Using Inheritance

- Let's build an application that organizes info about people!
  - 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

    # other methods

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

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

    # other methods
```

How `plist.sort()` works

- Python uses the timsort algorithm for sorting sequences – a highly-optimized combination of merge and insertion sorts that has very good average case performance
- The only knowledge needed about the objects being sorted is the result of a “less than” comparison between two objects
- Python interpreter translates `obj1 < obj2` into a method call on `obj1` → `obj1.__lt__(obj2)`
- To enable sort operations on instances of a class, implement the `__lt__` special method

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 __lt__(self, other):
        """return True if self's ame 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

    # other methods

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

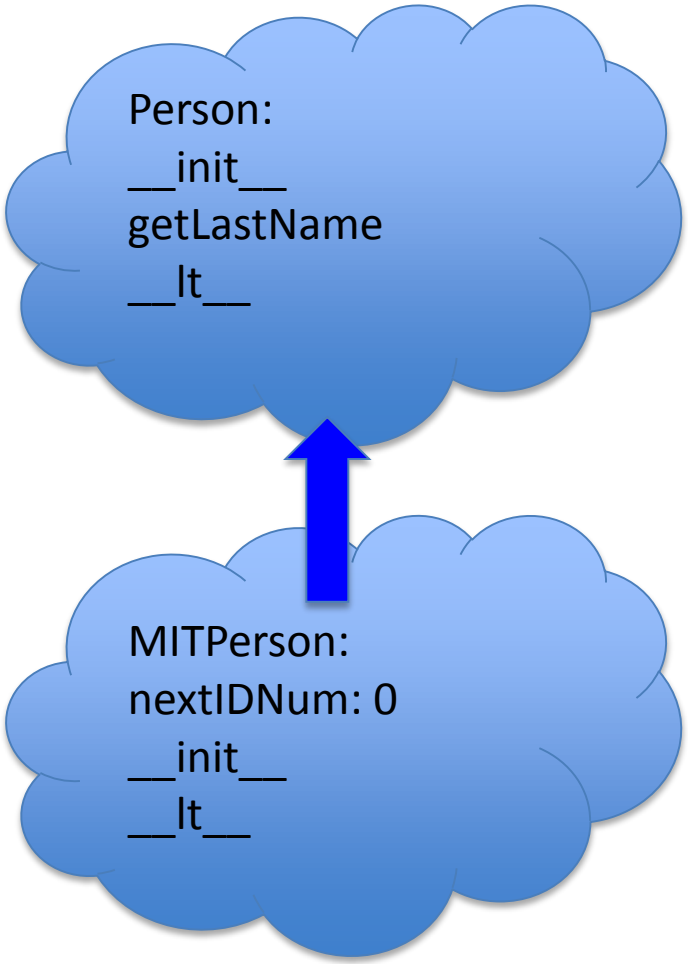
Using Inheritance

- Let's build 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

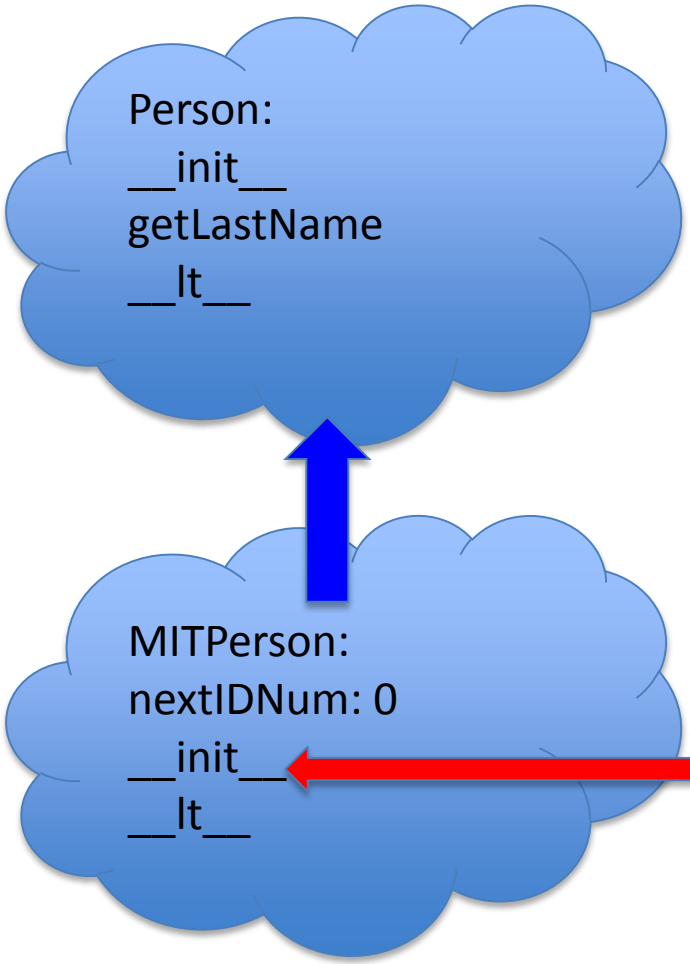
Building inheritance

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

    def __init__(self, name):
        Person.__init__(self, name) # initialize Person
attributes
        # new MITPerson attribute: a unique ID number
        self.idNum = MITPerson.nextIdNum
        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
```

Visualizing the hierarchy

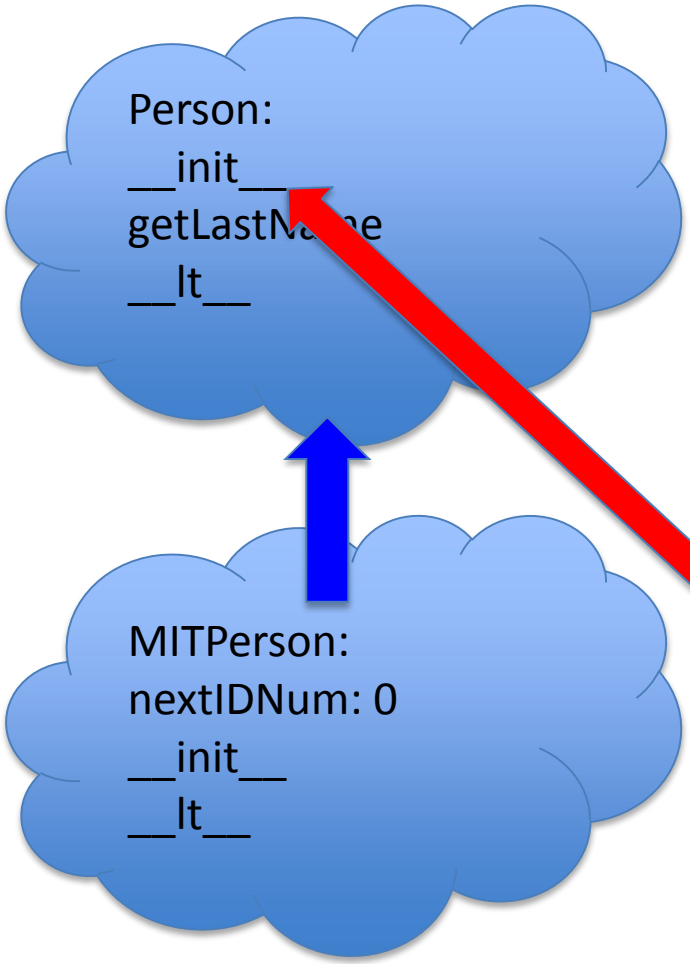


Visualizing the hierarchy



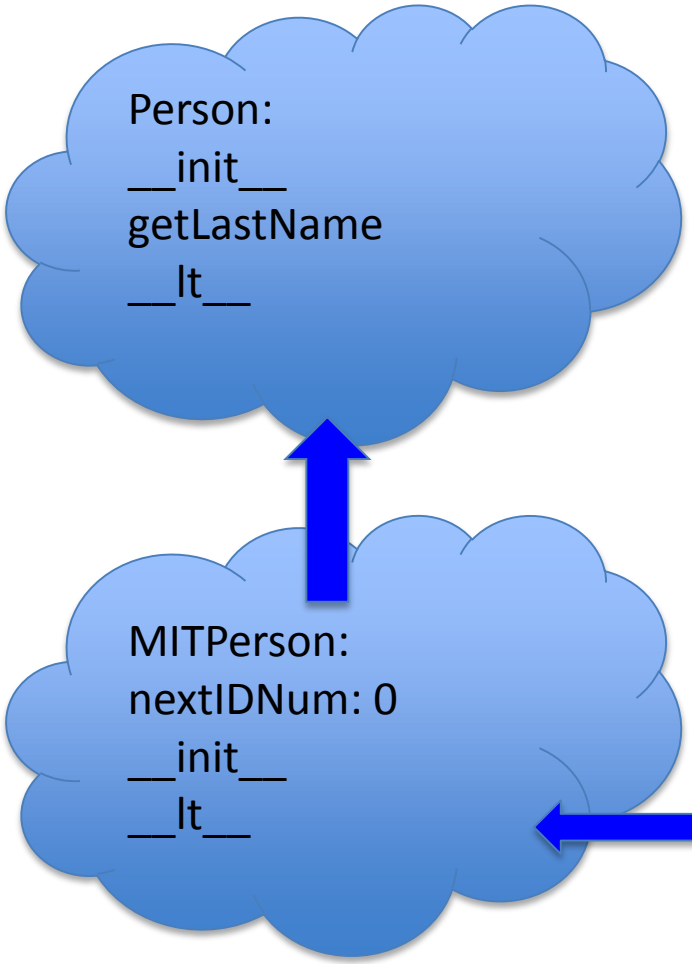
Calling MITPerson will use this __init__ procedure (because that is the one visible in MITPerson's environment)

Visualizing the hierarchy



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

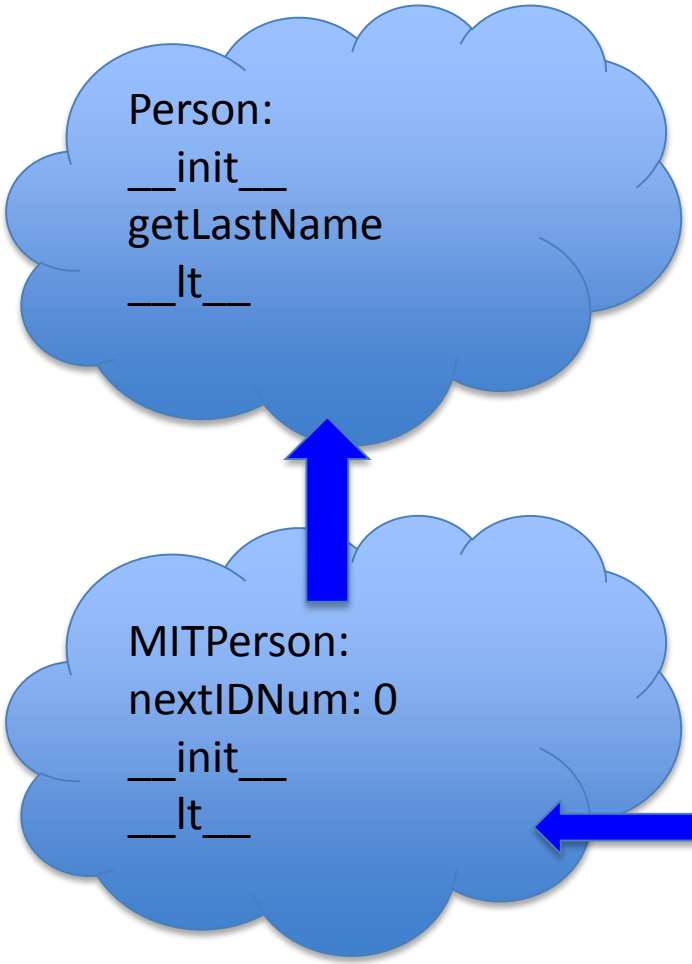
Visualizing the hierarchy



name	
birthday	
lastName	

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

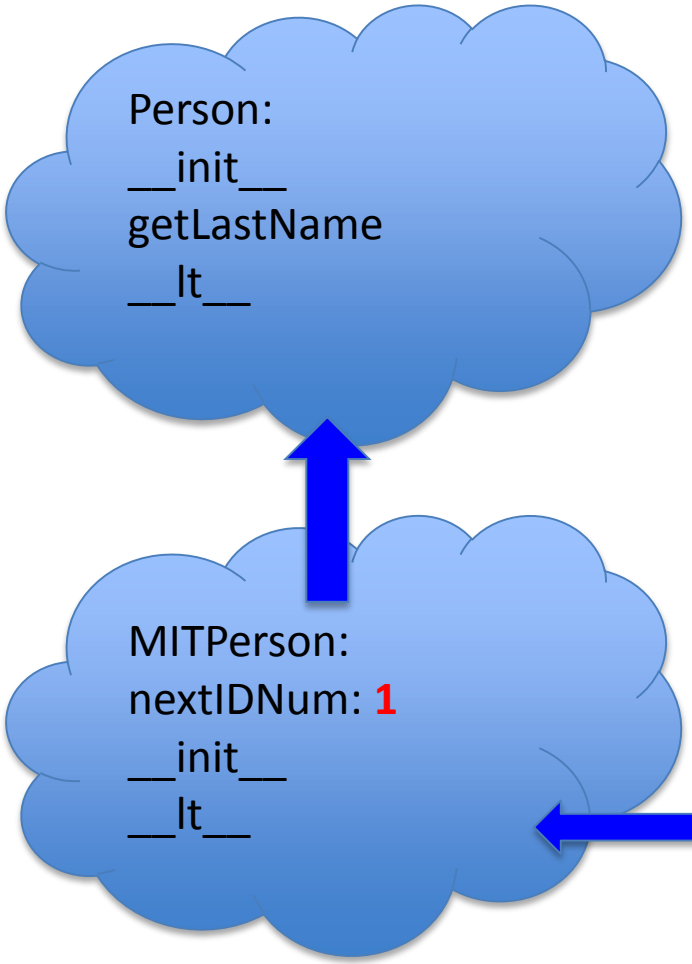
Visualizing the hierarchy



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



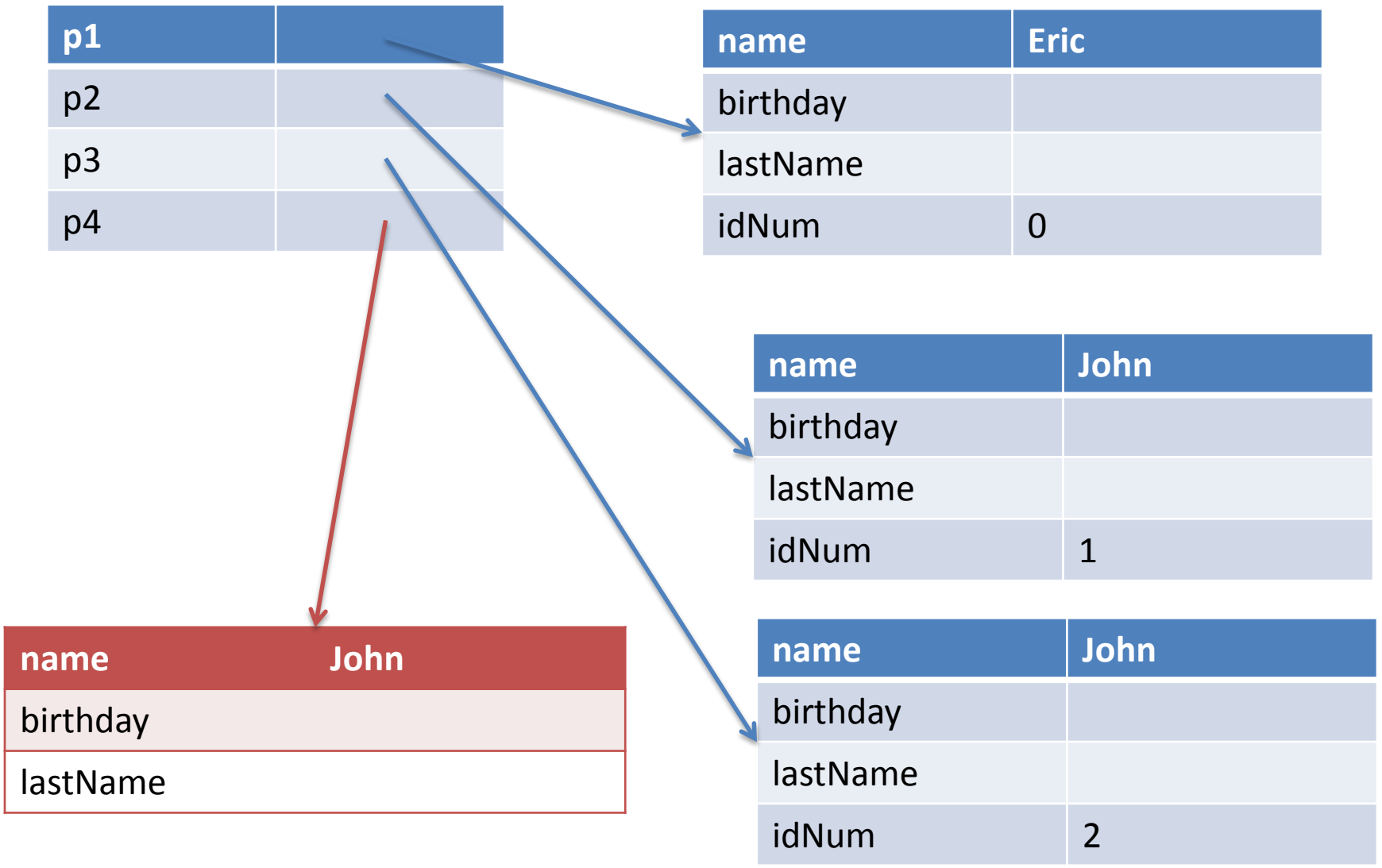
The rest of the original __init__ code calls
self.idNum = MITPerson.nextIdNum
which looks up nextIdNum in the MITPerson environment, and creates a binding in self (i.e. the instance)
And then updates nextIdNum in the MITPerson environment

name	
birthday	
lastName	
idNum	0

Examples of using this hierarchy

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

Visualizing the hierarchy



Suppose we want to compare things

- Note that MITPerson has its own `__lt__` method
- This 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

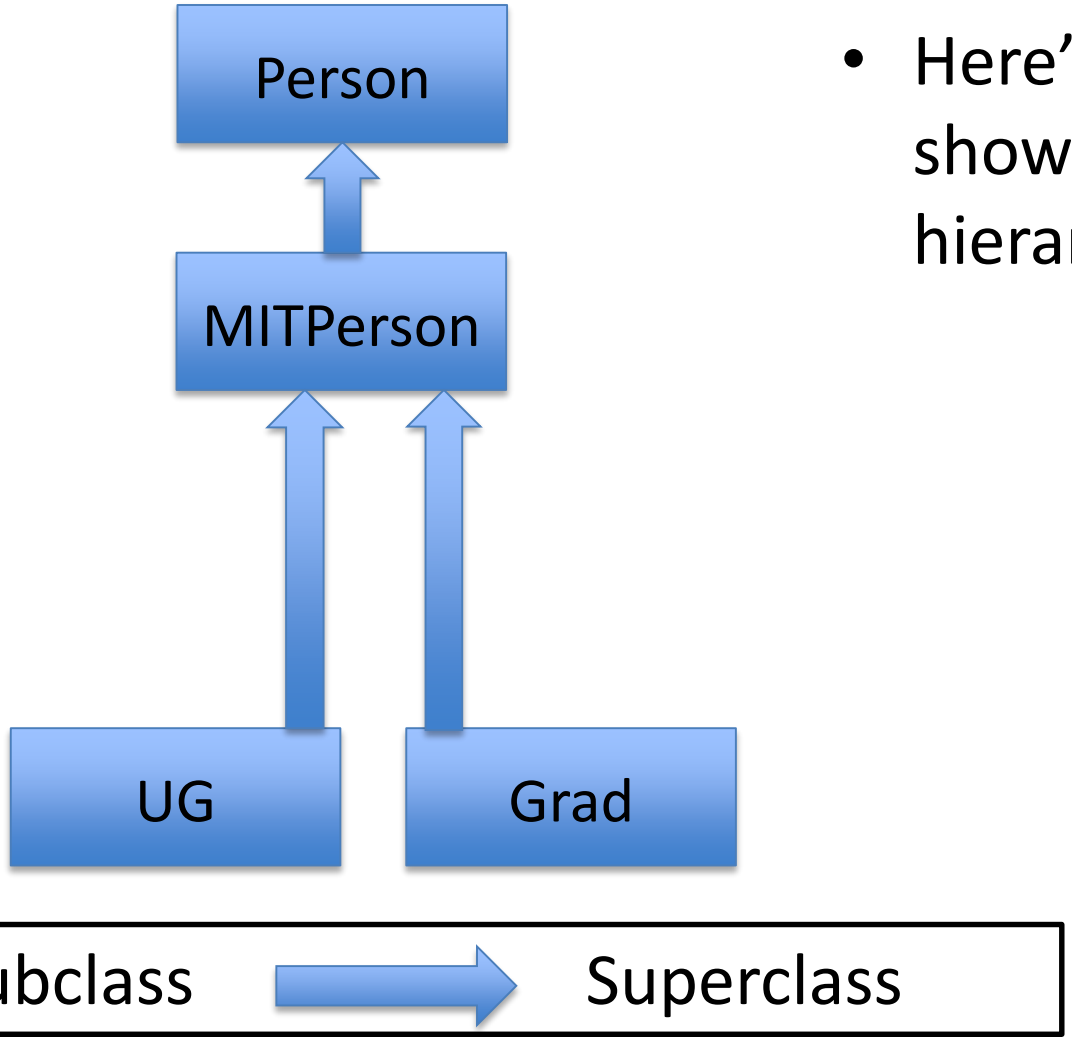
Using Inheritance

- Let's build 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 for the hierarchy

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

    def getClass(self):
        return self.year

class Grad(MITPerson):
    pass

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

Class Hierarchy & Substitution Principle



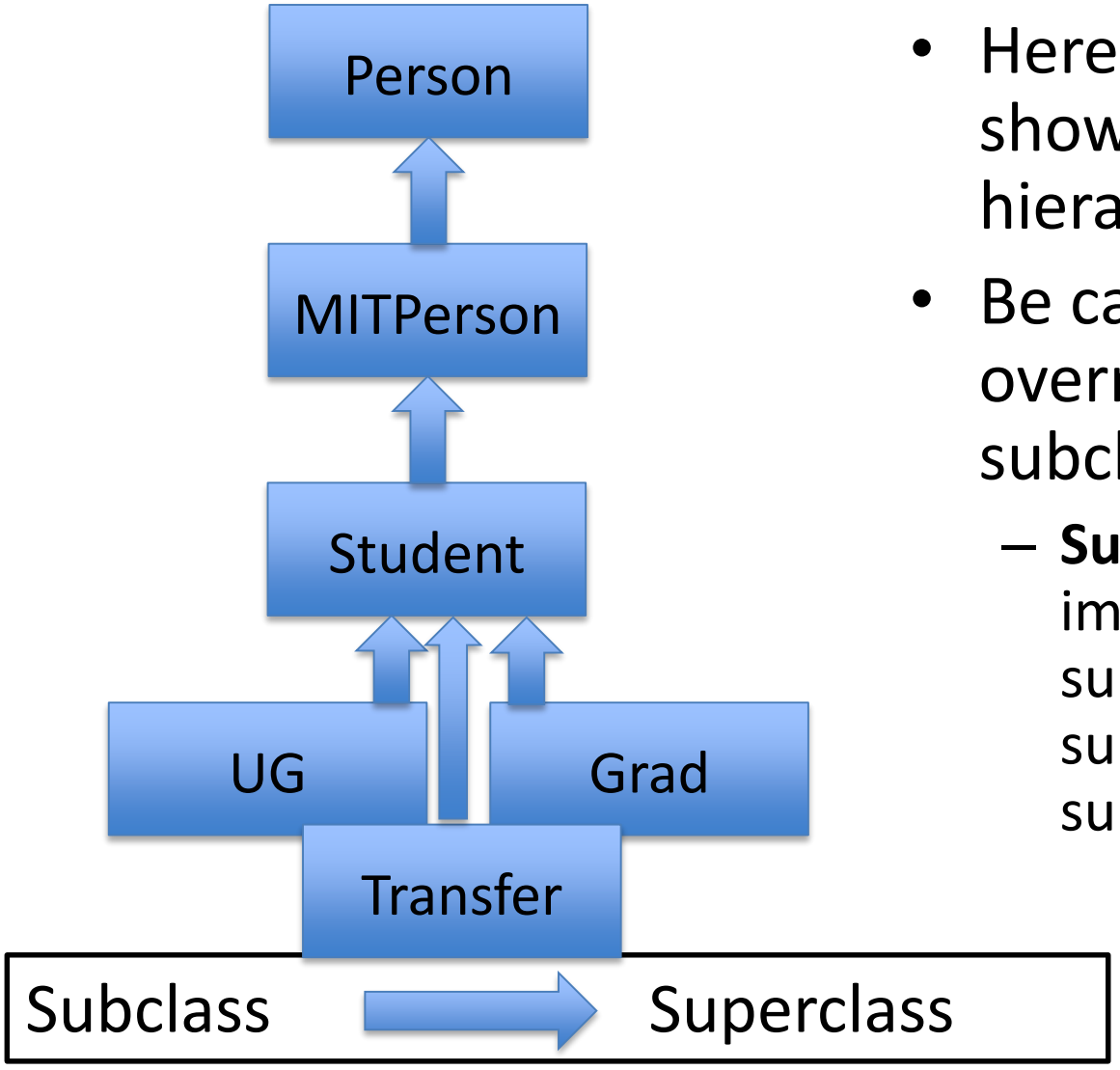
- Here's a diagram showing our class hierarchy

Cleaning up the hierarchy

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

    def getClass(self):
        return self.year

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

Class Hierarchy & Substitution Principle



- Here's a 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

Cleaning up the 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

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

In general, creating a class in the hierarchy that captures common behaviors of subclasses allows us to concentrate methods in a single place, and lets us think about subclasses as a coherent whole

Example class: A 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: A 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: A 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’)
```

Example: A 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
```

Using a gradebook without knowing internal details

```
def gradeReport(course):
    """Assumes: course if 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)
```

Setting up an example

```
ug1 = UG('Jane Doe', 2014)
ug2 = UG('John Doe', 2015)
ug3 = UG('David Henry', 2003)
g1 = Grad('John Henry')
g2 = Grad('George Steinbrenner')

six00 = Grades()
six00.addStudent(g1)
six00.addStudent(ug2)
six00.addStudent(ug1)
six00.addStudent(g2)

for s in six00.allStudents():
    six00.addGrade(s, 75)
six00.addGrade(g1, 100)
six00.addGrade(g2, 25)

six00.addStudent(ug3)
```

Using this example

- I could list all students using

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

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

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

- This 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 the 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 a `yield` statement is 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

```
>>> foo = genTest()

>>> 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

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

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

Using generators

- We 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
>>>
```

A 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
```

A 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?

- A 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)

Fix to Grades class

Before

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

After

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