COMP 204
Object Oriented Programming (OOP) - Inheritance

Yue Li
based on material from Mathieu Blanchette
Outline

Inheritance: using Bus as example

Inheritance in ecosim (A4): Pray and Predator
Inheritance

Motivation: We often need to create classes that are closely related but not identical to an existing class.

Example: We already have created a Bus class with

- attributes: station, capacity, passengers, terminus
- methods: _init_, move, unload, load, _str_

To represent a bus where passengers have to pay to board, we may want to add new attributes like the price of the ticket and the total amount of money present on the bus.

To represent an express bus that only stops at certain stops, we may want to add attributes about the stops the bus will make, and modify the load/unload methods accordingly.

Note: We want to continue to use all the other attributes and methods defined on the Bus class.
Inheritance

Bad approach: Code Duplication

▶ Create a completely separate PayBus class.
▶ Copy-paste the Bus class code into it.
▶ Add a new attribute cost_of_ticket and cash_onboard.
▶ **Bad** because:
  ▶ We now have two copies of the Bus code. If we want to make a change to the Bus class (e.g. bug fix, or improvement), we have to remember to make the same change to the PayBus class.
  ▶ Makes program large, difficult to understand.

Good approach: Inheritance

▶ Create a PayBus class that *inherits* the attributes and methods of the Bus class.
Inheritance

Parent Class
Attributes
station, capacity, passengers, terminus

Bus
Methods
move, unload, load
__str__

Inheritance Class
PayBus

Added Attributes
cost_of_ticket, cash

Overriden Methods
__init__
load # collect cash
__str__
The Bus *generic* class

**see bus\_generic.py**

```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

    def move(self):
        # code not shown

    def unload(self):
        # code not shown

    def load(self, waiting_line):
        # code not shown

    def \_\_str\_\_(self):
        # code not shown
```
Creating a subclass from the parent class

Define a subclass PayBus from the Bus class (see paybus0.py):

```python
from bus_generic import Bus

class PayBus(Bus):
    def __init__(self, price=2):
        Bus.__init__(self)
        self.cost_of_ticket = price  # cost of a ticket
        self.cash = 0  # the total cash onboard
```

- The PayBus class is a subclass of Bus because of this line:
  ```python
class PayBus(Bus):
```
- PayBus inherits the attributes and methods of the Bus class. Those get initialized by this line:
  ```python
  Bus.__init__(self)
  ```
  which calls the `__init__` method of the parent Bus class.
- Since we call the method directly on the class rather than on an object, `self` needs to be explicitly passed as an argument.
- PayBus extends the Bus class by adding two new attributes: `cost_of_ticket` and `cash`
PayBus class

The PayBus class has 6 attributes:

▶ station, capacity, passengers, terminus are inherited from the Bus class
▶ cost_of_ticket (unique to the PayBus class)
▶ cash (unique to the PayBus class)

Methods:

▶ All of 4 non-initializer methods are inherited from the Bus class (move, unload, load, \_\_str\_\_)
▶ Therefore, we can directly use the methods already defined in the Bus class
▶ We can also override these methods (next)

```python
stm_bus = PayBus(price=2)
stm_bus.load([3, 4, 5, 2, 6, 2, 3])
stm_bus.station = 3
stm_bus.cash = 134
print(stm_bus)
# Bus at station 3 contains passengers [3, 4, 5, 2, 6].
```
Overriding methods from the generic class

Goal: Make new passengers pay price_of_ticket and add cash

Approach: Override the load() method of Bus (paybus.py)

class PayBus(Bus):
    def __init__(self, price=2):
        Bus.__init__(self)
        self.cost_of_ticket = price  # cost of a ticket
        self.cash = 0  # the total cash onboard

    def load(self, waiting_line):
        number_boarding = Bus.load(self, waiting_line)
        self.cash += number_boarding * self.cost_of_ticket
        return number_boarding

The new load() method first calls the load method of the parent class. It then updates the cash on the PayBus object.

stm_bus = PayBus(2)
stm_bus.load([3,4,5,2,6,2,3])
print("Cash = ", stm_bus.cash)  # Prints Cash = 10
We can also override the `__str__` method to make it print information about the amount of cash on board.

```python
class PayBus(Bus):
    def __init__(self, price=2):
        Bus.__init__(self)
        self.cost_of_ticket = price  # cost of a ticket
        self.cash = 0  # the total cash onboard

    def load(self, waiting_line):
        number_boarding = Bus.load(self, waiting_line)
        self.cash += number_boarding * self.cost_of_ticket
        return number_boarding

    def __str__(self):
        return Bus.__str__(self) +
               r"\nCost of ticket: " +
               str(self.cost_of_ticket) +
               r"; Cash collected: " + str(self.cash)
```
stm_bus = PayBus(2)
stm_bus.load([3, 4, 5, 2, 6, 2, 3])
print("Cash = ", stm_bus.cash)  # Prints Cash = 10

print(stm_bus)
# Bus at station 3 contains passengers [3, 4, 5, 2, 6]
# Cost of ticket: 2; Cash collected: 10

generic_bus = Bus()
stm_bus = PayBus(2)

print(generic_bus)
# Bus at station 0 contains passengers []

print(stm_bus)
# Bus at station 0 contains passengers []
# Cost of ticket: 2; Cash collected: 0

generic_bus.load([4, 2, 5, 3, 6, 4, 2, 4])
print(generic_bus)
# Bus at station 0 contains passengers [4, 2, 5, 3, 6]

stm_bus.load([4, 2, 5, 3, 6, 4, 2, 4])
print(stm_bus)
# Bus at station 0 contains passengers [4, 2, 5, 3, 6]
# Cost of ticket: 2; Cash collected: 10
Multiple inheritance classes from the same generic class

**Parent Class**

*Attributes*
- station
- capacity
- passengers
- terminus

*Methods*
- move
- unload
- load
- __str__

**Inheritance Class**

**PayBus**

*Added Attributes*
- cost_of_ticket
- cash

*Overridden Methods*
- __init__
- load # collect cash
- __str__

**ExpressBus**

*Added Attributes*
- stops

*Overridden Methods*
- __init__, unload, load

*New Methods*
- load_safe
ExpressBus class

```python
from bus_generic import Bus

class ExpressBus(Bus):
    def __init__(self, my_stops):
        Bus.__init__(self)
        self.stops = my_stops  # list of stations
```

- A class like Bus can have many different subclasses. We will create an ExpressBus subclass (see express_bus.py).
- An express bus differs from a normal bus in that it only stops at certain predetermined stop.

Note: We could also have decided that the ExpressBus class is a subclass of the PayBus class, if we needed the functionality of payments.
ExpressBus class

We now need to override the load and unload methods to allow boarding/unloading only at a station where the bus stops.

```python
class ExpressBus(Bus):
    def __init__(self, my_stops):
        Bus.__init__(self)
        self.stops = my_stops  # list of stations
        # where the bus will stop

    def unload(self):
        if self.station in self.stops:
            return Bus.unload(self)  # allow unloading
        else:
            return []  # no unloading

    def load(self, waiting_line):
        if self.station in self.stops:  # allow loading
            return Bus.load(self, waiting_line)
        else:
            return 0  # no loading
```

ExpressBus class

See the difference between the Bus and ExpressBus classes:

```
exp = ExpressBus([0, 2, 4]) # bus will stop only at 0, 2, 4
slow = Bus()
exp.load([5, 3, 1])
slow.load([5, 3, 1])
print(exp)  # Bus at station 0 has passengers [5, 3, 1]
print(slow) # Bus at station 0 has passengers [5, 3, 1]

exp.move()
slow.move()
exp.load([4, 3])   # Nobody gets loaded onto express bus
slow.load([4, 3])  # But passengers can board the slow bus
print(exp)  # Bus at station 1 has passengers [5, 3, 1]
print(slow) # Bus at station 1 has passengers [5, 3, 1, 4, 3]
```

Defining new methods (not overriding existing) for subclass

Subclasses can also have their own methods:

```python
def load_safe(self, waiting_line):
    # allows passengers to board only if
    # their destinations are among the express bus stops
    should_board = [p for p in waiting_line if p in self.stops]
    number_boarding = min(len(should_board),
                           self.capacity - len(self.passengers))
    people_boarding = should_board[0:number_boarding]
    self.passengers.extend(people_boarding)
    return number_boarding
```

load_safe() method only allows boarding for people whose destination is among the stops the express buss will make.

```python
exp = ExpressBus([0,2,4])
exp.load_safe([4,2,3,1,3,2])
print(exp)  # Bus at station 0 has passengers [4, 2, 2]
slow = Bus()
slow.load_safe([4,2,3,1,3,2])
#AttributeError: 'Bus' object has no attribute 'load_safe'
```
Outline

Inheritance: using Bus as example

Inheritance in ecosim (A4): Pray and Predator
A live demo: `ecosim_animation.py` (code provided in your A4)
Note how prays and predators behave differently in the simulation
Predator visual range: 10 cells away

Pray visual range: 2 cells away

Pray found!
Predator visual range: 10 cells away

Predator found! Run!

Predator visual range: 10 cells away
Animal Class

def __init__(self, terrain, id, position=()):
    self.id = id  # animal identifier
    self.age = 0
    self.age_max = 10  # animal life span
    self.age_spawn_min = 3  # min spawn age
    self.age_spawn_max = self.age_max  # max spawn age
    self.spawn_waiting = 0  # countdown for next spawn
    self.spawn_waiting_time = 3  # spawn recovery time
    self.hunger = 0  # hunger level of the animal
    self.hunger_max = 3  # max level of hunger
    self.visual_range = 2  # how far the animal can see
    self.position = Position(0, terrain.width-1, 0, terrain.height-1, position[0], position[1])

def starve():
    # 1-2 lines of code

def eat():
    # 1-2 lines of code

def grow():
    # 1-2 lines of code

def die():
    # 1-2 lines of code

def inspect():
    # 15-20 lines of code
class Predator(Animal):
    def __init__(self, terrain, id, position=(),
                 age_max=50, age_spawn_min=20,
                 age_spawn_max=32,
                 spawn_waiting_time=6,
                 hunger_max=30, visual_range=10):
        Animal.__init__(self, terrain, id, position)
        self.age_max = age_max
        self.age_spawn_min = age_spawn_min
        self.age_spawn_max = age_spawn_max
        self.spawn_waiting_time = spawn_waiting_time
        self.hunger_max = hunger_max
        self.visual_range = visual_range

    # predator can move to adjacent cell containing
    # a pray, a plant, or nothing
    # predator cannot move into another predator
    def move(self, terrain, visible_neighbors):
        # 100–120 lines of code
Pray is also an inheritance class of Animal

class Pray(Animal):
    def __init__(self, terrain, id, position=(),
        age_max=30, age_spawn_min=2,
        hunger_max=10, spawn_waiting_time=5,
        visual_range=2):

        Animal.__init__(self, terrain, id, position)

        self.age_max = age_max
        self.age_spawn_min = age_spawn_min
        self.age_spawn_max = self.age_max
        self.hunger_max = hunger_max
        self.spawn_waiting_time = spawn_waiting_time
        self.visual_range=visual_range

    def move(self, terrain, visible_neighbors):
        # 50-100 lines of code

We will talk about A4 once it is released.
We can write new classes that extend any existing class, including those defined in BioPython. Example: Define the MySeq class that extends the Seq class to add

- a list of confidence values (between 0 and 1) associated to each character in the sequence
- an average_confidence() method that computes the average confidence values for the sequence
- a gc_content() method that computes the fraction of bases that are either G or C
Extending BioPython classes

```python
from Bio.Seq import Seq

class MySeq(Seq):
    def __init__(self, seq, conf):
        Seq.__init__(self, seq)
        self.confidence = conf  # confidence values

        # Seq doesn't compute GC content so
        # we'll add that functionality
    def gc_content(self):
        return len([b for b in self if b in "GC"]) / len(self)

    def avg_confidence(self):
        return sum(self.confidence)/len(self.confidence)

seq1 = Seq("ACGTATG")
seq2 = MySeq("AAACG", [0.9, 0.8, 0.5, 1, 0.8])
print("GC content = ", seq2.gc_content())
print("Average confidence value = ", seq2.avg_confidence())
```