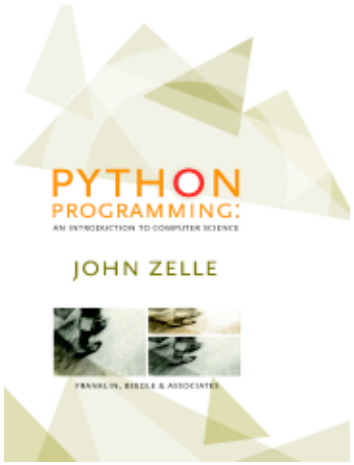
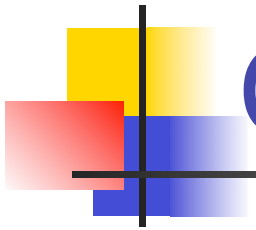
Python Programming: An Introduction To Computer Science

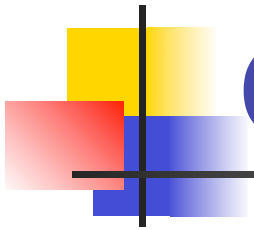


Chapter 10
Defining Classes



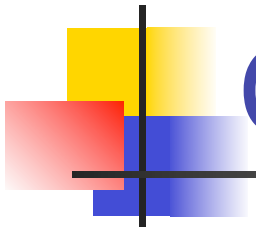
Objectives

- Appreciate how defining new classes provide structure for complex programs.
- Read and write Python class definitions.
- Understand concept of encapsulation and how it contributes to modular and maintainable programs.
- Write programs involving simple class definitions.



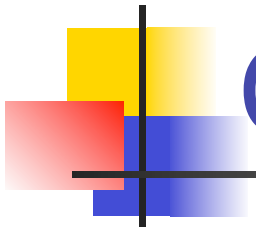
Quick Review of Objects

- 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 `sphere`. In this chapter we'll learn how to write our own classes to create novel objects.



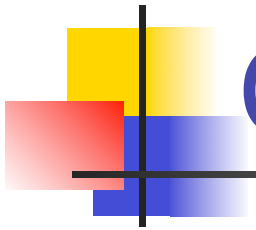
Quick Review of Objects

- 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 (state).
  2. A set of operations to manipulate that information (behavior).



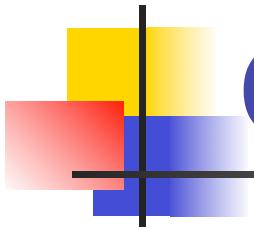
Quick Review of Objects

- 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.



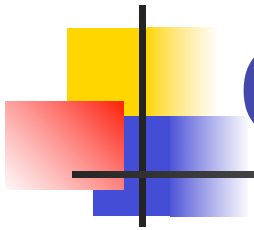
Quick Review of Objects

- A `sphere` object will have instance variables such as `pos`, which remembers the center point of the sphere, and `radius`, which stores the magnitude of the sphere's radius.
- VPython monitors the values of `pos` and `radius` to decide which pixels in the display should be colored.



Quick Review of Objects

- Changing the value of `pos` causes the sphere to move to a new position.
- 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.

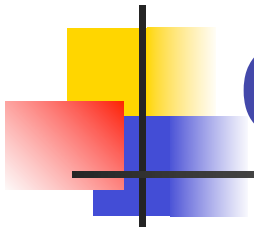


Quick Review of Objects

- 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 sphere object:

```
ball = sphere(pos=(0,0,0),radius=1)
```

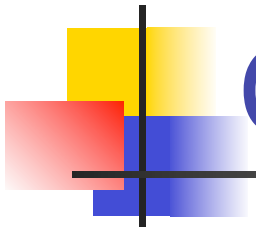
- `sphere`, the name of the class, is used to invoke the constructor.



Quick Review of Objects

```
ball = sphere(pos=(0,0,0),radius=1)
```

- This statement creates a new `sphere` instance and stores a reference to it in the variable `ball`.
- The parameters to the constructor are used to initialize some of the instance variables (`pos` and `radius`) inside the instance referred to by `ball`.



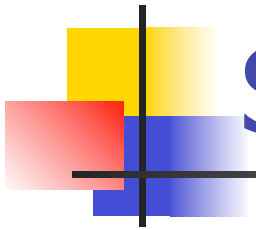
Quick Review of Objects

```
ball = sphere(pos=(0,0,0),radius=1)
```

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

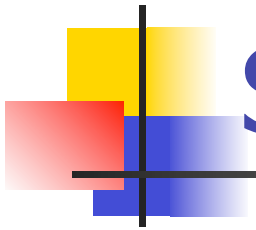
```
ball.rotate(angle=pi/4.)
```

Cannonball Program Specification



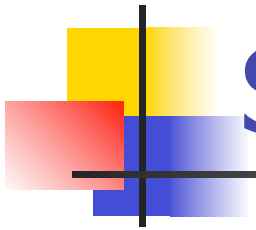
- Let's try to write a program that simulates the flight of a cannonball or other projectile.
- We're interested in how far the cannonball will travel when fired at various launch angles and initial velocities.

Cannonball Program Specification



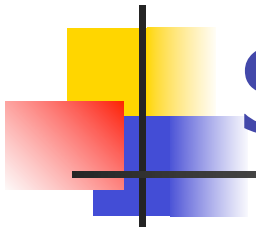
- The input to the program will be the launch angle (in degrees), the initial velocity (m/s), and the initial height (m) of the cannonball.
- The output will be the distance that the projectile travels before striking the ground (m).

Cannonball Program Specification

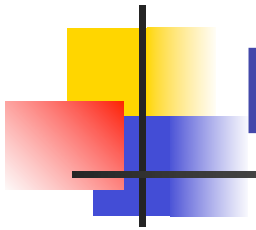


- The acceleration of gravity near the earth's surface is roughly 9.8 m/s/s.
- If an object is thrown straight up at 20 m/s, after one second it will be traveling upwards at 10.2 m/s. After another second, its speed will be .4 m/s. Shortly after that the object will start coming back down to earth.

Cannonball Program Specification

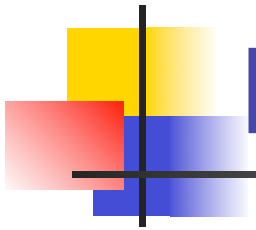


- Using calculus, we could derive a formula that gives the position of the cannonball at any moment of its flight.
- However, we'll solve this problem with simulation, a little geometry, and the fact that the distance an object travels in a certain amount of time is equal to its rate times the amount of time (d = r t).



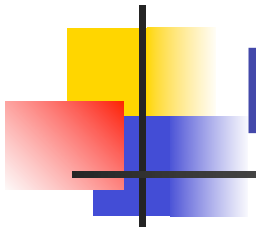
Designing the Program

- 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, z) where x is the distance from the starting point and y is the height above the ground.



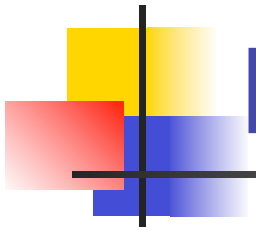
Designing the Program

- Suppose the ball starts at position (0,0,0), 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.



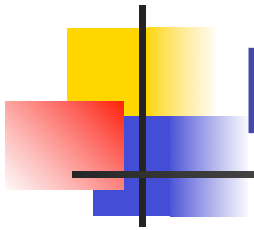
Designing the Program

- 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 velocity 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



Designing the Program

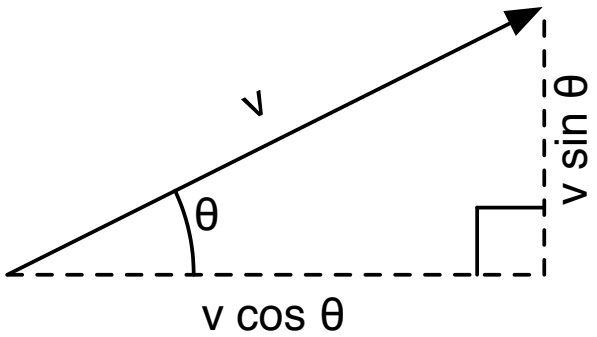
- Using step-wise refinement:

```
def main():
  angle = input("Enter the launch angle (in degrees): ")
  v0 = input("Enter the initial velocity (in meters/sec): ")
  h0 = input("Enter the initial height (in meters): ")
  dt = input("Enter the time interval between position calculations: ")
```

- Calculating the initial position for the cannonball is also easy. It's at distance `0` and height `h0`!

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

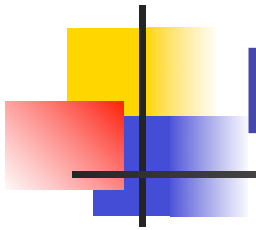
Designing the Program



- If we know the magnitude of the velocity and the angle θ, we can calculate

```
yvel = v*sin(theta)
```
and

```
xvel = v*cos(theta)
```



Designing the Program

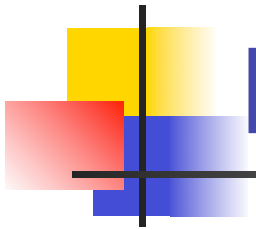
- Our input angle is in degrees, and the Python math library uses radians, so

```
theta = (angle * pi)/180.0
xvel = v0 * cos(theta)
yvel = v0 * sin(theta)
```

- In the main loop, we want to keep updating the position of the ball until it reaches the ground:

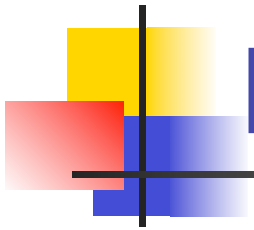
```
while ypos >= 0:
```

- We used >= so the loop will start if the ball starts out on the ground.



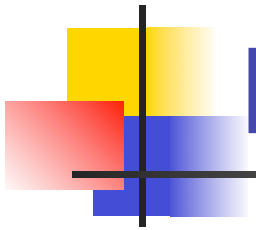
Designing the Program

- Each time through the loop we want to update the state of the cannonball to move it `dt` seconds farther.
- Since we assume there is no wind resistance, `xvel` remains constant.
- Say a ball is traveling at 30 m/s and is 50 m from the firing point. In one second it will be 50 + 30 m away. If the time increment is 0.1 second it will be 50 + 30*0.1 = 53 m.
- `xpos += xvel * dt`



Designing the Program

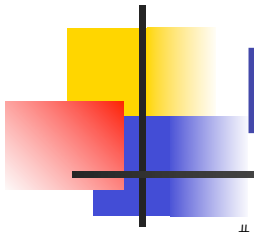
- Working with `yvel` is slightly more complicated since gravity causes the y-velocity to change over time.
- Each second, `yvel` must decrease by 9.8 m/s, the acceleration due to gravity.
- In 0.1 seconds the velocity will be 0.1(9.8) = .98 m/s.
- `yvel1 = yvel - 9.8 * dt`



Designing the Programs

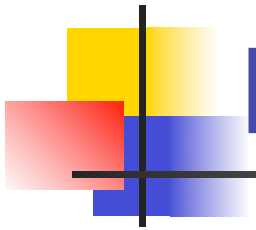
- To calculate how far the cannonball travels over the interval, we need to calculate its average vertical velocity over the interval.
- Since the acceleration due to gravity is constant, the distance traveled is simply the average of the starting and ending velocities times the length of the interval:

```
ypos += (yvel + yvel1) / 2.0 * dt
```



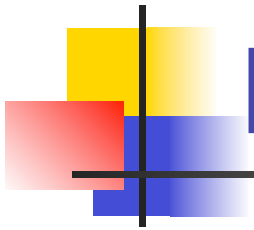
Designing Programs

```
# cball1.py
#   Simulation of the flight of a cannon ball (or other projectile)
#   This version is not modularized.

from math import pi, sin, cos
from visual import *

def main():
    angle = input("Enter the launch angle (in degrees): ")
    v0 = input("Enter the initial velocity (in m/s): ")
    h0 = input("Enter the initial height (in m): ")
    dt = input("Enter the time interval between position calculations: ")
    xpos = 0
    ypos = h0
    theta = (angle * pi)/180.0
    xvel = v0 * cos(theta)
    yvel = v0 * sin(theta)
    # visualize the initial state
    floor = box (pos=(0,0,0), length=50, height=0.5, width=4, color=color.blue)
    ball = sphere(pos=(xpos,ypos,0))
    while ypos >= 0:
        rate(100)
        xpos += xvel * dt
        yvel1 = yvel - 9.8 * dt
        ypos += (yvel + yvel1) / 2.0 * dt
        yvel = yvel1
        # visualize the ball's new position
        ball.pos.x = xpos
        ball.pos.y = ypos

    print "\nDistance traveled: %0.1f m." % (xpos)

main()
```



Modularizing the Program

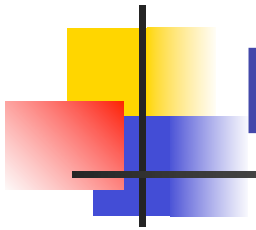
- During program development, we employed step-wise refinement (and top-down design), but did not divide the program into functions.
- While this program is fairly short, it is complex due to the number of variables.



Modularizing the Program

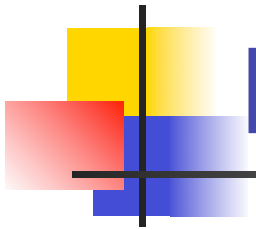
```
def main():
  angle, v0, h0, dt = getInputs()
  xpos, ypos = 0, h0
  xvel, yvel = getXYComponents(v0, angle)
  # visualize the initial state
  floor = box(pos=(0,0,0),length=50,height=0.5,width=4,color=color.blue)
  ball = sphere(pos=(xpos,ypos,0))
  while ypos >= 0:
    xpos, ypos, yvel = updateCannonBall(dt, xpos, ypos, xvel, yvel)
    # visualize the ball's new position
    ball.pos.x = xpos
    ball.pos.y = ypos

  print "\nDistance traveled: %0.1f meters." % (xpos)
```

- It should be obvious what each of these helper functions does based on its name and the original program code.



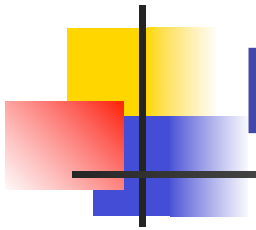
Modularizing the Program

- 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!



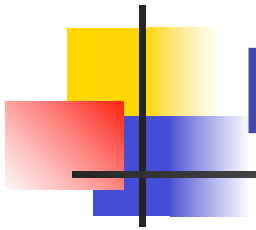
Modularizing the Program

- All four variables, plus `dt`, 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.

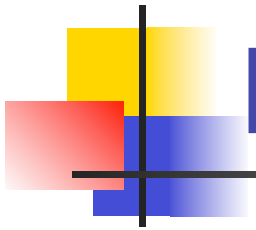


Modularizing the Program

- Using our object-based approach:

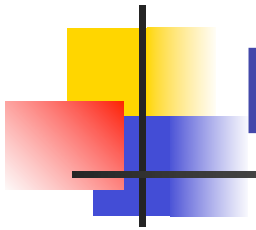
```
def main():
  angle, v0, h0, dt = getInputs()
  # visualize the initial state
  floor=box(pos=(0,0,0),length=50,height=0.5,width=4,color=color.blue)
  cball = Projectile(angle, v0, h0)
  while cball.getY() >= 0:
    rate(100)
    cball.update(dt)
  print "\nDistance traveled: %0.1f m." % (cball.getX())

main()
```

- To make this work we need a `Projectile` class that implements the methods `update`, `getX`, and `getY`.



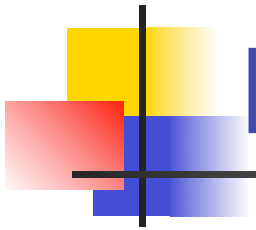
Example: Multi-Sided Dice

- 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 different numbers of sides.
- Let's design a generic class `MSDie` to model multi-sided dice.



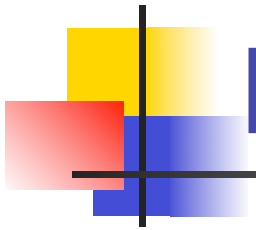
Example: Multi-Sided Dice

- 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.



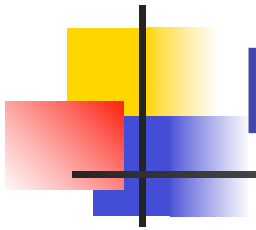
Example: Multi-Sided Dice

- 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.



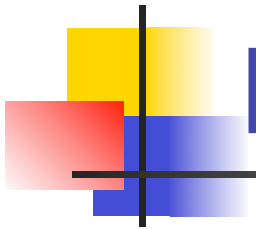
Example: Multi-Sided Dice

```
>>> die1 = MSDie(6)
>>> die1.getValue()
1
>>> die1.roll()
>>> die1.getValue()
5
>>> die2 = MSDie(13)
>>> die2.getValue()
1
>>> die2.roll()
>>> die2.getValue()
9
>>> die2.setValue(8)
>>> die2.getValue()
8
```



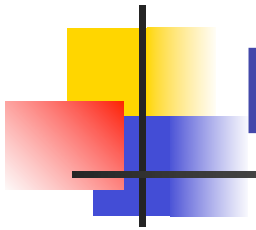
Example: Multi-Sided Dice

- 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.



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
```

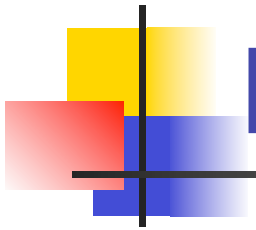


Example: Multi-Sided Dice

- Class definitions have the form

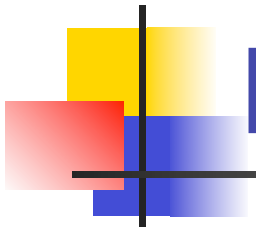
```
class <class-name>:
    <method-definitions>
```

- 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.



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:
  - `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`.

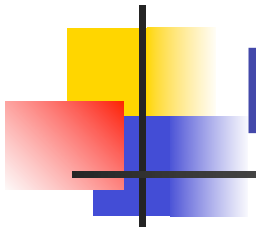


Example: Multi-Sided Dice

- The parameters of the method get assigned the arguments to the call. In the case of a method call, the first parameter refers to the object:

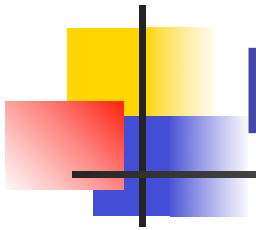
```
self = die1
value = 8
```

- The body of the method is executed.



Example: Multi-Sided Dice

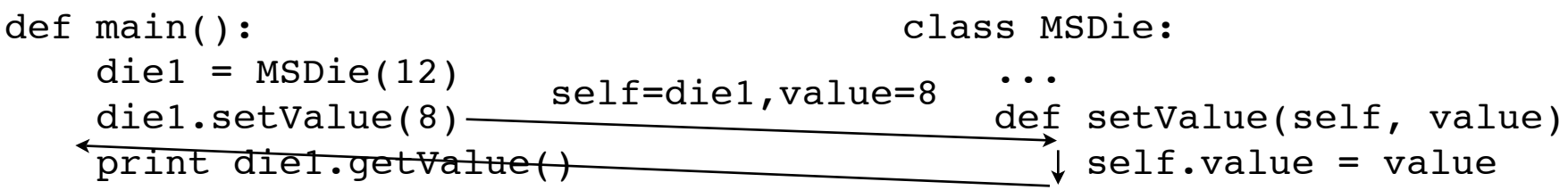
- Control returns to the point just after where the method was called. In this case, immediately following `die1.setValue(8)`.
- This method is called with one argument, but the method definition itself includes the `self` parameter as well as the `value` parameter.

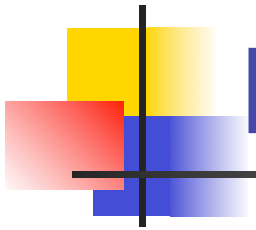


Example: Multi-Sided Dice

- The `self` parameter is a bookkeeping detail. We can refer to the first parameter as the self parameter and other parameters as normal parameters. So, we could say `setValue` uses one normal parameter.

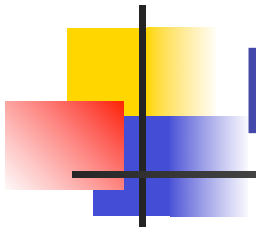
Example: Multi-Sided Dice





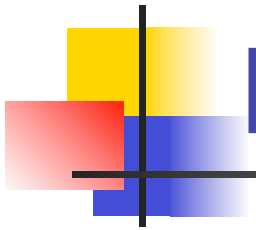
Example: Multi-Sided Dice

- 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`.



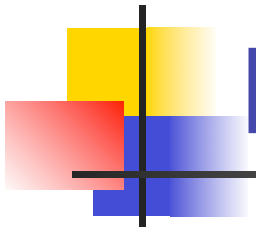
Example: Multi-Sided Dice

- 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.



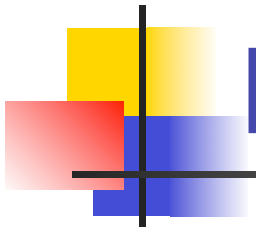
Example: Multi-Sided Dice

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

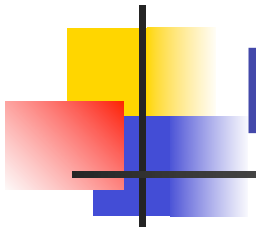
- Instance variables can remember the state of a particular object, and this information can be passed around the program as part of the object.
- This is different than local function variables, whose values disappear when the function terminates.



Example: The Projectile Class

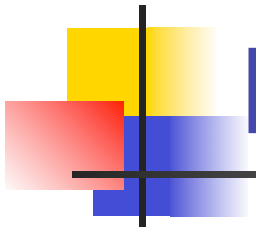
- This class will need a constructor to initialize instance variables, an `update` method to change the state of the projectile, and `getX` and `getY` methods that can report the current position.
- In the main program, a cannonball can be created from the initial angle, velocity, and height:

```
cball = Projectile(angle,v0,h0)
```



Example: The Projectile Class

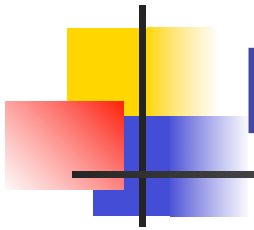
- The `Projectile` class must have an `__init__` method that will use these values to initialize the instance variables of `cball`.
- These values will be calculated using the same formulæ as before.



Example: The Projectile Class

```
class Projectile:
    def __init__(self, angle, velocity, height):
        self.xpos = 0
        self.ypos = height
        theta = pi * angle / 180.0
        self.xvel = velocity * cos(theta)
        self.yvel = velocity * sin(theta)
        # visualize the ball
        self.ball = sphere(pos=(self.xpos,self.ypos,0))
```

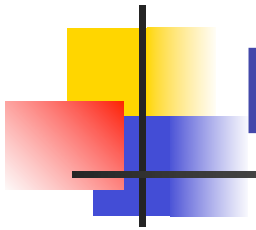
- We've created five instance variables (`self.*`). Since the value of `theta` is not needed later, it is a normal function variable.



Example: The Projectile Class

- The methods to access the X and Y position are straightforward.

```
def getY(self):
    return self.ypos

def getX(self):
    return self.xpos
```

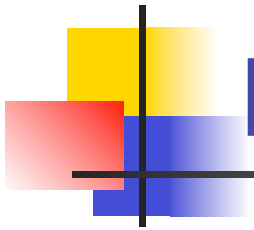


Example: The Projectile Class

- The last method is `update`, where we'll take the time interval and calculate the update X and Y values.

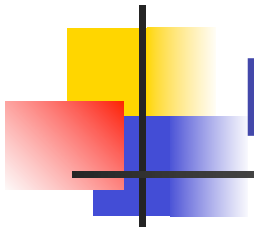
```
def update(self, dt):
    self.xpos += self.xvel * dt
    yvel1 = self.yvel - 9.8 * dt
    self.ypos += (self.yvel + yvel1) / 2.0 * dt
    self.yvel = yvel1
    # visualize the ball's new position
    self.ball.pos.x = self.xpos
    self.ball.pos.y = self.ypos
```

- `yvel1` is a temporary variable.



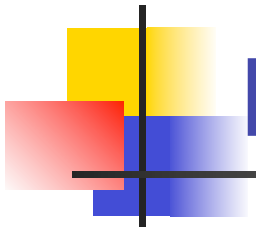
Encapsulating Useful Abstractions

- Defining new classes 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.



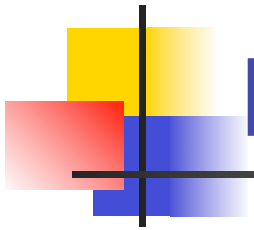
Encapsulating Useful Abstractions

- 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.



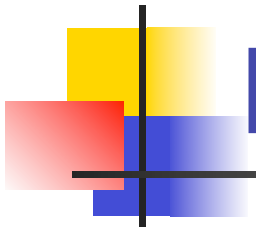
Encapsulating Useful Abstractions

- 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.



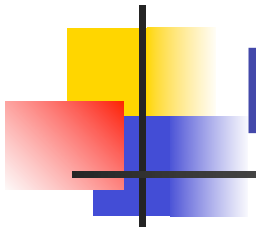
Encapsulating Useful Abstractions

- 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.



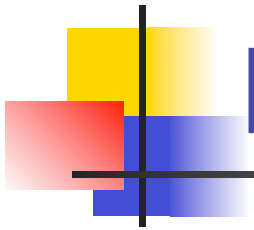
Putting Classes in Modules

- Sometimes we may program a class that is 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

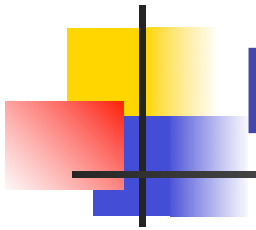
- You are already familiar with "#" to indicate comments explaining what's going on in a Python file.
- Python also has a special kind of commenting convention called the docstring. You can insert a plain string literal as the first line of a module, class, or function to document that component.



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`:

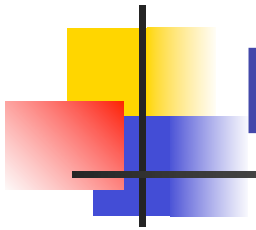
```
>>> import random
>>> print random.random.__doc__
random() -> x in the interval [0, 1).
```



Module Documentation

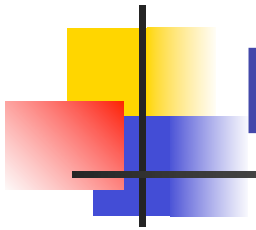
- Docstrings are also used by the Python online help system and by a utility called PyDoc that automatically builds documentation for Python modules. You could get the same information like this:

```
>>> import random
>>> help(random.random)
Help on built-in function random:

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



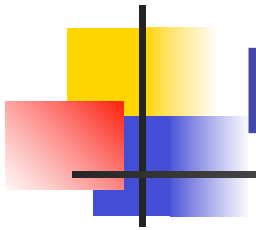
Module Documentation

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



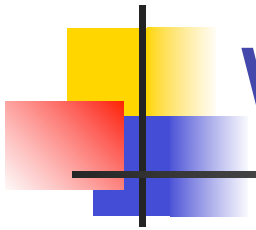
Module Documentation

```
# 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)
        self.ball = sphere(pos=(self.xpos, self.ypos, 0))
```



Module Documentation

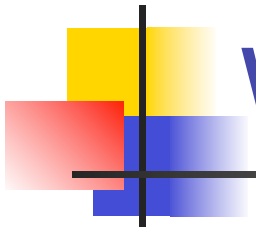
```
def update(self, dt):
    """Update the state of this projectile to move it dt
    seconds farther into its flight"""
    self.xpos += self.xvel * dt
    yvel1 = self.yvel - 9.8 * dt
    self.ypos += (self.yvel + yvel1) / 2.0 * dt
    self.yvel = yvel1
    # visualize the ball's new position
    self.ball.pos.x = self.xpos
    self.ball.pos.y = self.ypos

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
```



Working with Multiple Modules

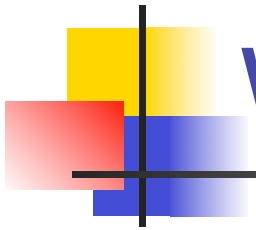
- Import from projectile module to solve original problem!

```
# cball4.py
#   Simulation of the flight of a cannon ball (or other projectile)
#   This version uses a separate projectile module file
from projectile import Projectile
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, dt = getInputs()
    floor = box (pos=(0,0,0), length=50, height=0.5, width=4,
color=color.blue)
    cball = Projectile(angle, vel, h0)
    while cball.getY() >= 0:
        rate(100)
        cball.update(dt)
    print "\nDistance traveled: %0.1f m." % (cball.getX())
```



Working with Multiple Modules

- If you are testing a multi-module Python program, you need to be aware that reloading a module may not behave as you expect.
- When Python first imports a given module, it creates a module object that contains all the things defined in the module (a namespace). If a module imports successfully (no syntax errors), subsequent imports do not reload the module. Even if the source code for the module has been changed, re-importing it into an interactive session will not load the updated version.



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.