CLASSES AND PYTHON

Sarah Khoja

WHAT ARE OBJECTS?

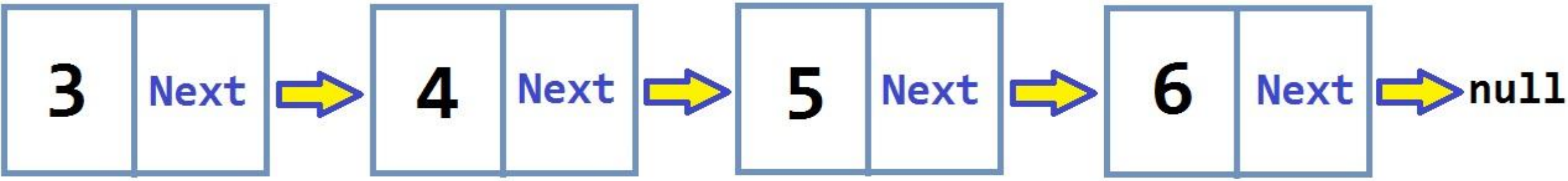
- You have data types- they are the templates
  - You can imagine it is a blueprint/mold of something
    - Integer: 1234
    - String: “Hello World”
    - Double: 123.4567
    - List: [2,3,5,7,11,13,17,19]
  - Each of the data types on the left, are blueprints
  - Each of the examples on the right are OBJECTS, instances of those data types
  - You have the mold, and when you fill it, you’ve created the object of that mold

OBJECT ORIENTED PROGRAMMING

- Python is OOP, so everything in python is an object
  - You can create new objects of the same type
    - Many int's, double's float's
  - You can manipulate objects
    - Assign values
  - Destroy objects
    - You can explicitly deleted (using del)
    - Or leave it to the garbage collection system that Python makes available
      - Python cleans up after memory spaces that are inaccessible, not going to be used again.

WHAT ARE OBJECTS MADE UP OF?

- Objects are a form of data abstraction that abstracts:
  - Internal representation:
    - Through data attributes
    - Specific variables attached to this object
    - Ex- human: data type
      - Objects of type human have the attributes of arms, legs, eyes, nose…
  - Interface:
    - Way of interacting with the object
    - Having methods– functions/procedures
    - You have an understanding of what the behavior is, but the implementation is hidden
      - Example- the data type human has the method of walking. You don't need to know how to walk, just use the walk function.


3
Next
4
Next
5
Next
6
Next
null

- Linked list: made up of nodes
- Each node is an object
- You have a class, or a template, which defines what a node should look like
  - Has an attribute of:
    - An integer
    - A pointer
  - Methods/fns
    - Add integer
    - Get node
    - Remove node
  - Every time you create an instance of that class, it is an object you are creating.

ADVANTAGES

- Follows the programming ideals- generalize, efficiency, organization
  - Bundle data into packages
  - Divide and conquer
    - When building, easier to test your different functionalities
    - Reduces complexity
  - Reuse code with classes

MAKING YOUR OWN CLASS

- You can define your own blueprints, or class.
  - You can then use this class by instantiating it– creating objects of it.
- How to create class:
  - Define the class name
  - Define the class attributes
  - Define the class methods
- How to use class:
  - Create an instance of fit
  - Then do operations on the instance
    - GetValue
    - SetValue

Defining Classes:

```
#A        #B              #C
class Coordinate(object):
   #define attributes here

#A-class definition
#B- name/type
#C- class parent
```

Defining Classes

1. This is similar to using
    a. def
2. Object--
    a. means that Coordinate is a Python object that inherits all its attributes
        i. Coordinate is a subclass of object
        ii. object is superclass of Coordinate

What are attributes?

- Data and functions that “belong” to the class
- Data attributes
  - example- a coordinate has two point (x, y)
- Methods: procedural attributes
  - functions that only work with this class.
  - how to interact with the object.
    - the distance between two objects of the coordinate class.
    - but you cannot apply this to the distance between two String objects.
-

Continue Creating Classes

- First- define how to create an instance of the object.
  - __init__
  - A special method used to initialize data attributes.
  - for example- the Coordinate class has two data attributes- x and y.
- In order to initialize these values, we can do it in the __init__ method.

```
class Coordinate(object):
  def __init__(self, x,y):
    self.x=x
    self.y=y
```

Continuing Creating Classes

- __init__ - special method to create an instance
  - of the class!
- self.x/self.y
  - two data attributes that are given to every Coordinate object
- self
  - the reference to an instance of the class
- x,y
  - the data that initializes a Coordinate object.

Create instance of a class

- c is an object of the class Coordinate
  - pass 3 and 4 into the __init__
  - the data attributes of an instance are called instance variables.
- Use the dot to access the data attributes x and y
- You do not need to provide arguments for 'self', python does this automatically.
  - passes in the object itself.

```
c=Coordinate(3,4)
origin=Coordinate(0,0)
print(c.x)
print(c.y)
print(origin.x)
```

Methods

1. These are the procedural attributes
   a. Functions that only work with the given class
2. First thing passed to the __init__ is the object itself.
   a. self
3. Access the attributes by using the dot operator
   a. data attributes
   b. procedural attributes

Defining Methods

1. self- refers to any instance, other is another instance.
2. use dot notation to access the data in each.

```
class Coordinate(object):
    def __init__(self,x,y):
        self.x=x
        self.y=y
    def distance(self, other):
        x_diff_sq=(self.x-other.x)**2
        y_diff_sq=(self.y-other.y)**2
        return(x_diff_sq+y_diff_sq)**0.5
```

Using Methods:

```
#c and zero are objects of the class Coordinate
c= Coordinate(3,4)
zero=Coordinate(0,0)

#Conventional way of using methods
print(c.distance(zero))

#translates to:
print(Coordinate.distance(c,zero))
```

Printing Objects

```
c= Coordinate(3,4)
print(c)
```

```
<Coordinate object at 0x7f11366fcdd0>
>
```

Printing

1. the information is unhelpful
    a. just provides the address where the object is stored.
2. We can define our own print function, which is more helpful in printing
    a. by redefining a special method-
        i. __str__
        ii. when you call print on the class object, python uses the __str__ function.
        iii. we can redefine the __str__ function to be helpful in printing
            1. example: when we call print(c) we want the following:
            2. <3,4>
        iv. Known as overriding the method __str__

```
class Coordinate(object):
  def __init__(self,x,y):
    self.x=x
    self.y=y
  def distance(self, other):
    x_diff_sq=(self.x-other.x)**2
    y_diff_sq=(self.y-other.y)**2
    return(x_diff_sq+y_diff_sq)**0.5


  #redefining the __str__ method
  #__str__ is called during print

  def __str__(self):
    return "<"+str(self.x)+","+str(self.y)+">"
    #this special method MUST return a string!!!

c= Coordinate(3,4)
print(c)
```

```
Python 2.7.10 (default, Jul 14 2015, 19:46:27)
[GCC 4.8.2] on linux
>
<3,4>
>
```

More overloading

1. When you want to use special operators like:
   a. +, -, ==, <, >, len(), print etc
   b. on classes, you have to overload them.
2. Define th eoverloaded methods in the class with double underscores before and after them.
   a. __add__(self,other) -> self + other
   b. __sub__(self, other) → self - other
   c. __eq__(self, other) -> self == other
   d. __lt__(self,other) -> len(self)
   e. __str__(self_ -> print(self)

Why classes and OOP?

1. help us bundle together objects that share:
   a. common attributes
   b. procedures that operate on those attributes
2. Uses Abstraction-
   a. to make distinction between how to implement an object versus how to use the object
3. Builds layers of object abstractions that inherit behaviors from other classes of objects.
4. Allows us to create our own objects on top of Python's basic classes.

Implementing vs Using Classes

1. Implementing is when you:
    a. define the class
    b. define data attributes
        i. what makes up the object?
    c. define methods
        i. how to use the object.
2. Using the Class:
    a. create instances of the object type
    b. do operations with the objects

Implementing vs Using

1. Implementing
    a. like a blueprint, our template/mold
2. Instance of the class:
    a. actually using that template to create an object of the class.
    b. why do we need this:
        i. example: humans all have hair, eyes, etc etc
        ii. but hair color, eye color, etc has different definitions
            1. blue, green etc color eyes.
            2. black, blonde, brown etc. hair color

```
class Animal(object):
  def __init__(self, age):
    self.age=age
    self.name=None

  #setters and getters for age
  def get_age(self):
    return self.age
  def set_age(self, newAge):
    self.age=newAge

  #setters and getters for name:
  def get_name(self):
    return self.name
  def set_name(self, newName=""):
    self.name=newName

  #for priting purposes:
  def __str__(self):
    return "animal: \nName:"+str(self.name)+"\nAge: "+
      str(self.age)



a=Animal(10)
print(a)
```

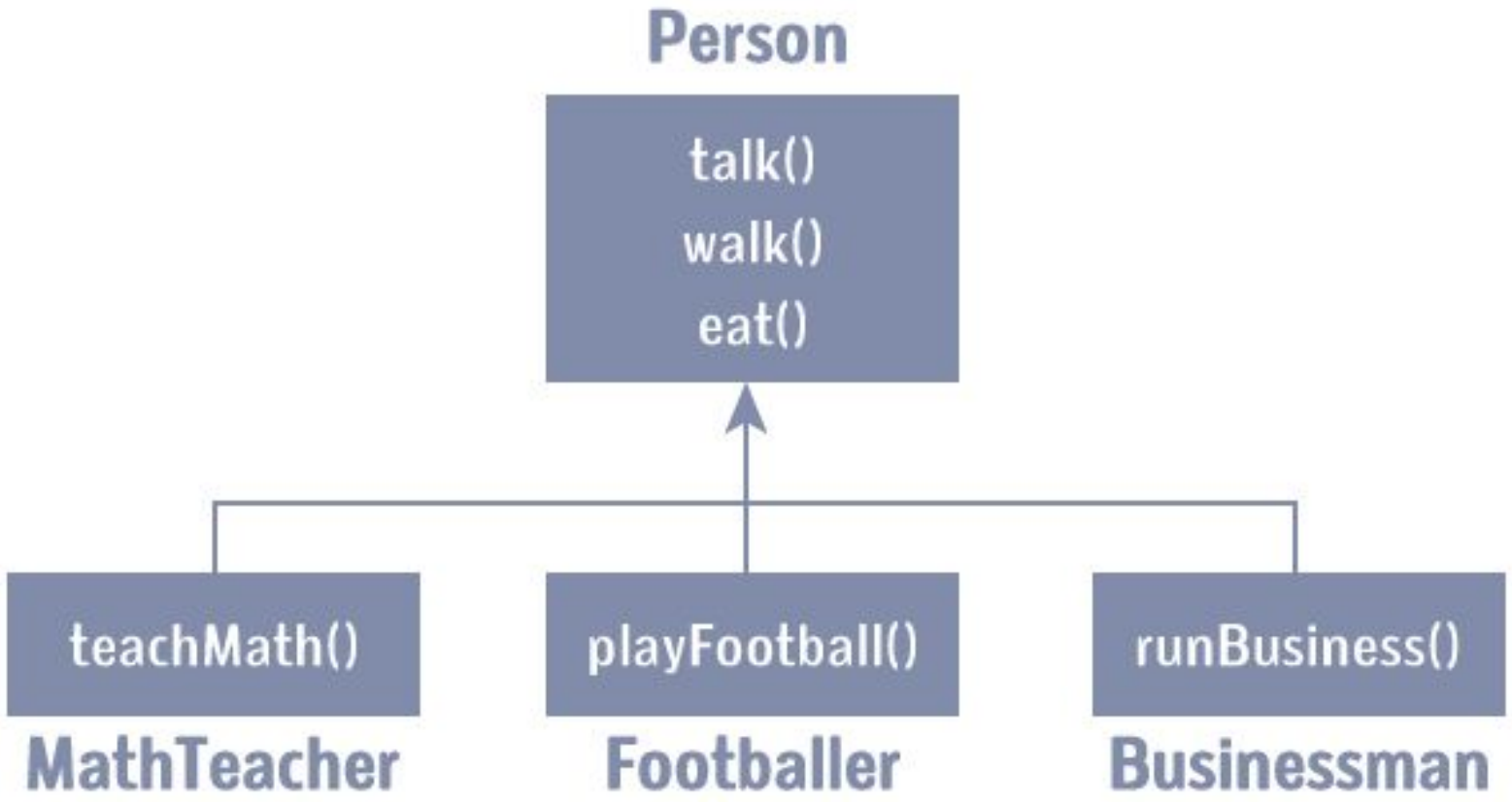
```
Python 2.7.10 (default,
[GCC 4.8.2] on linux
>
animal:
Name:None
Age: 20
>
```

```
a=Animal(10)
#not recommended
print(a.age)
#using the getter/setters
print(a.get_age())

a.set_age(20)
print(a.get_age())

b=Animal(5)
print(b.age)
```

Hiding Information

```
class Animal(object):
  def __init__(self,age):
    self.years=age
  def get_age(self):
    return self.years
  def set_age(self,age ):
    self.years=age
```

Hierarchies and Inheritance



Hierarchies and Inheritance

1. The superclass- the parent class, Person in this case.
2. The subclass- the child class, Math Teacher, Footballer, Businessman
    a. inherits everything that is defined in the parent class.
        i. all the data, and methods (behaviors)
    b. adds more information (more data attributes)
    c. adds more behavior (more methods)
    d. has the ability to override the behaviors defined in the superclass.

```
class Animal(object):
  def __init__(self,age):
    self.age=age
    self.name=None
  def get_age(self):
    return self.age
  def get_name(self):
    return self.name
  def set_age(self,newage):
    self.age=newage
  def set_name(self, newname=""):
    self.name=newname
  def __str__(self):
    return "animal:"+str(self.name)+ ":"
      + str(self.age)
```

```
#class Cat inherits from class Animal
#it inherits: init, age, name, get_age,
  get_name, set_age, set_name, str
class Cat(Animal):
  #defines its own new method
  def speak(self):
    print("meow" )
  #redefines what the __str__ method
    does
  def __str__(self):
    return "cat:"+str(self.name)+ ":"+
      str(self.age)
```

```
class Person(Animal):
  #constructor for the Person class
  def __init__(self, name,age):
    #call the Animal's constructor
    Animal.__init__(self, age)
    #set values for Person
    self.set_name(name)
    self.friends=[]
  def get_friends(self):
    return self.friends
  def add_friend(self,fname):
    if fname not in self.friends:
      self.friends.append(fname)
  def speak(self):
    print("hello")
  def age_diff(self, other):
    diff=self.age-other.age
    print(abs(diff), "year difference")
  def __str__(self):
    return "person:"+str(self.name)+ ":"
      +str(self.age)
```

```
class Dog(Animal):
  #class level variable
  #shared between all instances of the class!
  count=1
  def __init__(self, age, parent1=None, parent2=None):
    Animal.__init__(self, age)
    self.parent1=parent1
    self.parent2=parent2
    #assign the value in count to the dogId
    self.dogId= Dog.count
    #increment the count, so that the next dog will have a diff
      id
    Dog.count+=1
```

Define methods for Dog class

1. define the setters/getters for the data attributes in the dog class.
2.

```
def __add__(self, other):
  #recall- Dog.__init__(self, age, parent1, parent2)
  #age=0, parent1= self, parent2= other
  return Dog(0, self, other)
  #calling d3=d1+d2
    # now d3 is a new dog instance that is the child of d1
      and d2
```

Determine if two dogs are equal

booleans

```
def __eq__(self, other):
    parents_same = self.parent1.rid == other.parent1.rid \
                   and self.parent2.rid == other.parent2.rid
    parents_opposite = self.parent2.rid == other.parent1.rid \
                       and self.parent1.rid == other.parent2.rid
    return parents_same or parents_opposite
```