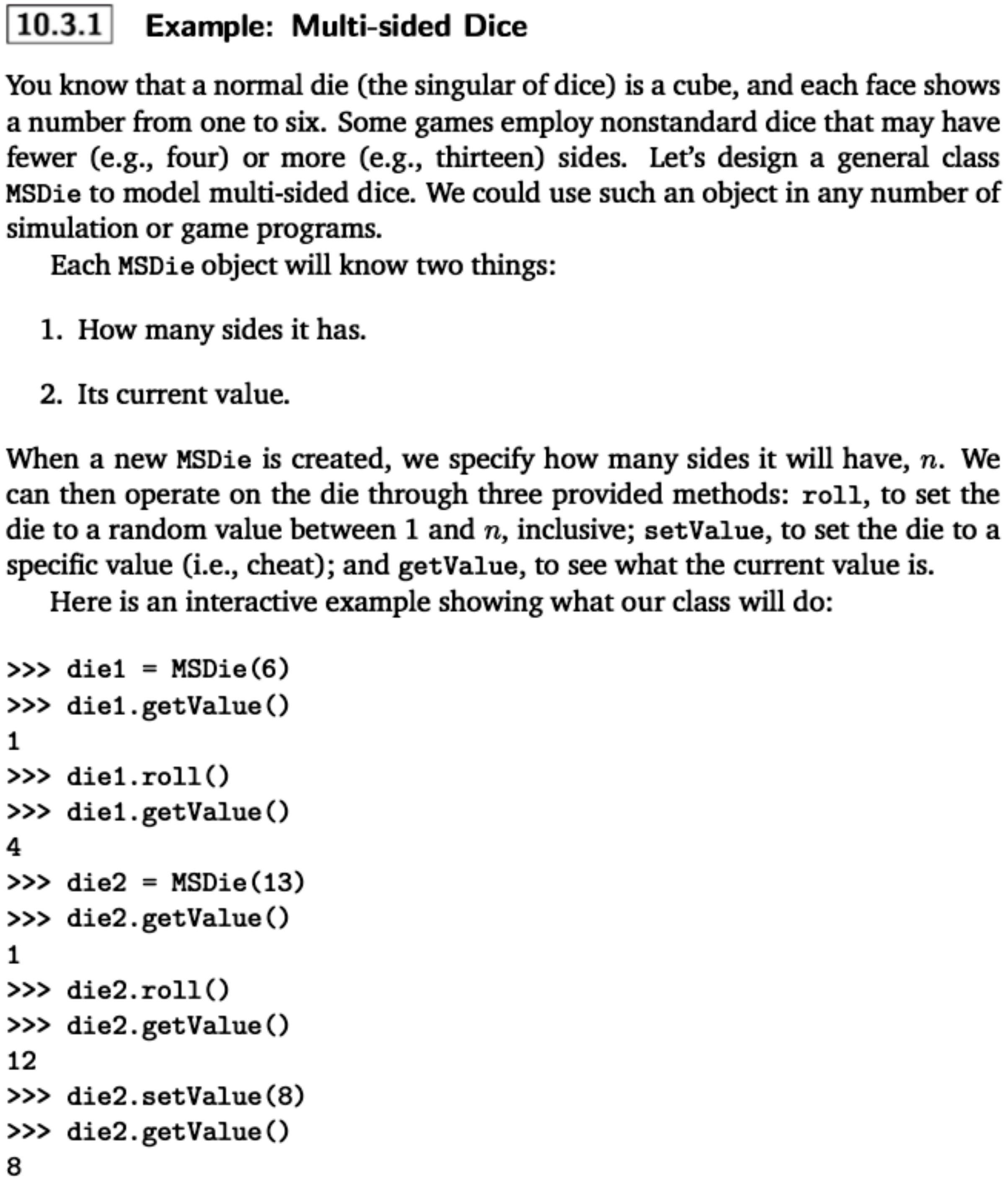
Table of contents

---

1. Class Definition

- class names begin in caps? convention?

(go to top)



10.3.1 Example: Multi-sided Dice

You know that a normal die (the singular of dice) is a cube, and each face shows a number from one to six. Some games employ nonstandard dice that may have fewer (e.g., four) or more (e.g., thirteen) sides. Let's design a general class `MSDie` to model multi-sided dice. We could use such an object in any number of simulation or game programs.

Each `MSDie` object will know two things:

1. How many sides it has.

2. Its current value.

When a new `MSDie` is created, we specify how many sides it will have, n. We can then operate on the die through three provided methods: `roll`, to set the die to a random value between 1 and n, inclusive; `setValue`, to set the die to a specific value (i.e., cheat); and `getValue`, to see what the current value is.

Here is an interactive example showing what our class will do:

```
>>> 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
```

Do you see how this might be useful? I can define any number of dice having arbitrary numbers of sides. Each die can be rolled independently and will always produce a random value in the proper range determined by the number of sides.

Using our object-oriented terminology, we create a die by invoking the `MSDie` constructor and providing the number of sides as a parameter. Our die object will keep track of this number internally using an instance variable. Another instance variable will be used to store the current value of the die. Initially, the value of the die will be set to be 1, since that is a legal value for any die. The value can be changed by the `roll` and `setValue` methods and returned from the `getValue` method.

Writing a definition for the `MSDie` class is really quite simple. A class is a collection of methods, and methods are just functions. Here is the class definition for `MSDie`:

```
# 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
```

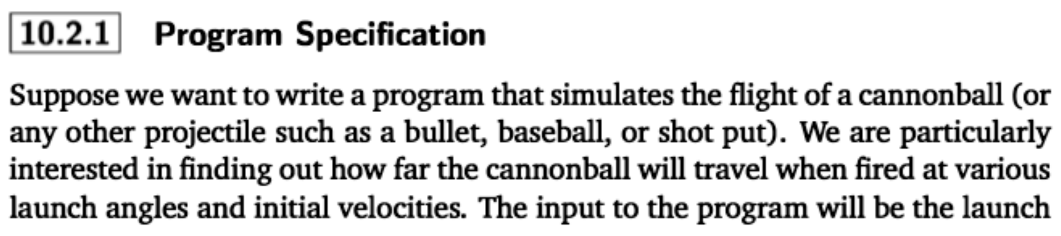
As you can see, a class definition has a simple form:

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

Each method definition looks like a normal function definition. Placing the

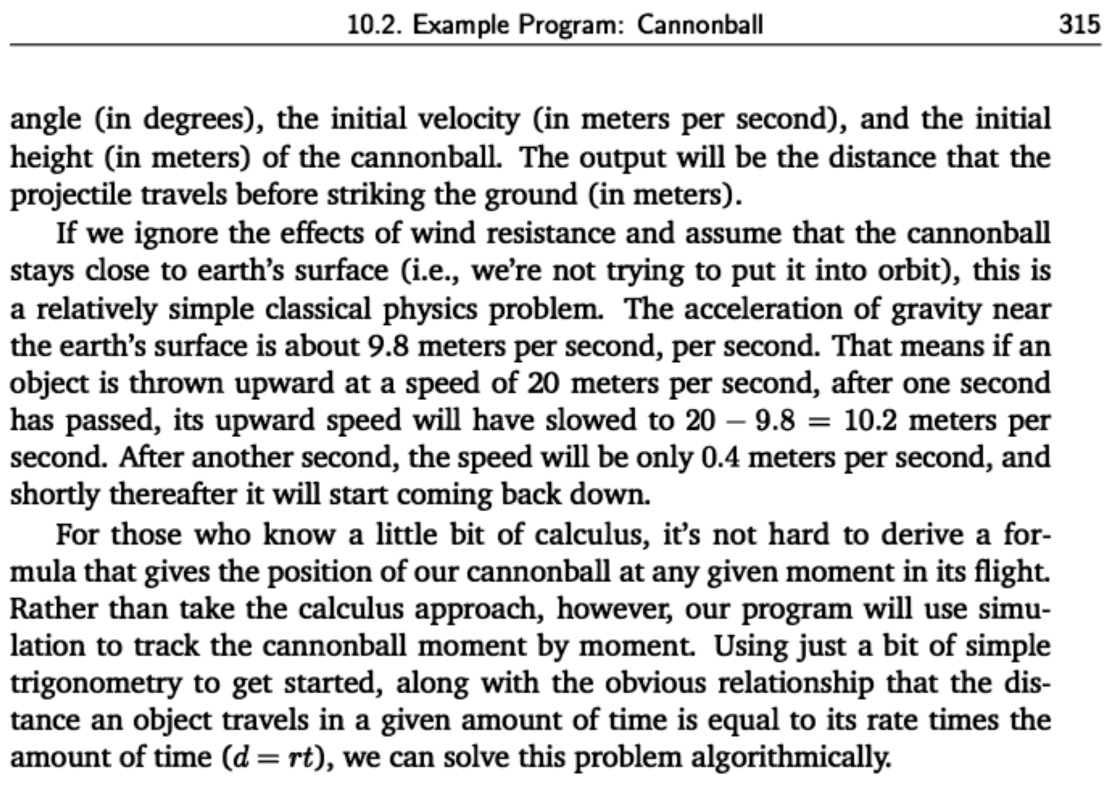
2. Projectile Class

(go to top)



10.2.1 Program Specification

Suppose we want to write a program that simulates the flight of a cannonball (or any other projectile such as a bullet, baseball, or shot put). We are particularly interested in finding out how far the cannonball will travel when fired at various launch angles and initial velocities. 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).

If we ignore the effects of wind resistance and assume that the cannonball stays close to earth's surface (i.e., we're not trying to put it into orbit), this is a relatively simple classical physics problem. The acceleration of gravity near the earth's surface is about 9.8 meters per second, per second. That means if an object is thrown upward at a speed of 20 meters per second, after one second has passed, its upward speed will have slowed to $20 - 9.8 = 10.2$ meters per second. After another second, the speed will be only 0.4 meters per second, and shortly thereafter it will start coming back down.

For those who know a little bit of calculus, it's not hard to derive a formula that gives the position of our cannonball at any given moment in its flight. Rather than take the calculus approach, however, our program will use simulation to track the cannonball moment by moment. Using just a bit of simple trigonometry to get started, along with the obvious relationship that the distance an object travels in a given amount of time is equal to its rate times the amount of time ($d = rt$), we can solve this problem algorithmically.

In [20]:
```
from math import *

class Projectile:

    def __init__(self, angle, velocity, height):
        self.theta = radians(angle)
        self.x_velocity = velocity * cos(self.theta)
        self.y_velocity = velocity * sin(self.theta)
        self.y_position = height
        self.x_position = 0

    def getX(self):
        return self.x_position

    def getY(self):
        return self.y_position

    def update(self, time):
        self.x_position += (time * self.x_velocity)
        y_vel_2 = self.y_velocity - (time*9.8)
        self.y_position += time * (y_vel_2 + self.y_velocity)/2
        self.y_velocity = y_vel_2
```

In [21]:
```
def getInputs():
    angle = float(input("Enter the launch angle (in degrees): "))
    velocity = float(input("Enter the initial velocity (in m/s): "))
    height = float(input("Enter the initial height (in meters): "))
    time = float(input("Enter the time interval between samples/positi

    return angle, velocity, height, time
```

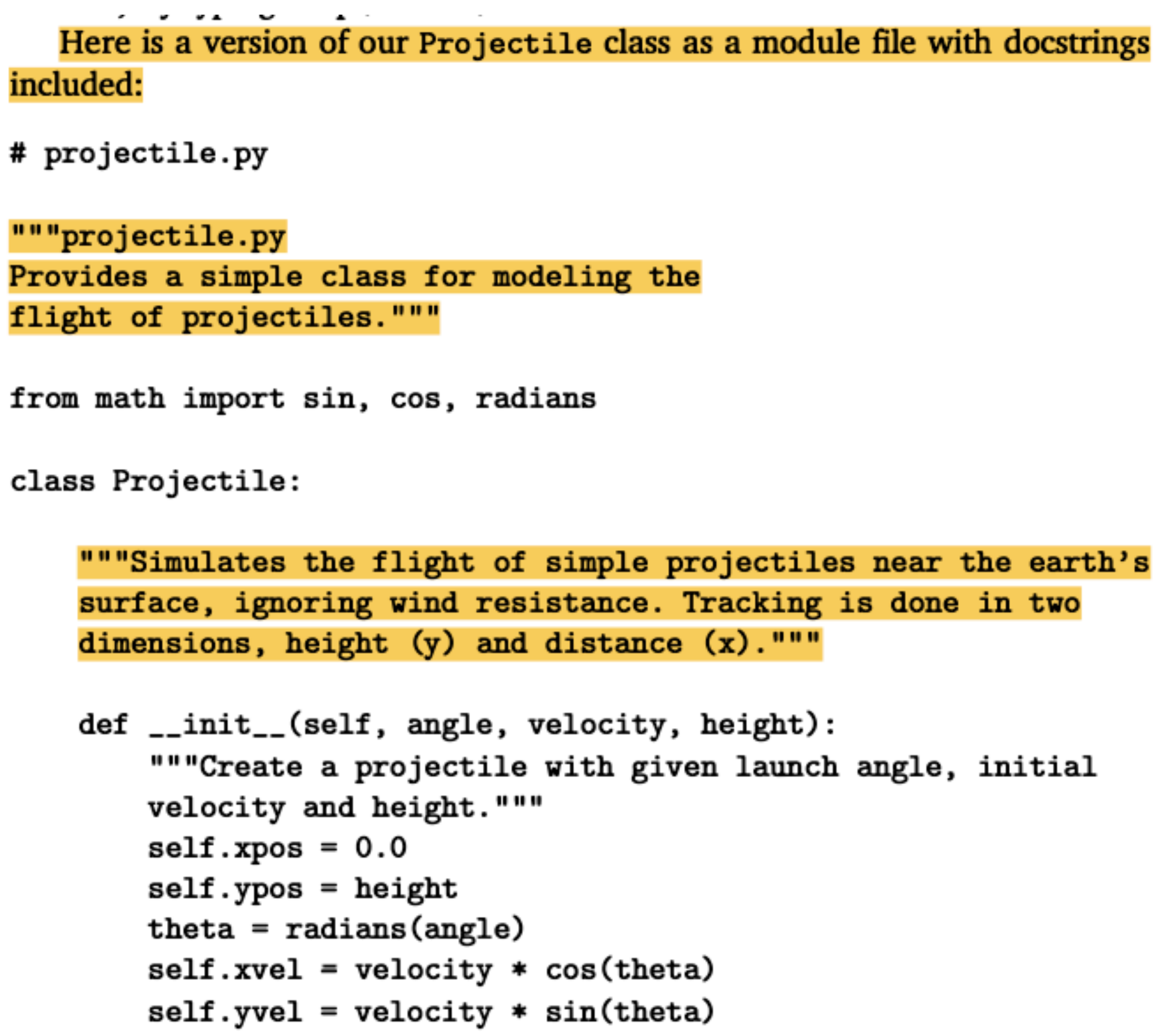
In [28]:
```
def main():
    angle, vel, height, time = getInputs()
    c_ball = Projectile(angle, vel, height)

    while c_ball.getY() >= 0:
        c_ball.update(time)

    print('Distance travelled: {0:0.1f} meters.'.format(c_ball.getX())
```

In [29]:
```
main()
```

```
Enter the launch angle (in degrees): 20
Enter the initial velocity (in m/s): 20
Enter the initial height (in meters): 0
Enter the time interval between samples/positions: 1
Distance travelled: 37.6 meters.
```

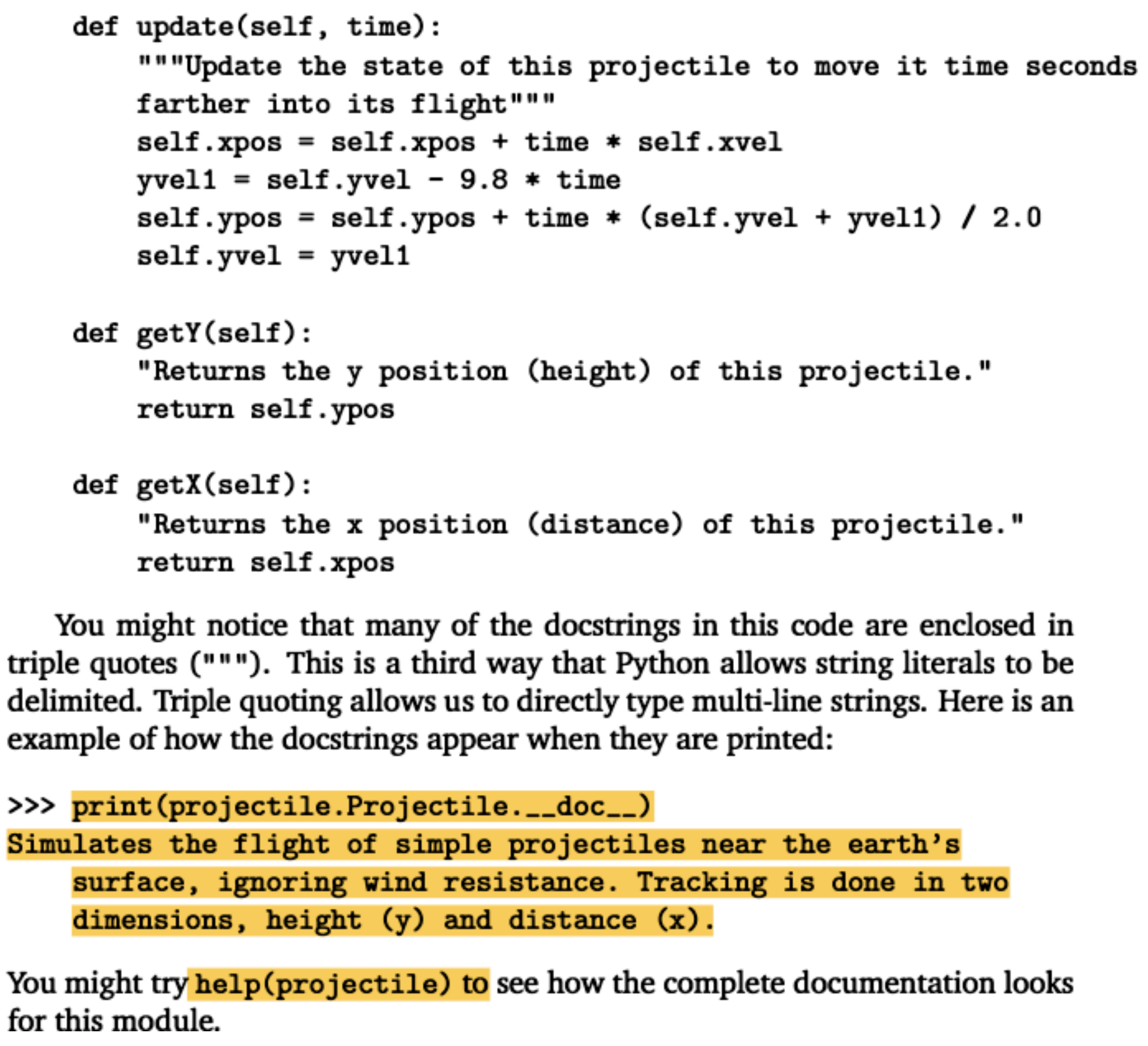
3. Docstring

(go to top)



Here is a version of our `Projectile` class as a module file with docstrings included:

```
# projectile.py

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

from math import sin, cos, radians

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 = radians(angle)
        self.xvel = velocity * cos(theta)
        self.yvel = velocity * sin(theta)
```



```
    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
        yvel1 = self.yvel - 9.8 * time
        self.ypos = self.ypos + time * (self.yvel + yvel1) / 2.0
        self.yvel = yvel1

    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
```

You might notice that many of the docstrings in this code are enclosed in triple quotes ("""). This is a third way that Python allows string literals to be delimited. Triple quoting allows us to directly type multi-line strings. Here is an example of how the docstrings appear when they are printed:

```
>>> print(projectile.Projectile.__doc__)
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).
```

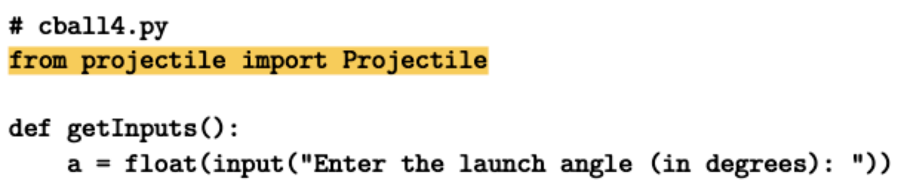
You might try `help(projectile)` to see how the complete documentation looks for this module.

In [ ]:

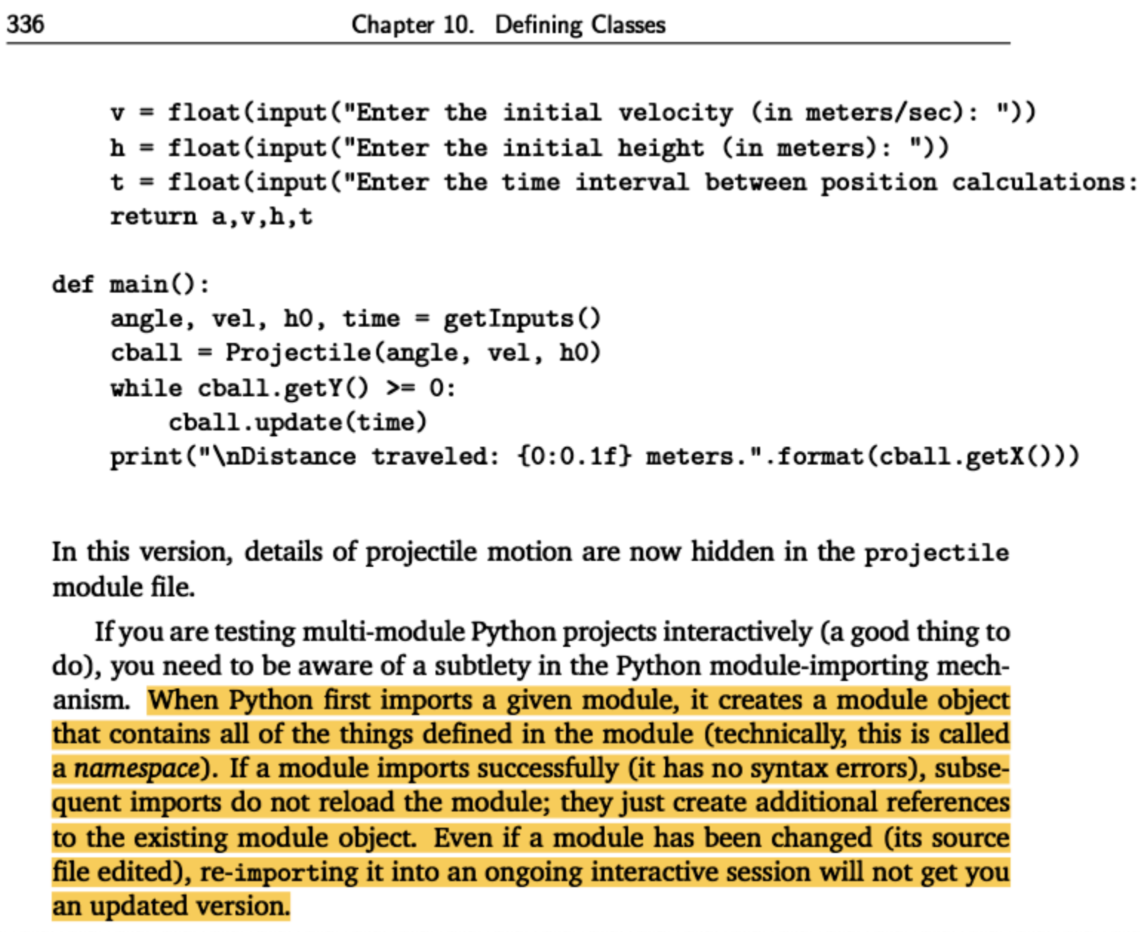
```

```

4. Importing Classes as modules

(go to top)



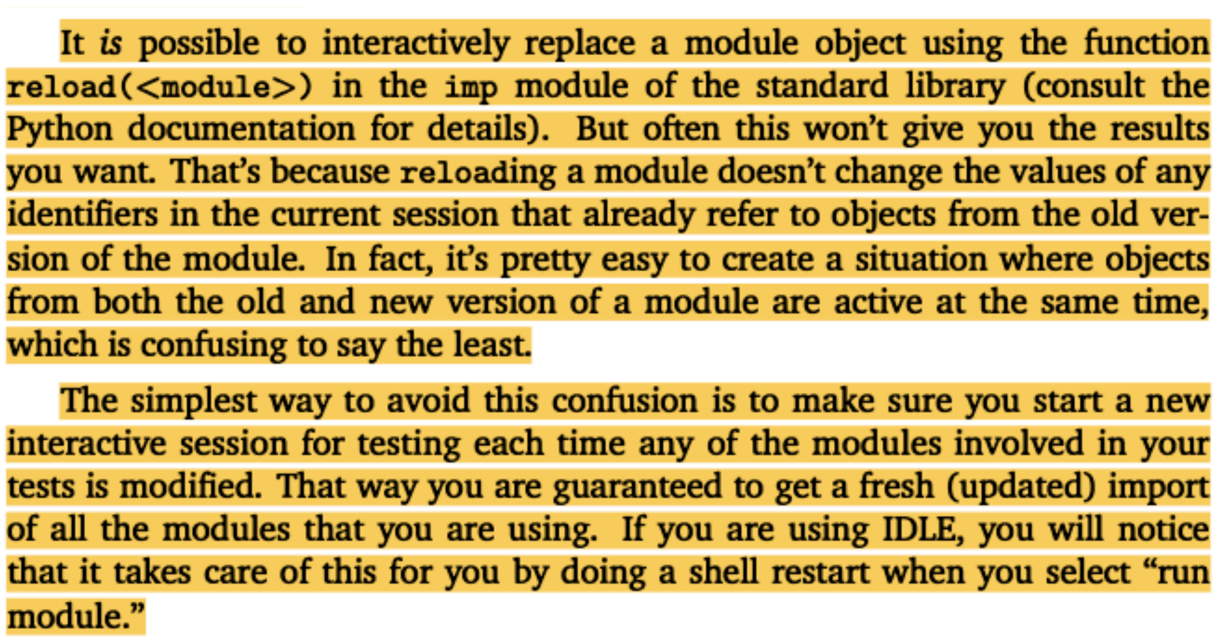
```
# cball4.py
from projectile import Projectile

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()))
```

In this version, details of projectile motion are now hidden in the `projectile` module file.

If you are testing multi-module Python projects interactively (a good thing to do), you need to be aware of a subtlety in the Python module-importing mechanism. When Python first imports a given module, it creates a module object that contains all of the things defined in the module (technically, this is called a *namespace*). If a module imports successfully (it has no syntax errors), subsequent imports do not reload the module; they just create additional references to the existing module object. Even if a module has been changed (its source file edited), re-`import`ing it into an ongoing interactive session will not get you an updated version.



It *is* possible to interactively replace a module object using the function `reload(<module>)` in the `imp` module of the standard library (consult the Python documentation for details). But often this won't give you the results you want. That's because `reload`ing a module doesn't change the values of any identifiers in the current session that already refer to objects from the old version of the module. In fact, it's pretty easy to create a situation where objects from both the old and new version of a module are active at the same time, which is confusing to say the least.

The simplest way to avoid this confusion is to make sure you start a new interactive session for testing each time any of the modules involved in your tests is modified. That way you are guaranteed to get a fresh (updated) import of all the modules that you are using. If you are using IDLE, you will notice that it takes care of this for you by doing a shell restart when you select "run module."

5. Title

(go to top)

6. Title

(go to top)

7. Title

(go to top)

---

8. Title

(go to top)

---

9. Title

(go to top)

---

10. Title

(go to top)

---

11. Title

(go to top)

12. Title

(go to top)

13. Title

(go to top)

14. Title

(go to top)

15. Title

(go to top)

16. Title

(go to top)

---

17. Title

(go to top)

---

18. Title

(go to top)

---

19. Title

(go to top)

---

20. Title

(go to top)

In [ ]:

In [ ]: