COMP 1531
Software Engineering Fundamentals

Week 04

Domain Modelling using OO Design Techniques & OO Programming with Python

So far, to create a domain model

- Do a noun/verb analysis to identify conceptual classes in the system
- not every conceptual class may be required
- some conceptual classes may become attributes
- Draw a CRC model
- Map the CRC model to a class diagram

Let us look at another case-study:

Case Study - 2

- UNSW has several departments. Each department is managed by a chair, who is a professor.
- Professors must be assigned to only one department. At least one professor teaches each course, but a professor may be on sabbatical and not teach any course.
- Each course may be taught more than once by different professors.
- We know of the department name, the professor name, the professor employee id, the course names, the course schedule, the term/year that the course is taught, the departments the professor is assigned to, the department that offers the course
- Draw a class diagram for the above case-study

Steps to develop a domain model for a system

1. Read the problem statement and identify classes
   - Abstract or tangible "things" in our problem domain (nouns and noun phrases) determined from requirement analysis
   - e.g., departments, chair, professor
2. Find associations
   - Verbs that join the nouns e.g., professor (noun) teaches (verb) students (noun)
3. Draw CRC diagram

Defining the CRC cards

Professor	
Knows Name *Knows* Employee ID *Knows assigned* Department *Teaches* Course	Department Course

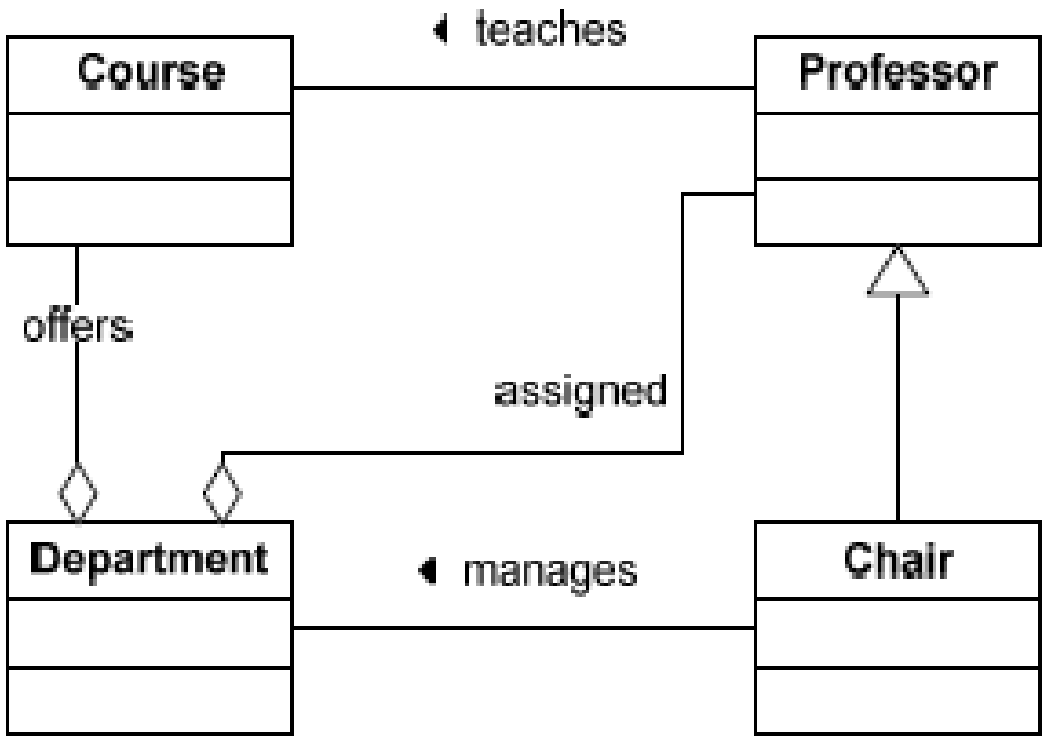
Department	
Managed by a Chair *Is Assigned* Professors *Offers* Courses *Knows* Department Name	Chair Professor Course

Course	
Offered by a Department *Taught by* Professor Knows schedule Knows term/year offered	Department Professor

Chair	
Manages a Department *Is a* Professor *Knows* Department Name	Department Professor

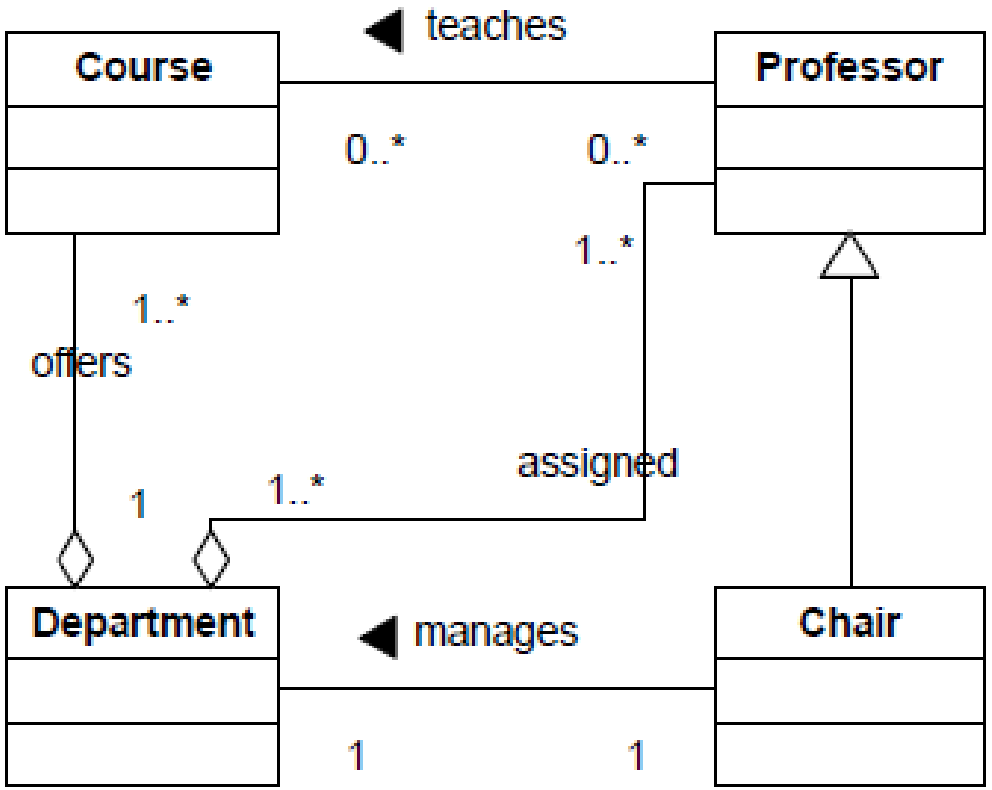
Steps to drawing the class diagram (contd…)

4. Draw the conceptual class diagram



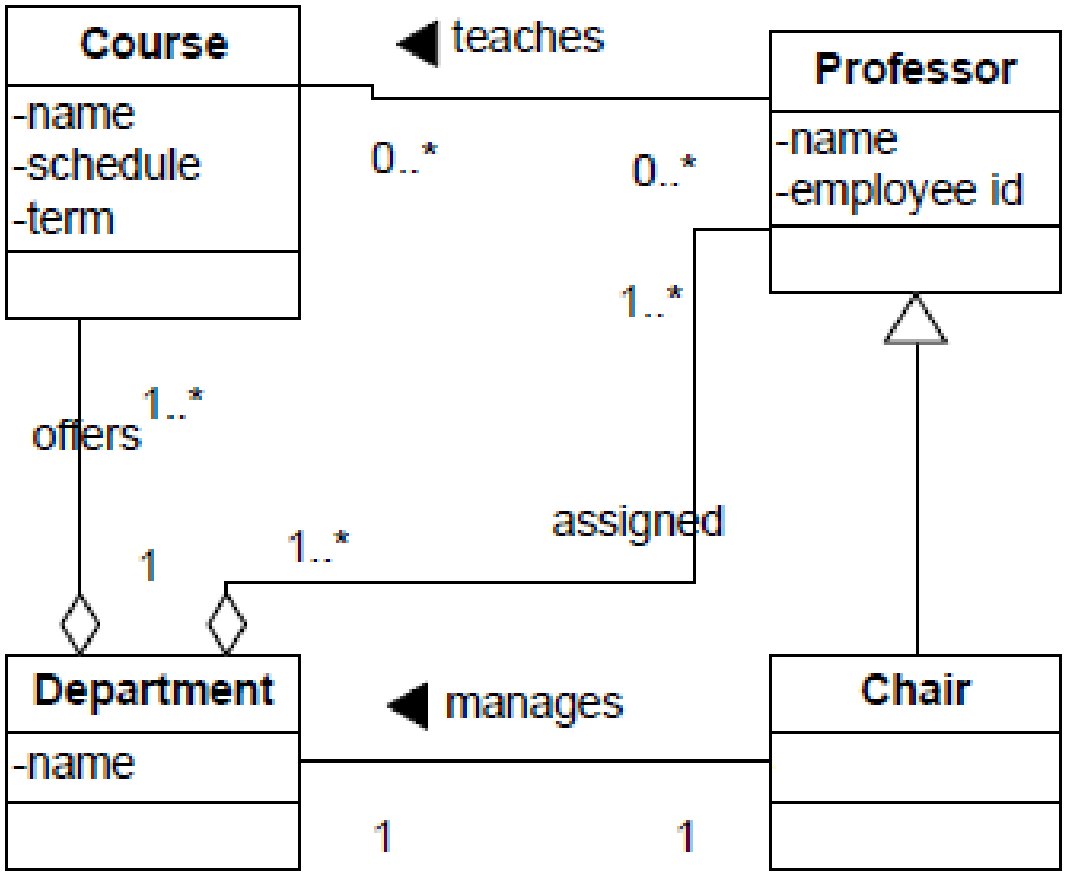
Steps to drawing the class diagram (contd…)

5. Fill in the multiplicity



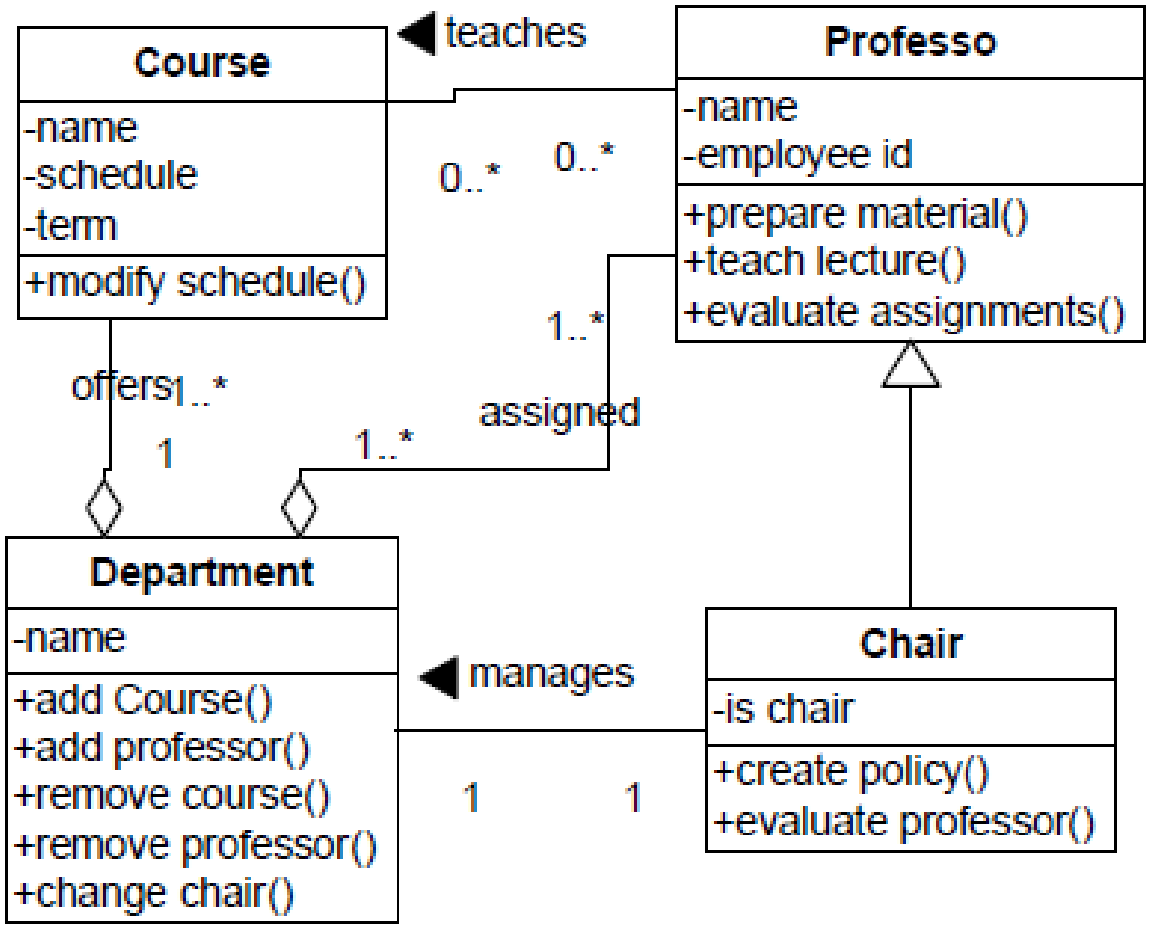
Steps to drawing the class diagram (contd…)

5. Identify attributes



Steps to drawing the class diagram (contd…)

5. Identify behaviours
6. Review class diagram and fine tune it



Walk through scenarios

7. Pick different use-case scenarios and do a scenario walk-through with the identified classes – identify any missing classes, attributes or methods

OO Programming with Python

- Creating classes
- Encapsulation
- Inheritance and abstract classes
- Implementing association, aggregation and composition

Defining a class in Python

Basic Python Syntax

- To create a class, use the keyword **class** followed by the name of the class e.g., **Account**

```
class ClassName:
   'Optional class documentation string'
   class_suite
```

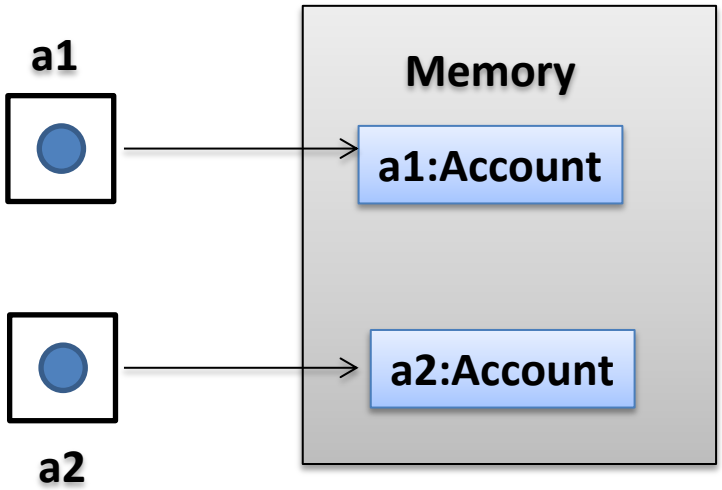
```
class Account:
      'Common base class for all bank accounts'
```

- An ***object instance*** is a specific realization of the class
  - Defining a class, does not actually create an object

Creating object instances

- A class is sometimes referred to as an object's type
- An object instance is a specific realization of the class
- Create an instance of the Account class as follows

```
class Account:
      'Common base class for all bank accounts'

# a1 and a2 are object instances
a1 = Account()
a2 = Account()
```

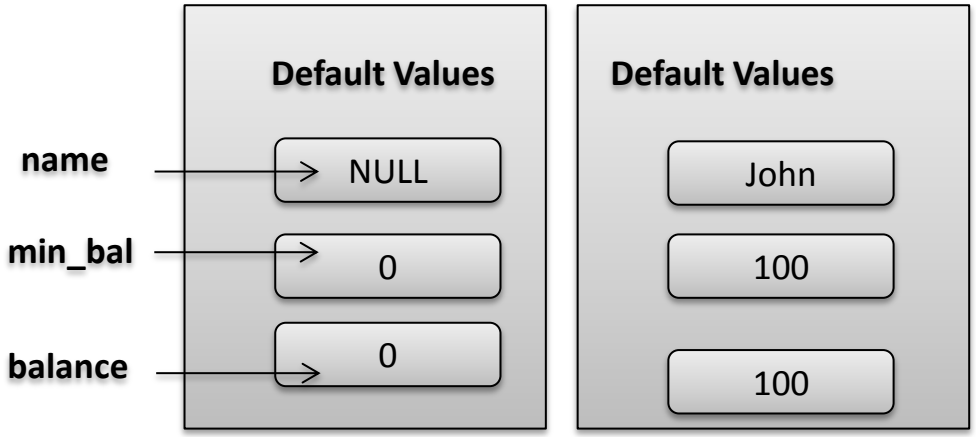


a1 == a2 -----> True or False?

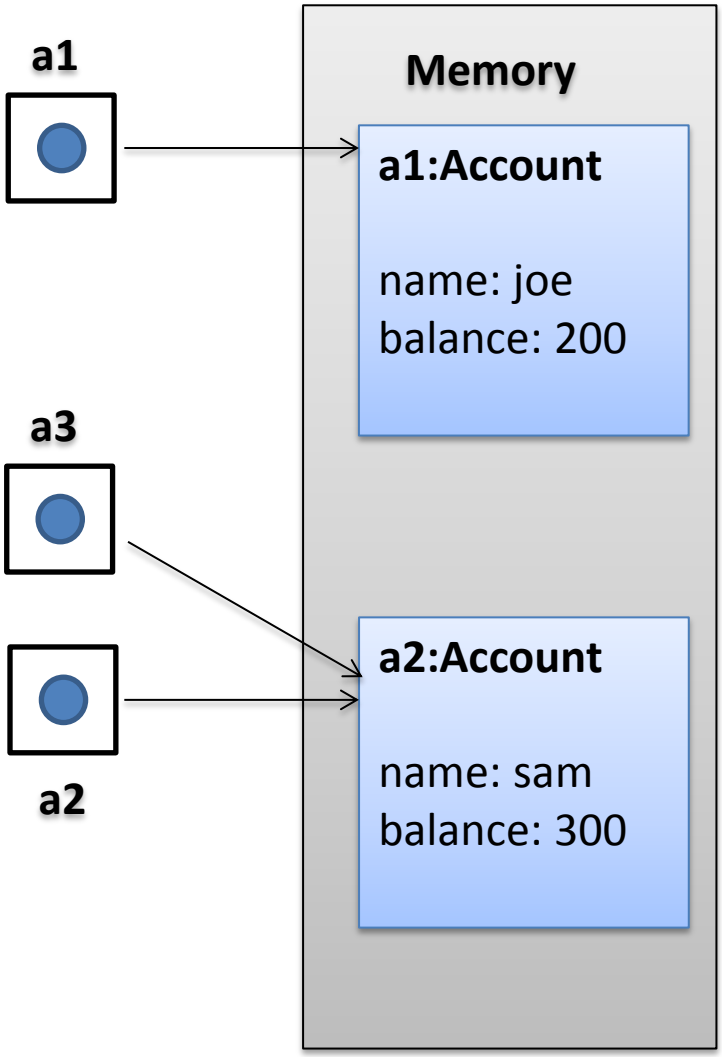
Constructor & Instance Variables

- A special method that creates an object instance and assign values (initialisation) to the attributes (instance variables)
- Constructors eliminate default values
- When you create a class without a constructor, Python automatically creates a default "no-arg" constructor for you

```
class Account:

    def __init__(self,name,min_bal):
        self.name = name
        self.min_bal = min_bal
        self.balance = min_bal

# a1 is an object instance
a1 = Account("John",100)
```



Object References



a1 == a2 -----> True or False?

Consider,
a3 = a1
a3 == a1 -----> True or False?

Instance Methods

- Similar to instance variables, methods defined inside a class are known as **instance methods**
- Methods define what an object can do.
- In Python, every instance method, must specify **`self`** (the specific object instance) as an argument to the method including the constructor (`__init__()`)

```
class Account:

    def __init__(self,name,min_bal):
        self.name = name
        self.min_bal = min_bal
        self.balance = min_bal

    def deposit(self, amount):
        self.balance += amount;

# a1 is an object instance
a1 = Account("John",100)
a1.deposit(120)
```

Encapsulation in Python

- Python does not support strong encapsulation. Attribute names are simply prefixed with a single underscore e.g., `_name` to signal that these attributes are **private** and must not be directly accessed by clients

```
class Account:

    def __init__(self,name,min_bal):
        self._name = name
        self._min_bal = min_bal
        self._balance = min_bal

    def get_name(self):
        return self._name

    def get_min_bal(self):
        return self._min_bal

    def set_min_bal(self,min_bal):
        self._min_bal = min_bal

    def get_balance(self):
        return self._balance

    def deposit(self, amount):
        self.balance += amount;
```

Recall why we need encapsulation important

1. Encapsulation ensures that an object's state is in a consistent state

```
class Account:

    def __init__(self,name,min_bal):
        self._name = name
        self._min_bal = min_bal
        self._balance = min_bal

    # define the getter and setter methods
    # ...

    def deposit(self, amount):
        self.balance += amount;

    def withdraw(self, amount):
        if self._balance - amount <= self._min_bal:
                print("Minimum balance must be maintained")
        else:
            self.balance -= amount

a1 = Account("John",100)
a1.withdraw(50)

a1._balance = 10
print("Current balance: {0}".format(str(a1._balance)))
```

breaking encapsulation and direct assignment of the *balance* attribute, potentially set the *balance* to an amount less than minimum balance, violating the business constraint

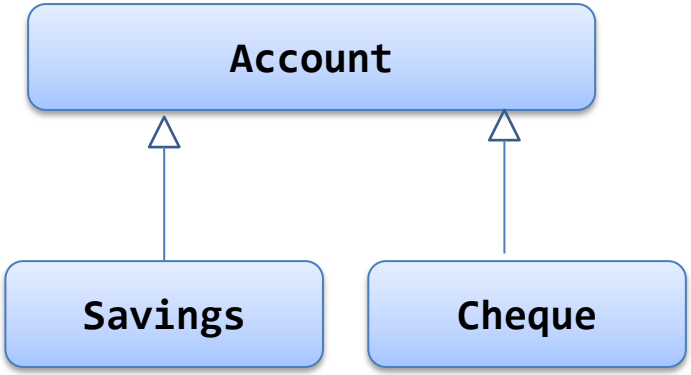
encapsulation enforces that *balance* is hidden and can only be changed through *deposit* and *withdraw* methods

Implementing Inheritance in Python

```
class Account(object):

        def __init__(self, name=None, min_bal=0):
             self._name = name
             self._balance = min_bal

        def get_name(self):
                return self._name
        def get_balance(self):
                return self._balance
        def set_balance(self,amount):
                self._balance = amount


class SavingsAccount(Account):
        def __init__(self,name,amount):
                Account.__init__(self,name,amount)
                self._saver_interest = 0.05
        def get_interest(self):
                return self._saver_interest

a2 = SavingsAccount("joe",1000)
print("{0}'s balance is {1} building interest at {2}:"
      .format(a2.get_name(),a2.get_balance(),a2.get_interest()))
```

Abstract Classes in Inheritance

- In the example below
  - `Savings` and `Cheque` both inherit from the base class `Account`
    - But `Account` is really not a real-world object
    - `Account` is a *concept* that represents some real-world objects like a Savings Account), so `Account` is said to be an abstract class
    - It does not make sense to create an instance of an abstract class



Abstract class – Vehicle
Defines common attributes e.g., *make, model, year, miles, wheels*