ISE 314X
Computer Programing for Engineers

Chapter 10
Classes and Object-Oriented Programming (OOP)

Yong Wang
Assistant Professor
Systems Science & Industrial Engineering
Binghamton University

BINGHAMTON UNIVERSITY
STATE UNIVERSITY OF NEW YORK

Objectives

- To understand Python class definitions
- To understand object-oriented programming

Objects

- An object is a data type that consists of the following attributes:
  - Instance Variables: A collection of related information
  - Methods: A set of operations to manipulate that information

Quick Review of Objects

- An object is an *instance* of a *class*
- New objects are created from a class by invoking a *constructor*

Example: Multi-Sided Dice

- Design a generic class MSDie to model a multi-sided die

BINGHAMTON UNIVERSITY
STATE UNIVERSITY OF NEW YORK

Example: Multi-Sided Dice

- Each `MSDie` object (an instance) will know two things (instance variables):
  - How many sides it has
  - Its current value

Example: Multi-Sided Dice

- Three methods that we can use to operate on the die:
  - `roll` – set the die to a random value between 1 and *sides*, inclusive
  - `getValue` – see what the current value is
  - `setValue` – set the die to a specific value on purpose

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

- Placing the function inside a class makes it a method
- `__init__` is the object constructor. Python calls this method to initialize a new `MSDie` instance
- The first parameter of a method is *always* named `self`

Example: Multi-Sided Dice

```
>>> from msdie import MSDie
>>> die1 = MSDie(6)
>>> die1.getValue()
1
>>> die1.roll()
>>> die1.getValue()
5
>>> die1.setValue(3)
>>> die1.getValue()
3
```

Example: Multi-Sided Dice

- When methods are called, we omit the first *self* parameter, and only provide other normal parameters

Example: Multi-Sided Dice

- Instance variables and methods are accessed by name using dot notation:

```
<object>.<instance-var>
<object>.<method>(<parameters>)
```

Difference between Instance Variables and Function Variables

- Instance variables can remember the state of the object
- This is different from local function variables, whose values disappear when the function terminates

Data Processing with Class

- Classes are useful for modeling real-world objects (person or things) with complex behaviors
- Example: An `Employee` class with information such as name, SSN, address, salary, etc.)
- Grouping information like this is called a *record*

Data Processing with Class

- Cumulative GPA
  - Quality points: credits times the letter grade numerical equivalent for each course

    A=4.0 A-=3.7 B+=3.3 B=3.0 B-=2.7

    C+=2.3 C=2.0 C-=1.7 D=1.0 F=0
  - Add up the quality points for all courses
  - Divide total quality points by the total credits for all the courses

Data Processing with Class

Courses	Credits	Letter Grades	Quality Points
ENG 101	3 credits	A(4.0)	3 x 4.0 = 12
SOC 102	2 credits	A-(3.7)	2 x 3.7 = 7.4
ANTH 103	1 credit	C+(2.3)	1 x 2.3 = 2.3
MATH 201	4 credits	B+(3.3)	4 x 3.3 = 13.2
ISE 314	4 credits	A(4.0)	4 x 4.0 = 16

Total credits = 14 Total quality points = 50.9

GPA = (Total quality points)/(Total credits) = 50.9/14 = 3.64

Data Processing with Class

- A data file (`students.dat`) contains grade information of multiple students
- Each line consists of a name, total credit hours, and total quality points

```
Adams, Henry        127     228
Comptewell, Susan   100     400
DibbleBit, Denny    18      41.5
Jones, Jim          48      155
Smith, Frank        37      125.3
```

- Find the student with the highest GPA

Data Processing with Class

```python
# gpa.py
# Program to find student with highest GPA
class Student:
    def __init__(self, name, creds, qpoints):
        self.name = name
        self.creds = float(creds)
        self.qpoints = float(qpoints)
```

Data Processing with Class

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

def getCreds(self):
    return self.creds

def getQPoints(self):
    return self.qpoints

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

Data Processing with Class

- To create a student record:

```
>>> from gpa import *
>>> S1 = Student("Smith, Frank",37,125.3)
>>> S1.name
'Smith, Frank'
>>> S1.creds
37.0
>>> S1.qpoints
125.3
```

Data Processing with Class

```
>>> S1.getName()
'Smith, Frank'
>>> S1.getCreds()
37.0
>>> S1.getQPoints()
125.3
>>> S1.gpa()
3.386486486486486 3
```

Data Processing with Class

- To find the student with the highest GPA, we can use an algorithm similar to finding the max of *n* numbers
- Look through the list one by one, keeping track of the highest GPA 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

```
def main():
    # open the input file for reading
    filename = input("Enter the name of the grade file: ")
    infile = open(filename, 'r')
    # set best to the record for the first student in the file
    best = makeStudent(infile.readline())
    # process subsequent lines of the file
    for line in infile:
        # turn the line into a student record
        s = makeStudent(line)
        # if this student is best so far, remember it.
        if s.gpa() > best.gpa():
            best = s
    infile.close()
    # print information about the best student
    print("The student with highest GPA:", best.getName())
    print("GPA:", best.gpa())

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

Data Processing with Class

```python
def makeStudent(infoStr):
    # infoStr is a tab-separated line: name creds qpoints
    # returns a corresponding Student object
    name, creds, qpoints = infoStr.split("\t")
    return Student(name, creds, qpoints)
```

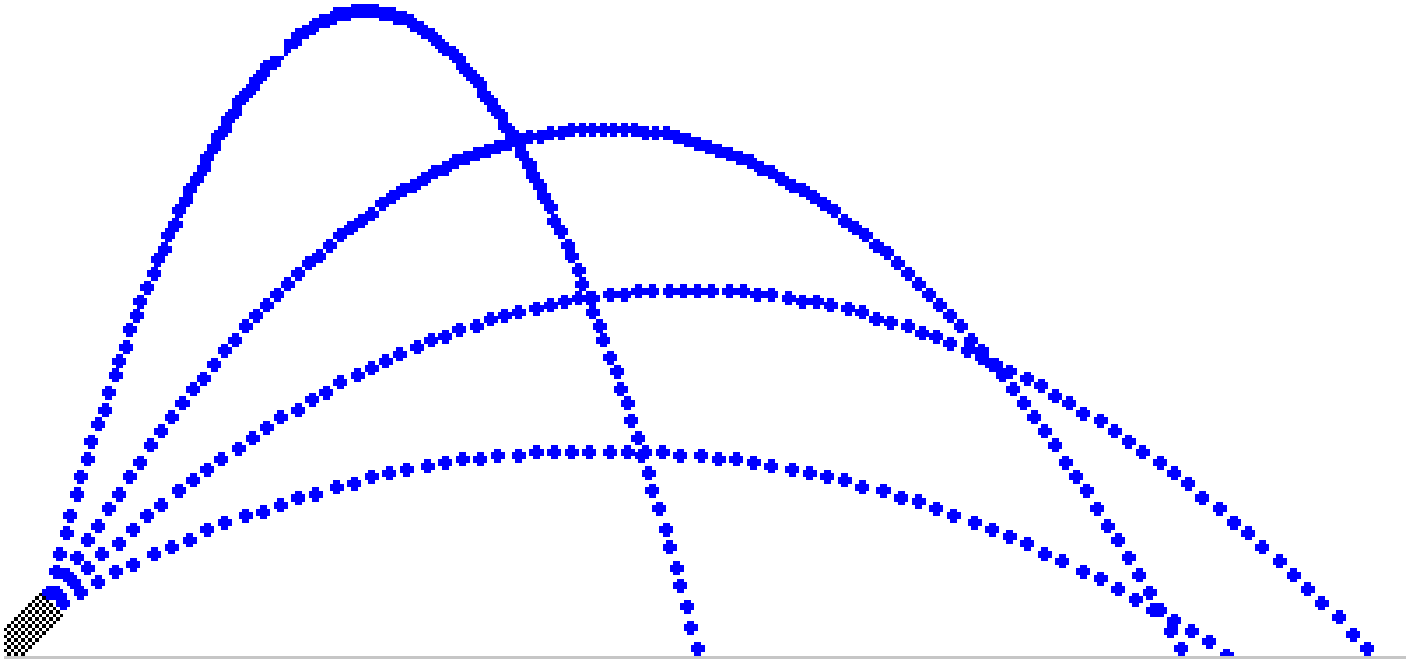
Data Processing with Class

- Run the program gpa.py

```
Enter name the grade file: students.dat
The student with highest GPA: Computewell, Susan
GPA: 4.0
```

Cannonball Program Specification

- Write a program that simulates the flight of a cannonball



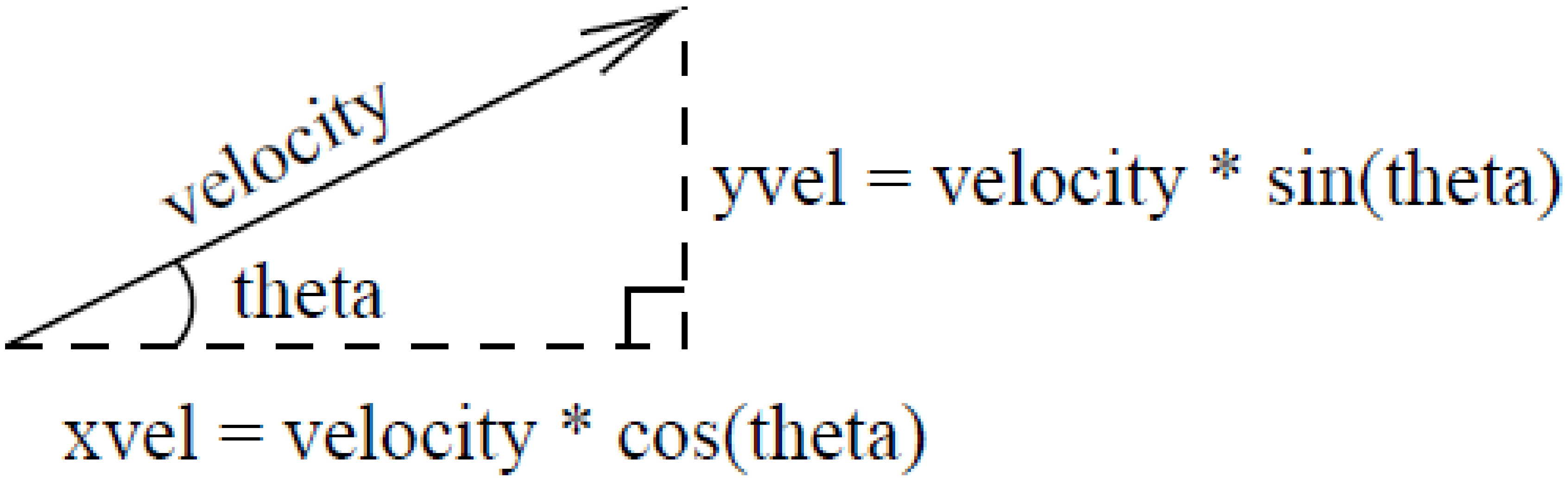
Cannonball Program Specification

- The input is the launch angle (in degrees) and the initial velocity (in m/s)
- The output is the trajectory positions (in meters) in two dimensions (*xpos*, *ypos*)

Designing the Program

- The acceleration of gravity is 9.8 m/s^2
- The ball always starts at position (0, 0)
- We check its position in both directions every 0.1 seconds

Designing the Program



Designing the Program

- We ignore wind resistance
- *xvel* will remain constant through the flight
- *yvel* will change over time due to gravity

Designing the Program

- Input the simulation parameters: angle, velocity, time interval
- Set the initial position of the cannonball: xpos, ypos
- Calculate the initial velocities in two directions: xvel, yvel
- While the cannonball is still flying:
  - Update the values of xpos, ypos, yvel for the time interval
  - Output xpos, ypos

Designing the Program

- In the main loop, we keep updating the position of the ball until `ypos ≤ 0` (i.e., hitting the ground)

Designing the Program

- Update yvel: Each `time` seconds, `yvel` must decrease by 9.8*`time` m/s due to gravity

```
yvel1 = yvel - 9.8 * time
```

- Update xpos:

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

- Update ypos:

```
ypos = ypos + time * (yvel+yvel1)/2
```

Designing Programs

```
# cball1.py
from math import pi, sin, cos, radians
def main():
    angle = eval(input("Enter the launch angle (in degrees):"))
    vel = eval(input("Enter the initial velocity (in meters/sec):"))
    time = eval(input("Enter time interval between calculations:"))
    theta = radians(angle)      # convert angle to radians
    xpos = 0                    # the initial position
    ypos = 0
    xvel = vel * cos(theta)
    yvel = vel * sin(theta)
    while ypos >= 0:            # loop until the ball hits the ground
        # update position and velocity in time seconds
        xpos = xpos + time * xvel
        yvel1 = yvel - time * 9.8
        ypos = ypos + time * (yvel + yvel1)/2.0
        yvel = yvel1
        print("(xpos,ypos): ({},{})".format(xpos, ypos))
main()
```

Designing Programs

```
Enter the launch angle (in degrees):30
Enter the initial velocity (in meters/sec):10
Enter time interval between calculations:0.1
(xpos,ypos): (0.8660254037844388,0.45099999999999996)
(xpos,ypos): (1.7320508075688776,0.8039999999999998)
(xpos,ypos): (2.5980762113533165,1.0589999999999997)
(xpos,ypos): (3.4641016151377553,1.2159999999999995)
(xpos,ypos): (4.330127018922194,1.2749999999999995)
(xpos,ypos): (5.196152422706634,1.2359999999999993)
(xpos,ypos): (6.062177826491073,1.0989999999999993)
(xpos,ypos): (6.928203230275512,0.8639999999999993)
(xpos,ypos): (7.794228634059952,0.5309999999999988)
(xpos,ypos): (8.660254037844391,0.09999999999999865)
(xpos,ypos): (9.526279441628831,-0.42900000000000016)
```

Modularizing the Program

- This program is complex due to the number of variables
- Divide the program into functions

Modularizing the Program

```python
# cball2.py
from math import pi, sin, cos, radians

def main():
    angle, vel, time = getInputs()
    xpos, ypos = 0, 0
    xvel, yvel = getXYComponents(vel, angle)
    while ypos >= 0:        # update position and velocity
        xpos,ypos,yvel= updateCannonBall(time,xpos,ypos,xvel,yvel)
        print("(xpos,ypos): ({},{})".format(xpos, ypos))

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

Modularizing the Program

```
def getXYComponents(vel, angle):
    theta = radians(angle)
    x = vel * cos(theta)
    y = vel * sin(theta)
    return x, y

def updateCannonBall(time, xpos, ypos, xvel, yvel):
        xpos = xpos + time * xvel
        yvel1 = yvel - 9.8 * time
        ypos = ypos + time * (yvel + yvel1)/2.0
        return xpos, ypos, yvel1

main()
```

Designing Programs

```
Enter the launch angle (in degrees):30
Enter the initial velocity (in meters/sec):10
Enter time interval between calculations:0.1
(xpos,ypos): (0.8660254037844388,0.45099999999999996)
(xpos,ypos): (1.7320508075688776,0.8039999999999998)
(xpos,ypos): (2.5980762113533165,1.0589999999999997)
(xpos,ypos): (3.4641016151377553,1.2159999999999995)
(xpos,ypos): (4.330127018922194,1.2749999999999995)
(xpos,ypos): (5.196152422706634,1.2359999999999993)
(xpos,ypos): (6.062177826491073,1.0989999999999999)
(xpos,ypos): (6.928203230275512,0.8639999999999999)
(xpos,ypos): (7.794228634059952,0.5309999999999988)
(xpos,ypos): (8.660254037844439,0.09999999999999865)
(xpos,ypos): (9.526279441628883,-0.4290000000000016)
```

Modularizing the Program

- This version of the program is modularized
- It is easier to follow, test, and maintain
- However, five parameters `xvel, yvel, xpos, ypos,` and `time`, are needed to compute and return the three new values