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
```

object is underlying basic Python class that has associated with it standard methods

```
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
```

create instances of class

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

getter

print out appropriately

BUILDING A CLASS (MORE)

```
import datetime
```

importing a library, a module that exists in Python supplies classes for manipulating dates and times

```
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
```

give a date of birth to a person by month, day, and year, datetime will automatically convert that into an internal representation (should not care about what the representation is)

get number of days since the person was born

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
```

defining my version of a built in method called less than

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
```

in the case of two people with the same last name, it has sorted it by the full name, whereas everywhere else it's simply sorted by last name.

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)
```

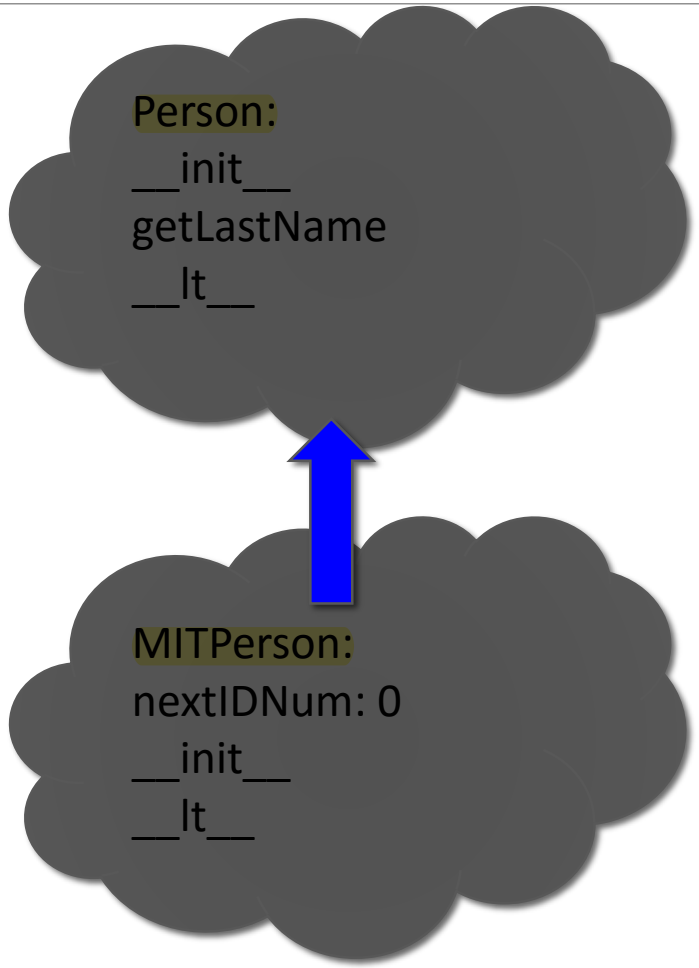
class attribute. definition of a data object that's built into the class belongs to class, not to instance

call the person-class initialization method to initialize the same kinds of things

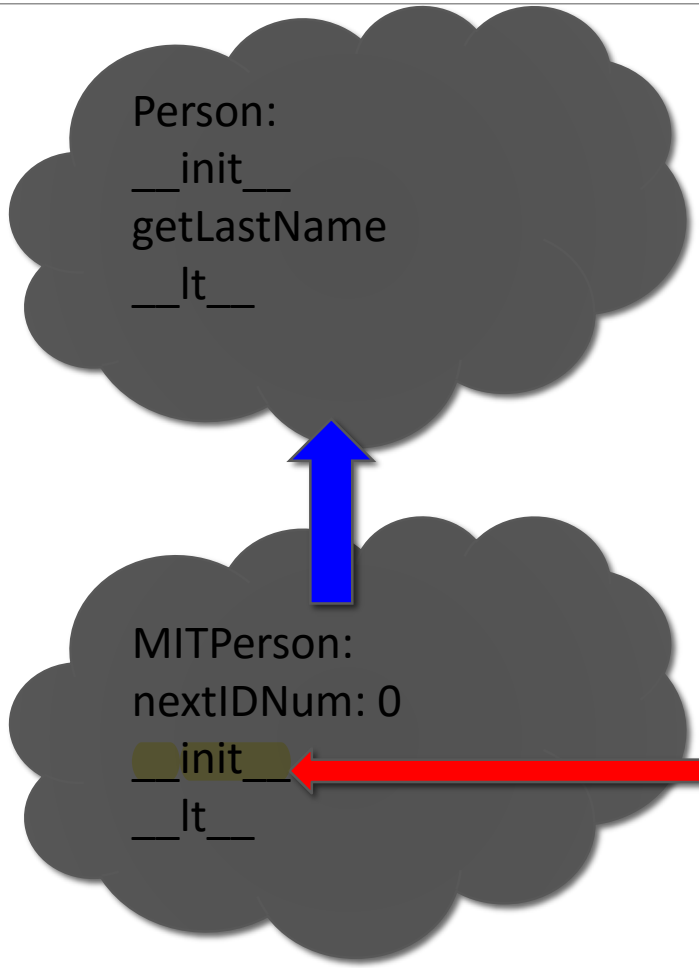
assign to this instance an ID number

increment global class data attribute by 1, so that the next person I create is going to have the next number in the sequence

VISUALIZING THE HIERARCHY

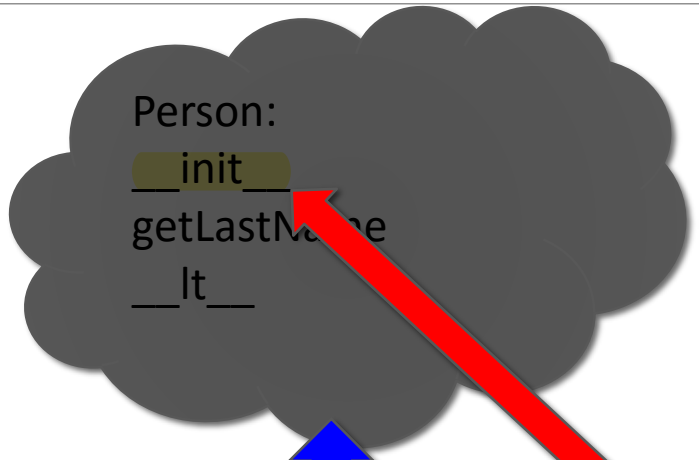


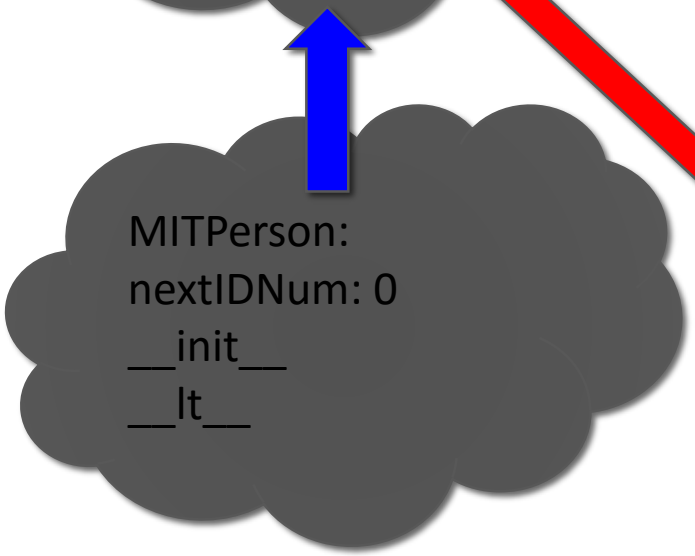
VISUALIZING THE HIERARCHY



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

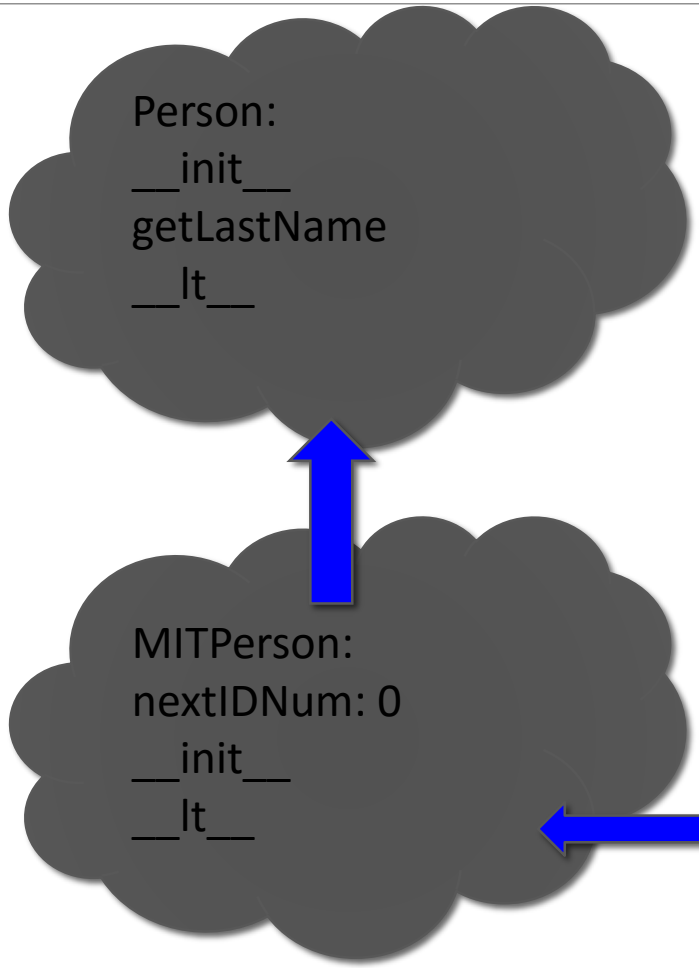
VISUALIZING THE HIERARCHY





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

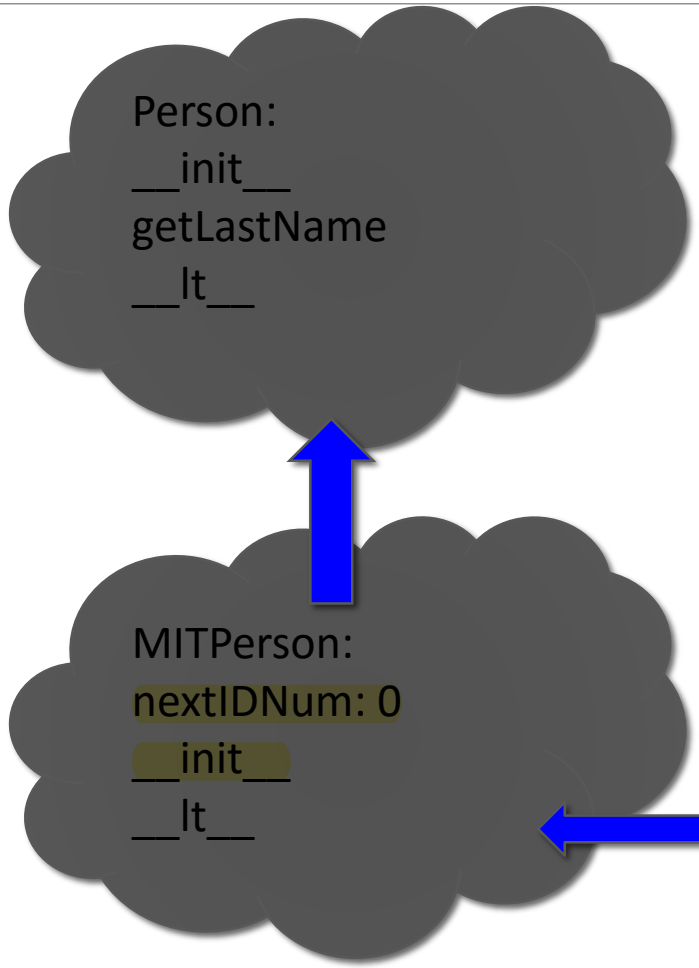
VISUALIZING THE HIERARCHY



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 associated with Person

name	
birthday	
lastName	

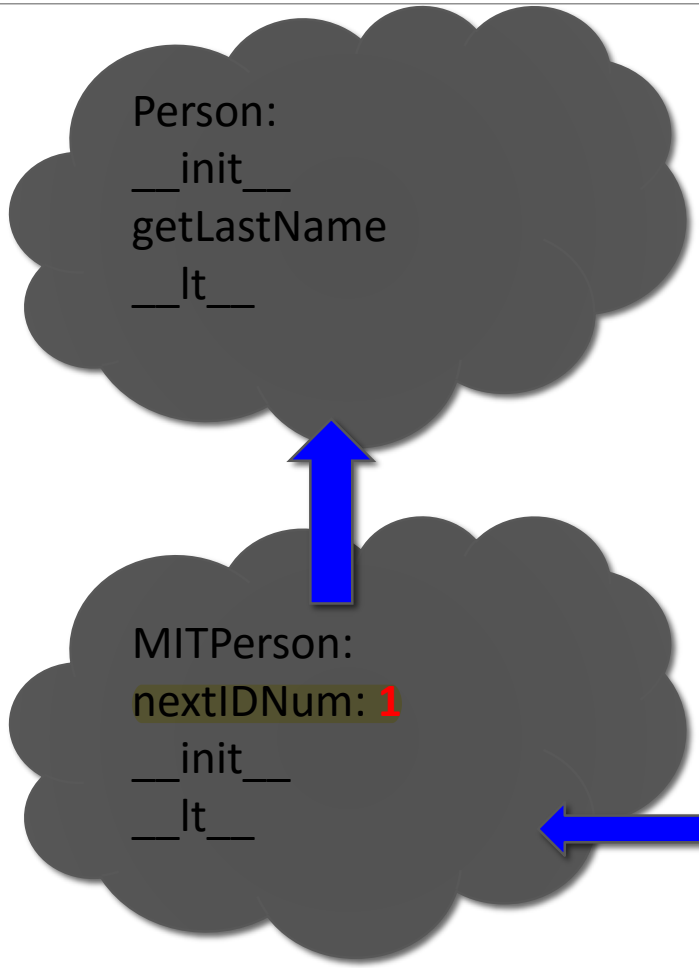
VISUALIZING THE HIERARCHY



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

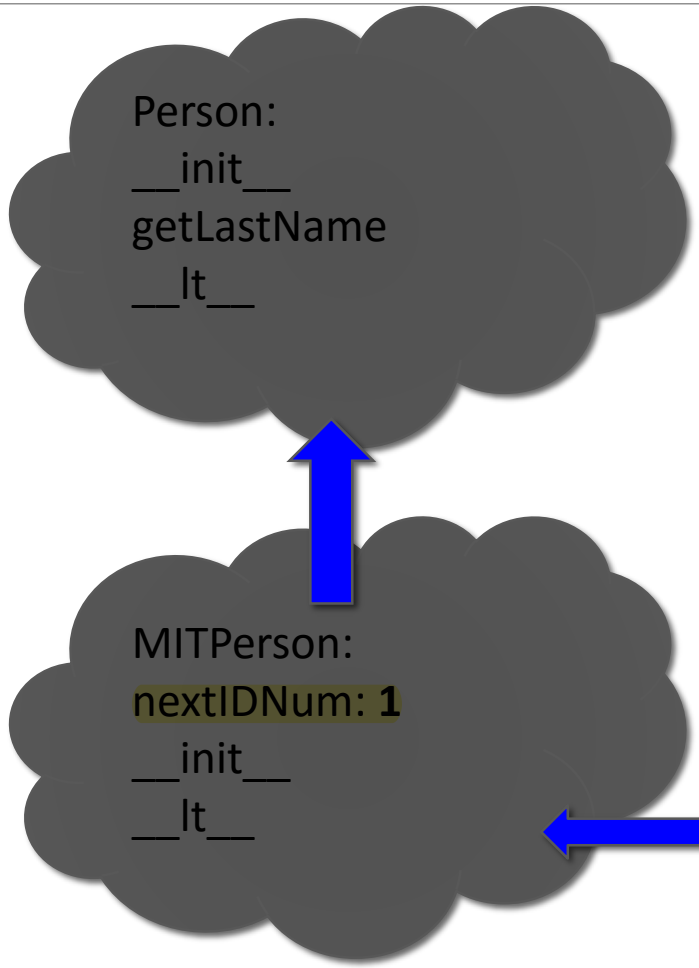


The rest of the original __init__ code calls
self.idNum = MITPerson.nextIdNum
which looks up nextIdNum in the MITPerson environment, and creates a binding in self (i.e. the instance)

And then updates nextIdNum in the MITPerson environment

name	
birthday	
lastName	
idNum	0

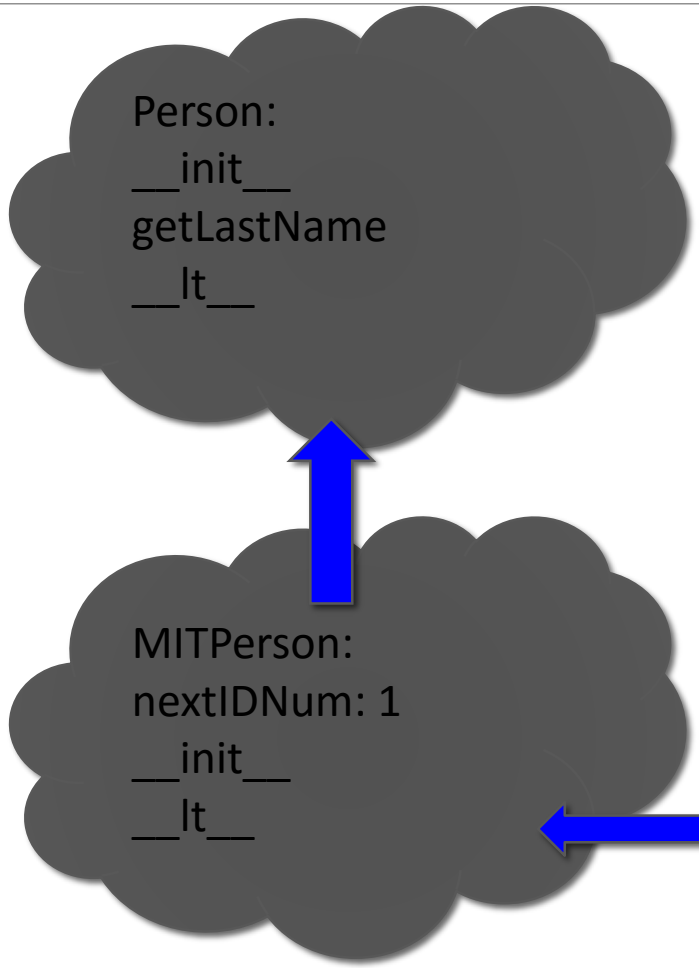
VISUALIZING THE HIERARCHY



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	

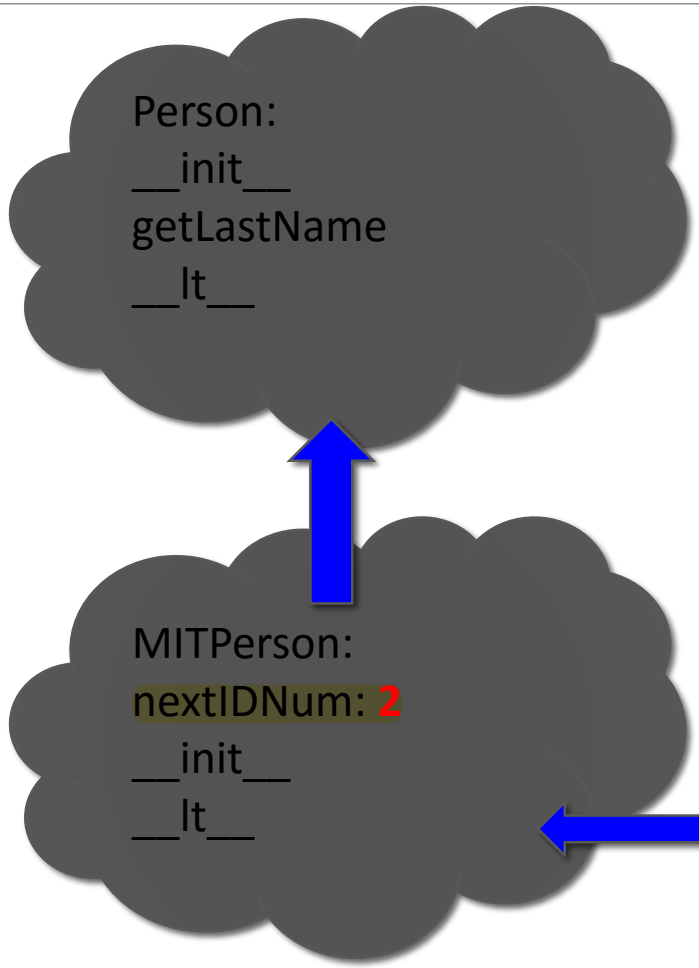
VISUALIZING THE HIERARCHY



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
```

sorting by ID number

EXAMPLE USING HIERARCHY

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

ID number can be important because the names may not be distinctive

bindings for P1, P2, and P3,
two instances of MIT people,
each with a unique ID number

p1	
p2	
p3	
p4	

name	Eric
birthday	
lastName	
idNum	0

name	John
birthday	
lastName	
idNum	1

P4 points to a different kind of object.
It's just a person-- no ID number
associated with them

name	John
birthday	
lastName	

name	John
birthday	
lastName	
idNum	2

TRY TO COMPARE

```
p1 < p2
```

```
True
```

sorting here should be done on the basis of ID number.
I created P1 before I created P2

```
p1 < p4
```

```
Attribute Error
```

Well, this is highlighting how a class or an instance of a class gets a method.
So why is it that in one case it works, and in another case it doesn't?

```
p4 < p1
```

```
False
```

HOW TO COMPARE?

- MITPerson has its own `__lt__` method compare by ID
- 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 compare by name
- thus, `p1 < p2` (self, other) will be converted into `p1.__lt__(p2)` (self, other) which applies the method associated with the type of `p1`, or the MITPerson version

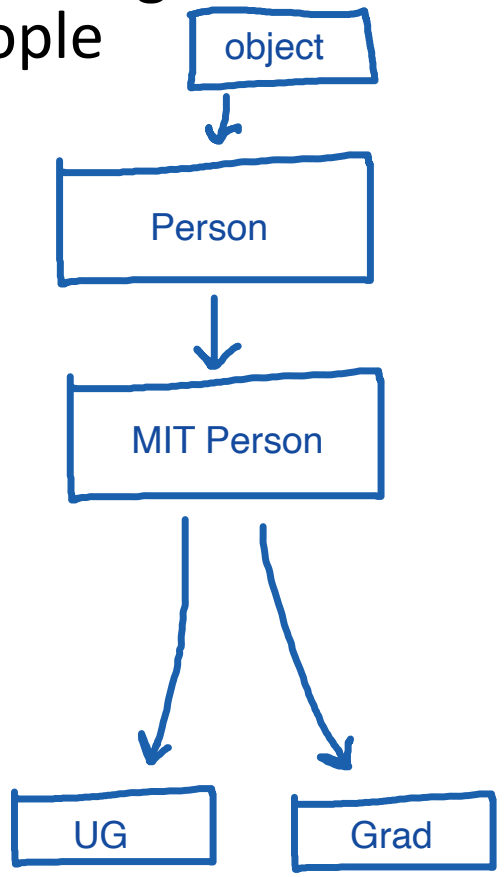
WHO INHERITS?

- 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

specific object/instance calling the method, in this case, self is the first one in that comparison, is the one that's going to define the type and therefore the method that we want to use

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
```

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

is somebody a student by simply saying, is this an instance of an undergrad or is this an instance of a grad student?

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

test for superclass checks for instances of subclasses

EXAMPLE

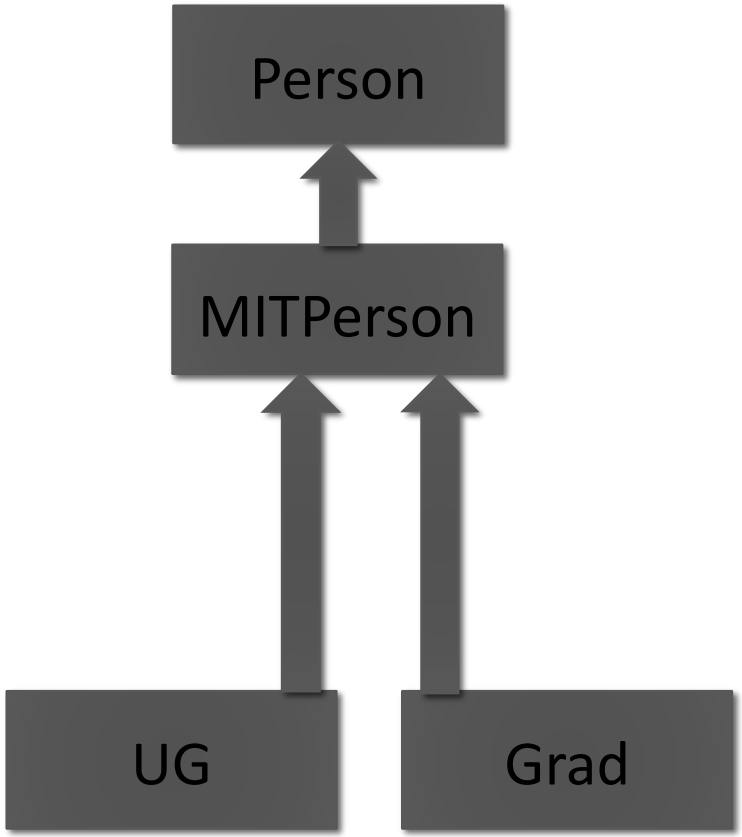
```
s1 = UG('Matt Damon', 2017)
s2 = UG('Ben Affleck', 2017)
s3 = UG('Lin Manuel Miranda', 2018)
s4 = Grad('Leonardo di Caprio')

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

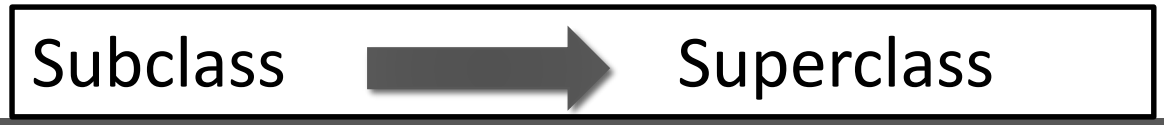
__str__ method of Person

undergraduate speak method uses the underlying MITPerson speak method, but adds something to it

SUBSTITUTION PRINCIPLE



Here's a diagram showing our class hierarchy



inherit methods up the hierarchy as we need them

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
```

One way to change it would be simply to add another clause to that expression that says, is this a transfer student?

```
class TransferStudent(MITPerson):
    pass
```

now I have to rethink isStudent

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

If I were to add in other kinds of students here, it's simply a matter of adding the class definition, ensuring that it inherits from student, and I don't have to change the hierarchy or the methods associated with them

CLEANING UP HIERARCHY

```
class Student(MITPerson):
    pass
```

It inherits from MITPerson. It's a Student

```
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)
```

each of these subclasses inherit from Student

better is to create a superclass that covers all students

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

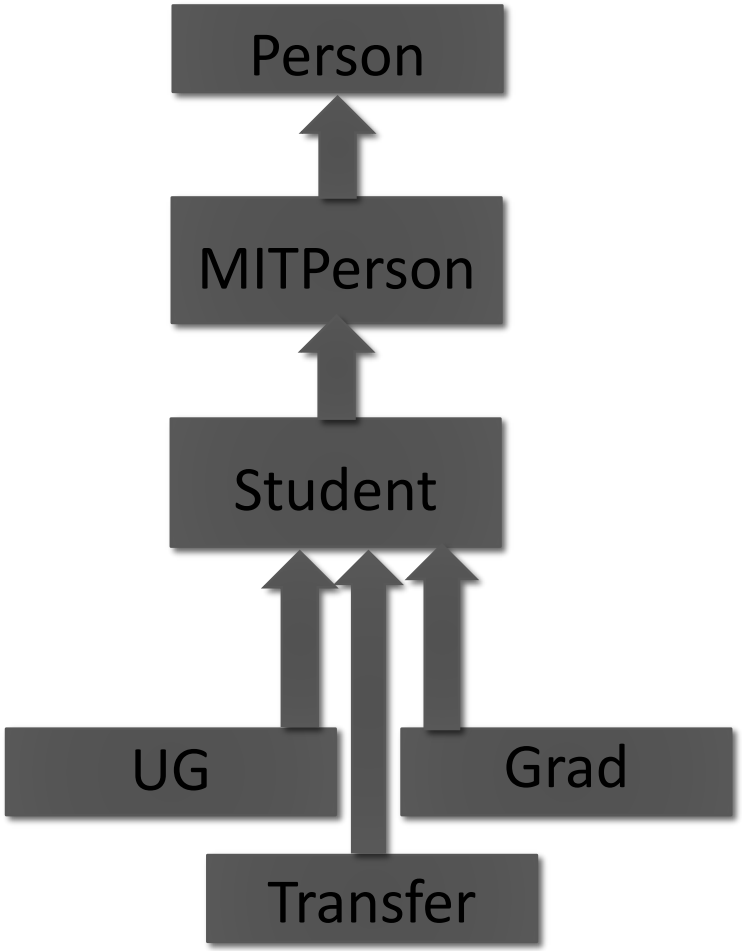
create a class that captures common behaviors of subclasses; concentrate methods in one place, think about subclasses as a coherent whole

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

but use that modularity to isolate changes to methods when I want to

A NEW CLASS OF OBJECT

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

use the inherited MIT Person initialization method to set it up, but I'll add in one more attribute

this will shadow MITPerson speak method

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

note use of MITPerson speak method

this method will shadow the MIT Person speak() method for professors, although it will use that inherited speak() method

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

note use of own speak method

use the base speak() method associated with the Professor but add other things in as we do it

Professor speak() method

each class has a speak() method. And in the hierarchy, we start with the class instance or the class of which we have an instance, use that speak() method. We may inherit from higher up in the hierarchy. But we have all of those pieces nicely contained to the instances, so speaking different kinds of objects do in a different way

EXAMPLE USAGE

MIT Person

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

Student

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

```
print(faculty.speak('hi there'))
```

Professor

```
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)
```

return the name rather than
just the last name as part of that speak() method

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
```

MIT Person case, Undergraduate is lower in the hierarchy, so in fact it would not have a change there

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

Faculty Member, it's in another place in the hierarchy, and it does not use the Student's method, it uses the MIT Person's method

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

***When I add a new student to the class, because it's getting appended to the end of the list, I'm going to change that self. issortedflag to false.

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
```

key value

because nothing is in list and dicionary yet, it 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  ***
```

student is an instance from another class

add a student to my class, first of all, check to make sure that that student is not alreadv in class

self.students is a list. simply append directly onto it, mutating that list

self.grades is a dictionary (going to be accessed by ID number). I create a new entry with key (look up instance student, and call method getIdNum on that instance to get out students ID) and value (empty list for grades)

self.grades is a dictionary, accesed through student ID with key (look up instance student, and call method getIdNum on that instance to get out students ID) and associated value (list of grades for that student)

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')
```

index into dict using IdNum; returns a list of grades

add to list; mutate existing list

handle if no entry in the dictionary for the student

```
    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;

return a copy of value (list of grades)

do things on that copy without destroying the original grades

handle if no entry in the dictionary for the student

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
```

if that flag for this instance is sorted, is false, I'm going to go ahead and sort

return a copy

so I don't accidentally destroy the underlying list stored inside of the instance

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)
```

returns copy of the list of all the students stored in course (of class Grades)

loop over that student list

use method to get data; preserves information hiding

run for every grade in the grades associated with that student

returns copy of the list of grades for a student stored in course (of class Grades)

return as string, with return between each student

list

handle if student hast no grades

because this is a big list of strings, I'm going to return it where I just break this up by inserting a carriage return in between each one of those strings

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')
```

create some instances of undergraduate and graduate students

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

create instance of Grades
-> a gradesbook called six00

add the students into that grade book
(in arbitrary number)

this class has within it a large number of other class instances. That's perfect, because if I change the behavior of those class instances, it won't change the behavior of the grade book

now dealing with this database, this structure of a grades book, without having to know the internal details

RUNNING AN EXAMPLE

student instance

add first grade

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

print(gradeReport(six00))
```

add additional grades

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

print(gradeReport(six00))
```

carriage return to print each entry on a different line

```
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
```

```
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

So in that case, this would still work, but this would not if I were to change that internal representation

- 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 be consistent of only using those methods to get out the internal data
- 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

I don't want to always be generating a list 100,000 long before I do something with it

iterables like lists are handy because you can read them as much as you wish, but you store all the values in memory and this is not always what you want when you have a lot of values

Generators are iterators, a kind of iterable you can only iterate over once. Generators do not store all the values in memory, they generate the values on the fly -> handy when you know your function will return a huge set of values that you will only need to read once (more memory efficient and faster)

GENERATORS

- any procedure or method with `yield` statement called a **generator** yield is a keyword that is used like return, except the function will return a generator

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

call to function returns an object that is of type generator, a generator instance

- `genTest()` ➔ `<generator object genTest at 0x201b878>` The next time I call for the next() method, it will go until I get to the next yield() method, in which case it will stop/suspend execution and return a value
- 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 keep doing that until we run out of yield(), then raise this exception

USING A GENERATOR

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

creates a generator

foo is now an object/instance of type generator

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

call method next() on generator object

It has gone until it found the first yield statement, it's returned that value, and it's stopped operation.

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

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

call method next() on generator object

It has gone until it found the next yield statement, it's returned that value, and it's stopped operation.

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

continue until the generator is considered empty, which happens when the function runs without hitting yield

USING GENERATORS

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

genTest(), that generator, is going to create something that will execute until I get to the first yield point and then return a value

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

inside the for loop, it's executing until it gets a value returned, yielding, and printing it out

```
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
```

no way to exit out of this while loop.
So it will simply, if I were to call it,
generate all of the Fibonacci numbers in turn

yield next which will
halt execution until I ask it to continue

when it returns, I'm going to move up one step.
What was the previous Fibonacci number
is now the second previous one.
Next is the previous Fibonacci number.
And I'm going to go back around the loop

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

runs until it gets to the next stopping point, yields up a value. set up in that while loopto be ready to compute the next one when I need it

- evaluating

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

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

Everything that can be done with generator can be done with a function

If we were to use a generator to iterate over a million numbers, how many numbers do we need to store in memory at once? -> need to store 2 numbers – one for the current value, and one for the max value (think about range())

WHY GENERATORS?

- generator separates the concept of computing a very long sequence of objects, from the actual process of computing them explicitly

  way of creating things as needed

- 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`

  better efficiency without changing the way we think about doing the computation: I can program as if that entire sequence is available to me. The computer is going to generate it as I need it

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

for all students in that list, just yield them up.
So one at a time, as I ask for them,
it will give me the next one without generating
the entire list as I go through them.