Object-­‐Oriented Programming in Python

Classes and Objects

Object-­‐Oriented Programming (OOP): A programming paradigm that involves designing programs around concepts represented as "objects"

• Python supports OOP through the provision of classes.

• Terminology

• Class: A collec on of func ons and a ributes,  a ached to a specific name, which represents an abstract concept.  • A ribute: A named piece of data (i.e. variable  associated with a class.  • Object: A single concrete instance generated from a  class

Instances of Classes

Classes can be viewed as factories or templates for genera ng new object instances. Each object instance takes on the proper es of the class from which it was created.

Instances of Classes

Crea ng Classes

Defining a class in Python is done using the class keyword, followed by an indented block with the class contents.

**General Format**

Class data

class <Classname>:

attributes

data1 = value1

...

dataM = valueM

def <function1>(self,arg1,...,argK):

<block>

...

def <functionN>(self,arg1,...,argK):

Class member functions

<block>

•

Defining Func ons in Classes

• A class defini on block can include mul ple func ons.

• These represent the func onality or behaviors that are associated with the class.

>>> class Maths: ... def subtract(self,i,j): ... return i-j ... ... def add(self,x,y): ... return x+y

Argument (self) refers to the object itself

Calling Func ons in Classes

• Using Class Func ons from Outside a Class Func ons are referenced by using the dot syntax:

tName>.<funcName>()

**<objectName>.<methodName>()** >>> m = Maths() >>> m.subtract(10,5) 5>>> m.add(6,7) 13

No need to specify value for self, Python does this automatically

Calling Func ons in Classes

• Using Class Func ons from Inside a Class When referring to func ons from within a class, we must always prefix the func (e.g. self.subtract()) **s from Inside a Class**

on name with self

tions the f

>>> class Maths: ... def subtract(self,i,j): ... return i-j ... ... def testsub(self): ... ... print print self.subtract(8,4)

self.s

Tell Tell Python Python to use to use function

function associated associated with with this this object

object

A ributes *Class* defined class *Class* defined class *attribute*

at *attribute* top at top of

of

>>> class Person: ... company = "ucd" ... ... def \_\_init\_\_(self): ... self.age = 23

*Instance* defined function.

*Instance* defined function.

inside *attribute*

inside *attribute* a class

a class

The The self self prefix prefix is

is always always required.

required.

>>> p1 = Person() >>> >>> p1 p2 = Person()

= Person() >>> >>> p2 p1.company = Person()

= "ibm" >>> >>> >>> 'ibm'

p1.company print print p2.company

p2.company

= "ibm"

'ibm'

>>> class Person: ... company = "ucd" ... ... def \_\_init\_\_(self): ... self.age = 23

>>> >>> >>> >>> >>> 23 >>> >>> >>> 23 p1 p2 p1.age print p1 p2 p1.age print = = Person() Person() = = p2.age = Person() Person()

p2.age

35 = 35

Change affects instance Change affects instance only only to (p2) to (p2)

instance instance the the associated

associated

attribute attribute age

age

Change to class attribute company affects Change all to instances class attribute (p1 and company

p2) affects all instances (p1 and p2)

Constructor

• When an instance of a class is created, the class constructor func on is automa cally called.

• The constructor is always named \_\_init\_\_()

• If • no It contains code for ini specific ini constructor al state (e.g. se is present, alizing a new instance of the class to a

an ng instance a empty object ribute values).

is created.

>>> class Person: ... def \_\_init\_\_( self, s ): ... self.name = s ... ... def hello( self ): ... print "Hello", self.name

Constructor function taking initial value for instance attribute name

>>> t = Person("John") >>> t.hello() Hello John

Calls \_\_init\_\_() on Person

Inheritance

Class inheritance is designed to model rela onships of the type **"**x **is a** y**"** (e.g. "a triangle is a shape")

Inheritance

The func ons and a ributes of a superclass are inherited by a subclass.

An inherited class can override, modify or augment the func ons and a ributes of its parent class.

Crea ng Subclasses

Simple superclass

>>> class Shape: ... def \_\_init\_\_( self ): ... self.color = (0,0,0)

Simple subclass

>>> class Rectangle(Shape): inheriting from Shape

... def \_\_init\_\_( self, w, h ): ... Shape.\_\_init\_\_( self ) ... self.width = w ... self.height = h ... ... def area( self ): ... return self.width\*self.height

Need to call constructor function in superclass

>>> r1 = Rectangle( 10, 5 ) >>> print r1.width 10>>> print r1.height 5>>> print r1.area() 50>>> print r1.color (0, 0, 0) Construct

object instance

Inherited attribute

Overriding

When inheri ng from a class, we can alter the behavior of the original superclass by "overriding" func ons (i.e. declaring func ons in the subclass with the same name).

Func ons in a subclass take precedence over func ons in a superclass.

Overriding Note: Functions in a subclass take precedence over functions in a superclass.

class Counter:

def \_\_init\_\_(self):

self.value = 0

def increment(self):

self.value += 1

class CustomCounter(Counter):

def \_\_init\_\_(self,size):

Counter.\_\_init\_\_(self) self.stepsize = size

Overriding

def increment(self): self.value += self.stepsize >>> cc = CustomCounter(4) >>> cc.increment() >>> cc.print\_current() Current value is 4

Calls increment() on CustomCounter not Counter

Composi on Classes can be built from other smaller classes,allowing us to model rela onships of the type **"**x **has a** y**”** (e.g. a (e.g. department has students). a department has students).

class Department:

def \_\_init\_\_( self ):

self.students = []

def enroll( self, student ):

self.students.append(student)

class Student:

def \_\_init\_\_( self,last,first ):

self.lastname = last self.firstname = first

>>> compsci = Department() >>> compsci.enroll( Student( "Smith", "John" ) ) >>> compsci.enroll( Student( "Murphy", "Alice" ) )

Create Student instances and add to Department instance

>>> for s in compsci.students: ... print "%s %s" % (s.firstname,s.lastname) ... John Smith Alice Murphy

Polymorphism

Two objects of different classes but suppor ng the same set of func ons or a ributes can be treated iden cally. The implementa ons may differ internally, but the outward "appearance" is the same.

Polymorphism • The implementations may differ internally, but the outward

"appearance" is the same.

Two different classes that contain the function area()

class Rectangle(Shape):

def \_\_init\_\_(self, w, h):

Shape.\_\_init\_\_(self)

self.width = w self.height = h

def area(self):

return self.width\*self.height

Instances of the two classes can be treated identically...

class Circle(Shape):

def \_\_init\_\_(self, rad): Shape.\_\_init\_\_(self)

self.radius = rad

def area(self):

return math.pi\*(self.radius\*\*2)

>>> l = [] >>> l.append( Rectangle(4,5) ) >>> l.append( Circle(3) ) >>> for someshape in l: ... print someshape.area() ... Result of area() in Rectangle 20Result of area() in Circle

28.2743338823