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

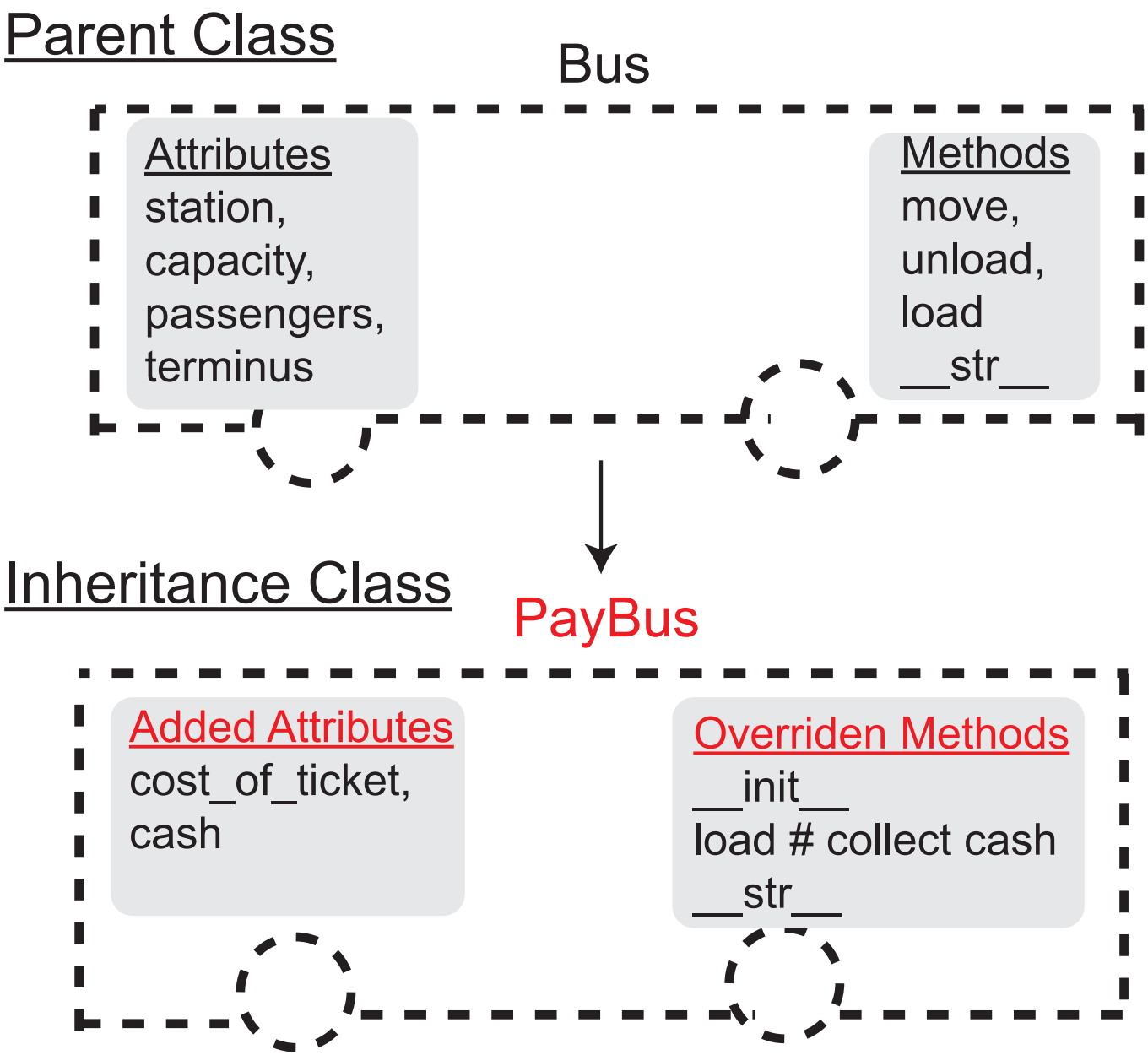
Bad approach: Code Duplication

- 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



The Bus *generic* class

see bus_generic.py

```
1  class Bus:
2      def __init__(self):
3          self.station = 0          # the position of the bus
4          self.capacity = 5         # the capacity of the bus
5          self.passengers = []      # the content of the bus
6          self.terminus = 5         # The last station
7
8      def move(self):
9          # code not shown
10
11     def unload(self):
12         # code not shown
13
14     def load(self, waiting_line):
15         # code not shown
16
17     def __str__(self):
18         # code not shown
```

Creating a subclass from the parent class

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

```
1  from bus_generic import Bus
2
3  class PayBus(Bus):
4      def __init__(self, price=2):
5          Bus.__init__(self)
6          self.cost_of_ticket = price # cost of a ticket
7          self.cash = 0               # the total cash onboard
```

- The PayBus class is a subclass of Bus because of this line:

  `class PayBus(Bus):`

- PayBus inherits the attributes and methods of the Bus class. Those get initialized by this line:

  `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)

```
9  stm_bus = PayBus(price=2)
10 stm_bus.load([3,4,5,2,6,2,3])
11 stm_bus.station = 3
12 stm_bus.cash = 134
13 print(stm_bus)
14 # 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`)

```
3  class PayBus(Bus):
4      def __init__(self, price=2):
5          Bus.__init__(self)
6          self.cost_of_ticket = price # cost of a ticket
7          self.cash = 0               # the total cash onboard
8
9      def load(self, waiting_line):
10         number_boarding = Bus.load(self, waiting_line)
11         self.cash += number_boarding * self.cost_of_ticket
12         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.

```
20 stm_bus = PayBus(2)
21 stm_bus.load([3,4,5,2,6,2,3])
22 print("Cash = ",stm_bus.cash) # Prints Cash = 10
```

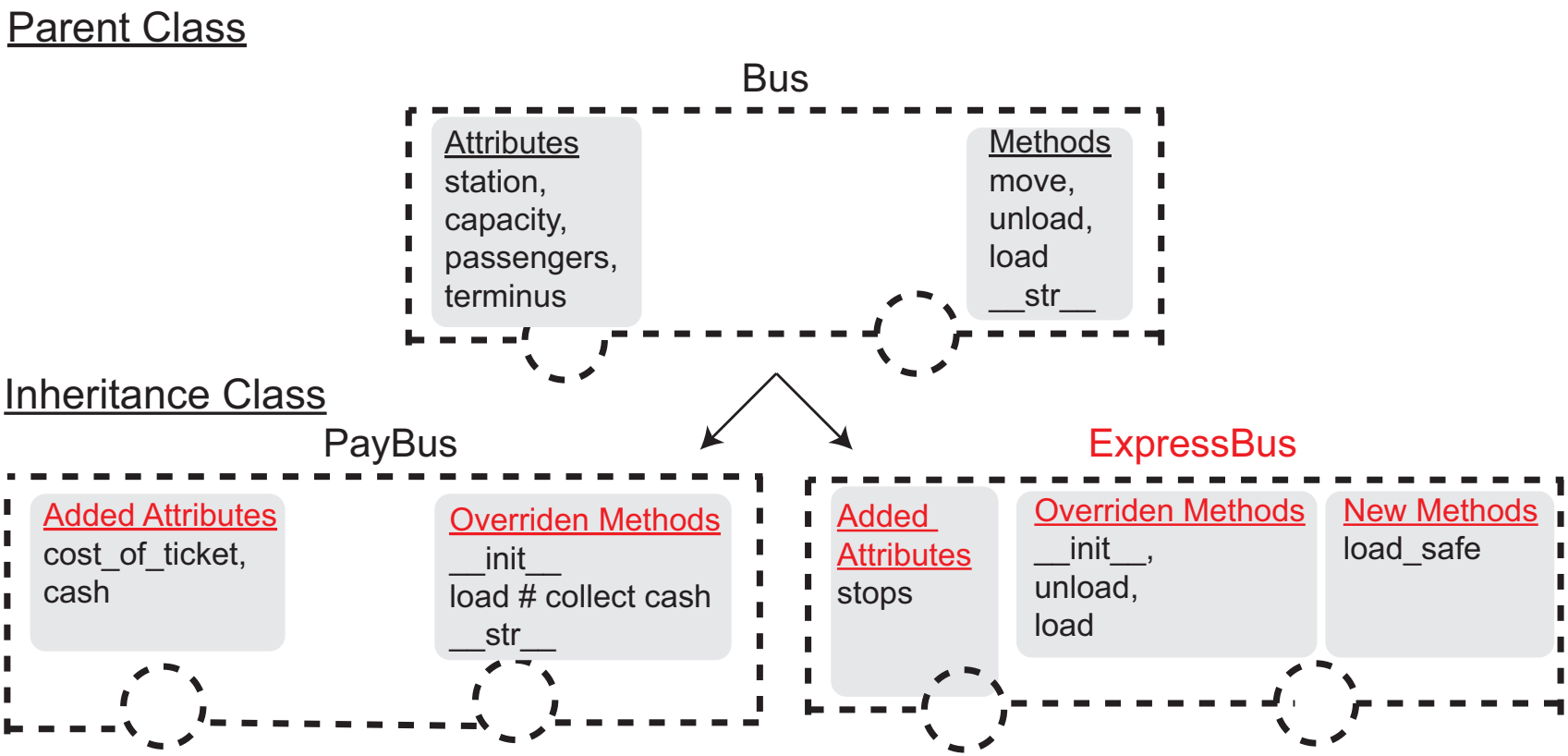
Overriding the `__str__` method from the generic class

We can also override the `__str__` method to make it print information about the amount of cash on board.

```
3  class PayBus(Bus):
4      def __init__(self, price=2):
5          Bus.__init__(self)
6          self.cost_of_ticket = price # cost of a ticket
7          self.cash = 0               # the total cash onboard
8
9      def load(self, waiting_line):
10         number_boarding = Bus.load(self, waiting_line)
11         self.cash += number_boarding * self.cost_of_ticket
12         return number_boarding
13
14     def __str__(self):
15         return Bus.__str__(self)+\
16                "\nCost of ticket: " +
                   ↪  str(self.cost_of_ticket) +\
17                "; Cash collected: " + str(self.cash)
```

```
20  stm_bus = PayBus(2)
21  stm_bus.load([3,4,5,2,6,2,3])
22  print("Cash = ",stm_bus.cash) # Prints Cash = 10
23
24  print(stm_bus)
25  #Bus at station 3 contains passengers [3, 4, 5, 2, 6]
26  #Cost of ticket: 2; Cash collected: 10
27
28  generic_bus = Bus()
29  stm_bus = PayBus(2)
30
31  print(generic_bus)
32  #Bus at station 0 contains passengers []
33
34  print(stm_bus)
35  #Bus at station 0 contains passengers []
36  #Cost of ticket: 2; Cash collected: 0
37
38  generic_bus.load([4,2,5,3,6,4,2,4])
39  print(generic_bus)
40  # Bus at station 0 contains passengers [4, 2, 5, 3, 6]
41
42  stm_bus.load([4,2,5,3,6,4,2,4])
43  print(stm_bus)
44  #Bus at station 0 contains passengers [4, 2, 5, 3, 6]
45  #Cost of ticket: 2; Cash collected: 10
```

Multiple inheritance classes from the same generic class



ExpressBus class

```
1  from bus_generic import Bus
2
3  class ExpressBus(Bus):
4      def __init__(self, my_stops):
5          Bus.__init__(self)
6          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.

```
3  class ExpressBus(Bus):
4      def __init__(self, my_stops):
5          Bus.__init__(self)
6          self.stops = my_stops # list of stations
7                                # where the bus will stop
8
9      def unload(self):
10         if self.station in self.stops:
11             return Bus.unload(self) # allow unloading
12         else:
13             return []               # no unloading
14
15     def load(self, waiting_line):
16         if self.station in self.stops:  # allow loading
17             return Bus.load(self, waiting_line)
18         else:                           # no loading
19             return 0
```

ExpressBus class

See the difference between the Bus and ExpressBus classes:

```
33 exp = ExpressBus([0,2,4]) # bus will stop only at 0, 2, 4
34 slow = Bus()
35 exp.load([5,3,1])
36 slow.load([5,3,1])
37 print(exp) # Bus at station 0 has passengers [5, 3, 1]
38 print(slow) # Bus at station 0 has passengers [5, 3, 1]
39
40 exp.move()
41 slow.move()
42 exp.load([4,3])  # Nobody gets loaded onto express bus
43 slow.load([4,3])  # But passengers can board the slow bus
44 print(exp)  # Bus at station 1 has passengers [5, 3, 1]
45 print(slow) # Bus at station 1 has passengers [5, 3, 1, 4,
   ↪  3]
46
```

Defining new methods (not overriding existing) for subclass

Subclasses can also have their own methods:

```
21  def load_safe(self, waiting_line):
22      # allows passengers to board only if
23      # their destinations are among the express bus stops
24      should_board = [p for p in waiting_line \
25                      if p in self.stops]
26      number_boarding = min(len(should_board),\
27                    self.capacity -len(self.passengers))
28      people_boarding = should_board[0:number_boarding]
29      self.passengers.extend(people_boarding)
30      return number_boarding
```

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

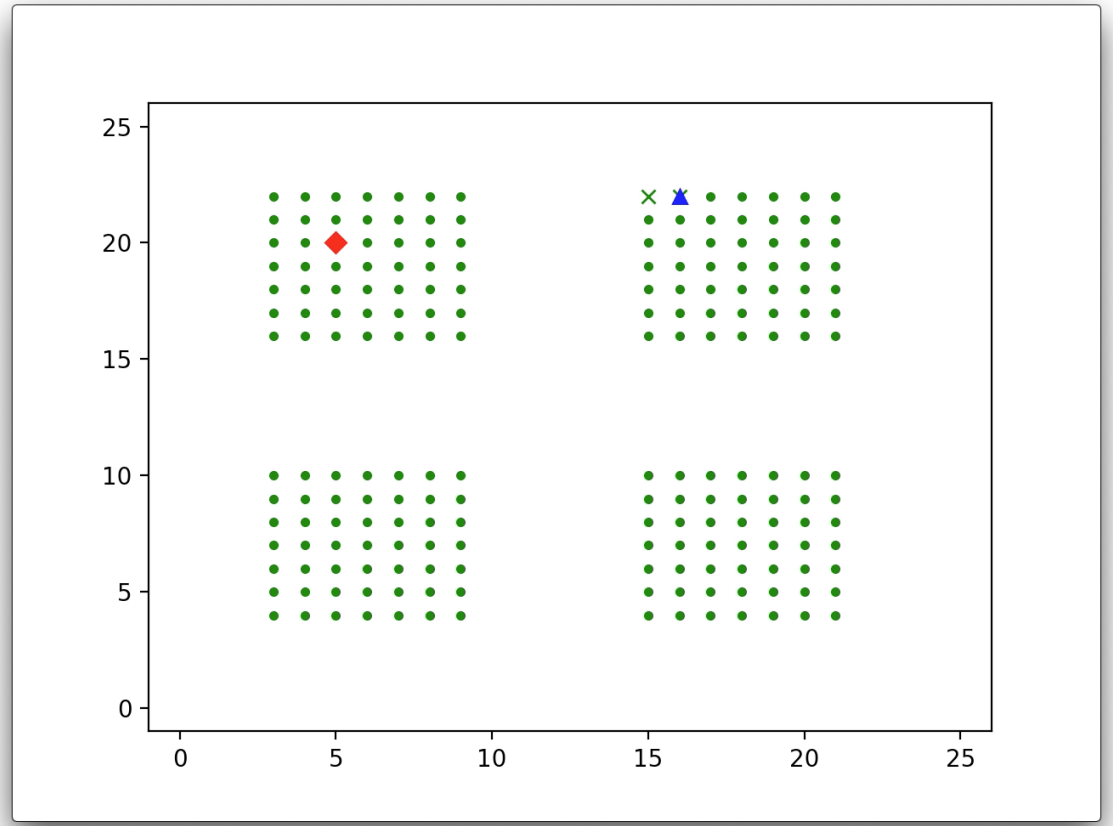
```
47  exp = ExpressBus([0,2,4])
48  exp.load_safe([4,2,3,1,3,2])
49  print(exp)  # Bus at station 0 has passengers [4, 2, 2]
50  slow = Bus()
51  slow.load_safe([4,2,3,1,3,2])
52  #AttributeError: 'Bus' object has no attribute 'load_safe'
```

Outline

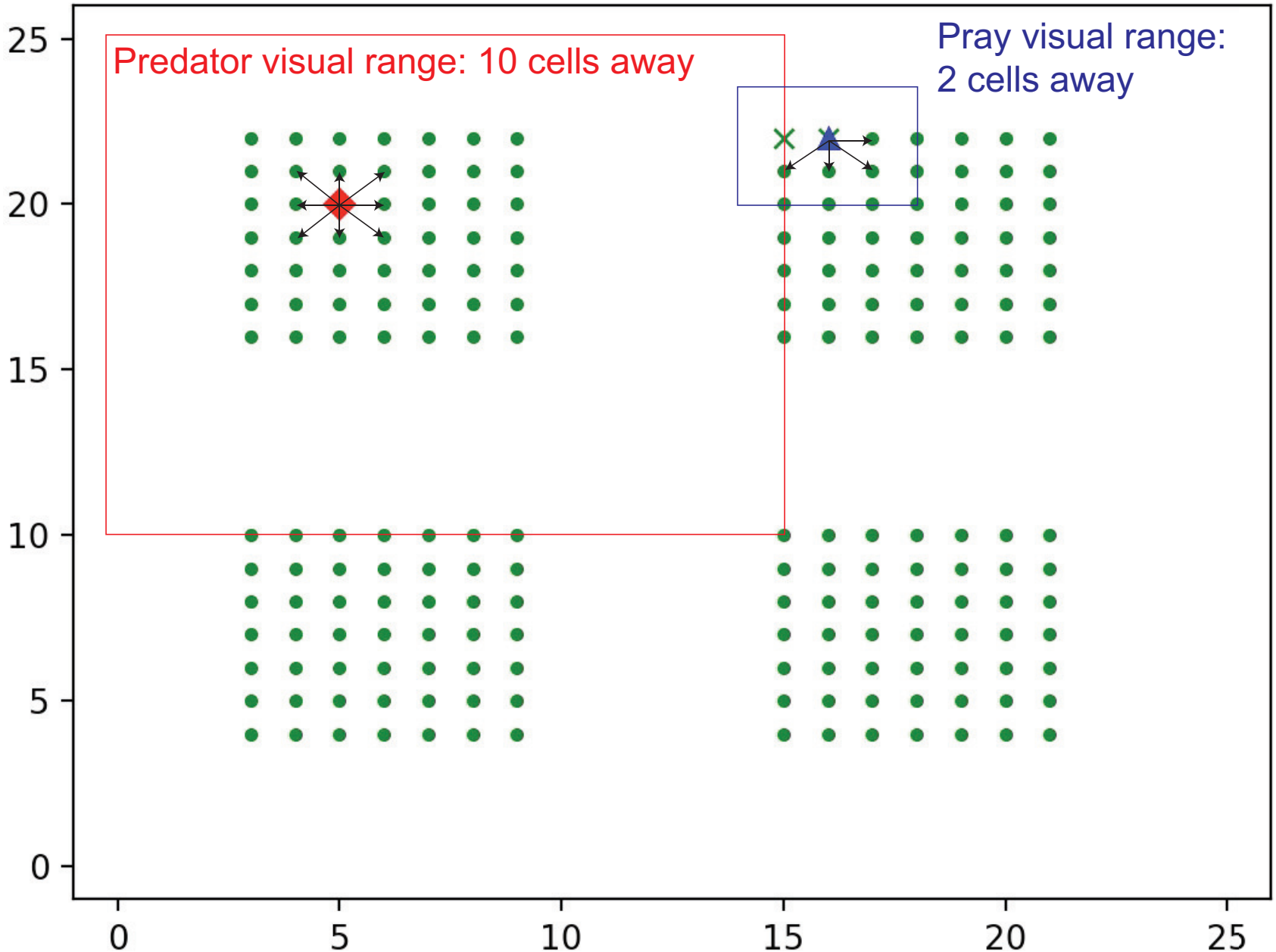
Inheritance: using `Bus` as example

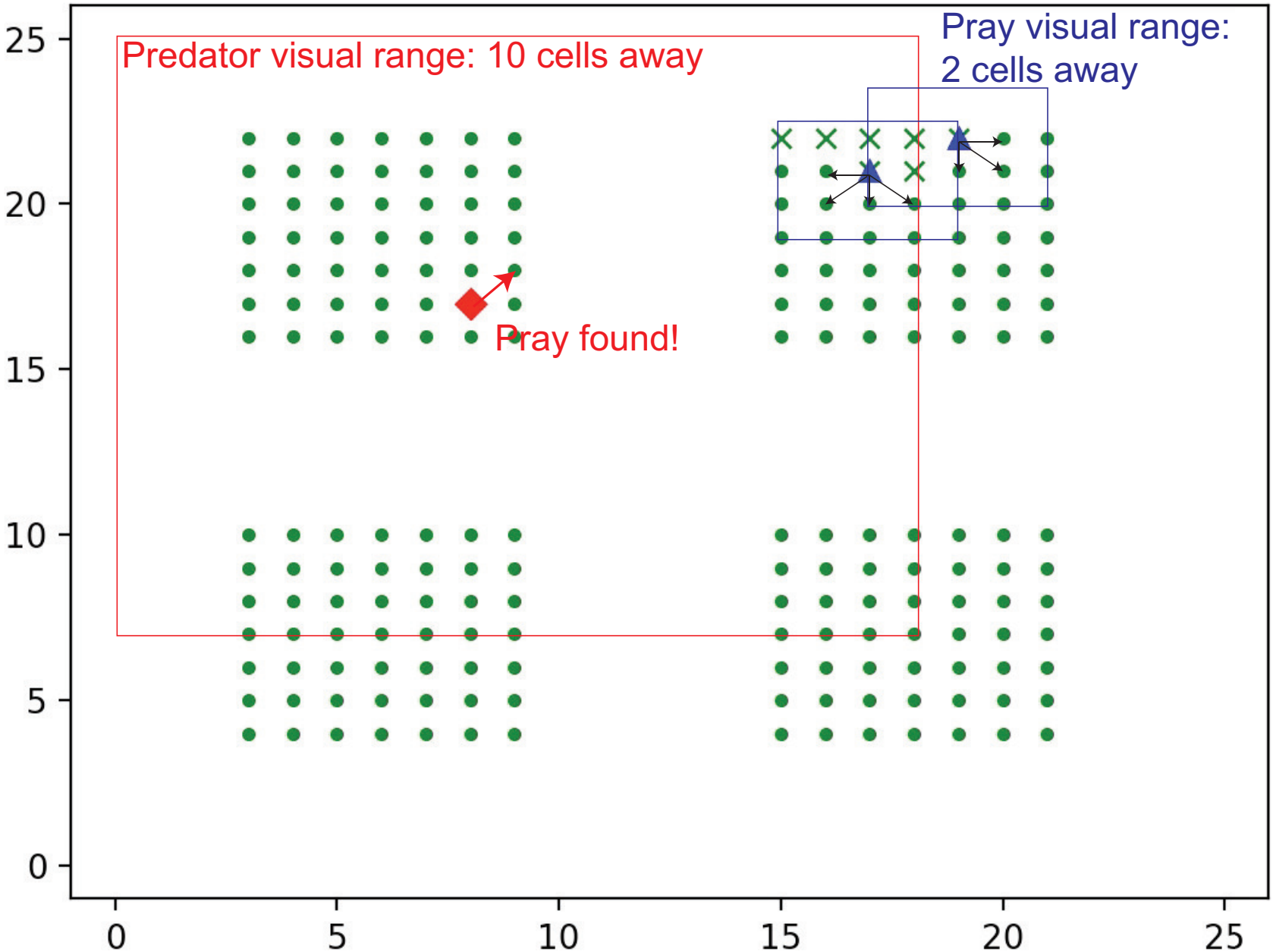
Inheritance in ecosim (A4): `Pray` and `Predator`

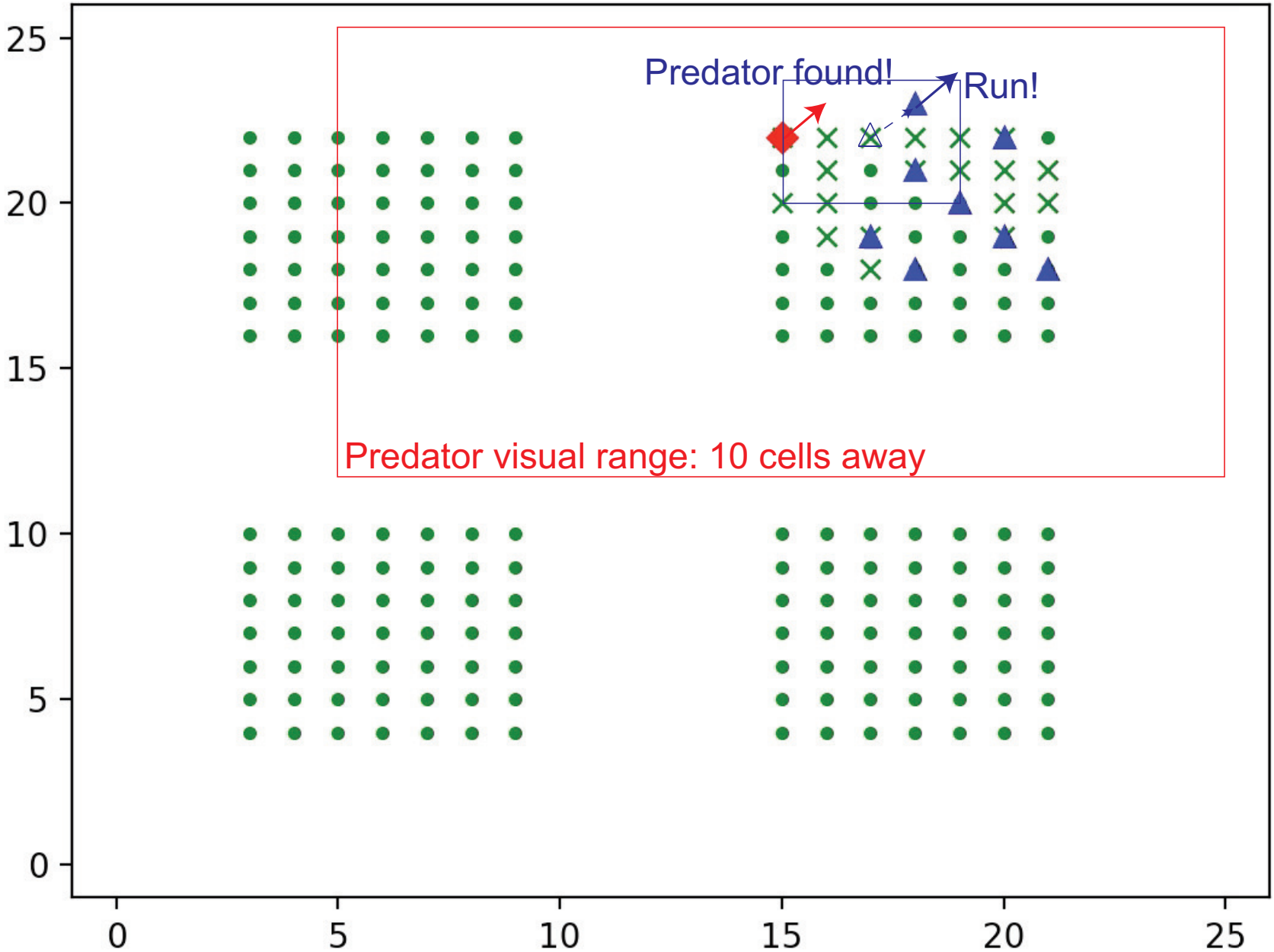
Pray and Predator (A4)



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
25
20
15
10
5
0
0
5
10
15
20
25


Predator visual range: 10 cells away
Pray visual range:
2 cells away
Pray found!
25
20
15
10
5
0
0
5
10
15
20
25


Predator found!
Run!
Predator visual range: 10 cells away
25
20
15
10
5
0
0
5
10
15
20
25

Animal Class

```
1  def __init__(self, terrain, id, position=()):
2      self.id = id # animal identifier
3      self.age = 0
4      self.age_max = 10 # animal life span
5      self.age_spawn_min = 3 # min spawn age
6      self.age_spawn_max = self.age_max # max spawn age
7      self.spawn_waiting = 0 # countdown for next spawn
8      self.spawn_waiting_time = 3 # spawn recovery time
9      self.hunger = 0 # hunger level of the animal
10     self.hunger_max = 3 # max level of hunger
11     self.visual_range=2 # how far the animal can see
12     self.position = Position(0, terrain.width-1, 0,
       ↪  terrain.height-1, position[0], position[1])
13 def starve():
14     # 1-2 lines of code
15 def eat():
16     # 1-2 lines of code
17 def grow():
18     # 1-2 lines of code
19 def die():
20     # 1-2 lines of code
21 def inspect():
22     # 15-20 lines of code
```

Predator is an inheritance class of Animal

```
1  class Predator(Animal):
2      def __init__(self, terrain, id, position=(),
3                   age_max=50, age_spawn_min=20,
4                   age_spawn_max=32,
5                   spawn_waiting_time=6,
6                   hunger_max=30, visual_range=10):
7
8          Animal.__init__(self, terrain, id, position)
9          self.age_max = age_max
10         self.age_spawn_min = age_spawn_min
11         self.age_spawn_max = age_spawn_max
12         self.spawn_waiting_time = spawn_waiting_time
13         self.hunger_max = hunger_max
14         self.visual_range=visual_range
15
16     # predator can move to adjacent cell containing
17     # a pray, a plant, or nothing
18     # predator cannot move into another predator
19     def move(self, terrain, visible_neighbors):
20         # 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.

Extending existing classes written by others

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

```
1  from Bio.Seq import Seq
2
3  class MySeq(Seq):
4      def __init__(self, seq, conf):
5          Seq.__init__(self, seq)
6          self.confidence = conf  # confidence values
7
8      # Seq doesn't compute GC content so
9      # we'll add that functionality
10     def gc_content(self):
11         return len([b for b in self if b in "GC"]) /
           ↪  len(self)
12
13     def avg_confidence(self):
14         return sum(self.confidence)/len(self.confidence)
15
16 seq1 = Seq("ACGTATG")
17 seq2 = MySeq("AAACG",[0.9, 0.8, 0.5, 1, 0.8])
18 print("GC content = ",seq2.gc_content())
19 print("Average confidence value = ",
   ↪  seq2.avg_confidence())
```