Introduction to Programming

Spring 2022

Functions

- **Quick Review of Objects**
- **Example Program: Cannonball**
- Defining New Classes
- Data Processing with Class
- Objects and Encapsulation
- Widgets
- Animated Cannonball

Quick Review of Objects

- In chapters 6, 7, and 8 we developed techniques for structuring the computations of the program.
  - Functions
  - Decision structures
  - Loop Structures
- 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 4 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)
```

- We can also get information about the object by using its methods.

```
myCircle.getCenter()
myCircle.getRadius()
```

Let's Create a simple Class Car

```python
from datetime import date
class Car:
    # constructor method - is called when creates an new object
    def __init__(self, p,y):
        self.plate = p
        self.year = int(y)
        self.age = 0
    # methods
    def getPlate(self): # accessor method
        return self.plate
    def getYear(self):  # accessor method
        return self.year
    def carAge(self): # mutator method
        today = date.today()
        self.age = today.year - self.year
        return self.age

```

```python
def main():
    # create a new Car object
    # Objectname = ClassName(parameters) - in this moment __init__ methos is called
    carObj1 = Car('WS100', 2000)  # plate and year
    carObj2 = Car('XX2222', 1999)
    carObj3 = Car('ZZ2W34', 2010)

    print("Car 1 = ", carObj1.getPlate(), carObj1.getYear(), carObj1.carAge() )
    print("Car 2 = ", carObj2.getPlate(), carObj2.getYear(), carObj2.carAge() )
    print("Car 3 = ", carObj3.getPlate(), carObj3.getYear(), carObj3.carAge() )


```

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.
- 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.

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.
- 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

1) Input the simulation parameters: angle, velocity, height, interval.

2) Calculate the initial position of the cannonball: xpos, ypos

3) Calculate the initial velocities of the cannonball: xvel, yvel

4) While the cannonball is still flying:

1) Update the values of xpos, ypos, and yvel for interval seconds further into the flight

5) Output the distance traveled as xpos

Designing the Program

- Using step-wise refinement:

```
def main():
    # Step 1
    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: "))
```

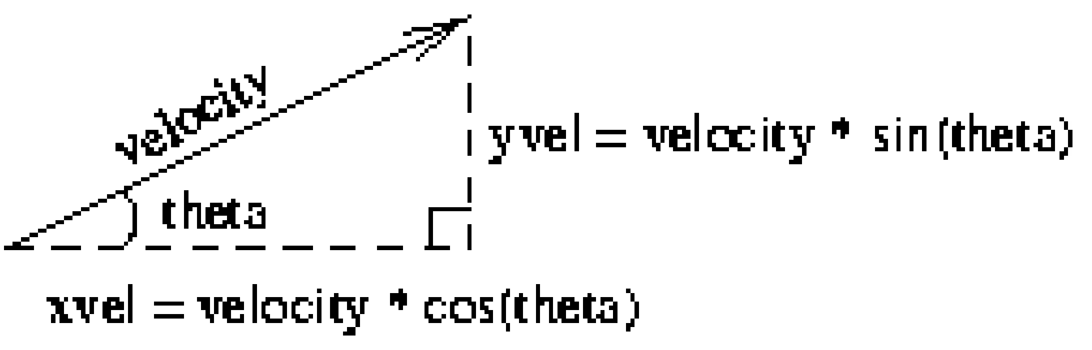
Designing the Program

- Step 2: Calculating the initial position for the cannonball is also easy. It’s at distance 0 and height h0!

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

Designing the Program

- Step 3:
- 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 * math.cos(theta)
yvel = vel * math.sin(theta)
```

Designing the Program

- Step 4: Loop:

```
while the cannonball is still flying:
```

- What does it means?

Designing the Program

- Step 4: Loop:

```
while the cannonball is still flying:
```

- What does it means?
- Cannonball will be flying as long as ypos (distance from the ground >= 0.0
- We use ypos >= 0 becauase initial height can be 0!

```
while ypos >=0:
```

Designing the Program

- Inside the loop – Each time through loop we want to calculate xpos, ypos, yvel

Designing the Program

- Since we assume there is no wind resistance, `xvel` remains constant.
- Say a ball is traveling at 30 m/s and is 0 m from the firing point.
- In one second it will be 30 meters away.
- If the time increment is .1 second it will be 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 Program

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

    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 `yvel1` 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,` 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`.