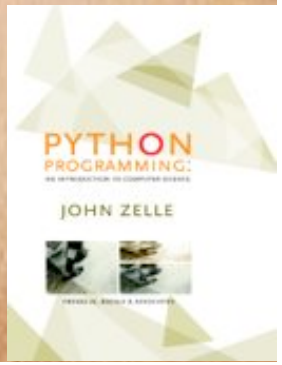
Python Programming: An Introduction To Computer Science

Chapter 10

Defining Classes

(These slides were heavily editted/ rewritten by Dan Fleck)



Coming up: Objectives

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.

What are 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.

What are 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.

Design of Circle object

- A `Circle` object will have instance variables
  - `center`, which remembers the center point of the circle,
  - `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.

Design of Circle

- `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.

Circle Class

```
class Circle:
```

Beginning of the class definition

```
    def __init__(self, center, radius):
        self.center = center
        self.radius = radius

    def draw(self, canvas):
        rad = self.radius
        x1 = self.center[0]-rad
        y1 = self.center[1]-rad
        x2 = self.center[0]+rad
        y2 = self.center[1]+rad
        canvas.create_oval(x1, y1, x2, y2, fill='green')

    def move(self, x, y):
        self.center = [x, y]
```

The constructor. This is called when someone creates a new Circle, these assignments create instance variables.

A method that uses instance variables to draw the circle

A method that sets the center to a new location and then redraws it

Creating a Circle

- 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([10,30], 20)
```
- `Circle`, the name of the class, is used to invoke the constructor.
- The constructor is ALWAYS named __init__ (it's a special name for this method)
- Notice: I do not pass "self" as a parameter… it is automatic!

Creating a Circle

```
myCircle = Circle([10,30], 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`.

Creating a Circle

```
myCircle = Circle([10,30], 20)
```

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

```
myCircle.draw(canvas)
myCircle.move(x,y)
```

Objects and Classes

- `myCircle = Circle([10,30], 20)`
- `myOtherCircle = Circle([4,60], 10)`

- myCircle and myOtherCircle are INSTANCES of the Class Circle also, myCircle and myOtherCircle are OBJECTS, instead of Classes

- The constructor is called when you do this line:
  - `myCircle = Circle([10,30], 20)`

Using the Circle

- from CircleModule import *

```
myCircle = Circle([10,30], 20)
print
  "CENTER :"+str(myCircle.center)
>>> CENTER :(10, 30)
```

To get an instance variable from an object, use: <<object>>.variable

What happens if the instance variable doesn't exist?

Using Instance Variables

```
myCircle = Circle([10,30], 20)
print "CENTER :"+str(circle.carl)
>>> AttributeError: Circle
  instance has no attribute
  'carl'
```

What happens if you set an instance variable that doesn't exist?

```
myCircle.bob = 234
```

Using Instance Variables

```
myCircle.bob = 234
```

What happens if you set an instance variable that doesn't exist?

Think: What happens if you assign ANY variable in python that doesn't exist?

```
john = 234
```

Python automatically creates a new variable if it doesn't exist. For instance variables this works the same… if you assign an instance variable that doesn't exist, Python just creates it

Summary: Using instance variables

- Creating new instance variables just means assigning them a value:
  - myCircle.bob = 234
  - self.bob = 234
- Using instance variables is done through dot notation:
  - val = myCircle.bob
  - val = self.bob

Using Operations in Objects

- Operations are created just like a function, but inside a class:

```
class Circle:
    def myFunction(self, p1, p2):
        << something >>>
    def function2(self, input1='55'):
        <<something>>
```

- To use operations, call them using dot notation:

```
myCircle.myFunction(actualP1, actualP2)
```

Note: self is automatically passed in to all operations… you never pass it in yourself!

What is ‘self’

- Self is a reference to the current instance. Self lets you access all the instance variables for the specific instance you’re working with.

Why use classes at all?

- Classes and object are more like the real world. They minimize the semantic gap.
- The semantic gap is the difference between the real world and the representation in a computer.
- Do you care how your TV works?
  - No… you are a user of the TV, the TV has operations and they work. You don't care how.

Why use classes at all?

- Classes model more closely real world objects… you must define them, but after that you just USE them without knowing (or caring) how they work inside.
- Do you care how a Button fills in each pixel to display itself on the screen?
  - No.. .you want to USE the button.

Why use classes at all?

- Classes and objects allow you to define an interface to some object (it's operations) and then use them without know the internals.

- Defining classes helps modularize your program into multiple objects that work together, that each have a defined purpose

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.
- If Python-masters change how the Button puts pixels on the screen, do you care?

Example: Bouncing Ball

- Lets try to create a bouncing ball class. Essentially this will be a ball that has a velocity and can bounce around a window.
- Specification
  - We want to specify initial position, velocity, color and bounds (where are the walls)
  - We then want to call an update method that moves the ball

Screen Layout

Example: Bouncing Ball

- class Ball:

  def __init__(self, xLoc, yLoc, xVel, yVel, color, leftWall,rightWall, topWall, bottomWall)

  # Should initialize everything

  def update()

  # Should move the ball and let it bounce appropriatly

Example: Bouncing Ball

- class Ball:

```
def __init__(self, xLoc, yLoc, xVelocity, yVelocity, color='green',  \
                    leftWall=0, rightWall=400, topWall=0, bottomWall=400):
    self.xLoc = xLoc
    self.yLoc = yLoc
    self.xVelocity = xVelocity
    self.yVelocity = yVelocity
    self.leftWall = leftWall
    self.rightWall = rightWall
    self.topWall = topWall
    self.bottomWall = bottomWall
    self.color = color
```

Example: Bouncing Ball

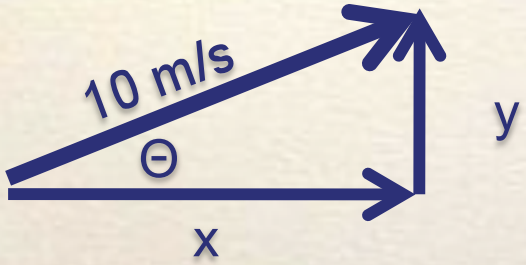
- class Ball:

```
# Draw the initial ball
  def draw(self, canvas):
      rad = 5
      x1 = self.xLoc-rad   # Top left corner
      y1 = self.yLoc-rad  # Top left corner
      x2 = self.xLoc+rad  # Bottom right corner
      y2 = self.yLoc+rad  # Bottom right corner

      # To create an oval, you specify the bounding box (top left and bottom right corner,
      # then the oval fills in the space.
      self.itm = canvas.create_oval(x1, y1, x2, y2, fill=self.color)
      self.canvas = canvas
```

Bouncing Ball: Physics 101

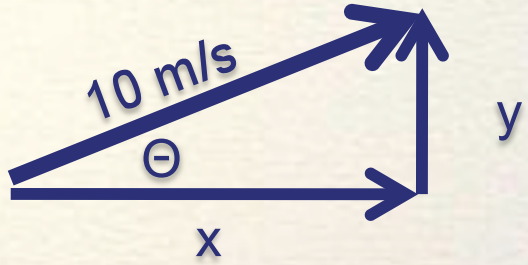
- gravity accelerates items at 9.8m/s^2
  - so every second you fall, your speed increases by 9.8m/s
- Our velocity has two components



  - Assuming Θ is 30 degrees
  - cos(Θ) = x / 10
  - sin(Θ) = y / 10

Bouncing Ball: Physics 101

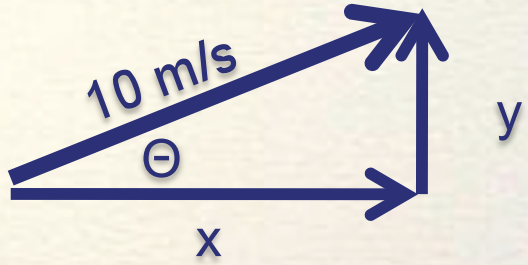
- Our velocity has two components



  - Assuming Θ is 30 degrees
  - cos(Θ) = x / 10
  - sin(Θ) = y / 10
    - x = 10 cos(30) =8.66 m/s
    - y = 10 sin (30) =0.5 m/s

Bouncing Ball: Physics 101

- Our velocity has two components



  - So, if our ball is travelling at 10 m/s, the y velocity is subject to gravity, but not the x. (we'll ignore wind resistance and all other factors)
  - So the first second we travel 8.66 meters in X and 0.5 meters in Y

Bouncing Ball: Physics 101

- Our update function will use simulation to keep the ball moving:
  - update():

```
# If we call update every second, then the change in X and Y directions are just their
# velocity (since it's in meters/second)
deltaX = 8.66 # Velocity in X direction never changes
yVelocity = yVelocity – 9.8 # Gravity
deltaY =  yVelocity

# Move the ball
self.canvas.move(self.itm, deltaX, deltaY)
```

This gives us a falling ball, how do we make it bounce?

Bouncing Ball: Physics 101

- If we hit the “floor”, change the yVelocity from positive to negative, and reduce it some (we bounce a little lower than we started)

```
# Bounce off the "floor"
if self.yLoc > self.bottomWall:
    self.yVelocity = -1 * self.yVelocity * self.bouncyness
    deltaY = self.bottomWall - self.yLoc # Make sure you’re above the floor!
else:
    deltaY = int(self.yVelocity)
self.yLoc += deltaY
```

Now we bounce up and down, what about left and right wall?

Bouncing Ball: Physics 101

- If we hit the left/right wall, just change our x direction

```
# Bounce off the "wall"
if self.xLoc > self.rightWall or self.xLoc < self.leftWall:
    self.xVelocity *= -1

deltaX = self.xVelocity/5
self.xLoc += deltaX
```

Great… but the balls should stop not keep rolling around

Bouncing Ball: Physics 101

- If we get to a very small yVelocity, just stop bouncing and rolling.

```
 # The ball isn't bouncing... stop!
if abs(self.yVelocity) < 10 and self.yLoc >= (self.bottomWall-5):
    self.yVelocity = 0
    self.xVelocity = 0
    return
else:
    self.yVelocity += 2 #9.8/5
```

Bouncing Ball: Physics 101

- Now it's easy to create a whole bunch of balls because they are Objects, and each will maintain it's own state (velocities)

```
for i in range(10):
    rcolor = '#%d%d%d' %(randint(0,9), randint(0,9), randint(0,9)) # Random color
    ball = Ball(randint(left,right), randint(top,bottom), randint(2,20),  \
                randint(2,20), color=rcolor, \
                leftWall=left, rightWall=right, topWall=top, bottomWall=bottom)
    ball.draw(canvas)
    balls.append(ball)
```

Design Summary

- Think about each “object” in your system
  - What behaviors should it have?
  - What information does it need to know? What information changes from one instance of this object to the next?
- There are many books on design strategies for object oriented programming!

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

- 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.

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
  ```

These are commonly called “getters”

- 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

Pseudocode:

```
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
import string

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

```
def main():
    filename = raw_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()
```

Why use getters?

- In general you should avoid using dot notation to read instance variables from anywhere except within the class itself.
- So, since main was not in the Class, we used getters instead.
- This is called data encapsulation. It hides your implementation inside the Class from the user of your class (main in this example).
- Doing it this way lets me update my class and change anything I want except the method names/parameters (it's signature)

Why use getters?

- Assume I have getter:

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

What if I want to store the name instead as first and last name in the class? Well, with the getter I only have to do this:

```
def getName(self):
    return self.firstname + self.lastname
```

If I had used dot notation outside the class, then all the code OUTSIDE the class would need to be changed because the internal structure INSIDE the class changed. Think about libraries of code… If the Python-authors change how the Button class works, do you want to have to change YOUR code? No! Encapsulation helps make that happen. They can change anything inside they want, and as long as they don't change the method signatures, your code will work fine.

Setters

- Anoter common method type are “setters”

```
def setAge(self, age):
    self.age = age
```

Why? Same reason + one more. I want to hide the internal structure of my Class, so I want people to go through my methods to get and set instance variables. What if I wanted to start storing people’s ages in dog-years? Easy with setters:

```
def setAge(self, age):
    self.age = age / 7
```

More commonly, what if I want to add validation… for example, no age can be over 200 or below 0? If people use dot notation, I cannot do it. With setters:

```
def setAge(self, age):
    if age > 200 or age < 0:
        # show error
    else:
        self.age = age / 7
```

Getters and Setters

- Getters and setters are useful to provide data encapsulation. They should be used!
- CS211 Preview: In Java you will be able to enforce access restrictions on your instance variables… you can (and should) make them private so Java itself enforces data encapsulation.
- So… does Python support “private” data? Yes (and no)

Hiding your private parts (in Python)

- You can create somewhat private parts in Python. Naming an instance variable with an __ (two underscores) makes it private.
- Example:

```
class Circle:
    def __init___(self):
        __name = ‘I am private’

def main():
    circ = Circle()
    nm = circ.name

Traceback (most recent call last):
  File "Private.py", line 16, in <module>
    main()
  File "Private.py", line 12, in main
    nm = circ.name
AttributeError: Circle instance has no attribute 'name’
```

Hiding your private parts (in Python)

- Be a little sneakier then.. use __name:
- Example:

```
class Circle:
    def __init___(self):
        __name = 'I am private'

def main():
    circ = Circle()
    nm = circ.__name
```

- Traceback (most recent call last):

```
  File "Private.py", line 16, in <module>
    main()
  File "Private.py", line 12, in main
    nm = circ.name
AttributeError: Circle instance has no attribute '__name'
```

Nice try, but that won't work!

Hiding your private parts (in Python)

- Be super sneaky then.. use _Circle__name:
- Example:

```
class Circle:
    def __init___(self):
        self.__name = 'I am private'

def main():
    circ = Circle()
    nm = circ._Circle__name
    print nm
```

- >>> I am private

Ahh… you saw my private parts… that was rude!

So, it is possible to interact with private data in Python, but it is difficult and good programers should know not to do it. Using the defined interfaces will make code more maintainable and safer to use

Private Instance Variables

- So it is good pratice to make all instance variables private by naming them beginning with two underscores.
- Then use getters to access them
- This also means you can create “read only” instance variables, but making them private and creating a getter but not a setter.

Helping other people use your classes

- Frequently, you will need to write classes other people will use
- Or classes you will want to use later, but have forgotton how

Answer: Document your class usage!

Putting Classes in Modules

- Sometimes we may program a class that could 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)`!
- 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
        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
```

PyDoc

- PyDoc The pydoc module automatically generates documentation from Python modules. The documentation can be presented as pages of text on the console, served to a Web browser, or saved to HTML files.

- pydoc –g # Launch the GUI