a mouse click, upon the circle. After such a click, the program continues where it left off, causing the light to turn green.

Because wait() was specifically called upon the circle instance, the program only continues with a click upon that circle. Mouse clicks elsewhere on the canvas will be ignored. It is possible to wait upon the canvas as a whole. For example, the above program could be augmented with the final two lines.

```
7 paper.wait( )
8 paper.close( )
```

In this case, after turning the light green it waits for another event, this time anywhere on the canvas. Note that such an event does not necessarily need to be on the background; clicking on any part of the canvas (including the circle) suffices.

In the first example, we used wait to control the timing of our program, but we did not take notice of the type of event or the location of the event. However, the wait() method provides a return value. To see this, try the following interactively in the interpreter:

```
>>> paper = Canvas()
>>> paper.wait()
```

After typing the second command, the program is waiting for us to trigger an event (notice that we do not yet have a subsequent prompt). If we go ahead and click on the canvas, we get the following continuation:

```
<cs1graphics.Event object at 0x6d52f0>
>>>
```

That return value is an instance of the Event class. Each event stores information about the type of event that occurred (e.g., mouse, keyboard), and additional characteristics. For example, a mouse event stores the coordinates of where the click occurred and which button was used. A keyboard event indicates which key was pressed. Figure 3.20 provides a brief overview of the accessors supported by the Event class. We will explore use of events more fully in Chapter 15, but for now we offer a simple example. The getMouseLocation() method returns a Point that represents the location of the mouse at the time the event occurred. For example, here is a program that adds a circle to the canvas centered at the indicated location.

Event	
getDescription( )	getTrigger( )
getMouseLocation( )	getKey( )

FIGURE 3.20: The Event class.

```
paper = Canvas( )
cue = paper.wait( ) # wait indefinitely for user event
ball = Circle(10, cue.getMouseLocation( ))
ball.setFill( 'red' )
paper.add(ball)
```

Widgets

We note in passing that the cs1graphics package includes several additional classes to support graphical user interfaces. Typically these tools are known as *widgets*. A TextBox allows a user to enter text that can later be retrieved from the program, and a Button serves as a clickable rectangle with a label. More complete documentation on their use is available online, and we will discuss their implementations in later chapters. For now, we provide the following brief demonstration.

```
paper = Canvas( )
nameInput = TextBox(150, 50, Point(100,10))
paper.add(nameInput)
submit = Button( 'Enter name', Point(100,80))
paper.add(submit)
submit.wait( )
welcome = Text( 'Hello, ' + nameInput.getMessage( ))
welcome.move(100, 150)
paper.add(welcome)
```

This interface displays a textbox and a button. When the button is pressed, it incorporates the characters previously entered into the text box into a newly displayed greeting.



A WORD OF WARNING

If you call wait() on an object such as a circle that has a transparent interior, it will only respond to clicks that take place *on the border*.

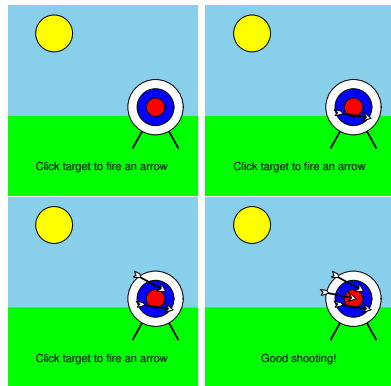


FIGURE 3.21: A few frames from the animation.

### 3.9 Case Study: Flying Arrows

We will combine the skills that we have learned to create an animated scene. It consists of a target and arrows flying at the target whenever the canvas is clicked on; see Figure 3.21 for a few images from the animation. The complete code for the program is given in Figure 3.22. In lines 1–13 we begin by setting up a sky blue canvas with grass and a sun. Next, in lines 15–33, we assemble the target as a separate layer. The local geometry of the target is designed so that the center of the bullseye coincides with the origin of the layer's coordinate system. Three concentric circles are added so that smaller circles appear in front of larger circles, with a path used to represent the legs holding up the target. Finally, the entire layer is added to the scene and positioned accordingly.

In our eventual animation, we will shoot three separate arrows. However, since all of the arrows have the same geometry and initial position, we create a model for the first (lines 36–48) and then clone it (lines 49 and 50). Each arrow is a layer with three components: a shaft, a tip and a fletching. The geometry is designed with the arrow facing right and with the tip of the arrow aligned with the origin of the layer. So when we rotate the arrow, the tip serves as the reference point. We note that none of the arrows are in the scene yet. We will add them one at a time, based upon the user's interactions.

To control the animation, we rely upon use of the wait method. For example, the call to `target.wait()` at line 56 causes the entire program to delay indefinitely. The program will continue only after the user clicks the mouse on the target. Since the target is a layer, the program recognizes a click on any portion of that target (including the legs). That causes the first arrow to be added to the scene (line 57) and the subsequent animation (lines 58–70)

```

1 from cs1graphics import *
2 from time import sleep
3
4 scene = Canvas()
5 scene.setBackgroundColor('skyBlue')
6 grass = Rectangle(200, 80, Point(100,160))
7 grass.setFill('green')
8 grass.setBorderColor('green')
9 grass.setDepth(100)
10 scene.add(grass)
11 sun = Circle(20, Point(50,30))
12 sun.setFill('yellow')
13 scene.add(sun)
14
15 target = Layer()
16 outside = Circle(30)
17 outside.setFill('white')
18 outside.setDepth(49)
19 target.add(outside)
20 middle = Circle(20)
21 middle.setFill('blue')
22 middle.setDepth(48)
23 target.add(middle)
24 inside = Circle(10)
25 inside.setFill('red')
26 inside.setDepth(47)
27 target.add(inside)
28 legs = Path(Point(-25,45), Point(0,0), Point(25,45))
29 legs.setBorderWidth(2)
30 target.add(legs)
31 target.move(160,110)
32 target.setDepth(75) # in front of grass; behind arrows
33 scene.add(target)
34
35 # prepare three arrows, but do not yet add to scene
36 arrow1 = Layer()
37 tip = Polygon(Point(0,0), Point(-8,5), Point(-5,0), Point(-8,-5))
38 tip.setFill('white')
39 arrow1.add(tip)
40 shaft = Path(Point(-30,0), Point(-5,0))
41 shaft.setBorderWidth(2)
42 shaft.setDepth(51)
43 arrow1.add(shaft)

```

FIGURE 3.22: The code for the arrow scene (continued on next page).

```

44 fletching = Polygon(Point(-30,0), Point(-33,-3), Point(-40,-3),
45                     Point(-36,0), Point(-38,3), Point(-36,3))
46 fletching.setFill('white')
47 arrow1.add(fletching)
48 arrow1.move(15,120)      # initial position
49 arrow2 = arrow1.clone()
50 arrow3 = arrow1.clone()
51
52 dialogue = Text('Click target to fire an arrow')
53 dialogue.move(100,170)
54 scene.add(dialogue)
55
56 target.wait()            # wait indefinitely for user event on target
57 scene.add(arrow1)
58 arrow1.rotate(-20)
59 sleep(0.25)
60 arrow1.move(41,-15)
61 arrow1.rotate(7)
62 sleep(0.25)
63 arrow1.move(41,-5)
64 arrow1.rotate(7)
65 sleep(0.25)
66 arrow1.move(41,5)
67 arrow1.rotate(7)
68 sleep(0.25)
69 arrow1.move(41,17)
70 arrow1.rotate(7)
71
72 target.wait()            # wait indefinitely for user event on target
73 scene.add(arrow2)
74 arrow2.rotate(-40)
75 sleep(0.25)
76 arrow2.move(39,-22)
77 arrow2.rotate(17)
78 sleep(0.25)
79 arrow2.move(39,-12)
80 arrow2.rotate(17)
81 sleep(0.25)
82 arrow2.move(39,3)
83 arrow2.rotate(17)
84 sleep(0.25)
85 arrow2.move(39,13)
86 arrow2.rotate(17)

```

FIGURE 3.22 (continuation): The code for the arrow scene (continued on next page).

```

87 scene.add(arrow3)
88 arrow3.rotate(-30)
89 sleep(0.25)
90 arrow3.move(37,-26)
91 arrow3.rotate(10)
92 sleep(0.25)
93 arrow3.move(37,-11)
94 arrow3.rotate(10)
95 sleep(0.25)
96 arrow3.move(37,6)
97 arrow3.rotate(10)
98 sleep(0.25)
99 arrow3.move(37,21)
100 arrow3.rotate(10)
101 dialogue.setMessage('Good shooting!')
102
103 scene.wait()             # wait for user event anywhere on canvas
104 scene.close()

```

FIGURE 3.22 (continuation): The code for the arrow scene.

as it moves toward the target. The second and third arrows are handled in similar fashion in lines 72–101, although with slightly different flights. Finally, we call `scene.wait()` at line 103, before closing the window. This has a similar effect to the earlier delays, but since `wait` is being invoked on the canvas as a whole, the user may click anywhere to continue.

To create a smoother animation we could have more frames with a smaller time step. Unfortunately, we would need many more lines of code if using the current programming style. In the next chapter, we will learn new techniques that could be used to better automate the animation of the arrow. Notice as well that the flight of the arrows is predetermined in our program; it has no relation to the location of the mouse click (though an interesting challenge for an exercise).



## FOR THE GURU

Python supports use of several graphics libraries. Our `cs1graphics` package is based upon another common library known as *Tkinter*. While `cs1graphics` is useful for many purposes, `Tkinter` is designed to support more industrial applications.

3.10 Chapter Review

3.10.1 Key Points

Graphics Primitives

- Creating a Canvas instance constructs a window. Various objects can be visualized by calling the add method of the canvas.
- The window for a canvas does not close unless the user manually closes the window through the operating system or the close method of the canvas is called.
- Coordinates are measured from the top left corner of the window. The x-coordinate specifies the number of pixels to the right of this point and the y-coordinate specifies the number of pixels below this point.
- Only Drawable objects can be added to a canvas.

Modifying Drawable Objects

- There are mutators to change the location, color, border color, position, and other features for each drawable object.
- A drawable object can be rotated, scaled, or flipped relative to its reference point. This reference point can be reconfigured using the adjustReference member function.

Depths

- When two or more drawable objects overlap on a canvas, the relative ordering of those objects is determined according to their specified depth attribute. Shapes with smaller depths are drawn in front of those with larger depths.

Layers

- A Layer represents a collection of objects that is treated as a single shape. They layer can be added, moved, rotated or scaled, just as with any other Drawable instance.
- Depths of objects within a layer only affects how the objects in the layer appear relative to each other. The depth of the layer controls whether all the shapes in that layer appears in front or behind objects that are not part of the layer.

Animation

- A time delay for an animation can be achieved by calling the sleep function imported from the time module.
- Flicker can be reduced in an animation by turning auto-refresh off for the canvas and calling refresh each time you want the canvas’s image rendered to the screen.

Events

- Calling the wait( ) method of the Canvas class causes the program to wait indefinitely until the user triggers an event on the window, such as a mouse click or keypress.
- Calling the wait( ) method on an individual drawable object waits until the user triggers and event specifically upon that particular object.
- The wait( ) function returns an Event instance that contains information about which mouse button or key was pressed, and the cursor location at that time.

3.10.2 Glossary

- canvas** A graphics window on which objects can be drawn.
- clone** A copy of an object.
- double buffering** A technique for avoiding flicker in animation by computing incremental changes internally before displaying a new image to the viewer.
- event** An external stimulus on a program, such as a user’s mouse click.
- event-driven programming** A style in which a program passively waits for external events to occur, responding appropriately to those events as needed.
- graphical user interface (GUI)** A design allowing a user to interact with software through a combination of mouse movements, clicks, and keyboard commands.
- pixel** The smallest displayable element in a digital picture.
- reference point** A particular location for a cs1graphics.Drawable instance that is used when determining the placement of the object upon a canvas’s coordinate system. The reference point remains fixed when the object is scaled, rotated, or flipped.
- RGB value** A form for specifying a color as a triple of integers representing the intensity of the red, green, and blue components of the color. Typically, each color is measured on a scale from 0 to 255.
- widget** An object that serves a particular purpose in a graphical user interface.

3.10.3 Exercises

Graphics Primitives

Practice 3.1: The following code fragment has a single error in it. Fix it.

```
1 from cs1graphics import *
2 screen = Canvas( )
3 disk = Circle( )
4 disk.setFillColor('red')
5 disk.add(screen)
```

Practice 3.2: Write a program that draws a filled triangle near the middle of a canvas.

For Exercise 3.3 through Exercise 3.7, try to find the errors *without* using a computer. Then try running them on the computer to help find the errors or confirm your answers.

Exercise 3.3: After starting Python, you immediately enter

```
can = Canvas(100,100)
```

Python reports an error with the last line saying

```
NameError: name 'Canvas' is not defined
```

What did you do wrong? How do you fix it?

**Exercise 3.4:** Assuming that you have already created an instance of the `Square` class called `sq`, what’s wrong with the statement

```
sq.setFillColor(Red)
```

Give two different ways to fix this statement.

**Exercise 3.5:** Assuming that you have already successfully created a `Canvas` instance called `can`, you enter the following to draw a blue circle centered in a red square:

```
sq = Square( )
sq.setSize(40)
sq.moveTo(30,30)
sq.setFillColor('Red')
can.add(sq)

cir = Circle( )
cir.moveTo(50,50)
cir.setRadius(15)
cir.setFillColor('Blue')
can.add(cir)
```

But the circle never appears. What’s wrong with the above program? Edit the program so it works as desired.

**Exercise 3.6:** Consider the following:

```
can = Canvas(200,150)

rect = Rectangle( )
rect.setWidth(50)
rect.setHeight(75)
rect.moveTo(25,25)

rect = Rectangle( )
rect.setWidth(100)
rect.setHeight(25)

can.add(rect)
can.add(rect)
```

Only one rectangle appears? Why? How would you get two different rectangles to show up? (There are several ways to fix this.)

### Layers

**Exercise 3.7:** The following runs but does not display anything. What is wrong?

```
can = Canvas( )
lay = Layer( )
sq = Square( )
lay.add(sq)
```

**Exercise 3.8:** Use the `Layer` class of the graphics library to create a pine tree. Make copies of the tree and use it to draw a forest of pine trees.

**Exercise 3.9:** Redo the smiley face described in the chapter as a layer. Test the code by rotating and scaling the layer and ensuring the face does not get distorted.

**Exercise 3.10:** Create an airplane as a layer and animate it flying across the screen and doing a loop. Hint: think carefully about the placement of the reference point.

### Graphics Scenes

**Exercise 3.11:** Use the graphics library to create a picture of your favorite animal with an appropriate background scene.

**Exercise 3.12:** Use the graphics library to draw an analog clock face with numbers at the appropriate locations and an hour and minute hand.

### Events

**Exercise 3.13:** Display the text "Click Me" centered in a graphics canvas. When the user clicks on the text, close the canvas.

**Exercise 3.14:** Create a program that draws a traditional traffic signal with three lights (green, yellow, and red). In the initial configuration, the green light should be on, but the yellow and red off (that is, black). When the user clicks the mouse on the signal, turn the green off and the yellow on; when the user clicks again, turn the yellow off and the red on.

**Exercise 3.15:** Write a program that displays a `Text` object graphically and adds characters typed on the keyboard to the message on the canvas.

**Exercise 3.16:** Write a program that allows the user to draw a path between five points, with each point specified by a mouse click on the canvas. As each successive click is received, display the most recent segment on the canvas.

### Projects

**Exercise 3.17:** Use the graphics library to draw an analog clock face with hour, minute, and second hands. Use the `datetime` module to start the clock at the current time and animate the clock so that it advances once per second.

**Exercise 3.18:** Write a game of Tic-tac-toe using a graphical interface. The program should draw the initial board, and then each time the mouse is clicked, determine the appropriate square of the game board for drawing an X or an O. You may allow the game to continue for each of the nine turns. (As a bonus, read ahead to Chapter 4 and figure out how to stop the game once a player wins.)

**Exercise 3.19:** Think of your own cool project and have fun with it.

# Photo processing with `cs1media`

Otfried Cheong, Edited by JinYeong Bak\*

February 29, 2016

## 1 First steps with `cs1media`

Suppose you have a photo taken with your digital camera on your computer, and upload it in elice. You can load this photo using:

```
from cs1media import *
img = load_picture("photos/geowi.jpg")
```

You have now created a picture object and assigned the name `img` to it. You can get some interesting information about the picture using the methods `size()` (which returns a tuple consisting of the width and height of the picture in pixels) and `title()` (which typically returns the filename). If you call the method `show()`, then Python will start a program to display the picture:<sup>1</sup>

```
>>> type(img)
<class 'cs1media.Picture'>
>>> img.size()
(500, 375)
>>> img.title()
'geowi.jpg'
>>> img.show()
```



## 2 Digital images

Digital images are stored as a rectangular matrix of tiny squares called *pixels* (picture elements). If the image has width `w` and height `h`, then there are

\*He updates this note for Python 3 and elice platform.  
Email: jy.bak@kaist.ac.kr

<sup>1</sup>'>>>' is the primary prompt in Python interactive mode. The elice platform do not support the interactive mode. To see the result, you use print function with each code or run it in your own computer.

`h` rows of pixels (numbered 0 to `h-1` from top to bottom) and `w` columns (numbered 0 to `w-1` from left to right).

The `cs1media` module represents each pixel as an *RGB color*, which is a triple of red, green, and blue light intensities. Each intensity is an integer between 0 (no light) and 255 (full light). So red is (255, 0, 0), and yellow is (255, 255, 0).

Many colors are predefined and can be accessed like this:

```
>>> Color.yellow
(255, 255, 0)
>>> Color.chocolate
(210, 105, 30)
```

To get a list of all predefined colors, use `help(Color)` when you have imported the `cs1media` module.

## 3 Color modifications

Looking at our photograph, it is rather dark. Let's make it lighter by increasing the light intensity of each pixel.

```
def make_lighter(img, factor):
    w, h = img.size()
    for y in range(h):
        for x in range(w):
            r, g, b = img.get(x, y)
            r = int(factor * r)
            g = int(factor * g)
            b = int(factor * b)
            if r > 255: r = 255
            if g > 255: g = 255
            if b > 255: b = 255
            img.set(x, y, (r, g, b))

img = load_picture("photos/geowi.jpg")
make_lighter(img, 1.5)
img.show()
```

To save a copy of the modified image, call

```
img.save_as("geowi_lighter.jpg")
```

When called as `make_lighter(img, 0.5)`, the function actually makes the picture *darker*.

With the same principle, we can create various image effects. For instance, we can create the sensation of a sunset by turning an image redder (by reducing the blue and green light intensity). We can create a negative image if we replace  $(r, g, b)$  by  $(255-r, 255-g, 255-b)$ :



To create a black-and-white image, we have to compute the “brightness” or *luminance* of each pixel. The total amount of light of a pixel seems to be  $r + g + b$ , but in fact this gives an unnatural result. The reason is that the human eye perceives blue to be much darker than green when the intensity of the light is the same. A good conversion to luminance is given by the following function:

```
def luminance(p):
    r, g, b = p
    return int(0.299*r + 0.587*g + 0.114*b)
```

Once we have the luminance  $v$  of a pixel, we can set it to the graytone pixel  $(v, v, v)$ .

Even though it’s called “black and white”, we have actually created graytone images. We can create images really using only black and white if we *threshold* the image: if the luminance is larger than some threshold, we set the pixel to white, otherwise to black:



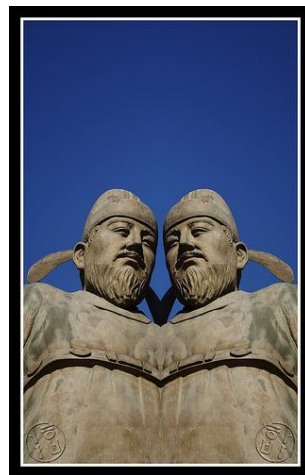
## 4 Reflections

When you look in a mirror, you see yourself mirrored. We can achieve this effect:

```
def mirror(img):
    w, h = img.size()
    for y in range(0, h):
        for x in range(w / 2):
            pl = img.get(x, y)
            pr = img.get(w - x - 1, y)
            img.set(x, y, pr)
            img.set(w - x - 1, y, pl)
```



Instead of reflecting the entire photo, we can take the left half, reflect it, and use it as the new right half:



It is interesting to apply this technique to photos of human faces, because faces are symmetric, but not perfectly so, and the symmetrized version looks interestingly different from the real face.



## 5 Pixelation

So far we modified every pixel of the photo independently. Let’s see what we can do if we look at several pixels together. For instance, we can take the average color for a group of pixels:

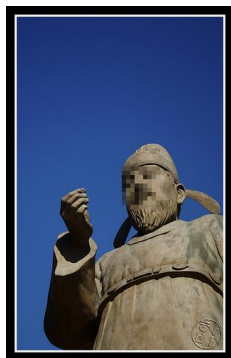


```
def blocks(img, step):
    w, h = img.size()
    norm = step * step
    for y in range(0, h - step, step):
        for x in range(0, w - step, step):
            # compute average
            r, g, b = 0, 0, 0
            for x0 in range(step):
                for y0 in range(step):
                    r0, g0, b0 = img.get(x+x0, y+y0)
                    r, g, b = r+r0, g+g0, b+b0
            r, g, b = r/norm, g/norm, b/norm
            # set block
            for x0 in range(step):
                for y0 in range(step):
                    img.set(x+x0, y+y0, (r, g, b))
```



This technique is often used to make part of an image unrecognizable, for instance car licence plates or faces on Google street view. To do that, we have to restrict the pixelation to a smaller rectangle of the photo:

```
def blocks_rect(img, step, x1, y1, x2, y2):
    w, h = img.size()
    for y in range(y1, y2 - step, step):
        for x in range(x1, x2 - step, step):
            # same as before
```



## 6 Blurring

If instead we compute a new color for each pixel by taking an average of a block around it, we can *blur* pictures. This is useful on pictures where pixels are visible because they have been magnified too much, etc. To implement this, we need two picture objects to work with, because we need the original pixel values to compute each new pixel:

```
def blur(img, radius):
    w, h = img.size()
    result = create_picture(w, h, Color.white)
    norm = (2 * radius + 1)**2
    for y in range(radius, h-radius):
        for x in range(radius, w-radius):
            # compute average
            r, g, b = 0, 0, 0
            for x0 in range(-radius, radius+1):
                for y0 in range(-radius, radius+1):
                    r0, g0, b0 = img.get(x+x0, y+y0)
                    r, g, b = r+r0, g+g0, b+b0
            r, g, b = r/norm, g/norm, b/norm
            result.set(x, y, (r, g, b))
    return result
```



## 7 Edge detection

The human visual system works by detecting *boundaries*. Even though natural objects like human faces have no sharp boundaries, we can easily recognize a child's line drawing of a face. When teaching a computer to recognize objects—an area called *computer vision*—one of the first steps is to recognize *edges* in the image.

A simple edge detection algorithm is the following: we compare the *luminances* of a pixel and the pixels to its left and above. If there is a big enough difference in both directions, then we have found an “edge” pixel. We can visualize this easily by drawing edge pixels black, and all other pixels white:



```
def edge_detect(img, threshold):
    w, h = img.size()
    r = create_picture(w, h)
    for y in range(1, h-1):
        for x in range(1, w-1):
            pl = luminance(img.get(x-1,y))
            pu = luminance(img.get(x,y-1))
            p = luminance(img.get(x,y))
            if (abs(pl - p) > threshold and
                abs(pu - p) > threshold):
                r.set(x, y, Color.black)
            else:
                r.set(x, y, Color.white)
    return r
```



```
def interpolate(t, c1, c2):
    r1, g1, b1 = c1
    r2, g2, b2 = c2
    r = int((1-t) * r1 + t * r2)
    g = int((1-t) * g1 + t * g2)
    b = int((1-t) * b1 + t * b2)
    return (r, g, b)

def gradient(img, c1, c2):
    w, h = img.size()
    for y in range(h):
        t = float(y) / (h-1) # 0 .. 1
        p = interpolate(t, c1, c2)
        for x in range(w):
            img.set(x, y, p)

img = create_picture(100, 100)
gradient(img, Color.yellow, Color.red)
```



## 8 Artificial images

So far we have modified pixel values based on the value of the pixels. We can also create entirely artificial images by setting pixels to whatever value we like. Let's first put our geese behind prison bars:

```
def prison(img, step):
    w, h = img.size()
    for y in range(0, h, step):
        for x in range(w):
            img.set(x, y, (0, 0, 0))
```



And now an entirely artificial image:

## 9 Cropping

We often need to transform photos in various ways. Cropping, for instance, means to cut out the part of the photo we are really interested in:



```
def crop(img, x1, y1, x2, y2):
    w1 = x2 - x1
    h1 = y2 - y1
    r = create_picture(w1, h1)
    for y in range(h1):
        for x in range(w1):
            r.set(x, y, img.get(x1+x, y1+y))
    return r

img = load_picture("photos/geowi.jpg")
r = crop(img, 67, 160, 237, 305)
```

Sometimes we are given a photo with a frame (like our statue above). We can leave it to the computer to find the right coordinates for cropping—all we need to do is remove the part of the photo that is similar in color to the frame color.

We first need a function to compare two colors:

```
def dist(c1, c2):
    r1, g1, b1 = c1
    r2, g2, b2 = c2
    return math.sqrt((r1-r2)**2 +
                     (g1-g2)**2 +
                     (b1-b2)**2)
```

The following function will then find the right x1 coordinate for cropping:

```
def auto_left(img, key, tr):
    w, h = img.size()
    x = 0
    while x < w:
        for y in range(h):
            if dist(img.get(x, y), key) > tr:
                return x
        x += 1
    return None
```

We do exactly the same for the top, bottom, and right side of the photo, and can finally crop like this:

```
def autocrop(img, threshold):
    # use top-left color as key
    key = img.get(0, 0)
    x1 = auto_left(img, key, threshold)
    x2 = auto_right(img, key, threshold)
    y1 = auto_up(img, key, threshold)
    y2 = auto_down(img, key, threshold)
    return crop(img, x1, y1, x2, y2)
```

It turns out our statue actually has two frames, a black outer frame and a whitish inner frame:

```
r1 = load_picture("photos/statue.jpg")
r2 = autocrop(r1, 60)
r3 = autocrop(r2, 200)
```



## 10 Rotations

If we have taken a photo while holding our camera sideways, then we need to rotate the final photo:

```
def rotate(img):
    # swap w and h:
    h, w = img.size()
    r = create_picture(w, h)
    for y in range(h):
        for x in range(w):
            r.set(x, y, img.get(h - y - 1, x))
    return r
```



## 11 Scaling

Very often we want to have a photo in a different resolution than what we have taken. We create a new picture with the right size, and then need to compute the color of each pixel of the scaled image by looking at the pixels of the original image.

Here is a version that is similar to the pixelation idea we used above: we take the average color for a block of pixels from the original image.

```
def scale_down(img, factor):
    w, h = img.size()
    w /= factor
    h /= factor
    result = create_picture(w, h)
    norm = factor ** 2
    for y in range(h):
        for x in range(w):
            # compute average
            r, g, b = 0, 0, 0
            x1 = x * factor
            y1 = y * factor
            for x0 in range(factor):
                for y0 in range(factor):
                    r0, g0, b0 = img.get(x1+x0, y1+y0)
                    r, g, b = r+r0, g+g0, b+b0
            r, g, b = r/norm, g/norm, b/norm
            result.set(x, y, (r, g, b))
    return result
```

This function can only scale down by an integer factor. To scale down by other factors or to scale up needs a different way to decide the color of each

pixel of the scaled image, possibly using interpolation to create a pleasing result.

## 12 Collage

A collage is a picture created by using (pieces of) existing pictures. For instance, let's start with this photo:



We'll take our statue above and paste it in at a suitable location:

```
def paste(canvas, img, x1, y1):
    w, h = img.size()
    for y in range(h):
        for x in range(w):
            canvas.set(x1+x,y1+y, img.get(x,y))
```

```
def collage1():
    trees = load_picture("trees1.jpg")
    r1 = load_picture("statue.jpg")
    r2 = scale_down(r1, 2)
    mirror(r2)
    paste(trees, r2, 100, 20)
    trees.show()
```

This works, but the result is not too exciting:



## 13 Chromakey

The background of the statue doesn't look natural. Can we get rid of this background? In this case, this is actually very easy, as the color of the background is very different from the color of the statue. We use the *chromakey* technique, used in the entertainment industry (<http://en.wikipedia.org/wiki/Chromakey>). Let's first try it by marking the statue's background in yellow:

```
def chroma(img, key, threshold):
    w, h = img.size()
    for y in range(h):
        for x in range(w):
            p = img.get(x, y)
            if dist(p, key) < threshold:
                img.set(x, y, Color.yellow)
```



Now all we need is an improved version of the `paste` function that will skip background pixels when copying an image onto the canvas. We need to pass the *color key* which we used for the background to the function:

```
def chroma_paste(canvas, img, x1, y1, key):
    w, h = img.size()
    for y in range(h):
        for x in range(w):
            p = img.get(x, y)
            if p != key:
                canvas.set(x1 + x, y1 + y, p)
```

Let's try this on our example:

```
def collage2():
    trees = load_picture("trees1.jpg")
    r1 = load_picture("statue.jpg")
    r2 = scale_down(r1, 2)
    mirror(r2)
    chroma_paste(trees, r2, 100, 20,
                  Color.yellow)
    trees.show()
```



## 14 Concatenating photos

Let's make a comic strip out of a list of given pictures. We first have to compute the size of the result—its width is the sum of the width of the given pictures, its height is the maximum height of any picture. We then create a new picture of the correct size, and copy the images at the right location one by one:

```
def concat(imglist):
    # compute height and width
    w = 0
    h = 0
    for img in imglist:
        w1, h1 = img.size()
        w += w1
        h = max(h, h1)
    r = create_picture(w, h, Color.white)
    x0 = 0
    for img in imglist:
        w1, h1 = img.size()
        for y in range(h1):
            for x in range(w1):
                r.set(x0 + x, y, img.get(x, y))
        x0 += w1
    return r
```

Here is how we can use this:

```
def make_strip():
    r1 = load_picture("geowi.jpg")
    r2 = load_picture("statue.jpg")
    r3 = load_picture("yuna.jpg")
    return concat([r1, r2, r3])
```

