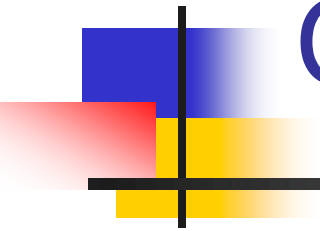
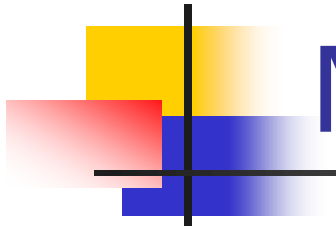


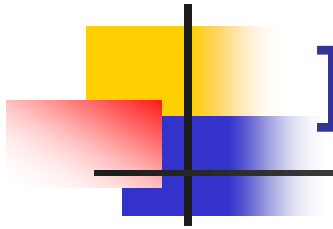
Computer Methods (MAE 3403)

Object Oriented Programming (OOP)



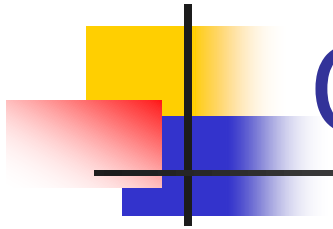
Motivation

- Other packages or modules are written in OOP using **class**.
  - np.array(), A.shape, scipy.integrate.solve_ivp, …
- OOP commonly used to write large programs/packages
  - simplifies the code with better readability
  - better describes the end goal
  - reusable
  - reduces potential bugs



Introduction

- Procedure-oriented programming (POP): list of instructions to achieve a certain functionality
  - Good for small and simple programs
- OOP: a completely different programming paradigm
- OOP isn't a must, but is a better option for large programs



OOP

- class: a template to define a logical grouping of data and functions
  - a people class, containing data (properties, attributes) such as name, age, and some methods (functions) to print ages and genders
- objects: combines attributes and methods, an instance of the class with actual values.
  - Iron man with age 35, Batman with age 33

Example: People class

```
class People():
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def greet(self):
        print("Greetings, " + self.name)
```

People class

- Data: name and age
- Method: greet

```
# initialize an object/instance "person1"
person1 = People(name = 'Iron Man', age = 35)
person1.greet()
print(person1.name)
print(person1.age)
```

```
# another independent instance
person2 = People(name = 'Batman', age = 33)
person2.greet()
print(person2.name)
print(person2.age)
```

Class

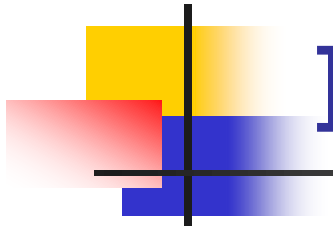
- A blueprint to define a logical grouping of data and methods

```
class ClassName(superclass):
    def __init__(self, arguments):
    # define or assign object attributes

    def other_methods(self, arguments):
    # body of the method
```

- self: must have as the first argument when you define a method. Refers to object itself, so that you can access attributes and other objects of the same object.

- Class name: CapWords
- Inherit from a `superclass'
- __init__: a special method that's run as soon as an object of a class is created, assigns initial values of attributes
- other_methods: define other functions



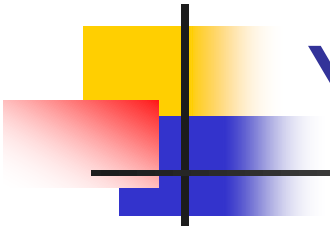
Import a class

- from Filename import ClassName
  - Directly use ClassName to create an instance/object
- import Filename
  - Use Filename.ClassName to create an instance/object for that class

Example

- Define a class named Student, with the attributes sid (student id), name, gender, type in the init method and a method called say_name to print out the student's name. All the attributes will be passed in except type, which will have a value as 'learning'.

```
class Student():
    def __init__(self, sid, name, gender):
        self.sid = sid
        self.name = name
        self.gender = gender
        self.type = 'learning'
    def say_name(self):
        print("My name is " + self.name)
```



You try

- Add a method report that print the student name as well as the student id. The method also has another argument score that will be passed in with a number between 0 and 100. Print the score too.

You try

- Add a method report that print the student name as well as the student id. The method also has another argument score that will be passed in with a number between 0 and 100. Print the score too.

```
def report(self, score):
    self.say_name()
    print("My id is: " + self.sid)
    print("My score is: " + str(score))
```

Object

- An instance of a defined class with actual values.

```
student1 = Student("001", "Susan", "F")
student2 = Student("002", "Mike", "M")
student1.say_name()
student2.say_name()
print(student1.type)
print(student1.gender)

student1.report(95)
student2.report(90)
```

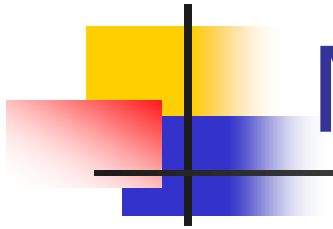
- Access attributes: student1.type
  - Try student1.+TAB
- Access methods: student1.say_name()

Class attributes

- Shared with all the instances created from one class.

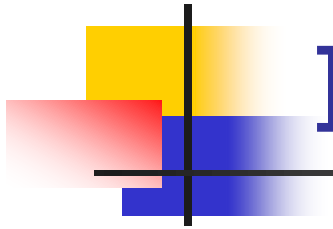
```
class Student():
    n_instances = 0
    def __init__(self, sid, name, gender):
        self.sid = sid
        self.name = name
        self.gender = gender
        self.type = 'learning'
        Student.n_instances += 1
    def num_instances(self):
        print(f'We have {Student.n_instances}-instance in total')
```

```
student1 = Student("001", "Susan", "F")
student1.num_instances()
student2 = Student("002", "Mike", "M")
student1.num_instances()
student2.num_instances()
```



More unique concepts

- Inheritance
  - Build a relationship between classes
- Encapsulation
  - Hide some private details of a class from other objects
- Polymorphism
  - Use a common operation in different ways



Inheritance

- Define a class that inherits all the methods/attributes from another class
  - child class vs. parent class(superclass)

class ClassName(superclass)

- Parent class is more general while child class is a specific type of the parent class.

Example

```
class Sensor():
    def __init__(self, name, location, record_date):
        self.name = name
        self.location = location
        self.record_date = record_date
        self.data = {}
    def add_data(self, t, data):
        self.data['time'] = t
        self.data['data'] = data
        print(f'We have {len(data)} points saved')
    def clear_data(self):
        self.data = {}
        print('Data cleared!')
```

```
import numpy as np
sensor1 = Sensor('sensor1', 'OSU', '2019-01-01')
data = np.random.randint(-10, 10, 10)
sensor1.add_data(np.arange(10), data)
print(sensor1.data)
```

A specific sensor: accelerometer

```
class Accelerometer(Sensor):
        def show_type(self):
                print('I am an accelerometer!')
acc = Accelerometer('acc1', 'OKC', '2019-02-01')
acc.show_type()
data = np.random.randint(-10, 10, 10)
acc.add_data(np.arange(10), data)
print(acc.data)
```

- Inheritance: shares the same attributes and methods as Sensor
- Have a different method show_type
  - Extended the superclass

Overriding a method

```
class OSUAcc(Accelerometer):
        def show_type(self):
            print(f'I am {self.name}, created at OSU!')
acc_osu = OSUAcc('OSUAcc', 'OSU', '2019-03-01')
acc_osu.show_type()
```

- Inherits from Accelerometer
- Override the show_type method in Accelerometer

Update attributes

- Inherit from the Sensor class, but add a new attribute brand

```
class NewSensor(Sensor):
        def __init__(self, name, location, record_date, brand):
            self.name = name
            self.location = location
            self.record_date = record_date
            self.brand = brand
            self.data = {}
new_sensor = NewSensor('OK', 'SWO', '2019-03-01', 'XYZ')
print(new_sensor.brand)
```

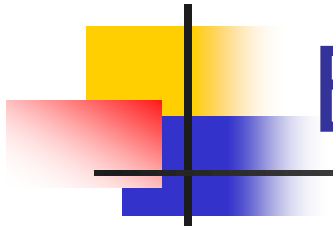
Without using much of the parent class!!

A simpler solution with super

- super avoid referring to the parent class explicitly

```
class NewSensor(Sensor):
    def __init__(self, name, location, record_date, brand):
        super().__init__(name, location, record_date)
        self.brand = brand
        self.data = {}
new_sensor = NewSensor('OK', 'SWO', '2019-03-01', 'XYZ')
new_sensor.brand
```

Use the superclass initialization method first for some attributes.
Then add the new attribute.



Encapsulation

- Restricting access to methods and attributes in class.
  - Hide complex details
  - Prevent data being modified by accident

- Use underscore as prefix, i.e., single _ or double __
  - single _: convention, should not be accessed directly
  - double __: cannot be accessed or modified directly

Example

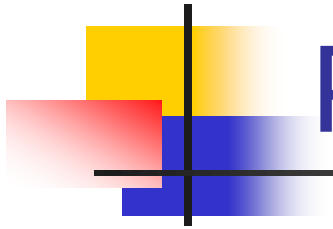
```
class Sensor():
    def __init__(self, name, location):
        self.name = name
        self._location = location
        self.__version = '1.0'
    # a getter function
    def get_version(self):
        print(f'The sensor version is {self.__version}')
    # a setter function
    def set_version(self, version):
        self.__version = version
```

```
sensor1 = Sensor('Acc', 'OSU')
print(sensor1.name)
print(sensor1._location)
print(sensor1.__version)
```

Use the "setter" and "getter" methods

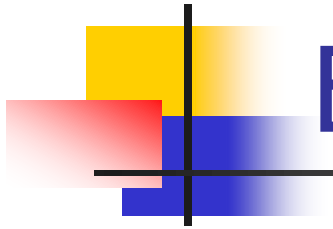
- The single and double underscores also apply to private methods in the same fashion.

```
sensor1.get_version()
sensor1.set_version('2.0')
sensor1.get_version()
```



Polymorphism

- Use a single interface with different underlying forms such as data types or classes
  - We can have commonly named methods across classes or child classes.
  - Parent class can have "abstract" methods: pass
- We override the method show_type in the OSUAcc. For parent class Accelerometer and child class OSUAcc, they both have a method named show_type, but they have different implementations.

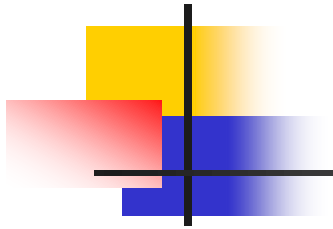


Examples

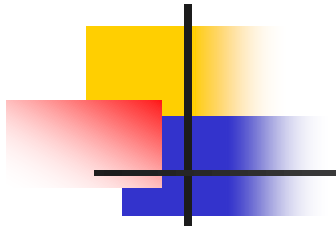
- Robot
- GPA Calculation
- Aerospace

Example 1

- Design a robot control system using Python Object-Oriented Programming.
- Create a base class called "Robot" with the following attributes and methods:
  - - Attributes: `name`, `battery_level`, `position`
  - - Methods:
    - __init__(self, name, battery_level, position)
    - move(self, distance) # reduce battery level
    - rotate(self, angle) # reduce battery level
    - perform_task(self) # empty method
    - display_status(self) # display name, battery_level, position

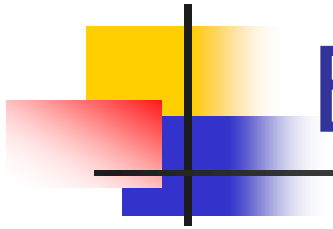


- Create three subclasses that inherit from `Robot`:
  - `CleaningRobot`, `SurveillanceRobot`, `AssemblyRobot`
- Each subclass should:
  - Implement a unique method related to its specific task
    - empty_dustbin # battery_level –1
    - night_vision_mode # battery_level -1
    - calibrate_arm # battery_level -1
  - Override the `perform_task()` method with a type-specific implementation



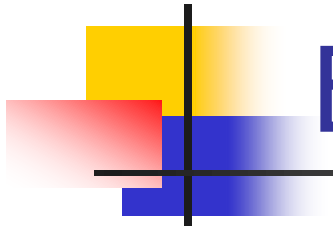
- Create a `RobotController` class that:
  - Has a list to store robots
  - Has methods to add robots, display all robot statuses, and execute tasks for all robots

- Create instances of different robot types and perform operations on them.



Example 2

- Load a txt file with students' grades for HW, exams
- Compute the final grade of each student
- Write the final grades to a file



Example 3

- Computation of aerodynamic lift and drag coefficients