# COMP 204

## Object Oriented Programming (OOP)

Mathieu Blanchette

# Object-Oriented Programming

- OOP is a way to write and structure programs to make them easier to design, understand, debug, and maintain.
- It allows putting together (i.e. encapsulating) all the data that pertains to a certain concept, along with the functions (called Methods) that operate on it.
- Nearly all large-scale software projects are written using OOP
- Became popular in the 90's

# 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)

We could also have needed a lot more information: name of driver, capacity of bus (different bus may have different capacities), etc.

Having all this data in separate dictionaries makes the code complex and slow.

# Classes

A **class** can also be thought of as a template for a user-defined compound type. It defines

- **Attributes:** what type of information we want to keep together
- **Methods:** what kinds of operations want to be able to perform on that data.

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

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

# Defining a class

A Python class is defined using: `class some_class_name:`

Within a class, we define *Methods*, which are functions that can be applied to objects of that class. Most classes contain a methods called `__init__(self)`, which defines and initializes the *attributes* of the class.

Example: A Bus class.

```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
```

The Bus class contains 4 attributes: station (int), capacity (int), passengers (list) and terminus (int).

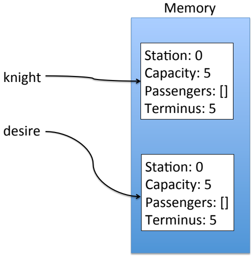
# Instantiating a class

*Instantiating a class* means creating an object from that class.

```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()  # Create an object of class Bus
desire = Bus()  # Create a second object of class Bus
```

Each object has its own set of attributes. The station, capacity, passengers, and terminus of knight and desire are different from each other.



Memory

knight

Station: 0
Capacity: 5
Passengers: []
Terminus: 5

desire

Station: 0
Capacity: 5
Passengers: []
Terminus: 5

# Using objects

We can evaluate and modify the values of attributes of an object.

```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()  # Create an object of class Bus
desire = Bus()  # Create a second object of class Bus

# We can change the value of an object's attributes
knight.station = 1

# We can evaluate an object's attribute
print(knight.station)  # 1
print(desire.station)  # 0
# update the station of desire
desire.station = knight.station + 2

# add a passenger to knight
knight.passengers.append(3)
print(knight.passengers) # [3]
print(desire.passengers) # []
```

# Initializer methods

```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
```

The initializer method (aka constructor) :

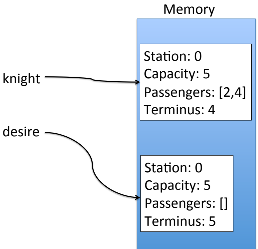
- Defines what the attributes of the class are, and how to initialize them.
- Created using syntax: def __init__(self):
- Gets executed when we create a new object of that class. For example: knight = Bus()
- Should always take at least one argument, called self.
  - Self refers to the object that is 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

```python
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(passengers=[2,4],terminus=4)

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



Memory

knight

Station: 0
Capacity: 5
Passengers: [2,4]
Terminus: 4

desire

Station: 0
Capacity: 5
Passengers: []
Terminus: 5

# Defining class methods

We can define other methods within a class.

Each method takes as first argument *self*, plus possibly more.

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

    # Increases station by one,
    # unless bus is already at terminus
    def move(self):
        if self.station< self.terminus:
            self.station+=1

knight=Bus(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 write 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

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

    def move(self):
         # Same as before

    # Removes passengers who have reached their station
    # Returns number of passengers who disembark
    def unload(self):
        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(passengers=[2,4,2],terminus=4, station=2)

disembarked = knight.unload()  # disembarked is now 2,
                               # knight.passengers is [4]
```

# 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

    # Fills the bus with as many people in waiting_line as possible.
    # Returns the number of people who boarded
    def load(self, waiting_line):
        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=1,passengers=[2,4,2],terminus=4)

nb_loaded = knight.load([4,5,3,5,4,3])  # 2
print(knight.passengers)   # prints [2,4,2,4,5]
```

# 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.