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 class 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 five 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.
- 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

- 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.

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.

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

- 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 m/s. Shortly after that the object will start coming back down to earth.

Designing the Program

- 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: its 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 ball 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

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

    # Convert angle to radians
    theta = (angle * pi)/180.0
    xvel = vel * cos(theta)
    yvel = vel * sin(theta)

    # Initial position
    xpos = 0
    ypos = h0

    # Main loop
    while ypos >= 0.0:
        # Update position
        xpos = xpos + time * xvel
        ypos = ypos + yvel * time - 0.5 * 9.8 * time ** 2

    # Output the distance traveled
    print("xpos = ", xpos)
Designing the Programs

- To calculate how far the cannonball travels over the interval, we need to calculate its \textit{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:
  \[
y_{	ext{pos}} = y_{	ext{pos}} + \text{time} \times \left( \frac{y_{\text{vel}} + y_{\text{vel1}}}{2} \right)
  \]

# 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 = input("Enter the launch angle (in degrees): ")
    vel = input("Enter the initial velocity (in meters/sec): ")
    h0 = input("Enter the initial height (in meters): ")
    time = 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
        yvel1 = yvel - 9.8 * time
        ypos = ypos + time * (yvel + yvel1)/2.0
        yvel = yvel1
    
    print "Distance traveled: %0.1f meters." % (xpos)

main()

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.

# cball2.py
# Simulation of the flight of a cannon ball (or other projectile)
# This version is modularized.
from math import pi, sin, cos

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 "Distance traveled: %0.1f meters." % (xpos)

main()

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 \texttt{theta} and \texttt{yvel1} are local to \texttt{getXYComponents} and \texttt{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

- 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.

Using our object-based approach:

```python
def main():
    angle, vel, h0, time = getInputs()
    cball = Projectile(angle, vel, h0)
    while cball.getY() >= 0:
        cball.update(time)
    print "Distance traveled: %0.1f meters."
main()
```

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.

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

- 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.

```python
# 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
  ```python
  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 always named `self`, which is a reference to the object on which the method is acting.

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

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

```python
main:
  die1.setValue(8)
```

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
- 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
- Outside the class, the constructor is referred to by the class name:
  ```python
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
- 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
- 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)

data

Example: The Projectile Class

class Projectile:
    def __init__(self, angle, velocity, height):
        self.xpos = 0.0
        self.ypos = height
        theta = pi * angle / 180.0
        self.xvel = velocity * cos(theta)
        self.yvel = velocity * sin(theta)

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

data

Example: The Projectile Class

    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

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.

data

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.

E.g., 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

- Grouping 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.

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 $3 \times 4 = 12$ 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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Hours</th>
<th>QPoints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adams, Henry</td>
<td>127</td>
<td>228</td>
</tr>
<tr>
<td>Comptewell, Susan</td>
<td>100</td>
<td>400</td>
</tr>
<tr>
<td>DibbleBit, Denny</td>
<td>18</td>
<td>41.5</td>
</tr>
<tr>
<td>Jones, Jim</td>
<td>48.5</td>
<td>155</td>
</tr>
<tr>
<td>Smith, Frank</td>
<td>37</td>
<td>125.33</td>
</tr>
</tbody>
</table>

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

```python
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

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

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

- 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!

**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.

---

```python
def get_file_name_from_user()
    file = input("Enter the file name: ")
    return file

def open_file_for_reading(file)
    file_handle = open(file, 'r')
    return file_handle

def set_best_to_first_student()
    best = Student()
```

```python
def find_best_student(file)
    for student in file:
        if student.gpa() > best.gpa:
            best = student
    return best
```

```python
def print_info_about_best(student)
    print("Student Name: ", student.name)
    print("GPA: ", student.gpa())
```

---

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

Module Documentation

- To see the documentation for an entire module, try typing help(module_name)!
- The following code for the projectile class has docstrings.
Module Documentation

from math import pi, sin, cos

class Projectile:
    def __init__(self, angle, velocity, height):
        self.xpos = 0.0
        self.ypos = height
        theta = pi * angle / 180.0
        self.xvel = velocity * cos(theta)
        self.yvel = velocity * sin(theta)

    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

    def getY(self):
        return self.ypos

    def getX(self):
        return self.xpos

Module Documentation

def getInputs():
    a = input("Enter the launch angle (in degrees): ")
    v = input("Enter the initial velocity (in meters/sec): ")
    h = input("Enter the initial height (in meters): ")
    t = 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 "Distance traveled: \%0.1f meters." % (cball.getX())

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.

Widgets

- One very common use of objects is in the design of graphical user interfaces (GUIs).
- Back in chapter 5 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.

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
- Constructor – Create a button in a window. We will specify the window, 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
- 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
- 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
- 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.

```python
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 = 0
```

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

- 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.

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.

```python
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.

```python
def clicked(self, p):
    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 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.

```python
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()
```

- 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.

```python
# 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.:
        # d1 = GDie(myWin, Point(40,50), 20)
        # creates a die centered at (40,50) having sides of length 20.
        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

        # 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)

    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)

        # 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)
```

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
# Simple 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")
    # 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()

main()