COMP 204

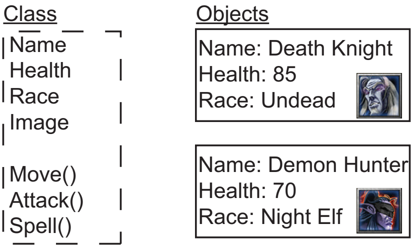
Object Oriented Programming (OOP)

Yue Li
based on material from Mathieu Blanchette

Object-Oriented Programming

- OOP is a way to write and structure programs to make them easier to design, understand, debug, and maintain.
- In OOP, computer programs manifest objects and interact with each other
- It encapsulates all the data fields that pertains under a certain concept, along with the functions (called Methods) that operate on them.
- Nearly all large-scale software projects are written using OOP

As an example:



Back to our bus simulation system

Remember our bus simulation code.
It had the information relative to a given bus dispersed over many variables:

- `bus_station` (dictionary mapping busID to stations)
- `bus_content` (dictionary mapping busID to list of people on board)

Limitations in non-OOP programs:

- Difficult to add new stuff to the program:
  - Capacity of bus (different bus may have different capacities)
  - Terminus (different bus may have different terminal stations)
  - Move speed (some bus may move to each stop at different speed)
- Having all these data in separate dictionaries makes the code complex and unintuitive.

Objected oriented programming is the solution to this problem

Classes

A **class** can also be thought of as a *template* that defines what type of information we can to keep together (*Attributes*), and what we want to do with it (*Methods*).

We have seen Python built-in classes (aka compound types):

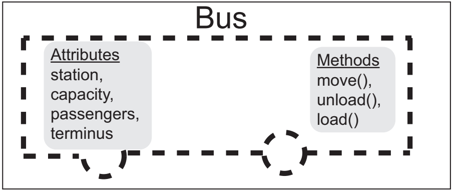
- `String`: Contains some data (the characters), and some methods that can be applied to that data (`isdecimal()`, `split()`, etc.)
- `List`: Contains a sequence of objects. Methods: `sort()`, `append()`, etc.
- `Dictionary`: Contains a set of tuple (key,value). Methods: `items()`, `keys()`, etc.

We will be learning how to create our own classes in a program

Objects

- To make use a class, we need to create objects of that class.
- An **object** is an instantiation of a class that contains all the data for a particular example of that class.

Class

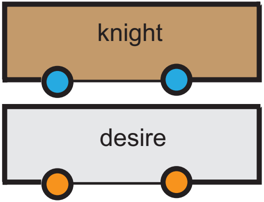


Bus

Attributes
station,
capacity,
passengers,
terminus

Methods
move(),
unload(),
load()

Objects



Example: list is a class, we can create objects of list

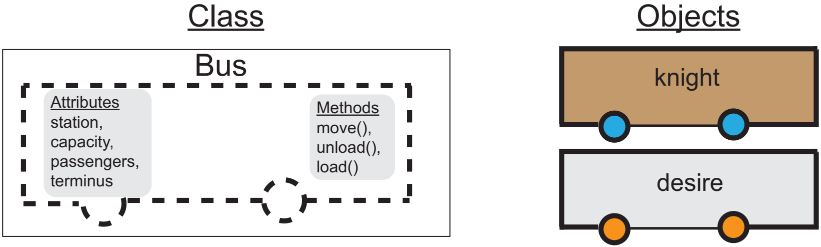
A class can have multiple objects instantiated from it, each with its own data, but all built from the same template.

```python
students_204 = list()  # could also write students = []
students_561 = list()

# students_204 and students_561 are two different objects of the same class

# We can store different values within them
# by calling the append method
students_204.append("Samy")
students_204.append("Nadia")

students_561.append("Yan")
students_561.append("Alina")

# we can call the sort method to sort them
students_204.sort()
# students_204 is now ["Nadia", "Samy"]
# students_561 is still ["Yan","Alina"]
```

OOP: Creating our own classes

Python allows us to define our own classes. A class contains two types of information:

- Attributes: Different pieces of information that will be stored within objects of that class. Attributes can be objects of any type: integers, Strings, Lists, Dictionaries, or objects belonging to user-defined classes.
- Methods: Functions that can be executed on objects of that class



OOP: Back to our bus simulation

We create a new class called Bus, which contains the following attributes: station, capacity, passengers, terminus.

```python
class Bus:
    def __init__(self):
        self.station = 0          # the position of the bus
        self.capacity = 5         # the capacity of the bus
        self.passengers = []      # the content of the bus
        self.terminus = 5         # The last station

# end of Bus class definition

knight = Bus()   # creates a object of class Bus,
                 # assigns it to variable knight
desire = Bus()   # creates a object of class Bus,
                 # assigns it to variable desire

print(knight.capacity)  # access attributes using the .

if desire.station==desire.terminus:
    print("Desire has reached its terminus")
```

Note: Each object of class Bus has its own set of values for these attributes.

Objects created from user-defined classes are mutable

We change the values of attributes of an object.

```
class Bus:
    def __init__(self):
        self.station = 0          # the position of the bus
        self.capacity = 5         # the capacity of the bus
        self.passengers = []      # the content of the bus
        self.terminus = 5         # The last station

# end of Bus class definition

knight = Bus()
desire = Bus()

# We can change the value of an object's attributes
knight.station = 1 # set knight station to 1
desire.station = knight.station + 1 # move to next station

knight.passengers.append(3) # add a customer, who is going
↪  to station 3
```

Initializer methods

```
class Bus:
    def __init__(self):
        self.station = 0          # the position of the bus
        self.capacity = 5         # the capacity of the bus
        self.passengers = []      # the content of the bus
        self.terminus = 5         # The last station
```

The initializer method (aka constructor) :

- We can define what the attributes of the class are, and how to initialize them.
- Created using syntax: `def __init__(self):`
- `def __init__(self)` gets executed when we create a new object of that class. For example: `knight = Bus()`
- `def __init__(self)` should always take at least one argument, called `self`.
  - `self` refers to the object that is being initialized.
  - When we write `self.capacity = 5`, this means: assign value 5 to the attribute capacity of the object being created.
- Any class definition should include an initializer method.

A more flexible initializer

Passing more arguments to the initializer

```
class Bus:
    def __init__(self, station=0, capacity=5,
                 passengers=[], terminus=5):
        self.station = station
        self.capacity = capacity
        self.passengers = passengers
        self.terminus = terminus
# end of Bus class definition


# We create an object of class Bus, initialized
# with station=0, capacity=5, passengers=[2,4], terminus=4
knight=Bus(station=0,capacity=5,passengers=[2,4],terminus=4)

desire=Bus() # creates an object of class Bus, initialized
             # with default values
```

Defining class methods

We can define *methods* within a class.

Each method takes as argument self, plus possibly more.

```python
class Bus:
    def __init__(self, ...):
        # Same as before

    # Define the move method, which moves
    # the bus up by one station
    def move(self):
        if self.station < self.terminus:
            self.station+=1

knight=Bus(station=0,capacity=5,passengers=[2,4],terminus=4)
desire=Bus()

knight.move()   # knight.station is now 1
knight.move()   # knight.station is now 2
desire.move()   # desire.station is now 1
```

To call a method on an object, we do `my_object.my_method()`.

Note: All methods take self as first argument. However, when calling the method, it is *not* explicitly provided as an argument. Instead, self refers to the object on which the method is called.

One more methods: .unload()

```
class Bus:
     def __init__(self, ...):
        # Same as before

    def move(self):
         # Same as before

    def unload(self):
        # removes passengers who have reached their station
        # Returns number of passengers who disembark
        out=[d for d in self.passengers if d==self.station]
        self.passengers = [d for d in self.passengers \
                           if d!=self.station]
        return len(out)

knight=Bus(station=0,capacity=5,passengers=[2,4,2],term=4)
knight.move()   # knight.station is now 1
knight.move()   # knight.station is now 2

disembarked = knight.unload()  # disembarked is now 2
```

One last methods: .load()

```python
class Bus:
    def __init__(self, ...):
         # Same as before
    def move(self):
         # Same as before
    def unload(self):
         # Same as before

    def load(self, waiting_line):
         # lets people in witing_line embark, until bus full
         # Returns the number of people who boarded
        number_boarding = min( len(waiting_line),\
                         self.capacity-len(self.passengers))
        people_boarding = waiting_line[0:number_boarding]
        self.passengers.extend(people_boarding)
        return number_boarding

knight=Bus(station=0,capacity=5,passengers=[2,4,2],terminus=4)
knight.move()   # knight.station is now 1
knight.move()   # knight.station is now 2
disembarked = knight.unload()
print(disembarked)   # prints 2
print(knight.passengers) # prints [4]

nb_loaded = knight.load([4,5,3,5,4,3])
print(knight.passengers)   # prints [4,4,5,3,5]
```

Putting it all together

See `busSim_object_oriented.py`

Notice how much simpler the simulation loop becomes!

Advantage: All the code that pertains to the bus behavior is in the Bus class. The programmer of the simulation loop does not need to know all the details of the Bus class. It only needs to know how to use its methods properly.