OOP: An Example

USING INHERITANCE

- explore in some detail an example of building an application that organizes info about people
- start with a Person object
  - Person: name, birthday
    - get last name
    - sort by last name
    - get age

BUILDING A CLASS

```
import datetime

class Person(object):
    def __init__(self, name):
        """create a person called name"""
        self.name = name
        self.birthday = None
        self.lastName = name.split(' ')[-1]

    def getLastName(self):
        """return self's last name"""
        return self.lastName

    def __str__(self):
        """return self's name"""
        return self.name
```

name is a string, so split into a list of strings based on spaces, then extract last element

BUILDING A CLASS (MORE)

```
import datetime

class Person(object):
    def __init__(self, name):
        """create a person called name"""
        self.name = name
        self.birthday = None
        self.lastName = name.split(' ')[-1]

    def setBirthday(self,month,day,year):
        """sets self's birthday to birthDate"""
        self.birthday = datetime.date(year,month,day)

    def getAge(self):
        """returns self's current age in days"""
        if self.birthday == None:
            raise ValueError
        return (datetime.date.today() - self.birthday).days
```

BUILDING A CLASS (MORE)

```
class Person(object):
    def __init__(self, name):
        """create a person called name"""
        self.name = name
        self.birthday = None
        self.lastName = name.split(' ')[-1]

    def __lt__(self, other):
        """return True if self's name is lexicographically
           less than other's name, and False otherwise"""
        if self.lastName == other.lastName:
            return self.name < other.name
        return self.lastName < other.lastName

    def __str__(self):
        """return self's name"""
        return self.name
```

EXAMPLE

```
p1 = Person('Mark Zuckerberg')
p1.setBirthday(5,14,84)
p2 = Person('Drew Houston')
p2.setBirthday(3,4,83)
p3 = Person('Bill Gates')
p3.setBirthday(10,28,55)
p4 = Person('Andrew Gates')
p5 = Person('Steve Wozniak')

personList = [p1, p2, p3, p4, p5]
```

EXAMPLE OF SORTING BY <

```
for e in personList:
    print(e)
Mark Zuckerberg
Drew Houston
Bill Gates
Andrew Gates
Steve Wozniak
```

```
personList.sort()

for e in personList:
    print(e)
Andrew Gates
Bill Gates
Drew Houston
Steve Wozniak
Mark Zuckerberg
```

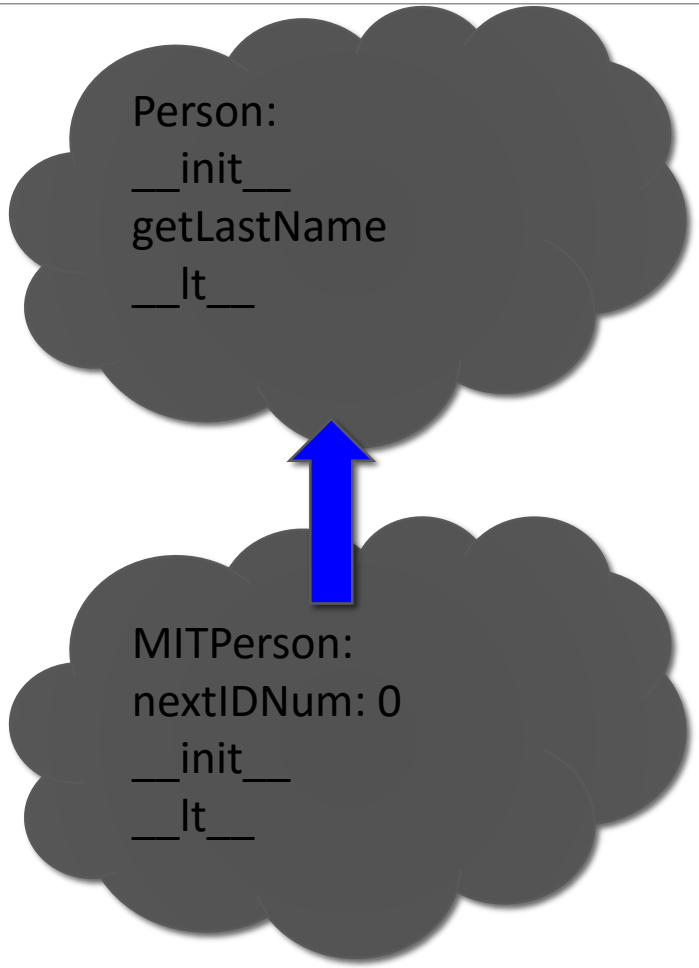
USING INHERITANCE

- explore in some detail an example of building an application that organizes info about people
  - Person: name, birthday
    - get last name
    - sort by last name
    - get age
  - MITPerson: Person + ID Number
    - assign ID numbers in sequence
    - get ID number
    - sort by ID number

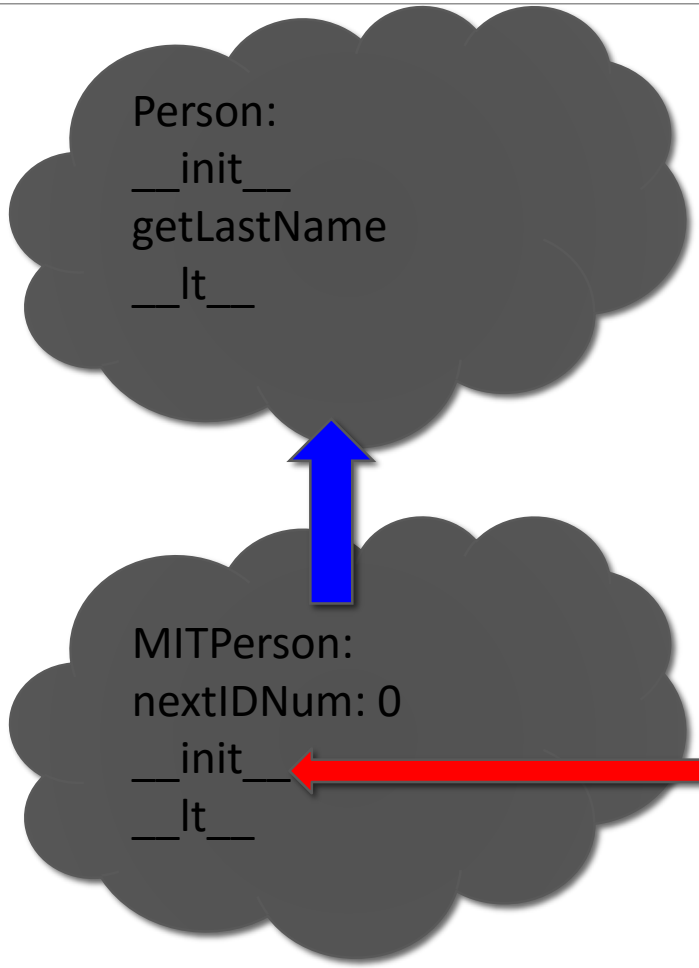
BUILDING INHERITANCE

```
class MITPerson(Person):
    nextIdNum = 0 # next ID number to assign

    def __init__(self, name):
        Person.__init__(self, name) # initialize Person attributes
        self.idNum = MITPerson.nextIdNum # MITPerson attribute: unique ID
        MITPerson.nextIdNum += 1

    def getIdNum(self):
        return self.idNum

    # sorting MIT people uses their ID number, not name!
    def __lt__(self, other):
        return self.idNum < other.idNum

    def speak(self, utterance):
        return (self.getLastName() + " says: " + utterance)
```

VISUALIZING THE HIERARCHY

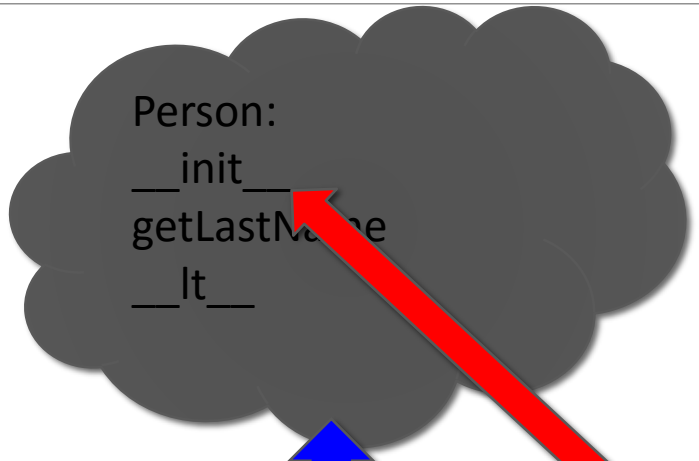


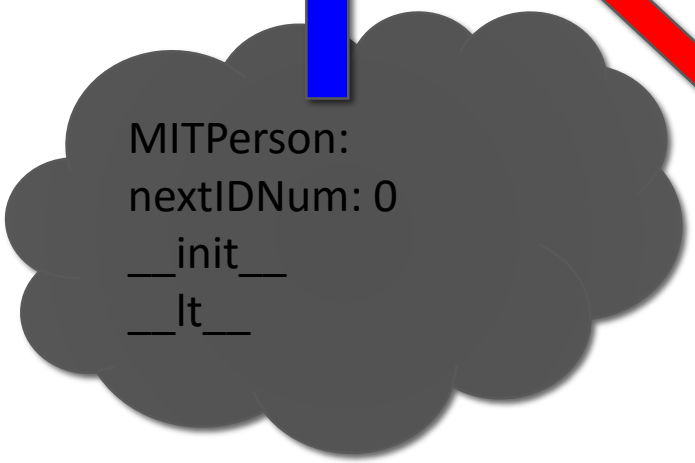
VISUALIZING THE HIERARCHY



Calling MITPerson will use this __init__ procedure (because that is the one visible in MITPerson's environment)

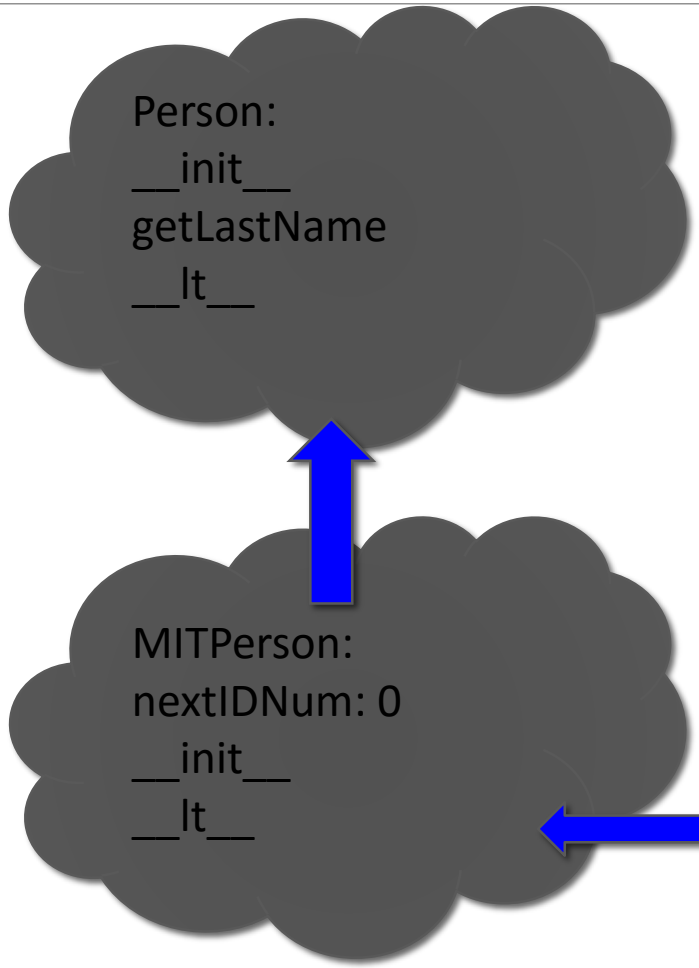
VISUALIZING THE HIERARCHY





That code uses
Person.__init__
which will in turn call this
procedure

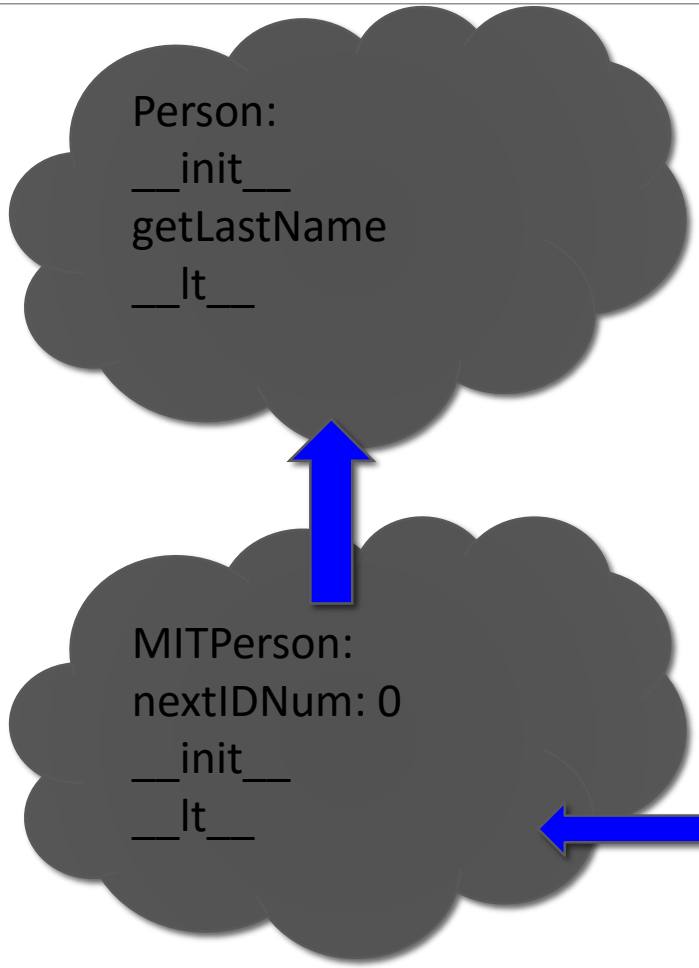
VISUALIZING THE HIERARCHY



And that creates an instance of MITPerson (because of the first call, which inherits from the class definition) but with bindings set by the inherited __init__ code

name	
birthday	
lastName	

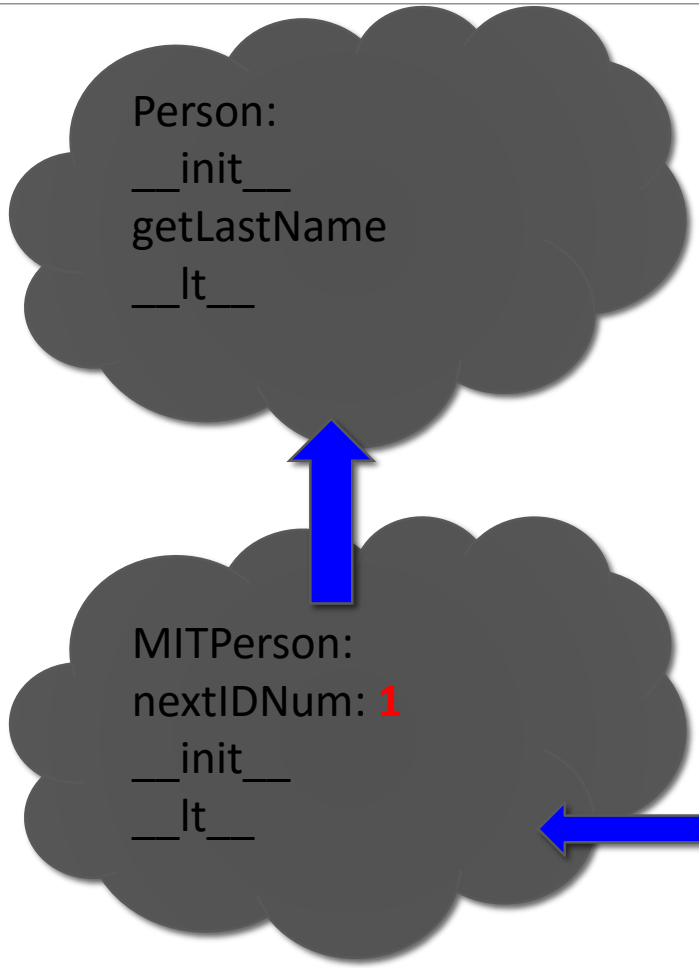
VISUALIZING THE HIERARCHY



The rest of the original __init__ code calls
self.idNum = MITPerson.nextIdNum
which looks up nextIdNum in the MITPerson environment, and creates a binding in self (i.e. the instance)

name	
birthday	
lastName	
idNum	0

VISUALIZING THE HIERARCHY

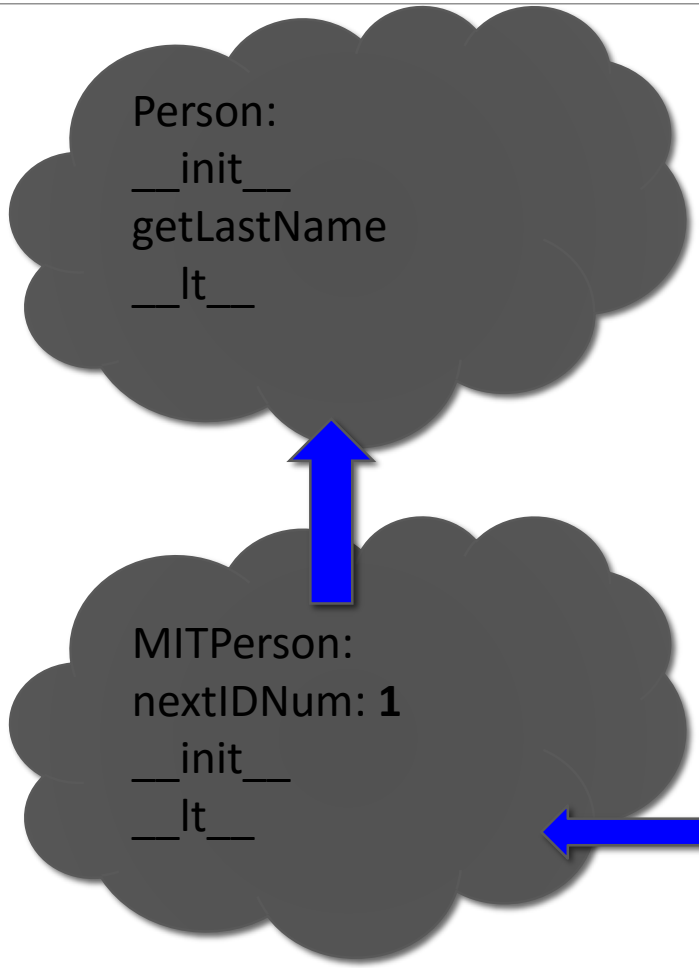


The rest of the original __init__ code calls
self.idNum = MITPerson.nextIdNum
which looks up nextIdNum in the MITPerson environment, and creates a binding in self (i.e. the instance)

And then updates nextIdNum in the MITPerson environment

name	
birthday	
lastName	
idNum	0

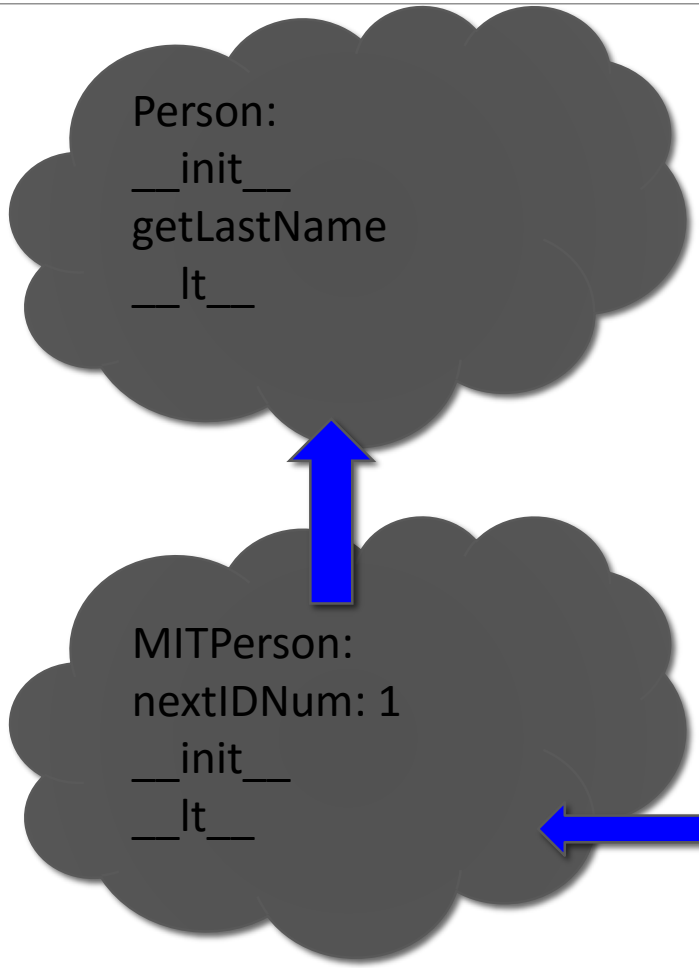
VISUALIZING THE HIERARCHY



Thus calling MITPerson a second time to create a second instance will execute the same sequence, but now nextIDNum is bound to 1

name	
birthday	
lastName	

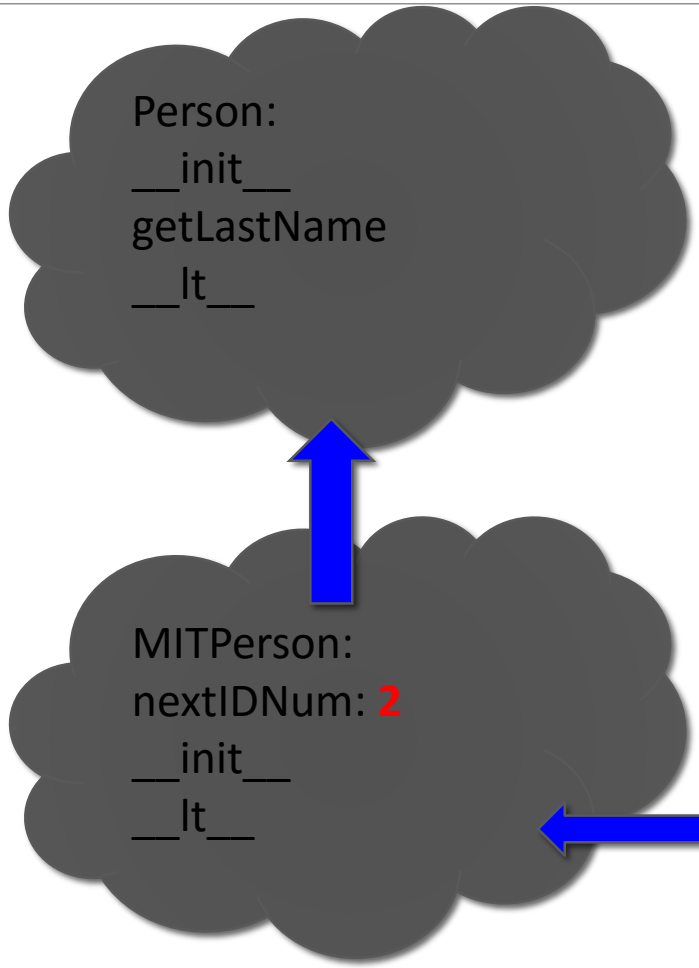
VISUALIZING THE HIERARCHY



As before, the rest of the original __init__ code calls self.idNum = MITPerson.nextIdNum which looks up nextIdNum in the MITPerson environment, and creates a binding in self (i.e. the instance)

name	
birthday	
lastName	
idNum	1

VISUALIZING THE HIERARCHY



As before, the rest of the original __init__ code calls self.idNum = MITPerson.nextIdNum which looks up nextIdNum in the MITPerson environment, and creates a binding in self (i.e. the instance)

And then updates nextIdNum in the MITPerson environment

name	
birthday	
lastName	
idNum	1

EXAMPLE

```
m3 = MITPerson('Mark Zuckerberg')
Person.setBirthday(m3,5,14,84)
m2= MITPerson('Drew Houston')
Person.setBirthday(m2,3,4,83)
m1 = MITPerson('Bill Gates')
Person.setBirthday(m1,10,28,55)

MITPersonList = [m1, m2, m3]
```

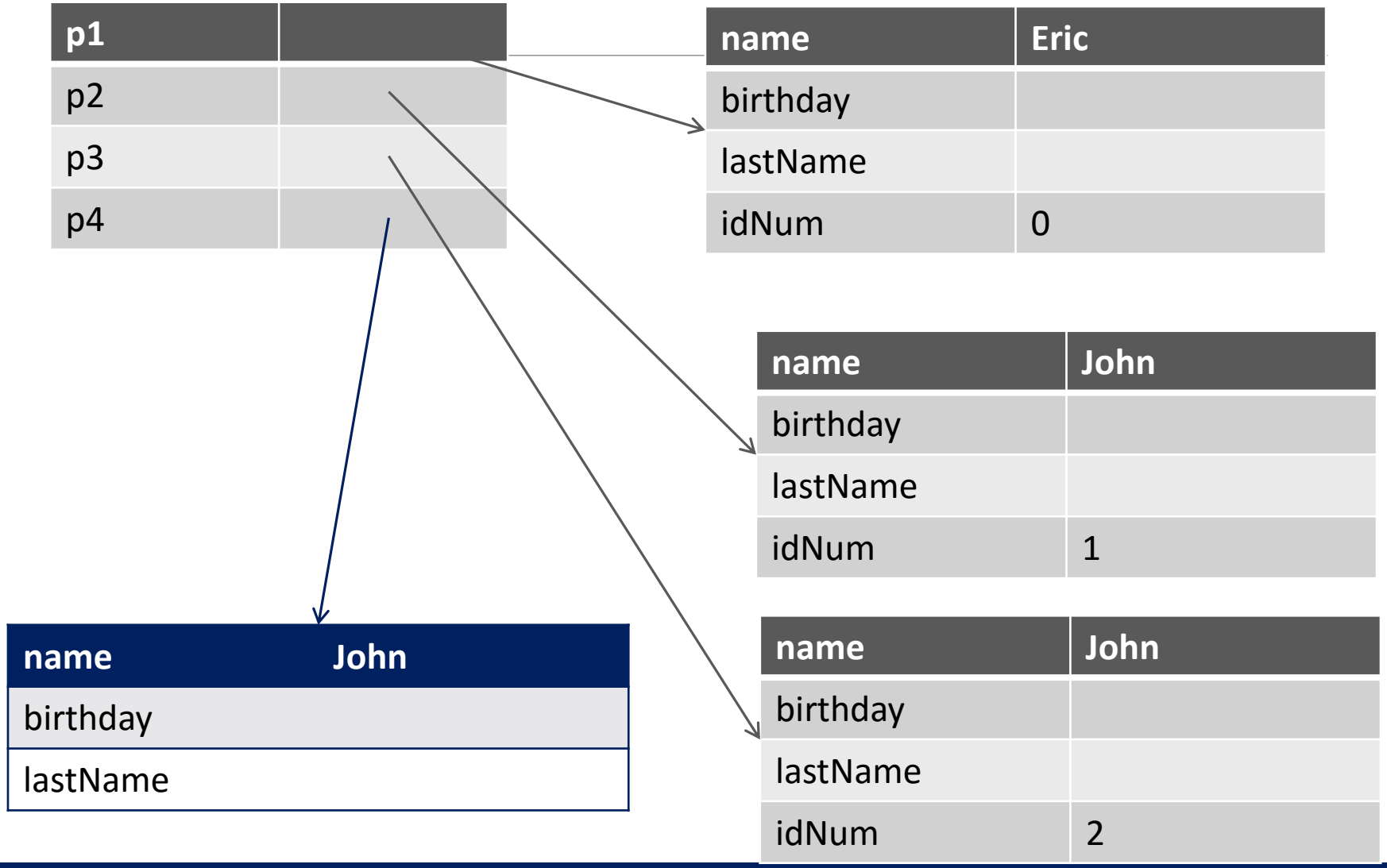
EXAMPLE OF SORTING BY <

```
for e in MITPersonList:
    print(e)
Bill Gates
Drew Houston
Mark Zuckerberg
```

```
MITPersonList.sort()

for e in MITPersonList:
    print(e)
Mark Zuckerberg
Drew Houston
Bill Gates
```

EXAMPLE USING HIERARCHY

```
p1 = MITPerson('Eric')
p2 = MITPerson('John')
p3 = MITPerson('John')
p4 = Person('John')
```


p1
p2
p3
p4
name
Eric
birthday
lastName
idNum
0
name
John
birthday
lastName
idNum
1
name
John
birthday
lastName
idNum
2
name
John
birthday
lastName

TRY TO COMPARE

```
p1 < p2
```

```
True
```

```
p1 < p4
```

```
Attribute Error
```

```
p4 < p1
```

```
False
```

HOW TO COMPARE?

- MITPerson has its own `__lt__` method
- method "shadows" the Person method, meaning that if we compare an MITPerson object, since its environment inherits from the MITPerson class environment, Python will see this version of `__lt__` not the Person version
- thus, `p1 < p2` will be converted into `p1.__lt__(p2)` which applies the method associated with the type of `p1`, or the MITPerson version

WHO INHERITS?

- Why does p4 < p1 work, but p1 < p4 doesn't?
  - p4 < p1 is equivalent to p4.__lt__(p1), which means we use the __lt__ method associated with the type of p4, namely a Person (the one that compares based on name)
  - p1 < p4 is equivalent to p1.__lt__(p4), which means we use the __lt__ method associated with the type of p1, namely an MITPerson (the one that compares based on IDNum) and since p4 is a Person, it does not have an IDNum

USING INHERITANCE

- explore in some detail an example of building an application that organizes info about people
  - Person: name, birthday
    - get last name
    - sort by last name
    - get age
  - MITPerson: Person + ID Number
    - assign ID numbers in sequence
    - get ID number
    - sort by ID number
  - Students: several types, all MITPerson
    - undergraduate student: has class year
    - graduate student

MORE CLASSES IN HIERARCHY

```
class UG(MITPerson):
    def __init__(self, name, classYear):
        MITPerson.__init__(self, name)
        self.year = classYear

    def getClass(self):
        return self.year

    def speak(self, utterance):
        return MITPerson.speak(self, " Dude, " + utterance)

class Grad(MITPerson):
    pass

def isStudent(obj):
    return isinstance(obj,UG) or isinstance(obj,Grad)
```

use the inherited `MITPerson` method to create an instance, which in turn will use the `Person` method

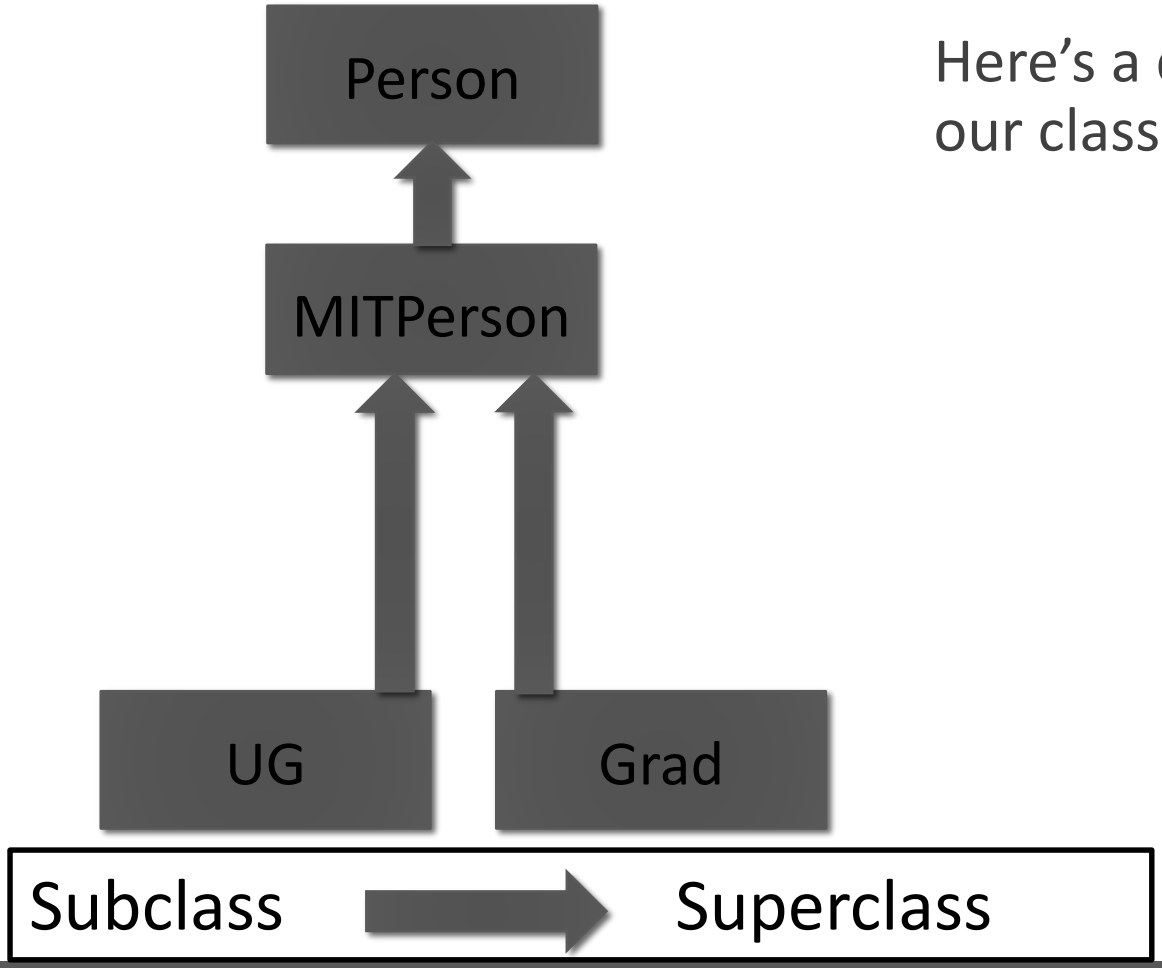
use the inherited `MITPerson` method to speak but with additional words

test for superclass checks for instances of subclasses

EXAMPLE

```
s1 = UG('Matt Damon', 2017)
s2 = UG('Ben Affleck', 2017)
s3 = UG('Lin Manuel Miranda', 2018)
s4 = Grad('Leonardo di Caprio')

print(s1)
print(s1.getClass())
print(s1.speak('where is the quiz?'))
print(s2.speak('I have no clue!'))
```

SUBSTITUTION PRINCIPLE



Here's a diagram showing our class hierarchy

ADDING ANOTHER CLASS

```
class UG(MITPerson):
    def __init__(self, name, classYear):
        MITPerson.__init__(self, name)
        self.year = classYear
    def getClass(self):
        return self.year
    def speak(self, utterance):
        return MITPerson.speak(self, " Dude, " + utterance)

class Grad(MITPerson):
    pass

class TransferStudent(MITPerson):
    pass

def isStudent(obj):
    return isinstance(obj,UG) or isinstance(obj,Grad)
```

now I have to rethink isStudent

CLEANING UP HIERARCHY

```
class Student(MITPerson):
    pass

class UG(Student):
    def __init__(self, name, classYear):
        MITPerson.__init__(self, name)
        self.year = classYear
    def getClass(self):
        return self.year
    def speak(self, utterance):
        return MITPerson.speak(self, " Dude, " + utterance)
class Grad(Student):
    pass
class TransferStudent(Student):
    pass

def isStudent(obj):
    return isinstance(obj,Student)
```

better is to create a superclass that covers all students

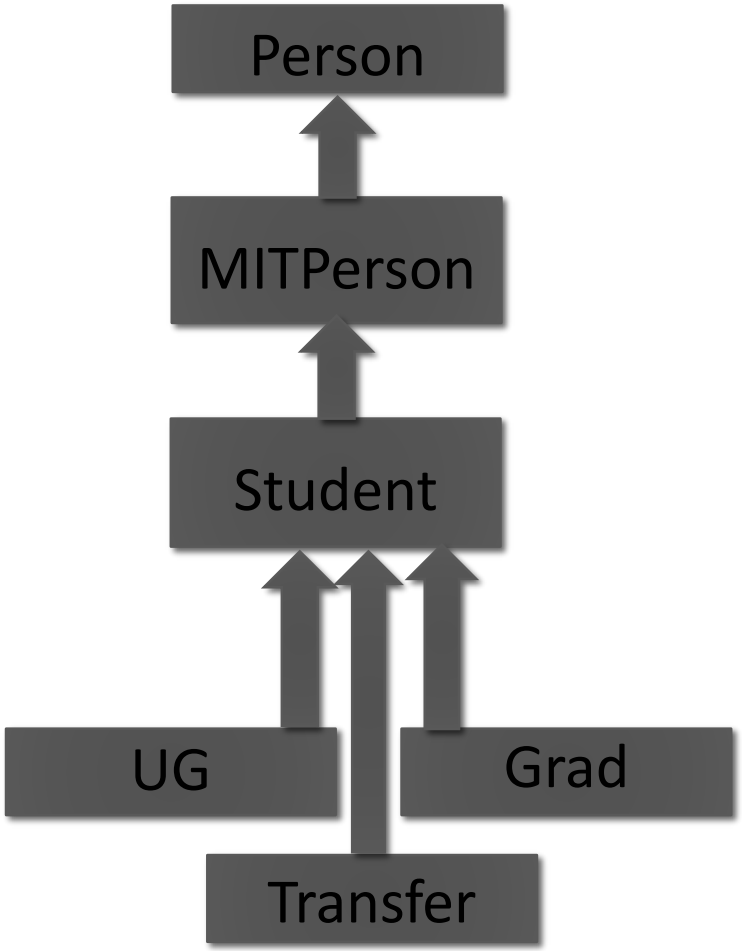
pass is a special keyword, says there is no expression in the body

create a class that captures common behaviors of subclasses; concentrate methods in one place, think about subclasses as a coherent whole

EXAMPLE

```
s1 = UG('Matt Damon', 2017)
s2 = UG('Ben Affleck', 2017)
s3 = UG('Lin Manuel Miranda', 2018)
s4 = Grad('Leonardo di Caprio')
S5 = TransferStudent('Robert deNiro')

print(s1)
print(s1.getClass())
print(s1.speak('where is the quiz?'))
print(s2.speak('I have no clue!'))
```

SUBSTITUTION PRINCIPLE



Here's an updated diagram showing our class hierarchy

Be careful when overriding methods in a subclass!

- **Substitution principle:** important behaviors of superclass should be supported by all subclasses

USING INHERITED METHODS

- add a Professor class of objects
  - also a kind of MITPerson
  - but has different behaviors
- use as an example to see how one can leverage methods from other classes in the hierarchy

A NEW CLASS OF OBJECT

```
class Professor(MITPerson):
    def __init__(self, name, department):
        MITPerson.__init__(self, name)
        self.department = department

    def speak(self, utterance):
        new = 'In course ' + self.department + ' we say '
        return MITPerson.speak(self, new + utterance)

    def lecture(self, topic):
        return self.speak('it is obvious that ' + topic)
```

this will shadow MITPerson speak method

note use of MITPerson speak method

note use of own speak method

EXAMPLE USAGE

```
print(m1.speak('hi there'))
Gates says: hi there

print(s1.speak('hi there'))
Damon says:  Dude, hi there

print(faculty.speak('hi there'))
Arrogant says: In course six we say hi there

print(faculty.lecture('hi there'))
Arrogant says: In course six we say it is obvious that
hi there
```

uses `MITPerson` `speak` method

uses `UG` `speak` method, which uses `MITPerson` method

uses `Professor` `speak` method, which uses `MITPerson` method

`Lecture` method uses `Professor` `speak` method

MODULARITY HELPS

- by isolating methods in classes, makes it easier to change behaviors
  - can change base behavior of MITPerson class, which will be inherited by all other subclasses of MITPerson
  - or can change behavior of a lower class in hierarchy

- change `MITPERSON's speak` method to

```
def speak(self, utterance):
    return (self.name + " says: " + utterance)
```

EXAMPLE USAGE

```
print(m1.speak('hi there'))
Mark Zuckerberg says: hi there

print(s1.speak('hi there'))
Matt Damon says:  Dude, hi there

print(faculty.speak('hi there'))
Doctor Arrogant says: In course six we say hi there

print(faculty.lecture('hi there'))
Doctor Arrogant says: In course six we say it is
obvious that hi there
```

changes to `MITPerson speak` method affect all classes use as base method for their own speak methods

MODULARITY HELPS

- by isolating methods in classes, makes it easier to change behaviors
  - can change base behavior of MITPerson class, which will be inherited by all other subclasses
  - or can change behavior of a lower class in hierarchy
- change `MITPERSON's speak` method to

```
def speak(self, utterance):
    return (self.name + " says: " + utterance)
```

- change `UG's speak` method to

```
def speak(self, utterance):
    return MITPerson.speak(self, " Yo Bro, " + utterance)
```

EXAMPLE USAGE

```
print(m1.speak('hi there'))
Mark Zuckerberg says: hi there

print(s1.speak('hi there'))
Matt Damon says:  Yo Bro, hi there

print(faculty.speak('hi there'))
Doctor Arrogant says: In course six we say hi there

print(faculty.lecture('hi there'))
Doctor Arrogant says: In course six we say it is
obvious that hi there
```

changes to `UG speak` method only affect classes that use it

EXAMPLE CLASS: GRADEBOOK

- create class that includes instances of other classes within it
- concept:
  - build a data structure that can hold grades for students
  - gather together data and procedures for dealing with them in a single structure, so that users can manipulate without having to know internal details

EXAMPLE: GRADEBOOK

```
class Grades(object):
    """A mapping from students to a list of grades"""
    def __init__(self):
        """Create empty grade book"""
        self.students = []  # list of Student objects
        self.grades = {}    # maps idNum -> list of grades
        self.isSorted = True # true if self.students is sorted

    def addStudent(self, student):
        """Assumes: student is of type Student
           Add student to the grade book"""
        if student in self.students:
            raise ValueError('Duplicate student')
        self.students.append(student)
        self.grades[student.getIdNum()] = []
        self.isSorted = False
```

EXAMPLE: GRADEBOOK

```
class Grades(object):

    def addGrade(self, student, grade):
        """Assumes: grade is a float
           Add grade to the list of grades for student"""
        try:
            self.grades[student.getIdNum()].append(grade)
        except KeyError:
            raise ValueError('Student not in grade book')

    def getGrades(self, student):
        """Return a list of grades for student"""
        try:    # return copy of student's grades
            return self.grades[student.getIdNum()][:]
        except KeyError:
            raise ValueError('Student not in grade book')
```

index into dict using IdNum; returns a list of grades

add to list; mutate existing list

index into dict using IdNum;

return a copy

EXAMPLE: GRADEBOOK

```
class Grades(object):


    def allStudents(self):
        """Return a list of the students in the grade book"""
        if not self.isSorted:
            self.students.sort()
            self.isSorted = True
        return self.students[:]
        #return copy of list of students
```

return a copy

USE GRADEBOOK WITHOUT KNOWING INTERNAL DETAILS

```
def gradeReport(course):
    """Assumes: course is of type grades"""
    report = []
    for s in course.allStudents():
        tot = 0.0
        numGrades = 0
        for g in course.getGrades(s):
            tot += g
            numGrades += 1
        try:
            average = tot/numGrades
            report.append(str(s) + '\'s mean grade is '
                          + str(average))
        except ZeroDivisionError:
            report.append(str(s) + ' has no grades')
    return '\n'.join(report)
```

use method to get data; preserves information hiding

return as string, with return between each student

SETTING UP AN EXAMPLE

```
ug1 = UG('Matt Damon', 2018)
ug2 = UG('Ben Affleck', 2019)
ug3 = UG('Drew Houston', 2017)
ug4 = UG('Mark Zuckerberg', 2017)
g1 = Grad('Bill Gates')
g2 = Grad('Steve Wozniak')

six00 = Grades()
six00.addStudent(g1)
six00.addStudent(ug2)
six00.addStudent(ug1)
six00.addStudent(g2)
six00.addStudent(ug4)
six00.addStudent(ug3)
```

RUNNING AN EXAMPLE

```
six00.addGrade(g1, 100)
six00.addGrade(g2, 25)
six00.addGrade(ug1, 95)
six00.addGrade(ug2, 85)
six00.addGrade(ug3, 75)

print(gradeReport(six00))
```

```
Matt Damon's mean grade is 95.0
Ben Affleck's mean grade is 85.0
Drew Houston's mean grade is 75.0
Mark Zuckerberg has no grades
Bill Gates's mean grade is 100.0
Steve Wozniak's mean grade is 25.0
```

```
six00.addGrade(g1, 90)
six00.addGrade(g2, 45)
six00.addGrade(ug1, 80)
six00.addGrade(ug2, 75)

print(gradeReport(six00))
```

```
Matt Damon's mean grade is 87.5
Ben Affleck's mean grade is 80.0
Drew Houston's mean grade is 75.0
Mark Zuckerberg has no grades
Bill Gates's mean grade is 95.0
Steve Wozniak's mean grade is 35.0
```

USING EXAMPLE

- could list all students using

```
for s in six00.allStudents():
    print(s)
```

- prints out the list of student names sorted by idNum
- why not just do

```
for s in six00.students:
    print(s)
```

- violates the data hiding aspect of an object, and exposes internal representation
  - If I were to change how I want to represent a grade book, I should only need to change the methods within that object, not external procedures that use it

COMMENTS ON EXAMPLE

- nicely separates collection of data from use of data
- access is through methods associated with the gradebook object
- but current version is inefficient – to get a list of all students, I create a copy of the internal list
  - let's me manipulate without change the internal structure
  - but expensive in a MOOC with 100,000 students

GENERATORS

- any procedure or method with `yield` statement called a **generator**

```
def genTest():
    yield 1
    yield 2
```

- `genTest()` ➔ `<generator object genTest at 0x201b878>`
- generators have a next() method which starts/resumes execution of the procedure. Inside of generator:
  - `yield` suspends execution and returns a value
  - returning from a generator raises a `StopIteration` exception

USING A GENERATOR

```
In [1]: foo = genTest()

In [2]: foo.__next__()
1
```

Execution will proceed in body of foo, until reaches first yield statement; then returns value associated with that statement

```
>>> foo.__next__()
2
```

Execution will resume in body of foo at point where stopped, until reaches next yield statement; then returns value associated with that statement

```
>>> foo.__next__()
Results in a StopIteration exception
```

USING GENERATORS

- can use a generator inside a looping structure, as it will continue until it gets a StopIteration exception:

```
>>> for n in genTest():
        print(n)


1
2
>>>
```

FANCIER EXAMPLE

```
def genFib():
    fibn_1 = 1 #fib(n-1)
    fibn_2 = 0 #fib(n-2)
    while True:
        # fib(n) = fib(n-1) + fib(n-2)
        next = fibn_1 + fibn_2
        yield next
        fibn_2 = fibn_1
        fibn_1 = next
```

FANCIER EXAMPLE

- evaluating

```
fib = genFib()
```

creates a generator object

- calling

```
fib.__next__()
```

will return the first Fibonacci number, and subsequence calls will generate each number in sequence

- evaluating

```
for n in genFib():
    print(n)
```

will produce all of the Fibonacci numbers (an infinite sequence)

WHY GENERATORS?

- generator separates the concept of computing a very long sequence of objects, from the actual process of computing them explicitly
- allows one to generate each new objects as needed as part of another computation (rather than computing a very long sequence, only to throw most of it away while you do something on an element, then repeating the process)
- have already seen this idea in `range`

FIX TO GRADES CLASS

```
def allStudents(self):
    if not self.isSorted:
        self.students.sort()
        self.isSorted = True
    return self.students[:]
    #return copy of list of students
```

Before

```
def allStudents(self):
    if not self.isSorted:
        self.students.sort()
        self.isSorted = True
    for s in self.students:
        yield s
```

After