Object Oriented Programming (OOPs)

Md Shad Akhtar

Assistant Professor

INDRAPRASTHA INSTITUTE *of*
INFORMATION TECHNOLOGY
DELHI

Data Types

- Scalar: int, float, char, bool
  - Store simple data
- Structured: list, set, tuple, dict
  - Stores slightly more complex data
  - Pre-defined functions to operate on these types

- Real world data are much more complex than what we can store them in scalar or structured types.

- We can create user-defined data types using the scalar and structured types
- Also, we can define custom functions to operate on them

Examples of User Types

- **Stack**: Last in first out - very useful in many applications
  - Operations: push, pop, isempty
- **Queues**: First in first out - most queuing systems use it
  - Operations: Enqueue, Dequeue, isempty
- **Binary search tree** - a root with a left sub tree which has smaller values, and a right subtree with larger values
  - Operations: Add/Remove element, search for an element, join trees, …
- **Graph** is a common way to represent many problems - e.g. cities with roads between them, any network (friendship, computer, …)..
  - ops desired: add a node, delete a node, add a link, delete a link, find path from A to B, find shortest path (if edges have values), check if a node exists, find all nodes connected to a node, …
- **Complex numbers** - we will use this as an initial example
  - Operations: add, subtract, findreal, findimag, …

Examples…

- In some problems, we might want to model a real world data object, for example a car along with its associated details and functions.
- Students in an Institute - we can create a dictionary with all values and then write functions on it; alternatively, we can define a Type student and then define as many students we want
    - Operations: What is the name, SGPA (sem), CGPA, graduated, list of courses, …
- While implementing an e-commerce site, may want to provide abstractions for customer, catalog, shopping_cart, …
- In a computer game, we can have characters, guns of different types, targets of different types, ….

Defining a Class (i.e. a new Type)

- Python allows defining a new type as a class
- To define a type for complex numbers:

```
class Complex:

    <body of the class definition>
```

- Declares that Complex is a type - like list, dict, set, int, ..
- As a type, objects of this type can be created and reference to them assigned to variables
- Generally, class Names start with a capital letter

Objects of a Class

- Class defines a new type (like list, set, etc), with a set of valid operations, and some attributes on which ops work
- We can create objects of this type and assign them to variables - as with language defined types like dict, sets, lists, int, ..
- An object of a class can be created and assigned to a var by

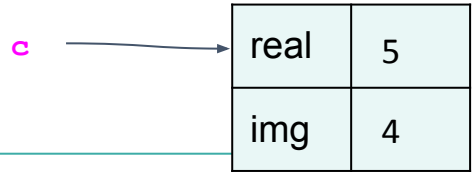
```
c1 = Complex()
```

- An object of type Complex is created and the var c1 points to that object
- Role of variable is same as with all types; the nature of object is now of Complex type

Class Methods and Attributes

- A class also has methods - functions which define the operations on this new type
  - Only these methods can be executed on objects of this class
  - Complex will have methods like: add, subtract, getreal, getimag, …
- A class has attributes - these are variables in its scope, accessed from within the class to implement the methods of the class
  - E.g. Complex has attributes real and imag
  - These attributes define the state of an object of this class
- As Python does not have explicit var declaration, attributes are defined in methods in the class, usually the __init__

Class



- Class is a user-defined data type
  - A collection of a set of attributes (values)

Class 'Complex'

- Attributes
  - Real
  - Imaginary

```
class Complex:
      real = 0
      img = 0
```

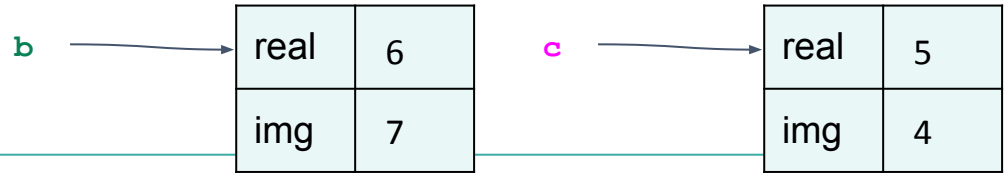
Object

```
c = Complex()
print(c.real) # prints 0
print(c.img) # prints 0
```

```
c.real = 5
c.img = 4
```

- To use/access attributes of the class, we need to a create an **instance of the class** *aka* **object**.
- All attributes (and functions) must be accessed through the object ONLY.

Class



- Class is a user-defined data type
  - A collection of a set of attributes (values)

Class 'Complex'

- Attributes
  - Real
  - Imaginary

Object

Another object

```
class Complex:
      real = 0
      img = 0
```

```
c = Complex()
b = Complex()
```

```
c.real = 5
c.img = 4
b.real = 6
b.img = 7
```

A better way is to define and initialize through the constructor function (a special function that gets automatically called when an object is created) of the class, i.e., `__init__()`

Class

C1 → | real | 5 |
|---|---|
| img | 4 |

- Class is a user-defined data type
  - A collection of a set of attributes (values)

Class 'Complex'

- Attributes
  - Real
  - Imaginary

Reference to the current object

```
class Complex:
    def __init__(self, x, y):
        self.real = x
        self.img = y

c1 = Complex(5, 4)
```

We also need some functions to operate on these attributes.

Class

- Class is a user-defined data type
  - A collection of a set of attributes (values)
  - A collection of functions to operate on attributes

Class 'Complex'

- Attributes
  - Real
  - Imaginary
- Methods
  - printComplex()

```
class Complex:
    def __init__(self, x, y):
        self.real = x
        self.img = y

    def printComplex(self):
        print(f'{self.real} + {self.img}i')
```

```
c1 = Complex(6, 7)
c1.printComplex() # prints 6 + 7i
```

Object – instance of a class

- To use/access attributes of the class or to call these functions, we need to a create an instance of the class.
  - All functions and attributes can be accessed through the object ONLY.

```
c1 = Complex(5, 4)
c2 = Complex(6, 7)
c1.add(c2)
c1.printComplex() # prints 11+11i
```

Internally, a function call is treated like this:

```
c1.add(c2) ⇒ Complex.add(c1, c2)
c1.printComplex() ⇒ Complex.printComplex(c1)
```

```
Complex.printComplex(c1) # prints 11+11i
```

```
class Complex:
    def __init__(self, x, y):
        self.real = x
        self.img = y

    def add(self, c):
        self.real += c.real
        self.img += c.img

    def printComplex(self):
        print(f'{self.real}+{self.img}i')
```

First argument must always be the current object

Object – instance of a class

- To use/access attributes of the class or to call these functions, we need to a create an instance of the class.
  - All functions and attributes can be accessed through the object ONLY.

```
c3 = Complex(9, 10)
c3.printComplex() # prints 9+10i
```

```
class Complex:
      def add(self, c):
            self.real += c.real
            self.img += c.img

      def __init__(shad, self, y):
            shad.real = self
            shad.img = y

      def printComplex(self):
            print(f'{self.real}+{self.img}i')
```

First argument must always be the current object regardless of the name.

Class, Object, Attribute, Methods

- A class is a collection of attributes and methods that operates on attributes.
  - It provides an abstraction on how data should be stored and managed
- Object is an instance of the class
  - We can create multiple objects of a single class
    - Each object creates and maintains its own state.
- Attributes and methods must be accessed through objects only.
- We can define as many methods as required
  - Each method must have the current object (`self`) as the first argument
- `__init__()` is a special function (also called constructor) that gets called at object creation.

Data Encapsulation or Information Hiding

- Class - encapsulates data (attributes) and provides operations
- Ideally, data should be hidden and can be accessed through methods only, e.g., `getAtt()`, `setAtt()`
- Python does not mandate it, but some languages do.

```
class Complex:
     def __init__(self, x, y):
          self.real = x
          self.__img = y

     def printComplex(self):
          print(f'{self.real}+{self.__img}i')
```

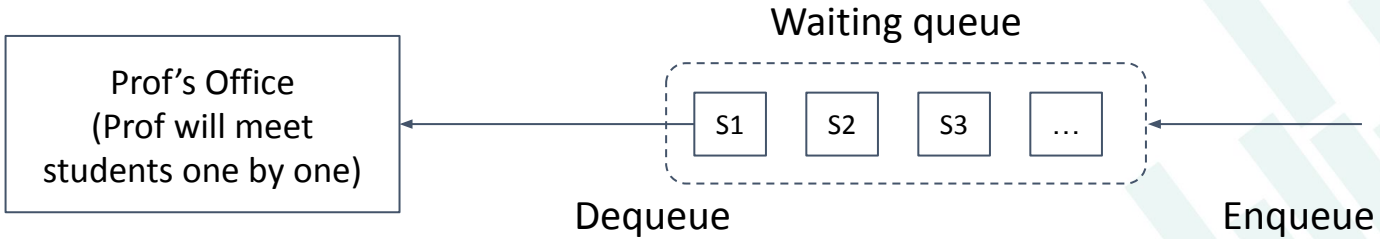
```
c4 = Complex(5, 10)
c4.printComplex() # prints 5+10i
print(c4.real) # prints 5
print(c4.__img) # Error: 'Complex' object
has no attribute '__img'
```

Normal attribute: accessible to the object inside and outside of the class.

Restricted attribute (__var): accessible to the object inside the class; but hidden to the object outside of the class.

A use-case for the queue

- Suppose we want to build a program to manage wait queue for a prof's office hour - in this application
  - A student comes, the assistant adds him/her to the queue
  - When prof is free, the asst removes the first student in the queue, prints info about the student; if queue is empty, then lets the prof know
  - If Prof wants to know many students are still waiting - give the number
  - When the office hour ends, gives a list of students who could not be served
  - …
- We will develop a Queue of persons to manage the wait queue of students for the prof
  - This Queue type can be used in other such applications also



Methods/Operations on Queue

- What operations do we want in this queue in which we will maintain the people (students) waiting
  - add(person): when a new person comes - he/she is added
  - remove(): returns the person who is to be now helped
  - isempty(): is the queue empty
- We also need methods to
  - Give the length of the queue: how many people are in the queue
  - Print the list of people waiting in the queue
- Let us now define a class Queue to implement this type - we will first implement the first few operations

Class Queue – w/o deleting item

```
class Queue:
 def __init__(self):
    self.qdata = []
    self.front = 0
    self.end = 0

 def add(self, obj):
    self.qdata.append(obJ)
    self.end += 1

 def remove(self):
    if self.isempty():
       return None
    else:
       obj = self.qdata[self.front]
       self.front += 1
    return obj

 def isempty(self):
    if self.front == self.end:
       return True
    else:
       return False
```

```
waitroom = Queue()
while True:
    op = input("1: add, 2: remove, -1: exit: ")
    op = int(op)
    if op == 1:
        roll_no = input("Give roll no: ")
        waitroom.add(roll_no)
    elif op == 2:
        obj = waitroom.remove()
        if obj == None:
            print("No student waiting")
        else:
            print(f'Next student: {roll_no}')
    elif op == -1:
        break
    else:
        print("Incorrect command, try again")
```

Class Queue – by deleting item

```
class Queue:
 def __init__(self):
    self.qdata = []

 def add(self, obj):
    self.qdata.append(obJ)

 def remove(self):
    return None if self.isempty() else self.qdata.pop(0)

 def isempty(self):
    return True if len(self.qdata) == 0 else False
```

No need to maintain explicit indexes:

Add at the end, i.e., append

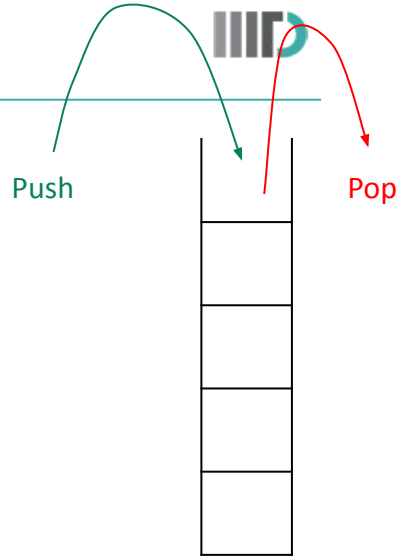
Remove from the beginning, i.e., index = 0

```
waitroom = Queue()
while True:
    op = input("1: add, 2: remove, -1: exit: ")
    op = int(op)
    if op == 1:
        roll_no = input("Give roll no: ")
        waitroom.add(roll_no)
    elif op == 2:
        obj = waitroom.remove()
        if obj == None:
            print("No student waiting")
        else:
            print(f'Next student: {roll_no}')
    elif op == -1:
        break
    else:
        print("Incorrect command, try again")
```

Using the Queue – office hour

- The program / appl is used by the prof's asst - who gives commands
- This app does not need to know internals of how the Queue is implemented
  - Just like you don’t need to know how list, dir … are implemented
- Code for class Queue can be written by one programmer and provided to another, who can use it by importing it as a module
  - We have put the user code in the same file for simplicity. We will see its use as a module.

Another example: Stack



- A collection of items with two operations only
  - push(item) – insert an item onto the stack
  - pop() – delete an item from the stack
- Constraint:
  - Both these operation must be done at the top of the stack.
    - We can't access any location other than the first location

How to implement it or more precisely which datatype to use?

- List
  - But list allows many other functions (e.g., insert at any place, removal from any place, etc.) which would violate the stack operation
- Class with a list attribute
  - **Restrict illegal access** to the list by not defining methods that would violate the stack operation.

```
class Stack:
      item = []
      def __init__(self, elements = []):
            self.item.extend(elements)

      def push(self, element):
            self.item.append(element)

      def pop(self):
            return self.item.pop()

      def __str__(self):
            return ','.join(map(str, self.item))

      def __len__(self):
            return len(self.item)

      def empty(self):
            self.item = []
```

```
s = Stack([1,2,3,4])
s.push(5)

print(s)  # calls special function __str__(). prints '1,2,3,4,5'
print(len(s))  # calls special function __len__(). prints '5'
print(s.pop())  # remove the top element '5' and prints it.
print(s)  # prints 1,2,3,4
s.empty()  # clear the stack
print(s)  # prints empty stack
```

Dunder (**d**ouble **under**score) functions – implementation to work with built-in functions/operators, e.g., print(), len(), +, *, etc.

They gets called automatically when these functions/operators are invoked.

Few dunder methods

Operation	Dunder method
+	object.__add__(self, other)
-	object.__sub__(self, other)
*	object.__mul__(self, other)
==	object.__eq__(self, other)
!=	object.__ne__(self, other)
>=	object.__ge__(self, other)
print()	object.__str__(self)
len()	object.__len__(self)

Operation	Dunder method
+	object.__add__(self, other)

```
class Complex:
  def __init__(self, x, y):
    self.real = x
    self.img = y

  def add(self, c1):
    real = self.real + c1.real
    img = self.img + c1.img
    return Complex(real, img)

def __add__(self, c1):
    real = self.real + c1.real
    img = self.img + c1.img
    return Complex(real, img)

  def __str__(self):
    return f'{self.real} + {self.img}i'
```

```
c1 = Complex(51, 4)
c2 = Complex(6, 7)
c3 = c1.add(c2) # Calls add()
print(c3) # prints 57 + 11i
c4 = c1 + c2 # Calls __add__()
print(c4) # prints 57 + 11i
```

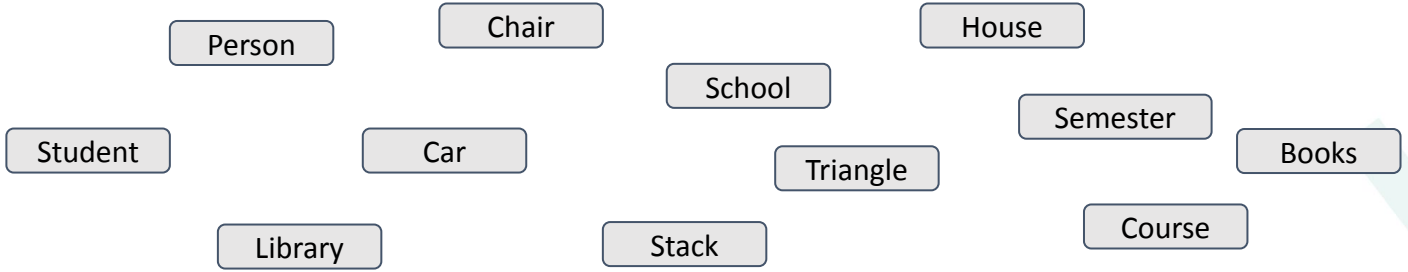
Observe, we did not call functions explicitly:

- `__init__()` : Called on object creation
- `__add__()` : Called on + operator
- `__str__()` : Called through print()

What does a Class represent?

- An entity —abstraction of an entity – through a set of attributes and methods to work with these attributes.

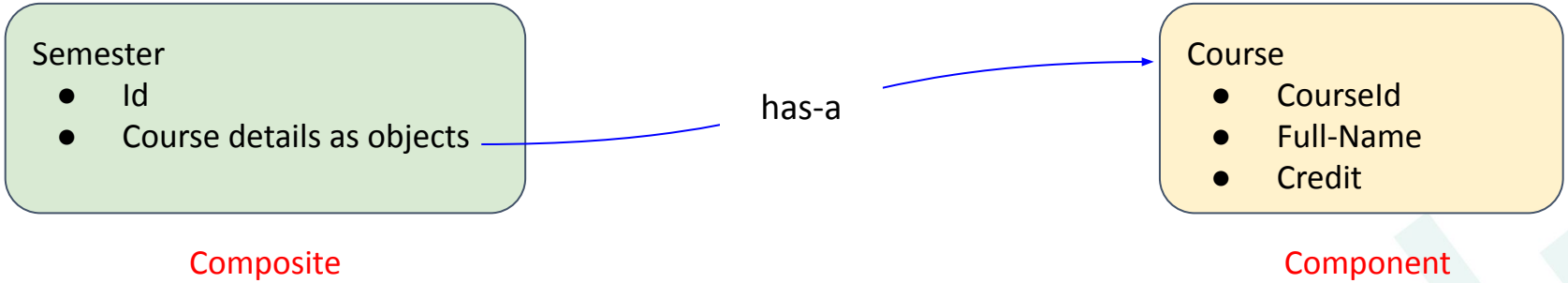
- While solving a (complex) problem, always think of an entity which can be defined by a set of attributes.
  - Keep all those attributes which are necessary for solving the problem.



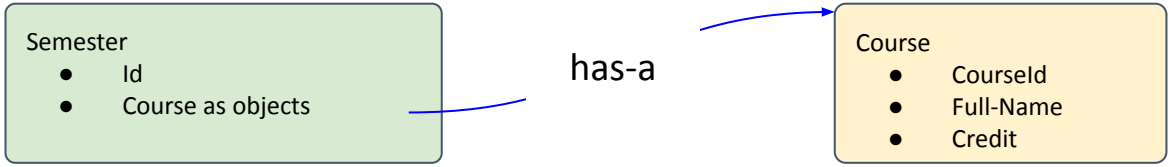
… Literally, anything which has some attributes can be defined as a class

Composition

- A class can have attributes of any types – including objects of other user-defined classes.
  - This is called composition.
- This enables the ability to handle more complex problems



Composition




```
class Semester:
 def __init__(self, id, courses):
   self.id = id
   self.courses = courses

 def addCourse(self, course):
   self.courses.append(course)

 def removeCourse(self, id):
   for course in self.courses:
     if id == course.id:
       self.courses.remove(course)
       break

 def display(self):
   s = f'SemesterID: {self.id}, NumCourses: {len(self.courses)}'
   for course in self.courses:
     s += f'\n\t {course.display()}'
   return s
```

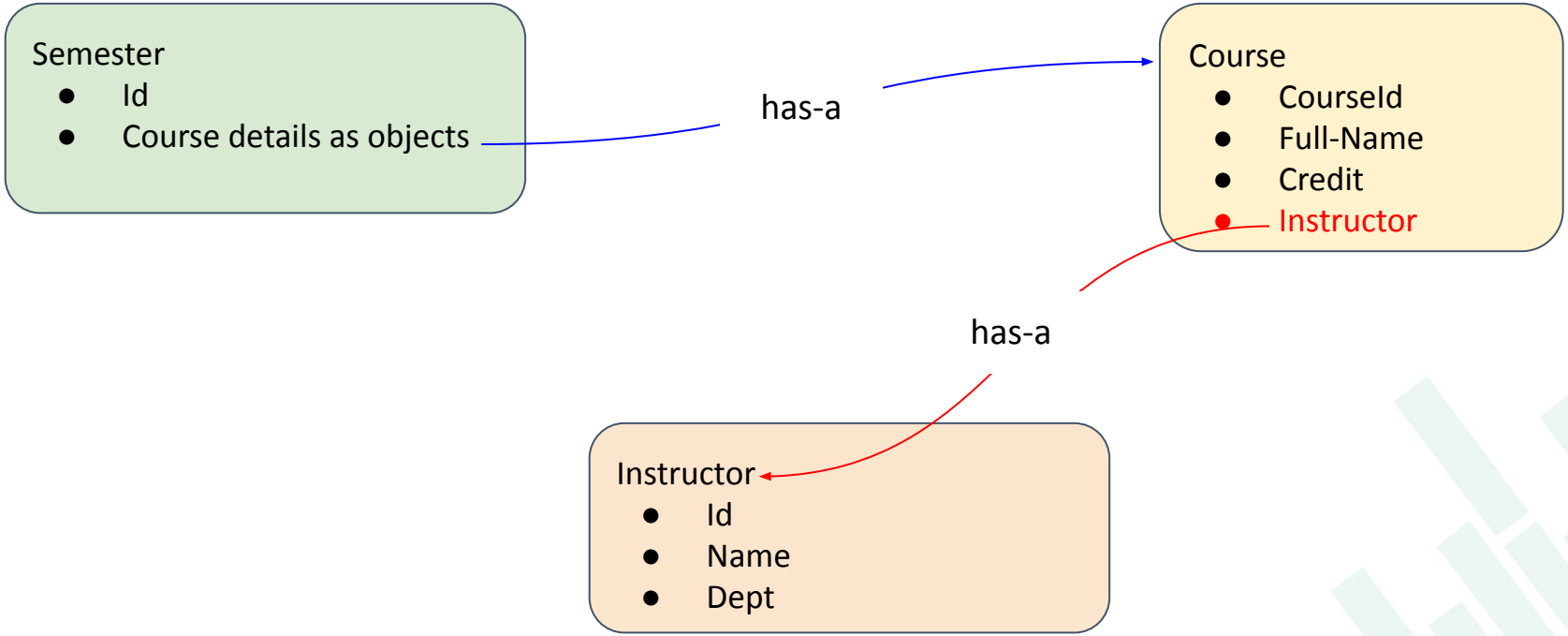
```
class Course:
 def __init__(self, id, name, credit):
     self.id = id
     self.name = name
     self.credit = credit

 def display(self):
   return f'CourseID: {self.id}, CourseName:
{self.name}, Credit: {self.credit}'
```

```
c1 = Course('CSE101', 'IP', 4)
c2 = Course('MTH101', 'Mathematics', 4)
c3 = Course('CSD101', 'HCI', 4)
sem = Semester('Monsoon', [c1, c2, c3])
print(sem.display())
c4 = Course('COM101' 'Comm', 4)
sem.addCourse(c4)
```

```
SemesterID: 'Monsoon', NumCourses: 3
        CourseID: CSE101, CourseName: IP, Credit: 4
        CourseID: MTH101, CourseName: Mathematics, Credit: 4
        CourseID: CSD101, CourseName: HCI, Credit: 4
```

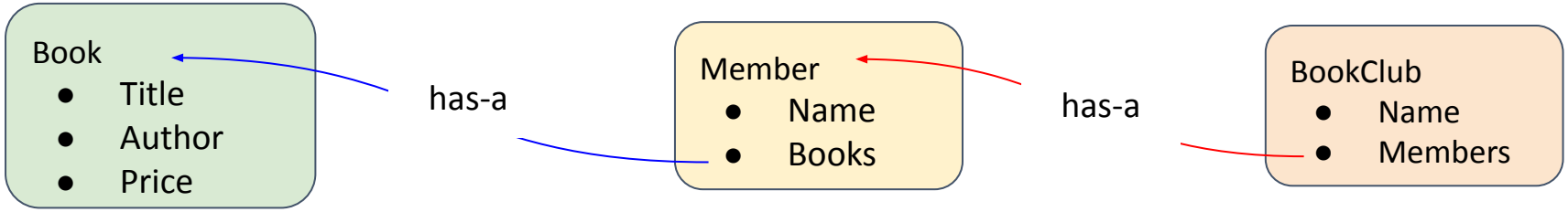
Composition

- Each course must be taught by an instructor



Composition: Example

- BookClub
  - A book club has members, who owns books



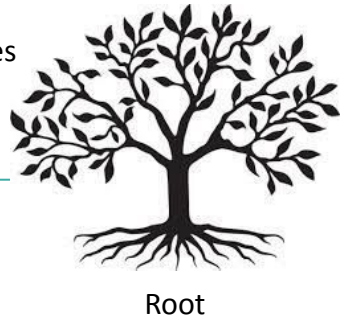
```python
class Book:
    def __init__(self, title, author, price):
        self.title = title
        self.author = author
        self.price = price
```

```python
class Member:
    def __init__(self, name, books):
        self.name = name
        self.books = books
    def add_book(self, book):
        self.books.append(book)
    def remove_book(self, book):
        self.books.remove(book)
```

```python
class BookClub:
    def __init__(self, name, members):
        self.name = name
        self.members = members
    def add_member(self, member):
        self.members.append(member)
    def remove_member(self, member):
        self.members.remove(member)
    def find_book(self, book):
        for member in self.members:
            if book in member.books:
                return member
    def transfer_book(self, m1, m2, book):
        m1.remove_book(book)
        m2.add_book(book)
```

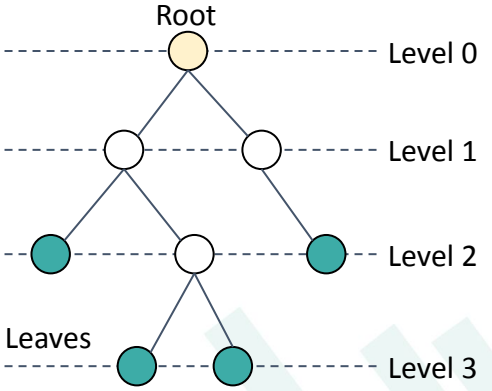
```python
if __name__ == "__main__":
    book1 = Book("The Great Gatsby", "F. Scott Fitzgerald", 10)
    book2 = Book("The Catcher in the Rye", "J. D. Salinger", 8)
    book3 = Book('The 4-Hour Workweek', 'Tim Ferriss', 15)
    book4 = Book('The Lean Startup', 'Eric Ries', 10)

    member1 = Member("David", [book2])
    member2 = Member("Aaron", [book3, book4])
    member3 = Member("Emily", [book1])

    book_club = BookClub("IIITD Book Club", [])
    book_club.add_member(member1)
    book_club.add_member(member2)
    book_club.add_member(member3)

    member = book_club.find_book(book1)
    print(member.name) # Emily
    book_club.transfer_book(member3, member2, book)
    member = book_club.find_book(book)
    print(member.name) #Aaron
```

Composition Example: Tree



- Tree
  - A data structure which is composed of nodes (vertices)
  - Each tree has a root node
  - Each node (including root) can have children nodes
    - A node can have only one parent node
  - A node without a child is called a leaf node
  - Each node can store some values

- Binary tree
  - A tree in which each node can have maximum two children – either 0, 1, or 2 childs.
    - Hence, the notion of left and right child



Composition Example: Tree

- How can we represent a 'Tree'?
  - Note: All nodes are the same – they should behave exactly in the same way.

Class 'Node'

- Attributes
  - data
  - Left_child
  - Right_child
- Methods
  - updateVal(val)

```
class Node:
    #Attributes: data, lchild, rchild
    def __init__(self, val):
        self.data = val
        self.lchild = None
        self.rchild = None
    def updateVal(self, val):
        ...
```

```
root = Node(15)
n1, n2 = Node(9), Node(17)
root.lchild(n1)
root.rchild(n2)
```