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

How can we do this?

► A bad approach: Duplication
  ► Create a PayBus class.
  ► Copy-paste the Bus class code into it.
  ► Add a new attribute \texttt{cost\_of\_ticket} and \texttt{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 \textit{inherits} the attributes and methods of the Patient class.
The Bus class

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

To create a new class PayBus that is a subclass of the Bus class:

```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
```

Notes:
The PayBus class is a subclass of Bus because of line 1:
```python
class PayBus(Bus):
```
PayBus inherits the attributes and methods of the Bus class. Those get initialized by the line:
```python
Bus.__init__(self)
```
which calls the `__init__` method of the parent class. Note: Since we are calling 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
The PayBus class has 6 attributes (station, capacity, passengers, terminus, cost_of_ticket, cash) and 5 methods (_init_, move, unload, load, __str__). They can be used like with any other class:

```python
stm_bus = PayBus(2)
stm_bus.load([3, 4, 5, 2, 6, 2, 3])
stm_bus.station = 3
stm_bus.cash = 134
print(stm_bus)
```

# Prints: Bus at station 3 contains passengers [3, 4, 5, 2, 6].

OK, so our PayBus class has this extra attribute cash, but the Bus methods don’t know about it...
Overriding the load method

Goal: Make new passengers pay price_of_ticket when they board, add to cash

Approach: Create a new load() method in the PayBus class. It will override the load() method of the parent class.

```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
```

Notes:
The load() method first calls the load method of the parent class. It then updates the amount of cash on the bus.

```python
stm_bus = PayBus(2)
stm_bus.load([3,4,5,2,6,2,3])
print("Cash = ", stm_bus.cash)  # Prints Cash = 10
```
Overriding the __str__ method

We should also revise 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) + \
               '. Amount of cash: ' + str(self.cash)
```
ExpressBus class

A class like Bus can have many different subclasses. We will create an ExpressBus subclass.

An express bus differs from a normal bus in that it only stops at certain predetermined stop.

```python
class ExpressBus(Bus):
    def __init__(self, my_stops):
        Bus.__init__(self)
        self.stops = my_stops  # list of stations
        # where the bus will 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 only allow boarding/unloading if we are at a station where the bus stops.

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:  # no loading
            return 0
ExpressBus class

See the difference between the Bus and ExpressBus classes:

```python
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 contains passengers [5, 3, 1]
print(slow) # Bus at station 0 contains 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 contains passengers [5, 3, 1]
p"
**ExpressBus class**

Subclasses can also define their own methods. Example: `Load_safe()` method only allows boarding for people whose destination is among the stops the express buss will make.

```python
def load_safe(self, waiting_line):
    # only allows boarding for passengers
    # whose destination is one of the stops
    # of the express bus
    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]

exp = ExpressBus([0, 2, 4])
exp.load_safe([4, 2, 3, 1, 3, 2])
print(exp)  # Bus at station 0 contains passengers [4, 2, 2]

slow = Bus()
slow.load_safe([4, 2, 3, 1, 3, 2])
# AttributeError: 'Bus' object has no attribute 'load_safe'
```
Extending BioPython classes

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
from Bio.Seq import Seq

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

    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())