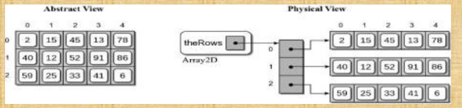
Implementing the 2-D Array

- There are various approaches that can be used to implement a 2-D array.
- Use a single 1-D array with the elements arranged by row or column.
- Use a 1-D array of 1-D arrays.
- Use lists

Array of Arrays Implementation

- Each row is stored within its own 1-D array.
- A 1-D array is used to store references to each row array.



How are the dimensions represented? Number of rows, number of columns?

2-D Array Implementation

array.py

```
class Array2D :
  def __init__( self, n_rows, n_cols ):
    self._the_rows = Array( numRows )
    for i in range( n_rows ) :
      self._the_rows[i] = Array( n_cols )

  def num_rows( self ):
    return len( self._the_rows )

  def num_cols( self ):
    return len( self._the_rows[0] )

  def clear( self, value = 0):
    for row in range( self.num_rows() ):
      row.clear( value )
```

2-D Array Implementation

- Subscript notation:

  `y = x[r, c]   x[r, c] = z`

- Subscripts are passed to the methods as a tuple.
- Must verify the size of the tuple.

2-D Array Implementation

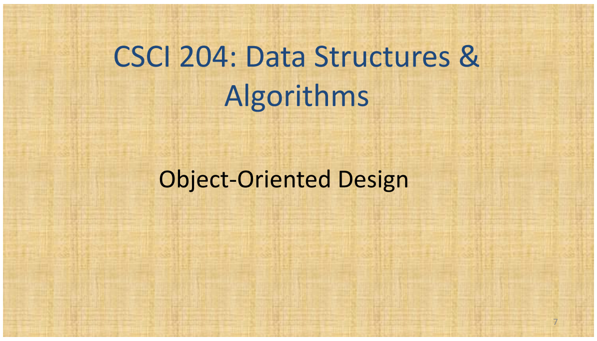
array.py

```
class Array2D :
# ...

  def __getitem__( self, ndx_tuple ):
    assert len(ndx_tuple) == 2, "Invalid number of array subscripts."
    row = ndx_tuple[0]
    col = ndx_tuple[1]
    assert row >= 0 and row < self.num_rows() \
       and col >= 0 and col < self.num_cols(), \
           "Array subscript out of range."
    the_row_array = self._the_rows[row]
    return the_row_array[col]
```

2-D Array Implementation

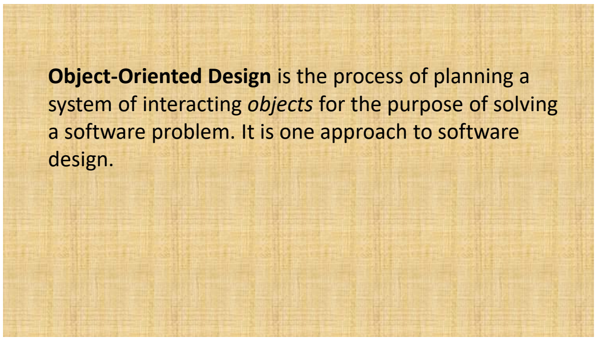
array.py

```
class Array2D :
# ...

  def __setitem__( self, ndx_tuple, value ):
    assert len(ndx_tuple) == 2, "Invalid number of array subscripts."
    row = ndx_tuple[0]
    col = ndx_tuple[1]
    assert row >= 0 and row < self.num_rows() \
       and col >= 0 and col < self.num_cols(), \
           "Array subscript out of range."
    the_row_array = self._the_rows[row]
    the_row_array[col] = value
```

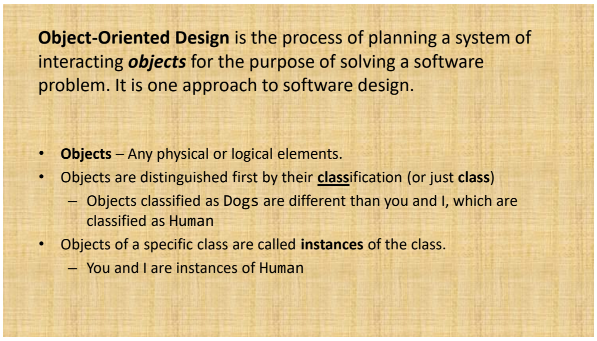


CSCI 204: Data Structures & Algorithms

Object-Oriented Design

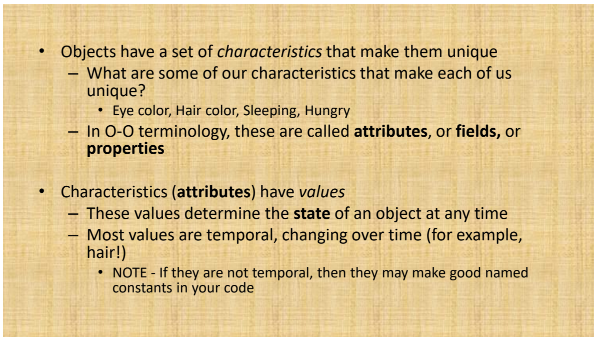


Object-Oriented Design is the process of planning a system of interacting *objects* for the purpose of solving a software problem. It is one approach to software design.

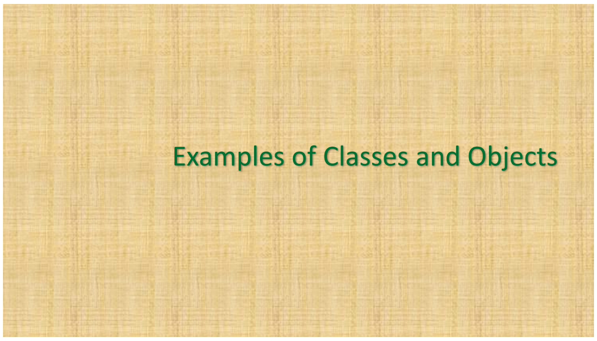


Object-Oriented Design is the process of planning a system of interacting ***objects*** for the purpose of solving a software problem. It is one approach to software design.

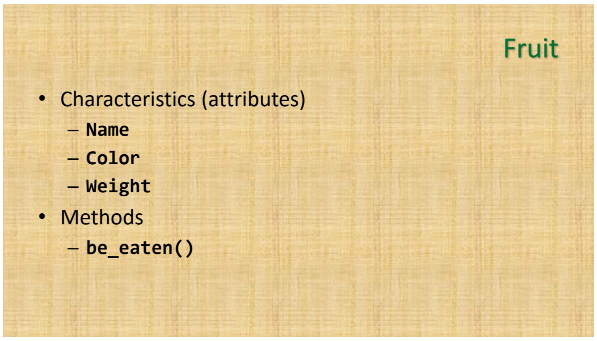
- **Objects** – Any physical or logical elements.
- Objects are distinguished first by their **classification** (or just **class**)
  - Objects classified as Dogs are different than you and I, which are classified as Human
- Objects of a specific class are called **instances** of the class.
  - You and I are instances of Human



- Objects have a set of *characteristics* that make them unique
  - What are some of our characteristics that make each of us unique?
    - Eye color, Hair color, Sleeping, Hungry
  - In O-O terminology, these are called **attributes**, or **fields,** or **properties**
- Characteristics (**attributes**) have *values*
  - These values determine the **state** of an object at any time
  - Most values are temporal, changing over time (for example, hair!)
    - NOTE - If they are not temporal, then they may make good named constants in your code

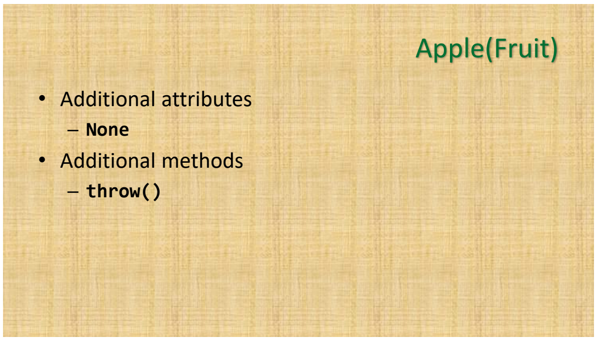


Examples of Classes and Objects



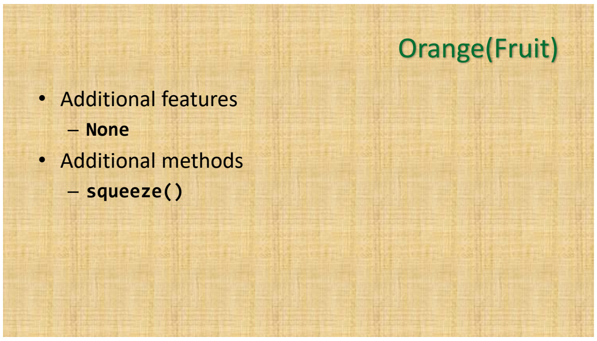
Fruit

- Characteristics (attributes)
  - **`Name`**
  - **`Color`**
  - **`Weight`**
- Methods
  - **`be_eaten()`**



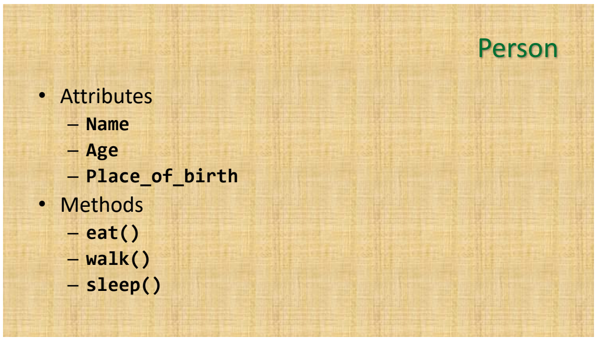
Apple(Fruit)

- Additional attributes
  - **None**
- Additional methods
  - **`throw()`**



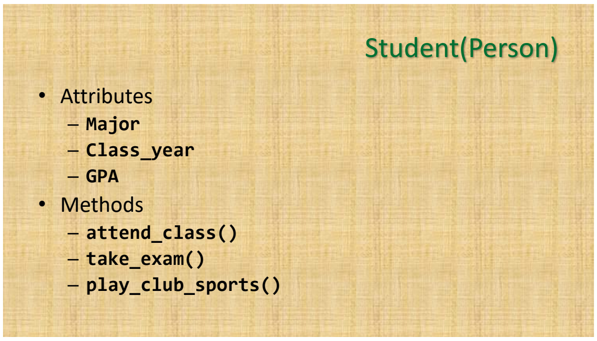
Orange(Fruit)

- Additional features
  - **None**
- Additional methods
  - **`squeeze()`**



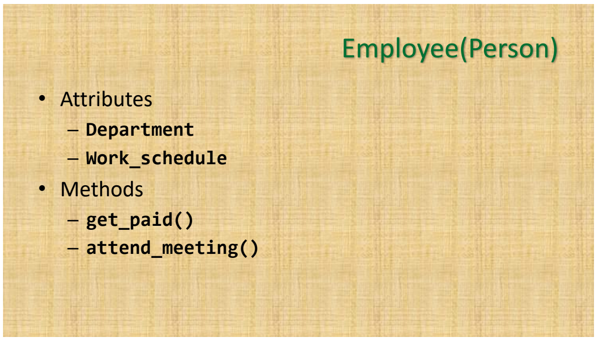
Person

- Attributes
  - **`Name`**
  - **`Age`**
  - **`Place_of_birth`**
- Methods
  - **`eat()`**
  - **`walk()`**
  - **`sleep()`**



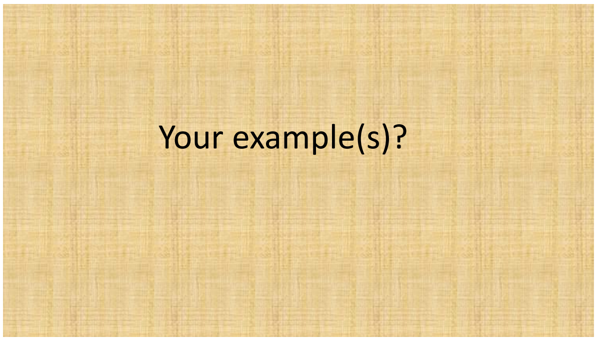
Student(Person)

- Attributes
  - **`Major`**
  - **`Class_year`**
  - **`GPA`**
- Methods
  - **`attend_class()`**
  - **`take_exam()`**
  - **`play_club_sports()`**



Employee(Person)

- Attributes
  - **`Department`**
  - **`Work_schedule`**
- Methods
  - **`get_paid()`**
  - **`attend_meeting()`**



Your example(s)?

Encapsulation and O-O design

- **Encapsulation**
  - The grouping of data and methods together into one package in such a way that the internal representation of the object is hidden
  - All interaction with the object is performed only through the object's methods
  - **Why is encapsulation an important part of the design process?**
    - **An object should always manage its own internal state!**
    - **An object is responsible for itself and how it carries out its own actions**

Encapsulation Example

- Our Array class example:
  - How Array class is defined is hidden, whether an array of ctype objects, or a Python list
  - To the outside world, all we need to know is how to use it

```
grades = Array2D(7, 3)
```

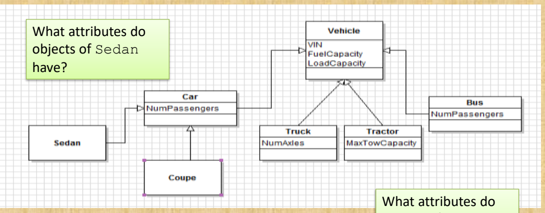
OO Design: Coupling vs. Cohesion

- **Coupling** – (aka dependency) – the degree to which each object relies on all of the other objects in the system
- **Cohesion** – the degree to which all of the functionality in an object are related
- What does a good OOD strive for?
  - **Low coupling**
    - High coupling means high interclass dependencies
    - Minimize coupling to avoid a "snowball effect" of change in one class
  - **High cohesion**
    - All public data and methods should all be related directly to the concept the class represents

Relationship: Inheritance

- The strongest class relationship
- Models the **"is-a"** relationship
- From an SE view, inheritance is POWERFUL, yet simple concept.
  - Idea – extend what you already have by adding only those capabilities / features you need
  - It can save an enormous amount of development time through **code reuse!**

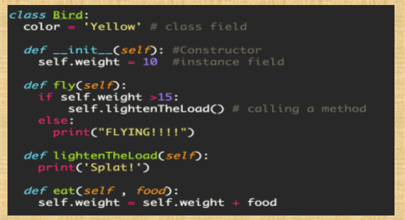
Example: Vehicles



What attributes do objects of `Sedan` have?

What attributes do objects of `Truck` have?

Code Example for Class Bird



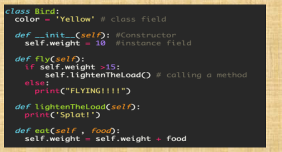
```
class Bird:
  color = 'Yellow' # class field

  def __init__(self): #Constructor
    self.weight = 10  #instance field

  def fly(self):
    if self.weight >15:
        self.lightenTheLoad() # calling a method
    else:
      print("FLYING!!!!")

  def lightenTheLoad(self):
    print('Splat!')

  def eat(self , food):
    self.weight = self.weight + food
```

big_bird = Bird()

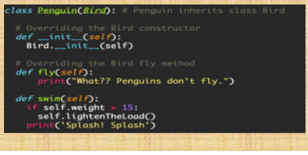
big_bird.fly()

big_bird.eat(20)

big_bird.fly()



```
class Bird:
  color = 'Yellow' # class field

  def __init__(self): #Constructor
    self.weight = 10  #instance field

  def fly(self):
    if self.weight >15:
        self.lightenTheLoad() # calling a method
    else:
      print("FLYING!!!!")

  def lightenTheLoad(self):
    print('Splat!')

  def eat(self , food):
    self.weight = self.weight + food
```



```
class Penguin(Bird): # Penguin inherits class Bird

  # Overriding the Bird constructor
  def __init__(self):
    Bird.__init__(self)

  # Overriding the Bird fly method
  def fly(self):
      print("What?? Penguins don't fly.")

  def swim(self):
    if self.weight > 15:
      self.lightenTheLoad()
    print('Splash! Splash')
```

Class Penguin that
inherits from class Bird

wheezy = Penguin()

wheezy.fly()

wheezy.eat(10)

wheezy.swim()

Design Exercise

- Take out your computer
- Write the code for class Vehicle and its subclasses Car and Truck in a file named *vehicle.py*
- Write the code in a separate file named *vehicle_app.py* for testing the Vehicle class that creates a few Car and Truck objects and prints their information.