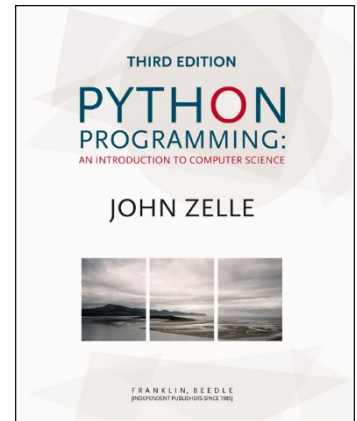


Python Programming: An Introduction To Computer Science



Chapter 10 Defining Classes



Objectives

- To appreciate how defining new classes can provide structure for a complex program.
- 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.



Objectives

- To be able to write programs involving simple class definitions.
- To be able to write interactive graphics programs involving novel (programmer designed) widgets.



Quick Review of Objects

- In the last three chapters we've developed techniques for structuring the *computations* of the program.
- We'll now take a look at techniques for structuring the *data* that our programs use.
- So far, our programs have made use of objects created from pre-defined classes such as `Circle`. In this chapter we'll learn how to write our own classes to create novel objects.



Quick Review of Objects

- In chapter four an *object* was defined as an active data type that knows stuff and can do stuff.
- More precisely, an object consists of:
 1. A collection of related information.
 2. A set of operations to manipulate that information.



Quick Review of Objects

- The information is stored inside the object in *instance variables*.
- The operations, called *methods*, are functions that “live” inside the object.
- Collectively, the instance variables and methods are called the *attributes* of an object.



Quick Review of Objects

- A `Circle` object will have instance variables such as `center`, which remembers the center point of the circle, and `radius`, which stores the length of the circle's radius.
- The `draw` method examines the `center` and `radius` to decide which pixels in a window should be colored.



Quick Review of Objects

- The `move` method will change the value of `center` to reflect the new position of the circle.
- 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.



Quick Review of Objects

- New objects are created from a class by invoking a *constructor*. You can think of the class itself as a sort of factory for stamping out new instances.
- Consider making a new circle object:
`myCircle = Circle(Point(0,0),20)`
- `Circle`, the name of the class, is used to invoke the constructor.



Quick Review of Objects

```
myCircle = Circle(Point(0,0), 20)
```

- This statement creates a new `Circle` instance and stores a reference to it in the variable `myCircle`.
- The parameters to the constructor are used to initialize some of the instance variables (`center` and `radius`) inside `myCircle`.



Quick Review of Objects

```
myCircle = Circle(Point(0,0), 20)
```

- Once the instance has been created, it can be manipulated by calling on its methods:

```
myCircle.draw(win)
```

```
myCircle.move(dx, dy)
```



Cannonball Program Specification

- Let's try to write a program that simulates the flight of a cannonball or other projectile.
- We're interested in how far the cannonball will travel when fired at various launch angles and initial velocities.



Cannonball Program Specification

- The input to the program will be the launch angle (in degrees), the initial velocity (in meters per second), and the initial height (in meters) of the cannonball.
- The output will be the distance that the projectile travels before striking the ground (in meters).



Cannonball Program Specification

- The acceleration of gravity near the earth's surface is roughly 9.8 m/s/s .
- If an object is thrown straight up at 20 m/s , after one second it will be traveling upwards at 10.2 m/s . After another second, its speed will be $.4 \text{ m/s}$. Shortly after that the object will start coming back down to earth.



Cannonball Program Specification

- Using calculus, we could derive a formula that gives the position of the cannonball at any moment of its flight.
- However, we'll solve this problem with simulation, a little geometry, and the fact that 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

- Given the nature of the problem, it's obvious we need to consider the flight of the cannonball in two dimensions: it's height and the distance it travels.
- Let's think of the position of the cannonball as the point (x, y) where x is the distance from the starting point and y is the height above the ground.



Designing the Program

- Suppose the ball starts at position $(0,0)$, and we want to check its position every tenth of a second.
- In that time interval it will have moved some distance upward (positive y) and some distance forward (positive x). The exact distance will be determined by the velocity in that direction.



Designing the Program

- Since we are ignoring wind resistance, x will remain constant through the flight.
- However, y will change over time due to gravity. The y velocity will start out positive and then become negative as the cannonball starts to fall.



Designing the Program

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

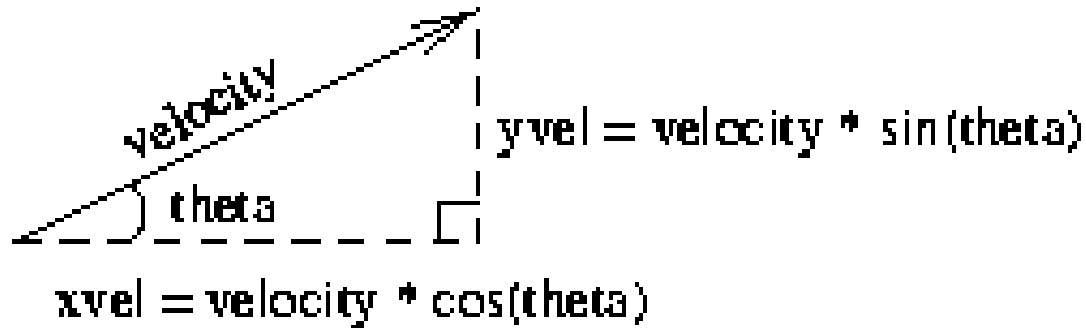
- Using step-wise refinement:

```
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: "))
```

- Calculating the initial position for the cannonball is also easy. It's at distance 0 and height h_0 !

```
xpos = 0  
ypos = h0
```

Designing the Program



- If we know the magnitude of the velocity and the angle theta, we can calculate $yvel = velocity * \sin(theta)$ and $xvel = velocity * \cos(theta)$.



Designing the Program

- Our input angle is in degrees, and the Python math library uses radians.
- ```
theta = math.radians(angle)
xvel = vel * cos(theta)
yvel = vel * sin(theta)
```
- In the main loop, we want to keep updating the position of the ball until it reaches the ground:  

```
while ypos >= 0.0:
```
- We used `>= 0` so the loop will start if the ball starts out on the ground.



# Designing the Program

---

- Each time through the loop we want to update the state of the cannonball to move it `time` seconds farther.
- Since we assume there is no wind resistance, `xvel` remains constant.
- Say a ball is traveling at 30 m/s and is 50 m from the firing point. In one second it will be 50 + 30 meters away. If the time increment is .1 second it will be  $50 + 30 * .1 = 53$  meters distant.
- `xpos = xpos + time * xvel`



# Designing the Program

---

- Working with `yvel` is slightly more complicated since gravity causes the `y`-velocity to change over time.
- Each second, `yvel` must decrease by 9.8 m/s, the acceleration due to gravity.
- In 0.1 seconds the velocity will decrease by  $0.1(9.8) = .98$  m/s.
- The velocity at the end of the time interval:  
 $yvel1 = yvel - time * 9.8$





# Designing the Programs

---

- To calculate how far the cannonball travels over the interval, we need to calculate its *average* vertical velocity over the interval.
- Since the velocity due to gravity is constant, it is simply the average of the starting and ending velocities times the length of the interval:

```
ypos = ypos + time * (yvel + yvel1)/2.0
```



# Designing Programs

---

```
cball1.py
Simulation of the flight of a cannon ball (or other projectile)
This version is not modularized.

from math import pi, sin, cos

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: "))

 radians = (angle * pi)/180.0
 xpos = 0
 ypos = h0
 xvel = vel * cos(radians)
 yvel = vel * sin(radians)
 while ypos >= 0:
 xpos = xpos + time * xvel
 yvell = yvel - 9.8 * time
 ypos = ypos + time * (yvel + yvell)/2.0
 yvel = yvell

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



# Modularizing the Program

---

- During program development, we employed step-wise refinement (and top-down design), but did not divide the program into functions.
- While this program is fairly short, it is complex due to the number of variables.



# Modularizing the Program

---

```
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))
```

- It should be obvious what each of these helper functions does based on their name and the original program code.



# Modularizing the Program

---

- This version of the program is more concise!
- The number of variables has been reduced from 10 to 8, since `theta` and `yvell` are local to `getXYComponents` and `updateCannonBall`, respectively.
- This may be simpler, but keeping track of the cannonball still requires four pieces of information, three of which change from moment to moment!



# Modularizing the Program

---

- All four variables, plus `time`, are needed to compute the new values of the three that change.
- This gives us a function with five parameters and three return values.
- Yuck! There must be a better way!



# Modularizing the Program

---

- There is a single real-world cannonball object, but it requires four pieces of information: `xpos`, `ypos`, `xvel`, `x` and `yvel`.
- Suppose there was a `Projectile` class that “understood” the physics of objects like cannonballs. An algorithm using this approach would create and update an object stored in a single variable.



# Modularizing the Program

---

- Using our *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()))
```

- To make this work we need a `Projectile` class that implements the methods `update`, `getX`, and `getY`.





# Example: Multi-Sided Dice

---

- A normal die (singular of dice) is a cube with six faces, each with a number from one to six.
- Some games use special dice with a different number of sides.
- Let's design a generic class `MSDie` to model multi-sided dice.



# Example: Multi-Sided Dice

---

- Each `MSDie` object will know two things:
  - How many sides it has.
  - It's current value
- When a new `MSDie` is created, we specify  $n$ , the number of sides it will have.



# Example: Multi-Sided Dice

---

- We have three methods that we can use 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.



# Example: Multi-Sided Dice

---

```
>>> die1 = MSDie(6)
>>> die1.getValue()
1
>>> die1.roll()
>>> die1.getValue()
5
>>> die2 = MSDie(13)
>>> die2.getValue()
1
>>> die2.roll()
>>> die2.getValue()
9
>>> die2.setValue(8)
>>> die2.getValue()
8
```



# Example: Multi-Sided Dice

---

- Using our object-oriented vocabulary, we create a die by invoking the `MSDie` *constructor* and providing the number of sides as a *parameter*.
- Our die objects will keep track of this number internally as an *instance variable*.
- Another *instance variable* is used to keep the current value of the die.
- We initially set the value of the die to be 1 because that value is valid for any die.
- That value can be changed by the `roll` and `setRoll` methods, and returned by the `getValue` method.



# Example: Multi-Sided Dice

---

```
msdie.py
Class definition for an n-sided die.

from random import randrange

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
```



# Example: Multi-Sided Dice

---

- Class definitions have the form

```
class <class-name>:
 <method-definitions>
```
- Methods look a lot like functions! Placing the function inside a class makes it a method of the class, rather than a stand-alone function.
- The first parameter of a method is *usually* named `self`, which is a reference to the object on which the method is acting.



# Example: Multi-Sided Dice

---

- Suppose we have a `main` function that executes `die1.setValue(8)`.
- Just as in function calls, Python executes the following four-step sequence:
  - `main` suspends at the point of the method application. Python locates the appropriate method definition inside the class of the object to which the method is being applied. Here, control is transferred to the `setValue` method in the `MSDie` class, since `die1` is an instance of `MSDie`.





# Example: Multi-Sided Dice

---

- The formal parameters of the method get assigned the values supplied by the actual parameters of the call. In the case of a method call, the first formal parameter refers to the object:  

```
self = die1
value = 8
```
- The body of the method is executed.



# Example: Multi-Sided Dice

---

- Control returns to the point just after where the method was called. In this case, it is immediately following `die1.setValue(8)`.
- Methods are called with one parameter, but the method definition itself includes the `self` parameter as well as the actual parameter.



# Example: Multi-Sided Dice

---

- The `self` parameter is a bookkeeping detail. We can refer to the first formal parameter as the *self* parameter and other parameters as *normal* parameters. So, we could say `setValue` uses one normal parameter.

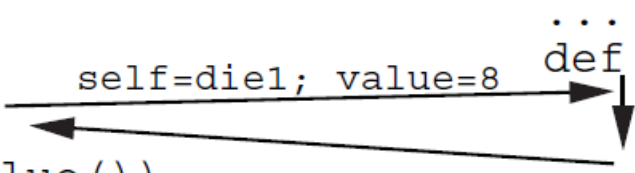


# Example: Multi-Sided Dice

---

```
def main():
 die1 = MSDie(12)
 die1.setValue(8)
 print(die1.getValue())

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





# Example: Multi-Sided Dice

---

- Objects contain their own data. Instance variables provide storage locations inside of an object.
- Instance variables are accessed by name using our dot notation:  
`<object>.<instance-var>`
- Looking at `setValue`, we see `self.value` refers to the instance variable `value` inside the object. Each `MSDie` object has its own `value`.



# Example: Multi-Sided Dice

---

- Certain methods have special meaning. These methods have names that start and end with two `_`'s.
- `__init__` is the object constructor. Python calls this method to initialize a new `MSDie`. `__init__` provides initial values for the instance variables of an object.



# Example: Multi-Sided Dice

---

- Outside the class, the constructor is referred to by the class name:  
`die1 = MSDie(6)`
- When this statement is executed, a new `MSDie` object is created and `__init__` is executed on that object.
- The net result is that `die1.sides` is set to 6 and `die1.value` is set to 1.



# Example: 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.





# Example: The Projectile Class

---

- This class will need a constructor to initialize instance variables, an `update` method to change the state of the projectile, and `getX` and `getY` methods that can report the current position.
- In the main program, a cannonball can be created from the initial angle, velocity, and height:  

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



# Example: The Projectile Class

---

- The `Projectile` class must have an `__init__` method that will use these values to initialize the instance variables of `cball`.
- These values will be calculated using the same formulas as before.



# 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 * cos(theta)
 self.yvel = velocity * sin(theta)
```

- We've created four instance variables (`self.???`). Since the value of `theta` is not needed later, it is a normal function variable.



# Example: The Projectile Class

---

- The methods to access the X and Y position are straightforward.

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

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



# Example: The Projectile Class

---

- The last method is `update`, where we'll take the time interval and calculate the updated X and Y values.

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

- `yvel1` is a temporary variable.



# Data Processing with Class

---

- A class is useful for modeling a real-world object with complex behavior.
- Another common use for objects is to group together a set of information that describes a person or thing.
  - Eg., a company needs to keep track of information about employees (an `Employee` class with information such as employee's name, social security number, address, salary, etc.)



# Data Processing with Class

---

- A grouping of information like this is often called a *record*.
- Let's try a simple data processing example!
- A typical university measures courses in terms of credit hours, and grade point averages are calculated on a 4 point scale where an “A” is 4 points, a “B” is three, etc.



# Data Processing with Class

---

- Grade point averages are generally computed using quality points. If a class is worth 3 credit hours and the student gets an “A”, then he or she earns  $3(4) = 12$  quality points. To calculate the GPA, we divide the total quality points by the number of credit hours completed.





# Data Processing with Class

---

- Suppose we have a data file that contains student grade information.
- Each line of the file consists of a student's name, credit-hours, and 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 |



# Data Processing with Class

---

- Our job is to write a program that reads this file to find the student with the best GPA and print out their name, credit-hours, and GPA.
- The place to start? Creating a `Student` class!
- We can use a `Student` object to store this information as instance variables.



# Data Processing with Class

---

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

- The values for `hours` are converted to `float` to handle parameters that may be floats, ints, or strings.
- To create a student record:  

```
aStudent = Student("Adams, Henry", 127, 228)
```
- The coolest thing is that we can store all the information about a student in a single variable!



# Data Processing with Class

---

- We need to be able to access this information, so we need to define a set of accessor methods.

```
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
```

- For example, to print a student's name you could write:  

```
print aStudent.getName()
```



# Data Processing with Class

---

- How can we use these tools to find the student with the best GPA?
- We can use an algorithm similar to finding the max of  $n$  numbers! We could look through the list one by one, keeping track of the best student seen so far!



# Data Processing with Class

---

```
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
```



# Data Processing with Class

---

```
gpa.py
Program to find student with highest GPA

class Student:

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

 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
```

```
def makeStudent(infoStr):
 name, hours, qpoints = infoStr.split("\t")
 return Student(name, hours, qpoints)

def main():
 filename = input("Enter name the grade
file: ")
 infile = open(filename, 'r')
 best = makeStudent(infile.readline())
 for line in infile:
 s = makeStudent(line)
 if s.gpa() > best.gpa():
 best = s
 infile.close()
 print("The best student is:",
best.getName())
 print ("hours:", best.getHours())
 print("GPA:", best.gpa())

if __name__ == '__main__':
 main()
```



# Encapsulating Useful Abstractions

---

- Defining new classes (like `Projectile` and `Student`) can be a good way to modularize a program.
- Once some useful objects are identified, the implementation details of the algorithm can be moved into a suitable class definition.





# Encapsulating Useful Abstractions

---

- The main program only has to worry about what objects can do, not about how they are implemented.
- In computer science, this separation of concerns is known as *encapsulation*.
- The implementation details of an object are encapsulated in the class definition, which insulates the rest of the program from having to deal with them.



# Encapsulating Useful Abstractions

---

- One of the main reasons to use objects is to hide the internal complexities of the objects from the programs that use them.
- From outside the class, all interaction with an object can be done using the interface provided by its methods.



# Encapsulating Useful Abstractions

---

- One advantage of this approach is that it allows us to update and improve classes independently without worrying about “breaking” other parts of the program, provided that the interface provided by the methods does not change.



# Putting Classes in Modules

---

- Sometimes we may program a class that could be useful in many other programs.
- If you might be reusing the code again, put it into its own module file with documentation to describe how the class can be used so that you won't have to try to figure it out in the future from looking at the code!



# Module Documentation

---

- You are already familiar with “#” to indicate comments explaining what’s going on in a Python file.
- Python also has a special kind of commenting convention called the *docstring*. You can insert a plain string literal as the first line of a module, class, or function to document that component.



# Module Documentation

---

- Why use a docstring?
  - Ordinary comments are ignored by Python
  - Docstrings are accessible in a special attribute called `__doc__`.
- Most Python library modules have extensive docstrings. For example, if you can't remember how to use `random`:

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



# Module Documentation

---

- Docstrings are also used by the Python online help system and by a utility called PyDoc that automatically builds documentation for Python modules. You could get the same information like this:

```
>>> import random
```

```
>>> help(random.random)
```

```
Help on built-in function random:
```

```
random(...)
```

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



# Module Documentation

---

- To see the documentation for an entire module, try typing `help(module_name)`!
- `"""` is a third way that Python allows string literals to be delimited, allowing us to type multi-line strings.
- The following code for the projectile class has docstrings.





# Module Documentation

---

```
projectile.py

"""projectile.py
Provides a simple class for modeling the flight of projectiles."""

from math import pi, sin, cos

class Projectile:

 """Simulates the flight of simple projectiles near the earth's
 surface, ignoring wind resistance. Tracking is done in two
 dimensions, height (y) and distance (x)."""

 def __init__(self, angle, velocity, height):
 """Create a projectile with given launch angle, initial
 velocity and height."""
 self.xpos = 0.0
 self.ypos = height
 theta = pi * angle / 180.0
 self.xvel = velocity * cos(theta)
 self.yvel = velocity * sin(theta)
```



# Module Documentation

---

```
def update(self, time):
 """Update the state of this projectile to move it time seconds
 farther into its flight"""
 self.xpos = self.xpos + time * self.xvel
 yvell = self.yvel - 9.8 * time
 self.ypos = self.ypos + time * (self.yvel + yvell) / 2.0
 self.yvel = yvell

def getY(self):
 "Returns the y position (height) of this projectile."
 return self.ypos

def getX(self):
 "Returns the x position (distance) of this projectile."
 return self.xpos
```



# Working with Multiple Modules

---

- Our main program can import from the projectile module in order to solve the original problem!

```
cball4.py
Simulation of the flight of a cannon ball (or other projectile)
This version uses a separate projectile module file

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()))
```



# Working with Multiple Modules

---

- If you are *interactively* testing a multi-module Python program, you need to be aware that reloading a module may not behave as you expect.
- When Python 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. Even if the source code for the module has been changed, re-importing it into an interactive session will not load the updated version.



# Working with Multiple Modules

---

- The easiest way – start a new interactive session for testing whenever any of the modules involved in your testing are modified. This way you’re guaranteed to get a more recent import of all the modules you’re using.
- If you’re using IDLE, you’ll notice it does this for you by doing a shell restart when you select “run module.”



# Widgets

---

- One very common use of objects is in the design of graphical user interfaces (GUIs).
- Back in chapter four we talked about GUIs being composed of visual interface objects known as *widgets*.
- The `Entry` object defined in our `graphics` library is one example of a widget.



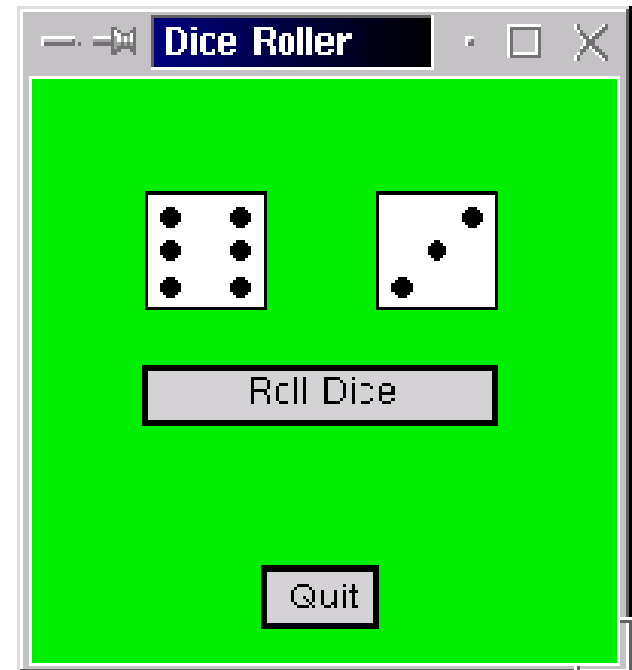
# Example Program: Dice Roller

---

- Let's build a couple useful widgets!
- Consider a program that rolls a pair of six-sided dice.
- The program will display the dice graphically and provide two buttons, one for rolling the dice and one for quitting the program.

# Example Program: Dice Roller

- There are two kinds of widgets: buttons and dice.
- The two buttons will be examples of the `Button` class, while the dice images will be provided by `dieView`.







# Building Buttons

---

- Most modern GUIs have buttons with 3-dimensional look and feel. Our simple graphics package does not have the machinery to produce buttons that appear to depress as they are clicked.
- All we can do is report back where the mouse was clicked after the click has been completed.



# Building Buttons

---

- Our buttons will be rectangular regions in a graphics window where user clicks can influence the behavior of the running application.
- We need a way to determine whether a button has been clicked.
- It would be nice to be able to activate and deactivate (gray-out) individual buttons.



# Building Buttons

---

- **Constructor** – Create a button in a window. We will specify the window in which the button will be displayed, the location/size of the button, and the label on the button.
- **Activate** – Set the state of the button to active.
- **Deactivate** – Set the state of the button to inactive.



# Building Buttons

---

- **Clicked**– Indicate if the button was clicked. If the button is active, this method will determine if the point clicked is inside the button region. The point will have to be sent as a parameter to the method.
- **getLabel**– Returns the label string of a button. This is provided so that we can identify a particular button.



# Building Buttons

---

- To support these operations, our buttons will need a number of instance variables.
- For example, buttons are drawn as a rectangle with some text centered on it. Invoking the `activate` and `deactivate` methods will change the appearance of the buttons.



# Building Buttons

---

- Saving the `Rectangle` and `Text` objects as instance variables means we will be able to control the width of the outline and color of the label.
- Let's try writing these methods and build up a list of possible instance variables! Once we have the list, we can write the constructor to initialize them.



# Building Buttons

---

- In `activate`, we can signal a button is active by making its outline thicker and making the label text black.

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

- Remember, `self` refers to the button object.
- Our constructor will have to initialize `self.label` as an appropriate `Text` object and `self.rect` as a rectangle object.
- `self.active` also has a Boolean instance variable to remember whether or not the button is currently inactive.



# Building Buttons

---

- The code for `deactivate` is very similar:

```
def deactivate(self):
 "Sets this button to 'inactive'."
 self.label.setFill('darkgrey')
 self.rect.setWidth(1)
 self.active = 0
```





# Building Buttons

---

- Let's work on the `clicked` method.
- The `graphics` package has the `getMouse` method to see if and where the mouse has been clicked.
- If an application needs to get a button click, it will have to first call `getMouse` and then see which button, if any, the point is inside of.



# Building Buttons

---

```
pt = win.getMouse()
if button1.clicked(pt):
 # Do button1 stuff
elif button2.clicked(pt):
 # Do button2 stuff
elif button3.clicked(pt):
 # Do button3 stuff
...
```

- The main job of the `clicked` method is to determine whether a given point is inside the rectangular button.



# Building Buttons

---

- The point is inside the button if its  $x$  and  $y$  coordinates lie between the extreme  $x$  and  $y$  values of the rectangle.
- This would be easiest if the button object had the min and max values of  $x$  and  $y$  as instance variables.



# Building Buttons

---

- ```
def clicked(self, p):  
    "RETURNS true if button active and p is inside"  
    return self.active and \  
           self.xmin <= p.getX() <= self.xmax and \  
           self.ymin <= p.getY() <= self.ymax
```
- For this function to return `True`, all three parts of the Boolean expression must be true.
- The first part ensures that only active buttons will return that they have been clicked.
- The second and third parts ensure that the x and y values of the point that was clicked fall between the boundaries of the rectangle.



Building Buttons

- The only part that is left is to write the constructor:

```
def __init__(self, win, center, width, height, label):
    """ Creates a rectangular button, eg:
        qb = Button(myWin, Point(30,25), 20, 10, 'Quit') """

    w,h = width/2.0, height/2.0
    x,y = center.getX(), center.getY()
    self.xmax, self.xmin = x+w, x-w
    self.ymax, self.ymin = y+h, y-h
    p1 = Point(self.xmin, self.ymin)
    p2 = Point(self.xmax, self.ymax)
    self.rect = Rectangle(p1,p2)
    self.rect.setFill('lightgray')
    self.rect.draw(win)
    self.label = Text(center, label)
    self.label.draw(win)
    self.deactivate()
```

- Buttons are positioned by providing a center point, width, and height



Building Buttons

- Buttons are positioned by providing a center point, width, and height.



Building Dice

- The purpose of the `DieView` class is to graphically display the value of a die.
- The face of the die is a square/rectangle, and the pips/spots on the die are circles.
- As before, the `DieView` class will have a constructor and a method.



Building Dice

- **constructor** – Create a die in a window. We will specify the window, the center point of the die, and the size of the die as parameters.
- **setValue** – Change the view to show a given value. The value to display will be passed as a parameter.



Building Dice

- Clearly, the hardest part of this will be to turn on the pips on the die to represent the current value of the die.
- One approach is to pre-place the pips, and make them the same color as the die. When the spot is turned on, it will be redrawn with a darker color.



Building Dice

- A standard die will need seven pips -- a column of three on the left and right sides, and one in the center.
- The constructor will create the background square and the seven circles. `setValue` will set the colors of the circles based on the value of the die.



Building Dice

```
# dieview.py
#     A widget for displaying the value of a die

from graphics import *

class DieView:
    """ DieView is a widget that displays a graphical representation
    of a standard six-sided die."""

    def __init__(self, win, center, size):
        """Create a view of a die, e.g.:
           dl = GDie(myWin, Point(40,50), 20)
        creates a die centered at (40,50) having sides
        of length 20."""

        # first define some standard values
        self.win = win
        self.background = "white" # color of die face
        self.foreground = "black" # color of the pips
        self.psize = 0.1 * size # radius of each pip
        hsize = size / 2.0 # half of size
        offset = 0.6 * hsize # distance from center to outer pip
```



Building Dice

```
# create a square for the face
cx, cy = center.getX(), center.getY()
p1 = Point(cx-hsize, cy-hsize)
p2 = Point(cx+hsize, cy+hsize)
rect = Rectangle(p1,p2)
rect.draw(win)
rect.setFill(self.background)

# Create 7 circles for standard pip locations
self.pip1 = self.__makePip(cx-offset, cy-offset)
self.pip2 = self.__makePip(cx-offset, cy)
self.pip3 = self.__makePip(cx-offset, cy+offset)
self.pip4 = self.__makePip(cx, cy)
self.pip5 = self.__makePip(cx+offset, cy-offset)
self.pip6 = self.__makePip(cx+offset, cy)
self.pip7 = self.__makePip(cx+offset, cy+offset)

self.setValue(1)
```



Building Dice

```
def __makePip(self, x, y):
    """Internal helper method to draw a pip at (x,y)"""
    pip = Circle(Point(x,y), self.psize)
    pip.setFill(self.background)
    pip.setOutline(self.background)
    pip.draw(self.win)
    return pip

def setValue(self, value):
    """ Set this die to display value."""
    # turn all pips off
    self.pip1.setFill(self.background)
    self.pip2.setFill(self.background)
    self.pip3.setFill(self.background)
    self.pip4.setFill(self.background)
    self.pip5.setFill(self.background)
    self.pip6.setFill(self.background)
    self.pip7.setFill(self.background)
```



Building Dice

```
# turn correct pips on
if value == 1:
    self.pip4.setFill(self.foreground)
elif value == 2:
    self.pip1.setFill(self.foreground)
    self.pip7.setFill(self.foreground)
elif value == 3:
    self.pip1.setFill(self.foreground)
    self.pip7.setFill(self.foreground)
    self.pip4.setFill(self.foreground)
elif value == 4:
    self.pip1.setFill(self.foreground)
    self.pip3.setFill(self.foreground)
    self.pip5.setFill(self.foreground)
    self.pip7.setFill(self.foreground)
```

```
elif value == 5:
    self.pip1.setFill(self.foreground)
    self.pip3.setFill(self.foreground)
    self.pip4.setFill(self.foreground)
    self.pip5.setFill(self.foreground)
    self.pip7.setFill(self.foreground)
else:
    self.pip1.setFill(self.foreground)
    self.pip2.setFill(self.foreground)
    self.pip3.setFill(self.foreground)
    self.pip5.setFill(self.foreground)
    self.pip6.setFill(self.foreground)
    self.pip7.setFill(self.foreground)
```



Building Dice

- Things to notice:
 - The size of the spots being $1/10$ of the size of the die was determined by trial and error.
 - We define and calculate various attributes of the die in the constructor and then use them in other methods and functions within the class so that if we wanted to change the appearance, all those values and the code to go with them is in one place, rather than throughout the class.



Building Dice

- `__makePip` is a helper function to draw each of the seven pips on the die. Since it is only useful within `DieView`, it's appropriate to make it a class method. It's name starts with `__` to indicate that its use is "private" to the class and is not intended to be used outside the class.



The Main Program

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

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")
```



The Main Program

```
# 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")

# 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()
```



The Main Program

- The visual interface is built by creating the two `DieViews` and two `Buttons`.
- The roll button is initially active, but the quit button is deactivated. This forces 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.

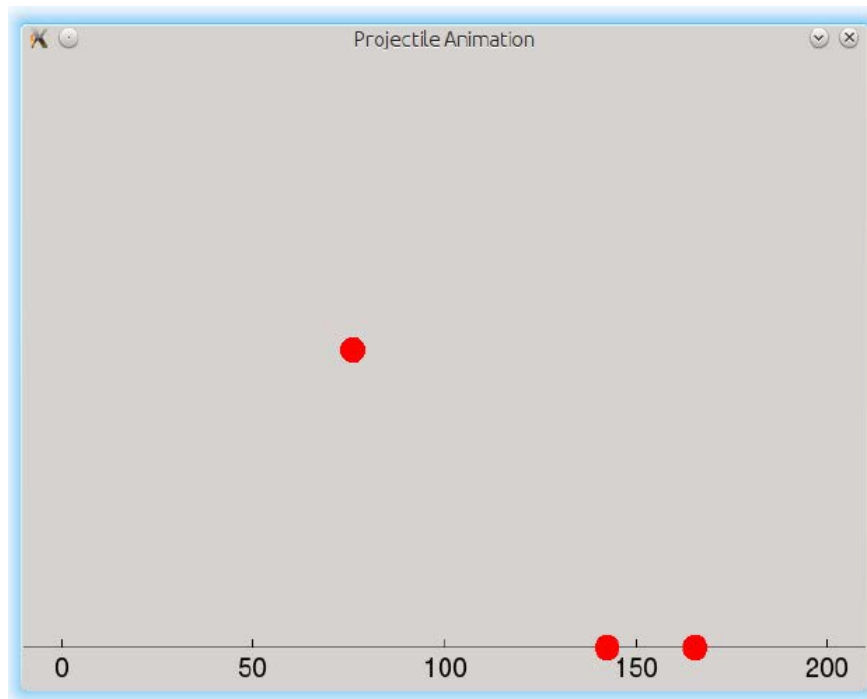


The Main Program

- The `if` within the loop ensures that the dice are rolled only when the user clicks the roll button.
- Clicking a point that is not inside any button causes the loop to iterate without doing anything.

Animated Cannon Ball

- Let's add a nicer interface to the cannon ball program.





Drawing the Animation Window

```
def main():

    # 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)
```



Drawing the Animation Window

- Did you notice the `autoflush=False`?
- The default behavior is for a graphics object to immediately update its appearance whenever it's asked to change, i.e. changing its color.
- By setting `autoflush` to `False`, we're telling the graphics library it's OK to allow commands to build up in the pipeline before performing them.



Drawing the Animation Window

- Why would we want the graphics commands to not occur immediately?
 - Graphics commands are relatively time consuming because they require communication with the underlying operating system to exchange information with the display hardware.
 - Rather than stopping the program many times to carry out a sequence of small graphics commands, they can be carried out together with just a single program interruption.



Drawing the Animation Window

- Another reason is that during animations, there may be many changes occurring on the screen that we need to synchronize. With `autoflush` off, we can make numerous changes that will all show up simultaneously when the update function is called.
- You will almost always want `autoflush` off for animations.



Creating a ShotTracker

- The next thing need is a graphical object that acts like a cannon ball.
 - We can use our `Projectile` class to model the flight of the cannon ball, but `Projectile` is not a graphics object!
 - We could use a `Circle`, but it doesn't know about projectile flight.
 - What we really need is something that has properties of both – let's create a `ShotTracker` that contains both a `Projectile` and a `Circle`.



Creating a ShotTracker

```
class ShotTracker:
```

```
    """ Graphical depiction of a projectile flight using a Circle """
```

```
    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)
```



Creating a ShotTracker

- Did you notice how the parameters have all the information we need to create both a `Projectile` and a `Circle` (`self.proj` and `self.marker`)?
- We need to ensure that whenever an update occurs, both the projectile and position of the circle are updated.
 - The projectile has an update method
 - For the marker, calculate how far it must move in the x and y directions.



Creating a ShotTracker

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

    self.proj.update(dt)
    center = self.marker.getCenter()
    dx = self.proj.getX() - center.getX()
    dy = self.proj.getY() - center.getY()
    self.marker.move(dx,dy)
```



Creating a ShotTracker

```
def getX(self):
    """ return the current x coordinate of the shot's center """
    return self.proj.getX()

def getY(self):
    """ return the current y coordinate of the shot's center """
    return self.proj.getY()

def destroy(self):
    """ undraw the shot """
    self.marker.undraw()
```



Creating an Input Dialog

- Before we can out a cannon ball in flight, we'll need to get the projectile parameters angle, velocity, and initial height from the user.
- A common way of getting user input in a GUI is to use a *dialog box*.
- A dialog box is a sort of mini GUI that serves as an independent component of a larger program.



Creating an Input Dialog

- The user can change the input values and select either “Fire!” to launch the cannon ball or “Quit” to exit the program.
- It’s useful to think of this dialog as just another object the main program can manipulate.
- It will have operations to create the dialog, allow a user to interact with it, and extract the user inputs.



Creating an Input Dialog

- We can create the window itself and draw its contents in the constructor.

```
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))
```



Creating an Input Dialog

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



Creating an Input Dialog

- The constructor accepts parameters that provide default values for the three inputs. That allows the program to seed the dialog with useful inputs as a prompt to the user.
- When it's time for the user to interact with the dialog, we need to make it go modal with its own event loop that waits for mouse clicks and does not exit until one of the buttons has been pressed.



Creating an Input 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!"
```

- The return value from the method is used to indicate which button was clicked to end the interaction.



Creating an Input 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
```

- Things to note:
 - For simplicity, all three inputs are retrieved in a single method call.
 - The strings from the entries are converted to floating point values.



Creating an Input Dialog

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

- This this class, getting values from the user will require just a few lines of code:

```
dialog = InputDialog(45, 40, 2)  
choice = dialog.interact()  
if choice == "Fire!" :  
    angle, vel, height = dialog.getValues()
```

- This has the flexibility of either popping up a new dialog each time input is required, or to keep a single dialog open and interact with it multiple times.



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)
```

```
    # 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)
```



The Main Event Loop

```
# 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()
```




The Main Event Loop

- Each pass through the event loop fires one cannon shot.

```
while 0 <= shot.getY() and -10 < shot.getX() <= 210:  
    shot.update(1/50)  
    update(50)
```

- This while loop keeps updating the shot until it hits the ground or leaves the window horizontally.
- Each time through, the position is updated to move it 1/50th of a second.



The Main Event Loop

- Since `autoflush` is `False`, changes won't appear in the window until the `update(50)` executes.
- The parameter to `update` specifies the rate at which updates are allowed – 50 here means the loop will spin around 50 times per second, establishing the effective frame rate for our simulation.
- The $1/50^{\text{th}}$ per second shot update combined with 50 updates per second gives us a real time simulation, i.e. our simulated cannon ball will stay in flight for the same clock time as the corresponding real cannon ball.



The Main Event Loop

- The big lesson: using separate classes to encapsulate functionality like tracking shots and interacting with the user makes the main program much simpler.
- One shortcoming of our approach is that we can only model the flight of one object at a time. This wouldn't be a suitable design for something like a video game where multiple objects would be in motion.