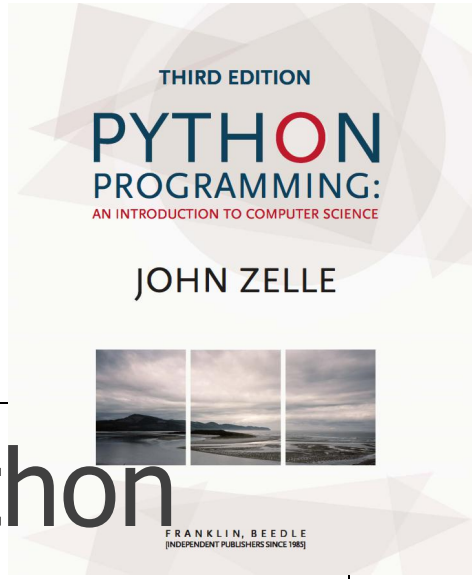


An Introduction to Python Programming

Chapter 10: Defining Classes

Objectives

- To be able to read and write **Python class definitions**.
- To understand the concept of **encapsulation** and how it contributes to building modular and maintainable programs.
- To be able to write interactive graphics programs involving novel (programmer designed) widgets.

Quick Review of Objects

- So far, our programs have made use of objects created from pre-defined class such as `Circle`. In this chapter we'll learn how to write our own classes to create novel objects.

Quick Review of Objects

- So far, in our programs an ***object*** was defined as an active data type that knows stuff and can do stuff.
- More precisely, an object consists of:
  - A collection of related information.
  - A set of operations to manipulate that information.
- ***instance variables , methods , attributes*** of an object.

```
myCircle = Circle(Point(0,0), 20)
myCircle.draw(win)
myCircle.move(dx, dy)
...
```

Quick Review of Objects

- All objects are said to be an ***instance* of some *class***. The class of an object determines which attributes the object will have.
- A **class** is a description of what its instances will know and do.
- **`Circle`**, the name of the class, is used to invoke the constructor.
- Once the instance has been created, it can be **manipulated by calling on its methods.**

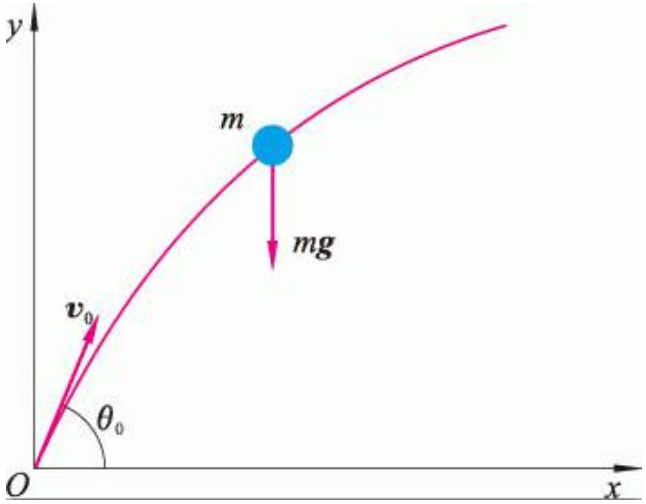
Cannonball Program Specification

- Let's simulate the flight of a cannonball :

How far the cannonball will travel when fired at various launch angles and initial velocities?

- The input: the **launch angle** , the **initial velocity** , the **initial height** of the cannonball.
- The output: the **distance** that the projectile travels before striking the ground .

Cannonball Program Specification



- **The distance** an object travels in a certain amount of time is equal to its rate times the amount of time (d = rt).

Designing the Program

- Suppose the ball starts at position (0,0).
- In a time interval it will have **moved some distance** upward (positive *y*) and forward (positive *x*). The exact distance will be determined by **the velocity in that direction**.

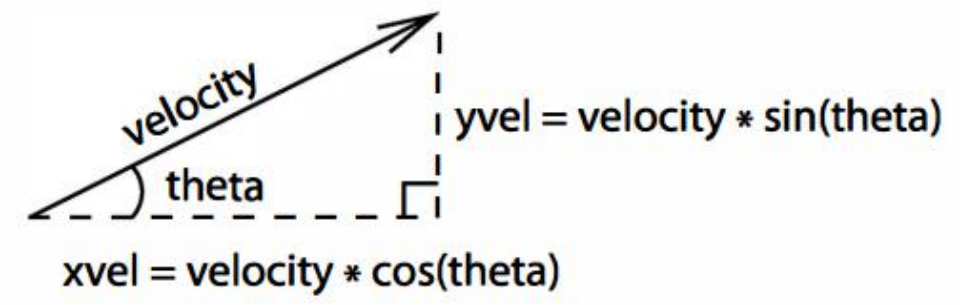
Designing the Program

- Here is a rough outline:

```
input the simulation parameters: angle, velocity, height, interval
calculate the initial position of the cannonball: xpos, ypos
calculate the initial velocities of the cannonball: xvel, yvel
while the cannonball is still flying:
    update the values of xpos, ypos, and yvel for interval seconds
        further into the flight
output the distance traveled as xpos
```

Designing the Program

- We need to calculate the x andy components of the initial velocity.



$$theta = \frac{\pi * angle}{180}$$

Designing the Program

- To update the horizontal position:

```
xpos = xpos + time * xvel
```

- To update the **yvel** and calculate **the average vertical velocity**:

```
yvel1 = yvel - time * 9.8

(yvel+yvel1)/2.0
```

Designing Programs

```
# cball1.py
from math import sin, cos, radians

def main():
    angle = float(input("Enter the launch angle (in degrees): "))
    vel = float(input("Enter the initial velocity (in meters/sec): "))
    h0 = float(input("Enter the initial height (in meters): "))
    time = float(input(
           "Enter the time interval between position calculations: "))
```

```
# convert angle to radians
theta = radians(angle)

# set the initial position and velocities in x and y directions
xpos = 0
ypos = h0
xvel = vel * cos(theta)
yvel = vel * sin(theta)

# loop until the ball hits the ground
while ypos >= 0.0:
    # calculate position and velocity in time seconds
    xpos = xpos + time * xvel
    yvel1 = yvel - time * 9.8
    ypos = ypos + time * (yvel + yvel1)/2.0
    yvel = yvel1

print("\nDistance traveled: {0:0.1f} meters.".format(xpos))
```

Modularizing the Program

- This program is complex due to **the number of variables**.
- Here's a version of the main algorithm **using helper functions**:

```
def main():
    angle, vel, h0, time = getInputs()
    xpos, ypos = 0, h0
    xvel, yvel = getXYComponents(vel, angle)
    while ypos >= 0:
        xpos, ypos, yvel = updateCannonBall(time, xpos, ypos, xvel, yvel)
    print("\nDistance traveled: {0:0.1f} meters.".format(xpos))
```

Modularizing the Program

- Still too much variables!
  - This may be simpler, but keeping track of the cannonball still requires **four pieces of information**, three of which change from moment to moment!
  - All four , plus `time`, are needed to compute the new values of the three that change.

Modularizing the Program

- Suppose there was a **`Projectile` class** that "understood" the physics of objects like cannonballs.
- Using the ***object-based*** approach:

```
def main():
    angle, vel, h0, time = getInputs()
    cball = Projectile(angle, vel, h0)
    while cball.getY() >= 0:
        cball.update(time)
    print("\nDistance traveled: {0:0.1f} meters.".format(cball.getX()))
```

Defining New Classes: Multi-Sided Dice

- Before designing a Projectile class, let's design a generic **class `MSDie`** to **model multi-sided dice**.
- Each **`MSDie`** object will know two things:
  - How many sides it has.
  - It's current value.
- We specify ***n***, the **number of sides** it will have.

Defining New Classes: Multi-Sided Dice

- We have **three methods** used to operate on the die:
  - `roll` – set the die to a random value between 1 and *n*, inclusive.
  - `setValue` – set the die to a specific value (i.e. cheat)
  - `getValue` – see what the current value is.

Defining New Classes:Multi-Sided Dice

```
>>> die1 = MSDie(6)
>>> die1.getValue()
1
>>> die1.roll()
>>> die1.getValue()
4
>>> die2 = MSDie(13)
>>> die2.getValue()
1
>>> die2.roll()
>>> die2.getValue()
12
>>> die2.setValue(8)
>>> die2.getValue()
8
```

Each die can be rolled independently and will always produce a random value in the proper range determined by the number of sides.

Defining New Classes:Multi-Sided Dice

- Writing a defnition for the **MSDie class**:

```
class MSDie:

    def __init__(self, sides):
        self.sides = sides
        self.value = 1

    def roll(self):
        self.value = randrange(1,self.sides+1)

    def getValue(self):
        return self.value

    def setValue(self, value):
        self.value = value
```

Defining New Classes: Multi-Sided Dice

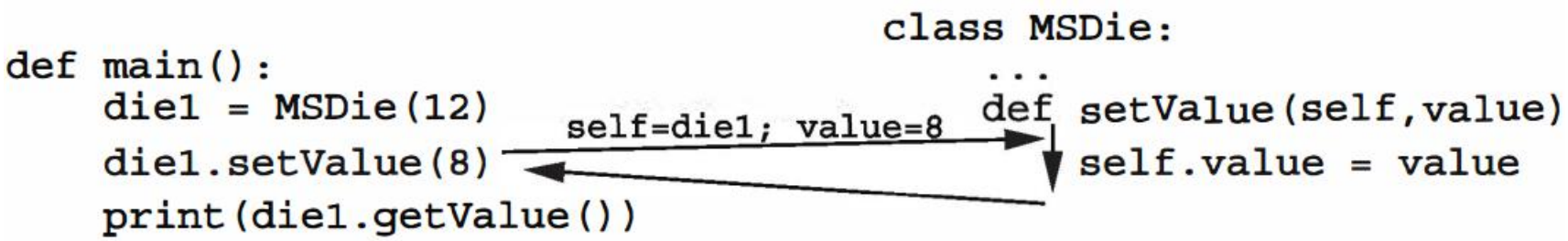
- **Class definitions** have the form:

```
class <class-name>:
    <method-definitions>
```

- The first parameter of a method is *always* named **`self`**, which is a reference to the object on which the method is acting.

Defining New Classes: Multi-Sided Dice

- Suppose we executes **`die1.setValue(8)`** in the **`main`** function.
- Python executes the following **four-step sequence**:
  - **`main`** suspends .Python **locates** the appropriate method definition **inside the class** of the object.
  - The **formal parameters** get assigned.
  - The body of the method is **executed**.
  - Control returns.



Defining New Classes: Multi-Sided Dice

- About the **`self` parameter** :
  - The **`self`** parameter is a **bookkeeping detail**.
  - It's a ***self*** parameter but not a ***normal*** parameters.
  - **Explicit** is better than **implicit**.

Defining New Classes: Multi-Sided Dice

- About **`__init__`** :
  - Certain methods have special meaning. These methods have names that start and end with **two _'s**.
  - It is the object contructor.It provides initial values for the instance variables of an object.
  - Outside the class, the constructor is **referred to by the class name**.When this statement is executed, a new **`MSDie` object** is created and **`__init__`** is executed.

```
die1 = MSDie(6)
```

Defining New Classes: Multi-Sided Dice

- **Instance variables** can remember the state of a particular object, and this information can be passed around the program as part of the object.
- This is **different** than **local function variables**, whose values disappear when the function terminates.

```
>>> die1 = Die(13)
>>> print(die1.getValue())
1
>>> die1.setValue(8)
>>> print(die1.getValue())
8
```

Example: The Projectile Class

- In the main program, a **cannonball** can be created from the **initial angle, velocity, and height**:

```
cball = Projectile(angle, vel, h0)
```

- The **`Projectile`** class must have an **`__init__`** method that will use these values to **initialize** the **instance variables** of **`cball`**.

Example: The Projectile Class

```
class Projectile:

    def __init__(self, angle, velocity, height):
        self.xpos = 0.0
        self.ypos = height
        theta = math.radians(angle)
        self.xvel = velocity * math.cos(theta)
        self.yvel = velocity * math.sin(theta)
```

- The value of **theta** is not needed after **__init __** terminates,so it is just a **normal (local) function variable**.

Example: The Projectile Class

- We need a couple of methods that **return the current position**.

```
def getX(self):
    return self.xpos

def getY(self):
    return self.ypos
```

Example: The Projectile Class

- The last method will **update** the state of the projectile.
- And the complete object-based solution to the cannonball problem will be seen on P326

```
def update(self, time):
    self.xpos = self.xpos + time * self.xvel
    yvel1 = self.yvel - time * 9.8
    self.ypos = self.ypos + time * (self.yvel + yvel1)/2.0
    self.yvel = yvel1
```

`yvel1` is a temporary variable.

Data Processing with Class

- Another common use for **objects** is to **group together a set of information** that describes a person or thing.
  - An **`Employee` class** with information such as employee's name, social security number, address, salary, etc.
  - A **student class** group information like name, address, credit hours, quality points,GPA, etc.
- Grouping information like this is often called a ***record***.

Data Processing with Class

- Suppose we have a **data file** that contains student grade information.

Student's Name	Credit-Hours	Quality Points
Adams, Henry	127	228
Comptewell, Susan	100	400
DibbleBit, Denny	18	41.5
Jones, Jim	48.5	155
Smith, Frank	37	125.33

- Our program can read this file to find the best student(according to GPA), print out **their name, credit-hours, and GPA**.

Data Processing with Class

- Courses are measured by a **4-point scale** where an "A' is 4 points, a "B" is 3 points, etc.
- GPA= total quality points / total credit hours .
  - If a class is worth 3 credit hours and the student gets an “A”, then he or she earns 3*4 = 12 quality points.

Data Processing with Class

- We can begin by **creating a Student class**.

```
class Student:
    def __init__(self, name, hours, qpoints):
        self.name = name
        self.hours = float(hours)
        self.qpoints = float(qpoints)
```

- Then we can make a record for Henry Adams:

```
aStudent = Student("Adams, Henry", 127, 228)
```

We can store all the information about a student in a single variable!

Data Processing with Class

- Next we must decide what methods a student object should have .

```
def getName(self):
    return self.name

def getHours(self):
    return self.hours

def getQPoints(self):
    return self.qpoints

def gpa(self):
    return self.qpoints/self.hours
```

Data Processing with Class

- To find the best student,Here's the algorithm for our program.

```
Get the file name from the user
Open the file for reading
Set best to be the first student
For each student s in the file
    if s.gpa() > best.gpa()
        set best to s
print out information about best
```

- The complete program is on P330

Data Processing with Class

- About the helper function called **makeStudent**.
  - This function is used to **create a record** for **the first student** in the file:

```
best = makeStudent(infile.readline())
```

  - It is called again inside the loop to **process each subsequent line** of the file:

```
s = makeStudent(line)
```

Programming Exercises

- **One unresolved issue**:This program only reports back a single student. If multiple students are tied for the best GPA, only the first one found is reported. Can you to modif the program so that it reports all students having the highest GPA?

Objects and Encapsulation: Encapsulating Useful Abstractions

- ***encapsulation:***
  - Once some useful objects are identified, the **implementation details** of the algorithm can be **moved into a suitable class definition**.
  - The main program only has to worry about **what objects can do**, not about how they are implemented.
  - This is **another application of abstraction** (ignoring irrelevant details), the essence of good design.

Objects and Encapsulation: Encapsulating Useful Abstractions

- Encapsulation is only a **programming convention** in Python.
  - Our class has **similar accessor methods** for **its instance variables**.
  - Stictly speaking, these methods are **not absolutely necessary** In Python, you can **access the instance variables** of any object with the regular dot notation.

Objects and Encapsulation: Encapsulating Useful Abstractions

```
>>> c = Projectile(60, 50, 20)
>>> c.xpos
0.0
>>> c.ypos
20
>>> c.xvel
25.0
>>> c.yvel
43.301270
```

- **Poor practice** to do this in programs.
- **Hide** the interal complexities of those objects.
- All interaction should use the **interface** provided by its methods.

Objects and Encapsulation: Putting Classes in Modules

- Put it into its **own module file** with **documentation** so that you won't have to looking at the code!
  - With **"#"** to indicate comments explaining
  - **("""")** allows us to directly type multi-line strings.
  - Python also has ***docstring***.

Objects and Encapsulation: Module Documentation

- **docstring**:
  - You can insert a plain string literal as the **first line** of a module, class, or function to document that component.
  - While ordinary comments are simply ignored by Python, **docstrings** are actually **carried along** during execution in a special attribute called **__doc __**.
  - You can use to **get help** on using the module or its contents.

```
>>> import random
>>> print (random.random.__doc__)
random() -> x in the interval [0, 1).
```

Objects and Encapsulation: Module Documentation

- Docstrings are also used by the **Pyhon online help system**
- A utility called **pydoc** can **automatically build documentation** for Python modules.

```
>>> import random
>>> help(random.random)
Help on built-in function random:

random(...) method of random.Random instance
    random() -> x in the interval [0, 1).
```

Objects and Encapsulation: Working with Multiple Modules

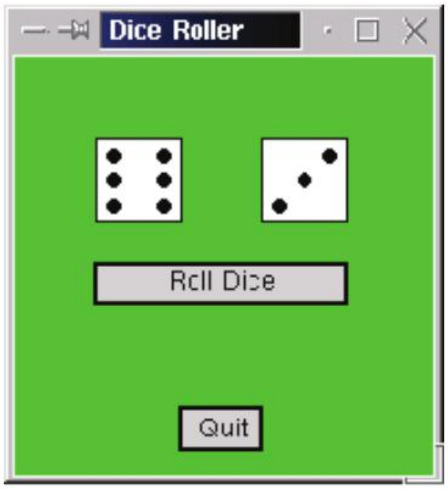
```
# cball4.py
from projectile import Projectile

def getInputs():
    a = float(input("Enter the launch angle (in degrees): "))
    v = float(input("Enter the initial velocity (in meters/sec): "))
    h = float(input("Enter the initial height (in meters): "))
    t = float(input("Enter the time interval between position calculations:
    return a,v,h,t

def main():
    angle, vel, h0, time = getInputs()
    cball = Projectile(angle, vel, h0)
    while cball.getY() >= 0:
        cball.update(time)
    print("\nDistance traveled: {0:0.1f} meters.".format(cball.getX()))
```

Objects and Encapsulation: Working with Multiple Modules

- The Python module-importing mechanism:
  - When first imports a given module, it creates a **module object** that contains all the things defined in the module (a ***namespace***).
  - If a module imports successfully (no syntax errors), **subsequent imports do not reload** the module.
  - If the source code for the module has been **changed**, **re-importing** it into an interactive session **will not updated** the version.
  - **Start a new interactive session** for testing the modified module.

Widgets:Dice Roller

- One very common use of objects ---**GUIs**.
- **Consider a program**: It **rolls a pair of standard dice** and **display** the dice graphically and **provide** two buttons, one for rolling and one for quitting.



Widgets:Dice Roller

- In this program:
  - two kinds of **widgets**: buttons and dice (instances of a button class).
  - the class providing a graphical view of the value of a die will be **DieView**.

Widgets:Building Buttons

- our buttons will support the **following methods**:
  - **Constructor** Creates a button in a window.
  - **Activate** sets the state of the button to active.
  - **Deactivate**
  - **Clicked** indicates whether the buton was clicked.
  - **GetLabel** returns the label string of the button.

Widgets:Building Buttons

- the **`activate`** and **`deactivate`** method:

```
def activate(self):
    "Sets this button to 'active'."
    self.label.setFill('black')
    self.rect.setWidth(2)
    self.active = True

def deactivate(self):
    "Sets this button to 'inactive'."
    self.label.setFill('darkgrey')
    self.rect.setWidth(1)
    self.active = False
```

Widgets:Building Buttons

- the **`clicked`** method:

 to determine whether a given point is **inside the rectangular button**.

```
def clicked(self, p):
    "Returns true if button is active and p is inside"
    return (self.active and
            self.xmin <= p.getX() <= self.xmax and
            self.ymin <= p.getY() <= self.ymax)
```

The complete class can be seen on P340-341

Widgets:Building Dice

- Our **DieView** class will have the following interface:
  - **constructor** creates a die in a window.
  - **setValue** changes the view to show a given value.

- The code for our **DieView class** is on P342.

Widgets:The Main Program

```
# roller.py
# Graphics program to roll a pair of dice. Uses custom widgets
# Button and DieView.

from random import randrange
from graphics import GraphWin, Point
from button import Button
from dieview import DieView

def main():
    # create the application window
    win = GraphWin("Dice Roller")
    win.setCoords(0, 0, 10, 10)
    win.setBackground("green2")

    # Draw the interface widgets
    die1 = DieView(win, Point(3,7), 2)
    die2 = DieView(win, Point(7,7), 2)
    rollButton = Button(win, Point(5,4.5), 6, 1, "Roll Dice")
    rollButton.activate()
    quitButton = Button(win, Point(5,1), 2, 1, "Quit")
```

Widgets:The Main Program

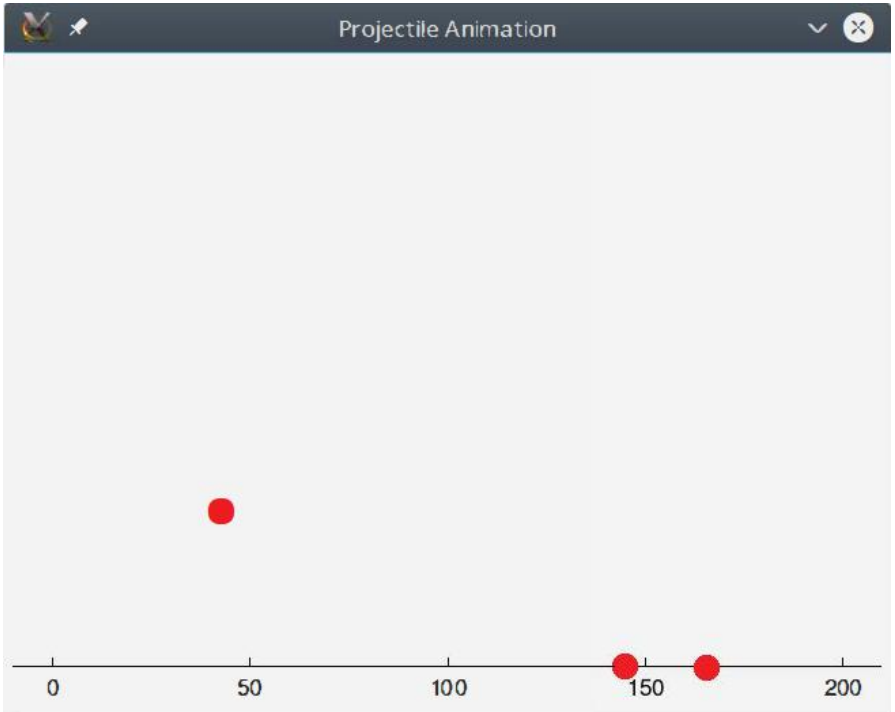
```
    # Event loop
    pt = win.getMouse()
    while not quitButton.clicked(pt):
        if rollButton.clicked(pt):
            value1 = randrange(1,7)
            die1.setValue(value1)
            value2 = randrange(1,7)
            die2.setValue(value2)
            quitButton.activate()
        pt = win.getMouse()

    # close up shop
    win.close()

main()
```

The Main Program

- Our **main program** is on P345.
- The visual interface is built by creating the two **`DieViews`** and two **`Buttons`**.
- The roll button is initially active, with the quit button deactivated,forcing the user to roll the dice at least once.
- The **event loop** is a sentinel loop that gets mouse clicks and processes them **until the user clicks on the quit button**.

Animated Cannonball

- Let's use our **new object ideas** to add a nicer interface to the **cannonball example** and make it have **a graphical interface.**



Animated Cannonball

- Drawing the Animation Window.

```
# create animation window
win = GraphWin("Projectile Animation", 640, 480, autoflush=False)
win.setCoords(-10, -10, 210, 155)

# draw baseline
Line(Point(-10,0), Point(210,0)).draw(win)

# draw labeled ticks every 50 meters
for x in range(0, 210, 50):
    Text(Point(x,-5), str(x)).draw(win)
    Line(Point(x,0), Point(x,2)).draw(win)
```

Animated Cannonball

- Creating a **ShotTracker**.
  - Use our existing **Projectile class**, but a it is **not a graphics object**.
  - **A Circle** is a good candidate,but it **does not know** how to flight.
  - Having elements of **both**, we can **define a suitable class** for it,calling a **ShotTracker**.

Animated Cannonball

- The **constructor** for the class:

```
def __init__(self, win, angle, velocity, height):
    """win is the GraphWin to display the shot. angle, velocity,
       and height are initial projectile parameters.
    """

    self.proj = Projectile(angle, velocity, height)
    self.marker = Circle(Point(0,height), 3)
    self.marker.setFill("red")
    self.marker.setOutline("red")
    self.marker.draw(win)
```

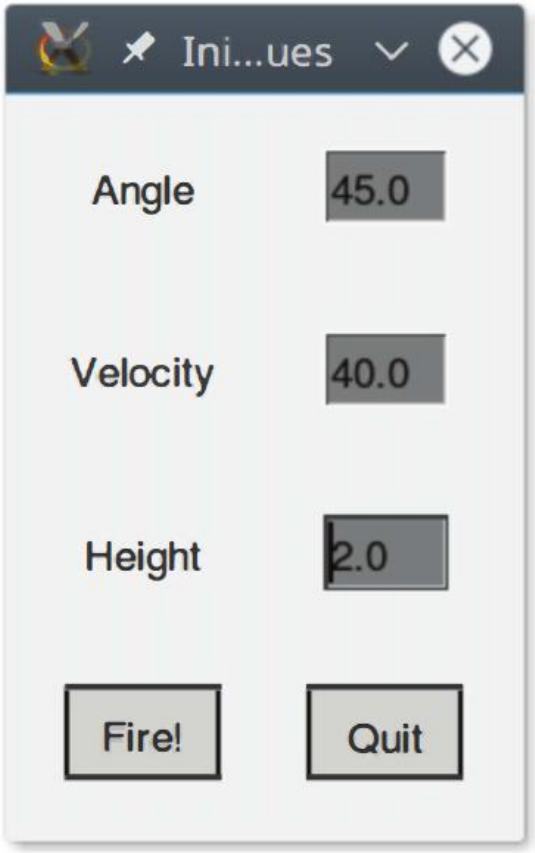
Animated Cannonball

- For the **circle** and **projectile**, we update both with the appropriate time interval..

```
def update(self, dt):
    """ Move the shot dt seconds farther along its flight """

    # update the projectile
    self.proj.update(dt)

    # move the circle to the new projectile location
    center = self.marker.getCenter()
    dx = self.proj.getX() - center.getX()
    dy = self.proj.getY() - center.getY()
    self.marker.move(dx,dy)
```

Animated Cannonball

- Creating an Input Dialog.



Animated Cannonball

```
class InputDialog:

    """ A custom window for getting simulation values (angle, velocity,
    and height) from the user."""

    def __init__(self, angle, vel, height):
        """ Build and display the input window """

        self.win = win = GraphWin("Initial Values", 200, 300)
        win.setCoords(0,4.5,4,.5)

        Text(Point(1,1), "Angle").draw(win)
        self.angle = Entry(Point(3,1), 5).draw(win)
        self.angle.setText(str(angle))

        Text(Point(1,2), "Velocity").draw(win)
        self.vel = Entry(Point(3,2), 5).draw(win)
```

Animated Cannonball

```
self.vel.setText(str(vel))

Text(Point(1,3), "Height").draw(win)
self.height = Entry(Point(3,3), 5).draw(win)
self.height.setText(str(height))

self.fire = Button(win, Point(1,4), 1.25, .5, "Fire!")
self.fire.activate()

self.quit = Button(win, Point(3,4), 1.25, .5, "Quit")
self.quit.activate()
```

Animated Cannonball

- Interact with the dialog:

```
def interact(self):
    """ wait for user to click Quit or Fire button
    Returns a string indicating which button was clicked
    """

    while True:
        pt = self.win.getMouse()
        if self.quit.clicked(pt):
            return "Quit"
        if self.fire.clicked(pt):
            return "Fire!"
```

Animated Cannonball

- Finally we add an operation to **get the data back** out and to **close up the dialog** .

```
def getValues(self):
    """ return input values """
    a = float(self.angle.getText())
    v = float(self.vel.getText())
    h = float(self.height.getText())
    return a,v,h

def close(self):
    """ close the input window """
    self.win.close()
```

Animated Cannonball

- The **Main Event Loop**

```
# file: animation.py

def main():

    # create animation window
    win = GraphWin("Projectile Animation", 640, 480, autoflush=False)
    win.setCoords(-10, -10, 210, 155)
    Line(Point(-10,0), Point(210,0)).draw(win)
    for x in range(0, 210, 50):
        Text(Point(x,-5), str(x)).draw(win)
        Line(Point(x,0), Point(x,2)).draw(win)
```

Animated Cannonball

```
# event loop, each time through fires a single shot
angle, vel, height = 45.0, 40.0, 2.0
while True:
    # interact with the user
    inputwin = InputDialog(angle, vel, height)
    choice = inputwin.interact()
    inputwin.close()

    if choice == "Quit": # loop exit
        break

    # create a shot and track until it hits ground or leaves window
    angle, vel, height = inputwin.getValues()
    shot = ShotTracker(win, angle, vel, height)
    while 0 <= shot.getY() and -10 < shot.getX() <= 210:
        shot.update(1/50)
        update(50)

win.close()
```

Programming Exercise

- Modify the cannonball simulation from the chapter so that it also calculates the maximum height achieved by the cannonball.