Elective Course

Course Code: CS4103

Autumn 2025-26

Lecture #13

Artificial Intelligence for Data Science

Week-4:

Basics of Python OOP & Exploring easyAI

Course Instructor:

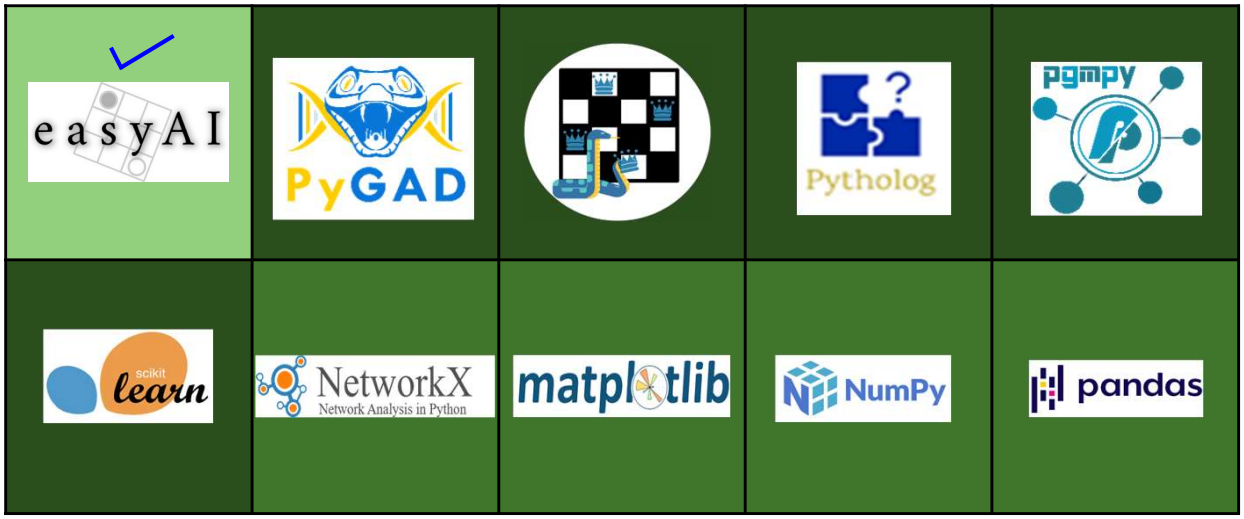
Dr. Monidipa Das

Assistant Professor

Department of Computational and Data Sciences

Indian Institute of Science Education and Research Kolkata, India 741246

A Few Python Libraries for AI

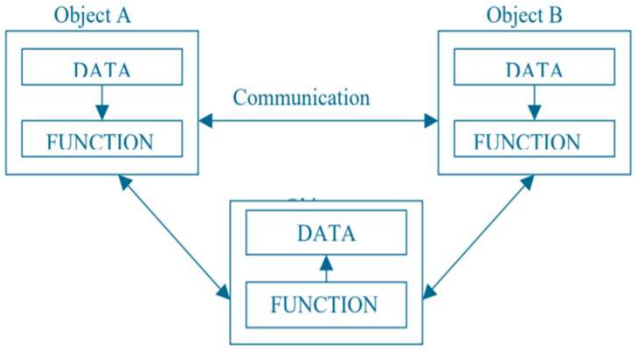


Object Oriented Programming (OOP) in Python

Object Oriented Programming (OOP)

- Promotes code *reusability* and *modular design* with organized methods and attributes.
- *Simplifies* complex programs by grouping related data and behavior.
- Helps developing *maintainable* and *scalable* applications

- **OOP Concepts in Python**
  - Class
  - Objects
  - Encapsulation
  - Inheritance
  - Polymorphism
  - Data Abstraction



Classes and Objects

- **Class:**
  - represents an abstract concept;
  - a collection of functions and attributes, attached to a specific name;
  - User-defined template for creating object
- **Object:**
  - A single concrete instance generated from a class
- **Attribute:**
  - A named piece of data (i.e. variable associated with a class.
- **Functions:**
  - A method that operate on data

Student

name roll_no dob marks	*attributes*
getTotalMarks() calculateGrade()	*methods*

student1
name: Nandini
roll_no: 23MS992
dob: 11-07-2005
marks: [92, 90, 89, 100]

student2
name: Alan
roll_no: 23MS995
dob: 16-12-2004
marks: [90, 95, 90, 96]

Creating Classes

- Defining a class in Python is done by using **`class`** keyword followed by an *indented block* with the class members

Data/
Attributes

```
class <Classname>:
    data1 = value1
    ...
    dataM = valueM

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

Methods/
Functions

Attributes in Class

```
class Student:

    department="CSE" # Class attribute

    def __init__(self, name, roll, dob, marks):
        self.name = name          # Instance attribute
        self.roll_no = roll       # Instance attribute
        self.dob = dob          # Instance attribute
        self.marks = marks        # Instance attribute
```

- `__init__` method is the **constructor** in Python, automatically called when a new object is created. It initializes the attributes of the class.
- `self` parameter is a reference to the current instance of the class. It allows us to access the attributes and methods of the object.
- Accessing attributes: **<objectName>.<attributeName>**

Defining Functions in Classes

- Functions represent the functionality or behaviors that are associated with the class.
- Argument (self) refers to the object itself
- Accessing class function from outside class: **<objectName>.<methodName>()**
- Accessing class function from inside class: **self.<methodName>()**

```
class Student:

    department="CSE"  # Class attribute

    def __init__(self, name, roll, dob, marks):
        self.name = name          # Instance attribute
        self.roll_no = roll       # Instance attribute
        self.dob = dob          # Instance attribute
        self.marks = marks        # Instance attribute

    def getTotalMarks(self):
        return sum(self.marks)

    def calculateGrade(self):
        t=self.getTotalMarks()/4  #sum(self.marks)
        if t>=90: return "A+"
        elif 80<=t<90: return "A"
        elif 70<=t<80: return "B+"
        elif 60<=t<70: return "B"
        elif 50<=t<60: return "C"
        elif 40<=t<50: return "D"
        else: return "F"
```

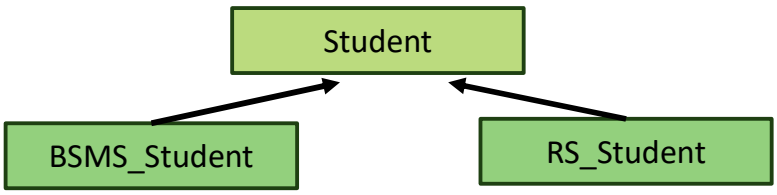
Creating Objects


```
student1 = Student ("Nandini","23MS992", "11-07-2005",[92, 90, 69, 60])
student2 = Student ("Alan", "23MS995", "16-12-2004", [90,95,90,96])

#accessing instance attributes and class attribute
print("Student-1 Details:",student1.name, student1.roll_no,student1.department)
print("Student-2 Details:",student2.name, student2.roll_no,student2.department)

print("Department name change....")
Student.department="CDS" #changing class attribute value

print("Student-1 Details:",student1.name, student1.roll_no,student1.department)
print("Student-2 Details:",student2.name, student2.roll_no,student2.department)

#accessing class function
print("Student-1 Grade:",student1.calculateGrade())
print("Student-2 Grade:",student2.calculateGrade())
```

```
Student-1 Details: Nandini 23MS992 CSE
Student-2 Details: Alan 23MS995 CSE
Department name change....
Student-1 Details: Nandini 23MS992 CDS
Student-2 Details: Alan 23MS995 CDS
Student-1 Grade: B+
Student-2 Grade: A+
```


Inheritance

- Inheritance allows a class (child class) to acquire properties and methods of another class (parent class)
- Class inheritance is designed to model relationships of the form "x is a type of y“



Parent class or Superclass

Child class or Subclass

- The functions and attributes of a superclass are Inherited by a subclass.
- An inherited class can override, modify or augment the functions and attributes of its parent class.
- **Function Overriding:** Altering/Extending method definition (changing behavior). The method in the subclass must have the same name, parameters, and return type as the method in the superclass. Subclass takes precedence over functions in a superclass

Inheritance: Example


```
class Student:

    department="CSE"  # Class attribute

    def __init__(self, name, roll, dob, marks):
        self.name = name
        self.roll_no = roll
        self.dob = dob
        self.marks = marks

    def getTotalMarks(self):
        return sum(self.marks)

    def calculateGrade(self):
        t=self.getTotalMarks()/4
        if t>=90: return "A+"
        elif 80<=t<90: return "A"
        elif 70<=t<80: return "B+"
        elif 60<=t<70: return "B"
        elif 50<=t<60: return "C+"
        elif 40<=t<50: return "D"
        else: return "F"
```

Method Overriding

```
class RS_Student(Student):

    def __init__(self, name, roll, dob, marks,t):
        Student.__init__(self, name, roll, dob, marks)
        self.thesisComp=t

    def calculateGrade(self):
        t=(self.getTotalMarks() + self.thesisComp)/5
        if t>=90: return "A+"
        elif 80<=t<90: return "A"
        elif 70<=t<80: return "B+"
        elif 60<=t<70: return "B"
        elif 50<=t<60: return "C+"
        elif 40<=t<50: return "D"
        else: return "F"
```


Inheritance: Example


```
rs1 = RS_Student ("Audrey","25RS994", "19-09-1999",[81, 78, 80, 82],84)

print("RS Student Details:",rs1.name, rs1.roll_no, rs1.department)
print("RS Student Total Subject Marks:",rs1.getTotalMarks())
print("RS Student Thesis Component Marks:",rs1.thesisComp)
print("RS Student Grade:",rs1.calculateGrade())
```

```
RS Student Details: Audrey 25RS994 CDS
RS Student Total Subject Marks: 321
RS Student Thesis Component Marks: 84
RS Student Grade: A
```

13

easyAI is an artificial intelligence framework for two-players abstract games

Defining Game

14

- To define a new game, make a subclass of the class **`easyAI.TwoPlayerGame`** and define the following methods:
  - `__init__(self, players, ...)`
  - `possible_moves(self)`
  - `make_move(self, move)`
  - `is_over(self)`
  - `show(self) #(optional)`
  - `scoring() #(optional)`
  - `...`

```
from easyAI import TwoPlayerGame

class GameClassName(TwoPlayerGame):
        def __init__(self, players):
         . . .
         def possible_moves(self):
        . . .
         def make_move(self, move):
        . . .
        def is_over(self):
        . . .
```

Source: https://zulko.github.io/easyAI/get_started.html

Defining Game: Example


```
from easyAI import TwoPlayerGame
from easyAI import Human_Player, AI_Player, Negamax

class TicTacToe(TwoPlayerGame):

    def __init__(self, players):
        self.players = players
        self.board = [0 for i in range(9)]
        self.current_player = 1 # player 1 starts.

    def possible_moves(self):
        return [i+1 for i,e in enumerate(self.board) if e==0]

    def make_move(self, move):
        self.board[int(move)-1] = self.current_player

    def is_over(self):
        return (self.possible_moves() == []) or self.lose()
```


Defining Game: Example


```
    def lose(self):
        """ Has the opponent "three in line ?" """
        return any( [all([(self.board[c-1]== self.opponent_index)
                          for c in line])
                          for line in [[1,2,3],[4,5,6],[7,8,9], # horiz.
                                       [1,4,7],[2,5,8],[3,6,9], # vertical
                                       [1,5,9],[3,5,7]]]) # diagonal
    def show(self):
        print ('\n'+'\n'.join([
                        ' '.join([['.','X','O'][self.board[3*j+i]]
                        for i in range(3)])
                 for j in range(3)]) )

    def scoring(self):
        return -100 if self.lose() else 0
```

Starting a Game


```
game = GameClassName(players = [player_1, player_2], *other_required_arguments)
game.play() # start the match !
```

The players can be:

Human_Player() (which will be asked interactively which moves it wants to play) or
AI_Player(algo)

```
from easyAI import Human_Player, AI_Player, Negamax
ai_algo = Negamax(6)
g=TicTacToe([Human_Player(),AI_Player(ai_algo)])
#other option: g=TicTacToe([Human_Player(), Human_Player()])
#other option: g=TicTacToe([AI_Player(ai_algo),AI_Player(ai_algo)])
g.play()
```


Outcome


```
. . .
. . .
. . .

Player 1 what do you play ? 5

Move #1: player 1 plays 5 :

. . .
. X .
. . .

Move #2: player 2 plays 1 :

O . .
. X .
. . .

Player 1 what do you play ? 3

Move #3: player 1 plays 3 :

O . X
. X .
. . .

Move #4: player 2 plays 7 :

O . X
. X .
O . .
```

```
Player 1 what do you play ? 4

Move #5: player 1 plays 4 :

O . X
X X .
O . .

Move #6: player 2 plays 6 :

O . X
X X O
O . .

Player 1 what do you play ? 2

Move #7: player 1 plays 2 :

O X X
X X O
O . .

Move #8: player 2 plays 8 :

O X X
X X O
O O .

Player 1 what do you play ? 9

O X X
X X O
O O X
```

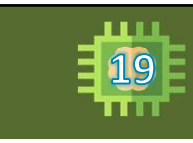
Human_Player()
vs. AI_Player(ai_algo)

```
. . .
. . .
. . .

Move #1: player 1 plays 1 :

X . .
. . .
. . .

Move #2: player 2 plays 5 :

X . .
. O .
. . .

Move #3: player 1 plays 2 :

X X .
. O .
. . .

Move #4: player 2 plays 3 :

X X O
. O .
. . .

Move #5: player 1 plays 7 :

X X O
. O .
X . .
```

```
Move #6: player 2 plays 4 :

X X O
O O .
X . .

Move #7: player 1 plays 6 :

X X O
O O X
X . .

Move #8: player 2 plays 8 :

X X O
O O X
X O .

Move #9: player 1 plays 9 :

X X O
O O X
X O X
```

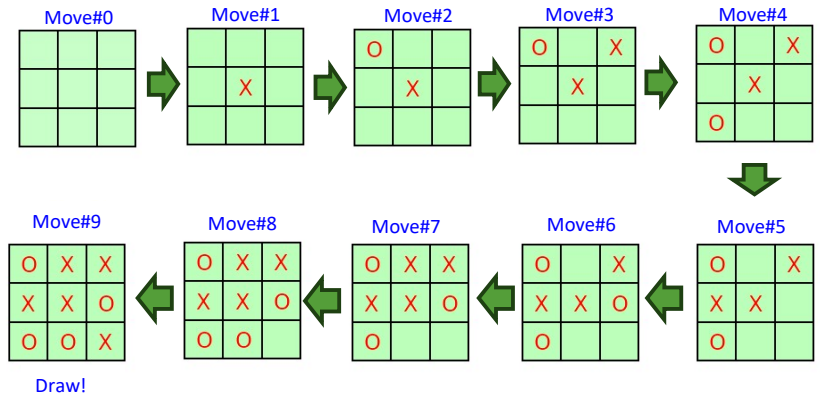
AI_Player(ai_algo)
vs.
AI_Player(ai_algo)

Scope of Enhancement




```
Player 1 what do you play ? 5
Move #1: player 1 plays 5 :
Move #2: player 2 plays 1 :
Player 1 what do you play ? 3
Move #3: player 1 plays 3 :
Move #4: player 2 plays 7 :
Player 1 what do you play ? 4
Move #5: player 1 plays 4 :
Move #6: player 2 plays 6 :
Player 1 what do you play ? 2
Move #7: player 1 plays 2 :
Move #8: player 2 plays 8 :
Player 1 what do you play ? 9
Move #9: player 1 plays 9 :
```

1) Adding visualization:



2) Developing new variants of two-player games

Questions?