The History of Objects

- Objects weren't always supported by programming languages
- Idea first originated at MIT in the 1960s and was officially incorporated in a few languages in the same decade
- OOP (Object Oriented Programming) has now become a core feature of nearly all languages
Object Oriented Programming (OOP)

- A certain style of computer programming
- Centered around data structures called “objects”
- Objects are a standard way to organize data
- Many pros and cons, but almost every language and decent sized project uses it
Using objects

- In Python everything is an object
- Object methods - string, list

```python
str1 = 'objects rule the world'
words = str1.split()
words.append('indeed')
print(' '.join(words))
```

objects rule the world indeed
Defining a Class

class Car():
    wheels = 4

print Car.wheels
myCar = Car() # instantiation
print myCar.wheels
print Car.wheels = 5 # change the class variable
print Car.wheels
print myCar.wheels
class Car():
    
    wheels = 4

    def __init__(self, color):
        self.color = color

    #print Car.color <-- AttributeError: class Car has no attribute 'color'

myCar = Car("red")
print myCar.color # red
class Car():
    wheels = 4
    def __init__(self, color):
        self.color = color
    def fade(self):
        self.color = self.color + "ish"

myCar = Car("red")
print myCar.color #red
myCar.fade()
print myCar.color #redish
class Car():
    wheels = 4
    def __init__(self, color):
        self.color = color
    def fade(self):
        self.color = self.color + "ish"

@staticmethod
def isOld(miles):
    if miles < 50000:
        return False
    return True

print Car.isOld(30000) # False
class Car():
    wheels = 4
    def __init__(self, color, horsepower):
        self.color = color
        self.engine = self.Engine(horsepower)

class Engine():
    def __init__(self, horsepower):
        self.horsepower = horsepower
    def getWatts(self):
        return self.horsepower * 745.7

myCar = Car('red', 400)
print myCar.engine.getWatts() #298280.0
import MODULENAME

def func1():
    BODY1
...
def funcn(a):
    BODYN

class Class1(object):
    CLASSBODY1
...
class ClassN(object):
    CLASSBODYN

# start of the program
MAINBODY

import modules like math

Function definitions

Class definitions

your "main" program
Graphics Objects

• Use graphics.py module

• Graphics objects available:
  – Point
  – Line
  – Circle
  – Oval
  – Rectangle
  – Polygon
  – Text
Creating an object

\[
p = \text{Point}(50, 20)
\]
\[
circle = \text{Circle}(p, 30)
\]

Class name constructs a point

Parameters \((x,y)\) coordinates

Point object

Circle object

Class name constructs a circle

Parameters center point \(p\) and radius 30

Objects can be passed as parameters too
Accessing Attributes and Methods

• Using dot (.)

```python
p = Point(50, 20)
print p.x, p.y
print p.getX(), p.getY()
```

50 20
50 20

attributes or instance variables

methods to get the values of the entries
Objects are mutable

1. \( p = \text{Point}(50, 20) \)
2. \( p.x = p.x - 20 \)
3. \( p2 = p \)
4. \( p2.x = p2.x + 10 \)
5. \( \text{print } p \text{.getX()}, p \text{.getY()} \)
Objects are mutable

1. \( p = \text{Point}(50, 20) \)
2. \( p.x = p.x - 20 \)
3. \( p2 = p \)
4. \( p2.x = p2.x + 10 \)
5. \( \text{print } p\text{.getX()}, p\text{.getY()} \)
Objects are mutable

1. \( p = \text{Point}(50, 20) \)
2. \( p.x = p.x - 20 \)
3. \( p2 = p \)
4. \( p2.x = p2.x + 10 \)
5. \( \text{print } p\text{.getX()}, p\text{.getY()} \)

\( p2 \) is an alias of \( p \), i.e. it refers to the same point object.
Objects are mutable

1. \( p = \text{Point}(50, 20) \)
2. \( p.x = p.x - 20 \)
3. \( p2 = p \)
4. \( p2.x = p2.x + 10 \)
5. \( \text{print } p\text{.getX()}, p\text{.getY()} \)

\( p2 \) is an alias of \( p \), i.e. it refers to the same point object.
Objects are mutable

1. `p = Point(50, 20)`
2. `p.x = p.x - 20`
3. `p2 = p`
4. `p2.x = p2.x + 10`
5. `print p.getX(), p.getY()`

`p2` is an alias of `p`, i.e. it refers to the same point object.
Scoping in functions

- Basic types - create a copy of the variable inside the function

```python
def move_by_10(x, y):
    x = x + 10
    y = y + 10

x = 10
y = 10
move_by_10(x, y)
print x, y
```

What does this print? 10 10
Scoping in functions

- Objects - create an alias of the variable inside the function

```python
def move_by_20(p):
    p.x = p.x + 20
    p.y = p.y + 20

p1 = Point(10, 10)
move_by_20(p1)
print p1.getX(), p1.getY()
```

What does this print?

30 30
Simple Graphics Program

from graphics import *

win = GraphWin('My Circle', 100, 100)
c = Circle(Point(50,50), 10)
c.setFill('red')
c.draw(win)

win.mainloop()
from graphics import *

win = GraphWin('My Circle', 150, 150)
c = Circle(Point(50, 50), 10)
c.setFill('red')
c.draw(win)

win.mainloop()
Simple Graphics Program

```python
from graphics import *

win = GraphWin('My Circle', 150, 150)
c = Circle(Point(50,50), 10)
c.setFill('red')
c.draw(win)
win.mainloop()
```
Simple Graphics Program

from graphics import *

win = GraphWin('My Circle', 150, 150)
c = Circle(Point(50,50), 10)
c.setFill('red')
c.draw(win)

win.mainloop()

every graphics program must end with this line;
it allows the window to process mouse clicks and keyboard input
User-defined types

• What if we want to create our own class?
• E.g. let's create a class that draws a car wheel. For simplicity, the wheel will look like this:
Wheel class

• Attributes
  • tire_circle
  • wheel_circle

• Methods
  • draw
  • move
  • get_size
  • get_center
  • set_color
Wheel Class Definition

class Wheel(object):
    def __init__(self, center, wheel_radius, tire_radius):
        self.tire_circle = Circle(center, tire_radius)
        self.wheel_circle = Circle(center, wheel_radius)

Special method (constructor): it is called when the object is constructed and sets the initial state of the object

defines the objects attributes
Wheel Class Definition

class Wheel(object):
    def __init__(self, center, wheel_radius, tire_radius):
        self.tire_circle = Circle(center, tire_radius)
        self.wheel_circle = Circle(center, wheel_radius)

• What is this self parameter?
• self is an alias to the object instance
• Must use it to access any of the object's attributes or methods
• it must always be the first parameter in a method signature
Wheel Class Definition

```python
class Wheel(object):

    def __init__(self, center, wheel_radius, tire_radius):
        self.tire_circle = Circle(center, tire_radius)
        self.wheel_circle = Circle(center, wheel_radius)
```

Attributes are defined inside the `__init__` method using the `self` parameter.
Attributes vs Local Variables

• Attribute
  – Defined in the `__init__` method
  – Belongs to a specific object
  – Exists as long as the containing object exists

• Local variable
  – Declared within a method or a function
  – Exists only during the execution of its containing method or function
class Wheel(object):

    def __init__(self, center, wheel_radius, tire_radius):
        self.tire_circle = Circle(center, tire_radius)
        self.wheel_circle = Circle(center, wheel_radius)

    def draw(self, win):
        self.tire_circle.draw(win)
        self.wheel_circle.draw(win)

    def move(self, dx, dy):
        self.tire_circle.move(dx, dy)
        self.wheel_circle.move(dx, dy)
class Wheel(object):
    ''' This class defines a wheel template with two circles.
    Attributes: tire_circle, wheel_circle
    '''

    def __init__(self, center, wheel_radius, tire_radius):
        self.tire_circle = Circle(center, tire_radius)
        self.wheel_circle = Circle(center, wheel_radius)

    def draw(self, win):
        self.tire_circle.draw(win)
        self.wheel_circle.draw(win)

    def move(self, dx, dy):
        self.tire_circle.move(dx, dy)
        self.wheel_circle.move(dx, dy)

    def set_color(self, wheel_color, tire_color):
        self.tire_circle.setFill(tire_color)
        self.wheel_circle.setFill(wheel_color)
Wheel Class Definition

```python
.......

def undraw(self):
    self.tire_circle.undraw()
    self.wheel_circle.undraw()

def get_size(self):
    return self.tire_circle.getRadius()

def get_center(self):
    return tire_circle.getCenter()
```
Using our Wheel class

```
win = GraphWin('Wheel', 320, 240)
w = Wheel(Point(100, 100), 50, 70)
w.draw(win)
w.set_color('gray', 'black')
w.undraw()
win.mainloop()
```
Using our Wheel class

```python
win = GraphWin('Wheel', 320, 240)
w = Wheel(Point(100, 100), 50, 70)
w.draw(win)

w.set_color('gray', 'black')
w.undraw()
win.mainloop()
```

What happened to the mysterious self parameter?

```python
def draw(self, win):
    self.tire_circle.draw(win)
    self.wheel_circle.draw(win)
```
Using our Wheel class

```python
win = GraphWin('Wheel', 320, 240)
w = Wheel(Point(100, 100), 50, 70)
w.draw(win)
w.set_color('gray', 'black')
w.undraw()
win.mainloop()
```
Using our Wheel class

```python
win = GraphWin('Wheel', 320, 240)
w = Wheel(Point(100, 100), 50, 70)
w.draw(win)
w.set_color('gray', 'black')
w.undraw()
win.mainloop()
```