DSP505: Programming Lab for Data Science and Artificial Intelligence

TPL616: Advanced Programming for DSAI

(Object Oriented Programming in Python)

Vishwesh Jatala

Assistant Professor

Department of CSE

Indian Institute of Technology Bhilai

vishwesh@iitbhilai.ac.in

Acknowledgement

This lecture notes are prepared using:

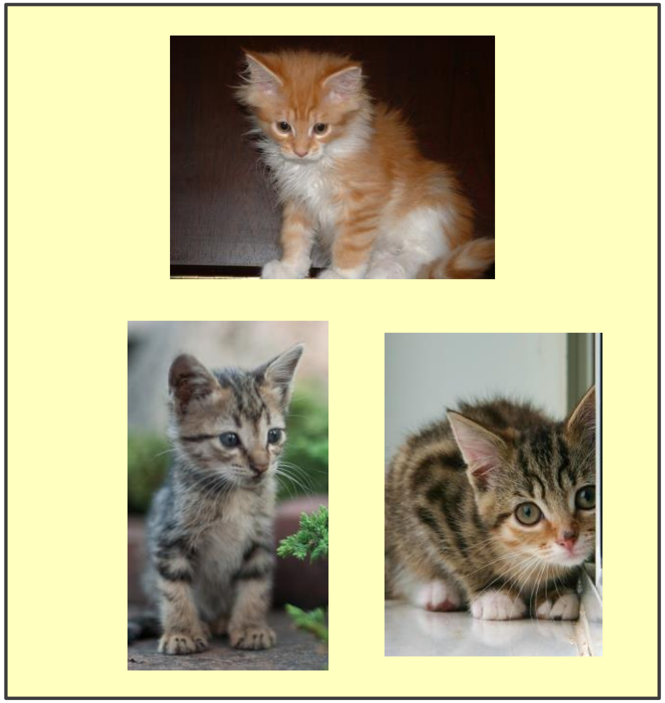
- MIT Opencourseware: https://ocw.mit.edu/https://ocw.mit.edu/
- Dr.Greene UCD School of Computer Science and Informatics, Dublin
- IIT Delhi
- Miscellaneous Internet Sources.

Introduction

Object Oriented Programming is a way of computer programming using the idea of “objects” to represents data and methods.

Introduction

- mimic real life
- group different objects part of the same type



Introduction

- Python supports many different kinds of data

```
1234  3.14159  "Hello"  [1, 5, 7, 11, 13]
{"CA": "California", "MA": "Massachusetts"}
```

- each is an **object**, and every object has:
  - a **type**
  - an internal **data representation** (primitive or composite)
  - a set of procedures for **interaction** with the object
- an object is an **instance** of a type
  - `1234` is an instance of an `int`
  - `"hello"` is an instance of a string

Introduction

- **EVERYTHING IN PYTHON IS AN OBJECT** (and has a type)
- can **create new objects** of some type
- can **manipulate objects**
- can **destroy objects**
  - explicitly using `del` or just “forget” about them
  - python system will reclaim destroyed or inaccessible objects – called “garbage collection”

What are objects?

- objects are **a data abstraction** that captures…

(1) an **internal representation**

- through data attributes

(2) an **interface** for interacting with object

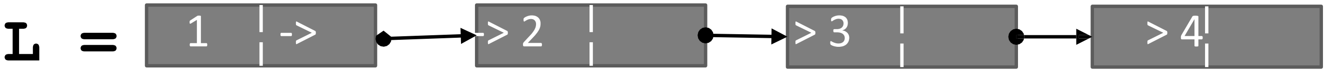
- through methods

  (aka procedures/functions)

- defines behaviors but hides implementation

Example: Lists

- how are lists **represented internally**? linked list of cells



follow pointer to the next index

- how to **manipulate** lists?
  - `del(L[i])`
  - `L.append(),L.extend(),L.count(),L.index(), L.insert(),L.pop(),L.remove(),L.reverse(), L.sort()`
- internal representation should be private
- correct behavior may be compromised if you manipulate internal representation directly

Advantages of OOP

- **bundle data into packages** together with procedures that work on them through well-defined interfaces
- Python supports the OOP through classes
- Classes make it easy to **reuse** code
  - many Python modules define new classes
  - inheritance allows subclasses to redefine or extend a selected subset of a superclass' behavior

Classes and Objects

- make a distinction between **creating a class** and **using an instance** of the class
- **creating** the class involves
  - defining the class name
  - defining class attributes
  - *for example, someone wrote code to implement a list class*
- **using** the class involves
  - creating new **instances** of objects
  - doing operations on the instances
  - *for example,* `L=[1,2]` *and* `len(L)`

Creating Classes

- use the `class` keyword to define a new type

name/type

```
class Coordinate:
    #define attributes here
```

class definition

- similar to `def`, indent code to indicate which statements are part of the **class definition**
- Create new a class and name it is as Coordinate.

Class Members

- data and procedures that "**belong**" to the class
- **data attributes**
  - think of data as other objects that make up the class
  - *for example, a coordinate is made up of two numbers*
- **methods** (procedural attributes)
  - think of methods as functions that only work with this class
  - how to interact with the object
  - *for example you can define a distance between two coordinate objects but there is no meaning to a distance between two list objects*

Attributes

- Class attributes
- Belongs to the class itself
- Shared by all instances of the classes
- Access it using ClassName.attribute or object.attribute

```
class Coordinate:
    count = 0
```

Attributes

- first have to define **how to create an instance** of object
- use a **special method called `__init__`** to initialize some data attributes

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

what data initializes a `Coordinate` object

parameter to refer to an instance of the class

special method to create an instance is double __ underscore

two data attributes for every `Coordinate` object

Creating an Instance of a Class

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

create a new object of type `Coordinate` and pass in 3 and 4 to the `__init__`

use the dot to access an attribute of instance `c`

- data attributes of an instance are called **instance variables**
- don’t provide argument for `self`, Python does this automatically

Attributes

Class attribute defined at top of class

```
>>> class Person:
...  company = "ucd"
...
...      def __init__(self):
...          self.age = 23
```

Instance attribute defined inside a class function.
The `self` prefix is always required.

```
>>> p1 = Person()
>>> p2 = Person()
>>> p1.age = 35
>>> print p2.age
23
```

Change to instance attribute age affects only the associated instance (p2)

```
>>> p1 = Person()
>>> p2 = Person()
>>> p1.company = "ibm"
>>> print p2.company
'ibm'
```

Change to class attribute company affects all instances (p1 and p2)

Constructor

- When an instance of a class is created, the class constructor function is automatically called.
- The constructor is always named __init__()
- It contains code for initializing a new instance of the class to a specific initial state (e.g. setting instance attribute values).

```
>>> 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`

What is a Method?

- procedural attribute, like a **function that works only with this class**
- Python always passes the object as the first argument
  - convention is to use **`self`** as the name of the first argument of all methods
- the "**.**" **operator** is used to access any attribute
  - a data attribute of an object
  - a method of an object

Define a Method for `Coordinate` Class

```
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
```

use it to refer to any instance

another parameter to method

dot notation to access data

- other than `self` and dot notation, methods behave

just like functions (take params, do operations, return)

How to Use a Method

```
def distance(self, other):
    # code here
```

method def

Using the class:

```
c = Coordinate(3,4)

zero = Coordinate(0,0)


print(c.distance(zero)
)
```

object to call method on

name of method

parameters not including `self` (`self` is implied to be `c`)

Representation of an object

```
>>> c = Coordinate(3,4)
>>> print(c)
<__main__.Coordinate object at 0x7fa918510488>
```

- **uninformative** print representation by default
- define a **`__str__` method** for a class
- Python calls the `__str__` method when used with `print` on your class object
- you choose what it does! Say that when we print a `Coordinate` object, want to show

```
>>> print(c)
<3,4>
```

Defining Your Own Print Method

```
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
    def __str__(s elf):
        return "<"+str(self.x)+","+str(self.y)+">"
```

name of special method

must return a string

Object Types

- can ask for the type of an object instance

```
>>> c = Coordinate(3,4)
>>> print(c)
<3,4>
>>> print(type(c))
<class __main__.Coordinate>
```

return of the __str__ method

the type of object c is a class Coordinate

Special Operators

- +, -, ==, <, >, len(), print, and many others

https://docs.python.org/3/reference/datamodel.html#basic-customization

- like `print`, can override these to work with your class
- define them with double underscores before/after

```
__add__(self, other)            self + other
__sub__(self, other)            self - other
__eq__(self, other)             self == other
__lt__(self, other)             self < other
__len__(self)                   len(self)
__str__(self)                   print self
```

... and others

Another Example

```
class Animal:
    def __init__(self, age):
        self.age = age

        self.name = None


myanimal = Animal(3)
```

Getter And Setter Methods

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

getter

```
    def get_age(self):
        self.age
    def get_name(self):
        return self.name
```

setter

```
    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)
```

- **getters and setters** should be used outside of class

to access data attributes

An Instance And Dot Notation (Recap)

- instantiation creates an **instance of an object**

```
a = Animal(3)
```

- **dot notation** used to access attributes (data and methods) though it is better to use getters and setters to access data attributes

```
a.age
```

```
a.get_age()
```

- access data attribute
- allowed, but not recommended

- access method
- best to use getters and setters

Information Hiding

- author of class definition may **change data attribute** variable names

replaced age data attribute by years

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

- if you are **accessing data attributes** outside the class and class **definition changes**, may get errors

- outside of class, use getters and setters instead

  use `a.get_age()` NOT `a.age`
  - good style
  - easy to maintain code
  - prevents bugs

Python Not Great At Information Hiding

- allows you to **access data** from outside class definition

```
print(a.age)
```

- allows you to **write to data** from outside class definition

```
a.age = 'infinite'
```

- allows you to **create data attributes** for an instance from outside class definition

```
a.size = "tiny"
```

- it’s **not good style** to do any of these!

Default Arguments

- **default arguments** for formal parameters are used if no actual argument is given

```
def set_name(self, newname=""):
    self.name = newname
```

- default argument used here

```
a = Animal(3)
a.set_name()
print(a.get_name())
```

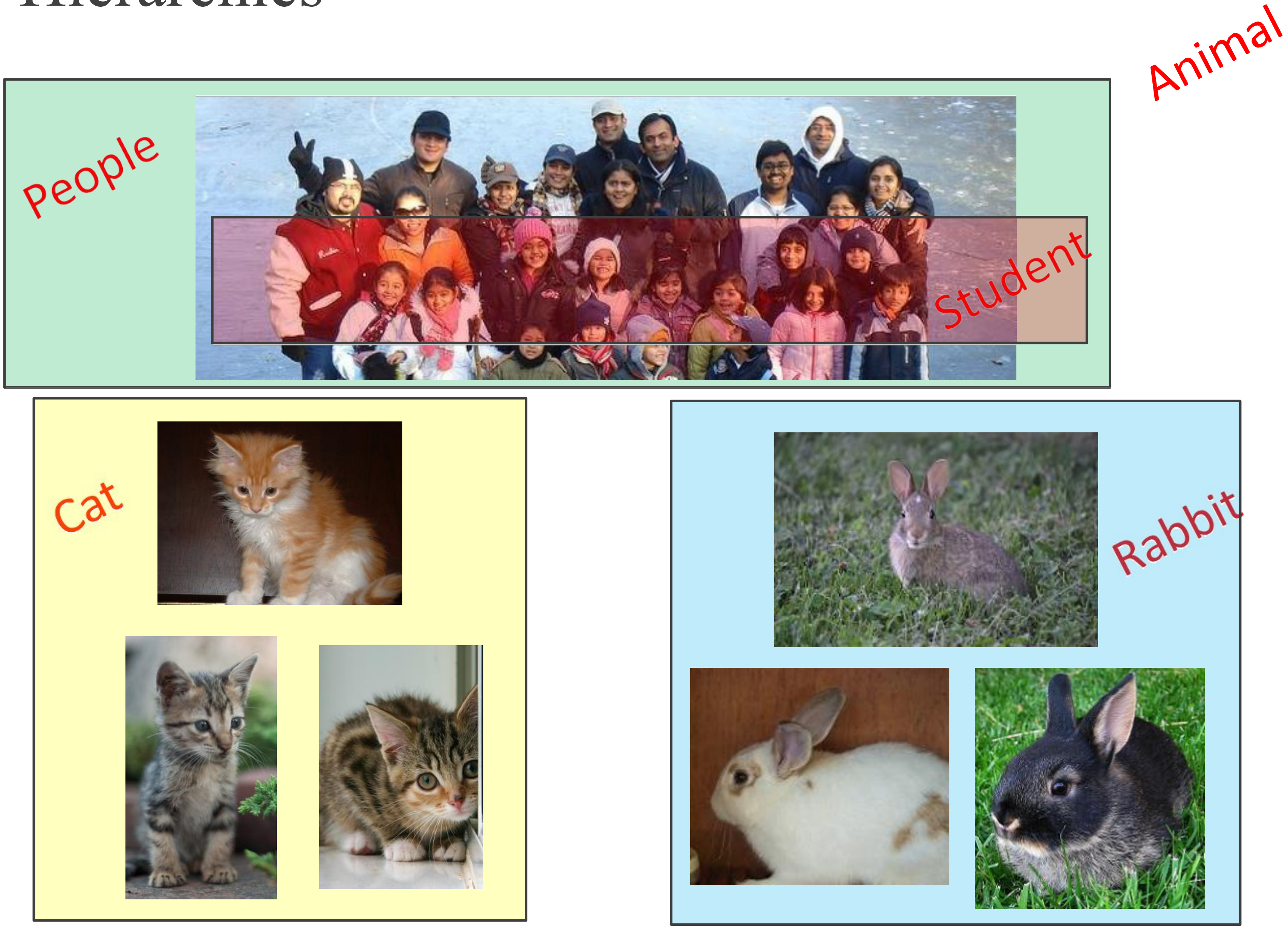
prints ""

- argument passed in is used here

```
a = Animal(3)
a.set_name("fluffy")
print(a.get_name())
```

prints "fluffy"

Hierarchies



Animal

People

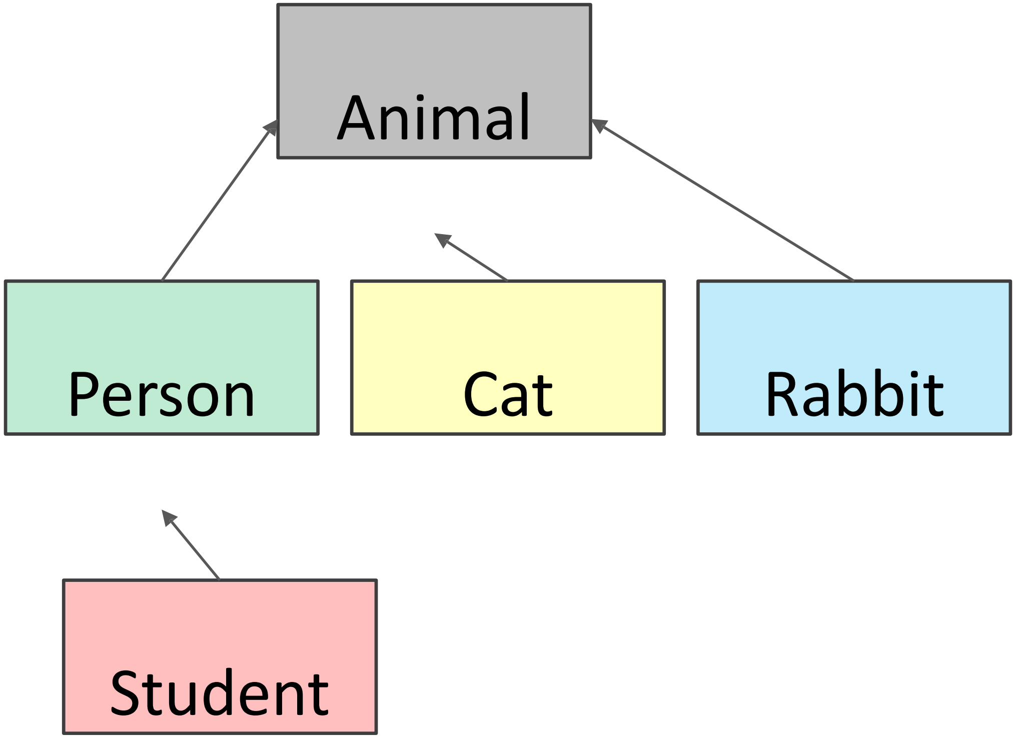
Student

Cat

Rabbit

Hierarchies

- **parent class** (superclass)
- **child class** (subclass)
  - **inherits** all data and behaviors of parent class
  - **add** more **info**
  - **add** more **behavior**
  - **override** behavior



Inheritance: Parent Class

```
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)
```

- everything is an object
- class `object` implements basic operations in Python, like binding variables, etc

Inheritance: Subclass

inherits all attributes of `Animal`:

__init__()

age, name

get_age(), get_name()

set_age(), set_name()

__str__()

```
class Cat(Animal):
    def speak(self):
        print("meow")
    def __str__(self):
        return
        "cat:"+str(self.name)+":"+str(self.age)
```

add new functionality via speak method

overrides __str__

- add new functionality with `speak()`
  - instance of type `Cat` can be called with new methods
  - instance of type `Animal` throws error if called with `Cat`’s new method
- `__init__` is not missing, uses the `Animal` version

Which Method To Use?

- subclass can have **methods with same name** as superclass
- for an instance of a class, look for a method name in **current class definition**
- if not found, look for method name **up the hierarchy** (in parent, then grandparent, and so on)
- use first method up the hierarchy that you found with that method name

```
class Person(Animal):
    def __init__(self, name, age):
        Animal.__init__(self, age)
        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)
```

parent class is `Animal`

call `Animal` constructor

call `Animal`'s method

add a new data attribute

new methods

override `Animal`'s __str__ method

```
import random

class Student(Person):
    def __init__(self, name, age, major=None):
        Person.__init__(self, name, age)
        self.major = major
    def change_major(self, major):
        self.major = major

    def speak(self):
        r = random.random()
        if r < 0.25:
            print("i have homework")
        elif 0.25 <= r < 0.5:
            print("i need sleep")
        elif 0.5 <= r < 0.75:
            print("i should eat")
        else:
            print("i am watching tv")
    def __str__(self):
        return
        "student:"+str(self.name)+":"+str(self.age)+":"+str(self.major)
```

bring in methods from `random` class

inherits `Person` and `Animal` attributes

adds new data

- I looked up how to use the `random` class in the python docs
- `random()` method gives back float in [0, 1)

Object Oriented Programming

- create your own **collections of data**
- **organize** information
- **division** of work
- access information in a **consistent** manner
- add **layers** of complexity
- like functions, classes are a mechanism for **decomposition** and **abstraction** in programming