6.00.1.x - Week 5 - Object Oriented Programming

October 25, 2020

1 Video 1: Object Oriented Programming

- Every **object** has:
  - A type.
  - An internal data representation (primitive or composite).
  - A set of procedures for interaction with the object.
- Each instance is a particular type of object.
  - 1234 is instance of an int.
  - a = 'hello= is an instance of a string.

- Everything in python is an **object** and has a type.
- Objects are a **data abstraction** that caputre:
  - Internal representation through data attributes.
  - Interface for interacting with object through methods (procedures), defines behaviors but hides implementation.
- Can **create new instances of objects**.
- Can also **destroy objects**.

Built in data objects:

- Lists.
- Tuples.
- Strings.

We want to explore ability to create our own data object types.

How are lists represented internally? They are a linked list of cells. But it doesn't matters. What matters is how to **manipulate** them. How do you do that?

- L[i], L[i:j], L[i,j,k]

- len(), min(), max(), etc.

Internal representation should be **private**. Correct behavior may be compromised if you manipulate it directly - instead you use **defined interfaces** to do so.

How to create your own objects with classes?

- Distinction: creating a class and using an instance of said class.
  - A list is a class. L1 = [1,2,3] is an instance of said class.
- Creating the class involves:
  - Defining the class name.
  - Defining class attributes.
- Using the class involves:
  - Creating new instances of objects.
  - Doing operations on the instances.

Advantages of OOP

- Bundle data into packages together with rpocedures that work on them through well-defined interfaces.
- Divide-and-conquer development.
  - Implement and test behavior of each class separately.
  - Increased modularity reduces complexity.
- Classes make it easy to reuse code.
  - Many Python modules define new classes.
  - Each class has a separate environment (no collision on function names).
  - Inheritance allows subclasses to redefine or extend a selected subset of a superclass' behavior.

2 Video 2: Class Instances

Use the class keyword to define a new type:

class Coordinate(object): (define attributes here)

- Similar to def, indent code to indicate which statements are part of the **class definition**
- 'class' shows class definition.
- 'Coordinate' is the class name.
- The word 'object' means that Coordinate is a Python object and inherits all its attributes.
  - Coordinate is a subclass of object.
  - Object is a superclass of Coordinate.

What are attributes?

- Data and procedures that 'belong' to the class.
  - Data attributes.
  - Procedural attributes (methods). Think of methods as functions that only work with this class.

How to create an instance of an object?

- Use a special method called **init** to initialize **some** data attributes:

[27]:
```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

- 'self' is a parameter to refer to an instance of the class.
- x,y is the data that initializes a 'Coordinate' object.

Example of actually creating an instance of a class:

[28]:
```
# Creating a new object of type Coordinate and passing in 3 and 4 to the␣
 ↪__init__ method.
c = Coordinate(3,4)
# Note that argument for 'self' is automatically supplied by Python.
origin = Coordinate(0,0)
print(c.x)
print(origin.x)
```

```
3
0
```

Think of c as pointing to a frame:

- Within the scope of that frame we bound values to data attribute variables.
- c.x is interpreted as getting the value of c (a frame) and then looking up the value associated with x within that frame (thus the specific value for this instance).

3 Video 3: Methods

What is a method?

- Procedural attribute, like a function that works only with this class.
- Python always passes the actual object as the first argument, convention is to use self as the name of the first argument of all methods.
- The "." operator is used to access any attribute:
  - A data attribute of an object.
  - A method of an object.

Example:

[29]:
```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def distance(self, other):
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5

c = Coordinate(3,4)
origin = Coordinate(0,0)
print(c.distance(origin))
```

```
5.0
```

How to use a method from the Coordinate class?

- c: object on which to call method.
- distance: name of method.
- origin: parameters not including self.

In this case, c.distance inherits the *distance* from the class definition, and automatically uses c as the first argument.

Another way:

Coordinate.distance gets the value of Coordinate, then looks up the value associated with distance (a procedure), then invokes it (which requires two arguments).

[30]:
```
# Name of class - Name of method - Both parameters, c and origin
print(Coordinate.distance(c,origin))
```

```
5.0
```

How to print a representation of an object?

- Uninformative print representation by default.
- Define a **str** method for a class.

[31]:
```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def distance(self, other):
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5
    def __str__(self):
        return "<" + str(self.x) + "," + str(self.y) + ">"

c = Coordinate(3,4)
# This prints it in a new way
print(c)
```

```
<3,4>
```

How to check if something is an instance of a class, in this case Coordinate?

[32]:
```
print(isinstance(c,Coordinate))
```

```
True
```

Special operators

Full list: https://docs.python.org/3/reference/datamodel.html#basic-customization

- Like print, can override these to work with your class.
- Define them with double underscores before/after:
  - **add**
  - **sub**
  - **eq**
  - **lt**
  - **str**
  - and many others.

4 Video 4: Classes Examples

Example #1: Fractions

- Internal representation is two ints: numerator, denominator.
- Interface a.k.a. methods a.k.a. how to interact with *Fractions* objects.
  - Print representation.
  - Add, subtract.
  - Convert to a float.

[33]:
```
# Create the class and create the print representation.
class fraction(object):
    def __init__(self,numer,denom):
        self.numer = numer
        self.denom = denom
    def __str__(self):
        return str(self.numer) + ' / ' + str(self.denom)

oneHalf = fraction(1,2)
twoThirds = fraction(2,3)
print(oneHalf)
print(twoThirds)
```

```
1 / 2
2 / 3
```

One additional thing that is very important is being able to access data attributes. It is important to use a method that gives me an attribute of the object, and not call the attribute of the object directly. Here's how it's done:

[34]:
```
class fraction(object):
    # This was already here
    def __init__(self,numer,denom):
        self.numer = numer
        self.denom = denom
    def __str__(self):
        return str(self.numer) + ' / ' + str(self.denom)
    # This is the new part
    def getNumer(self):
        return self.numer
    def getDenom(self):
        return self.denom

oneHalf = fraction(1,2)
print(oneHalf.getNumer())
# Can also invoke it in this way
print('another way')
```

```
fraction.getDenom(oneHalf)
```

```
1
another way
```

[34]:
```
2
```

[35]:
```
# Adding the sum and subtraction

class fraction(object):
    # This was already here
    def __init__(self, numer, denom):
        self.numer = numer
        self.denom = denom
    def __str__(self):
        return str(self.numer) + ' / ' + str(self.denom)
    def getNumer(self):
        return self.numer
    def getDenom(self):
        return self.denom
    # This is new
    def __add__(self, other):
        numerNew = other.getDenom() * self.getNumer() \
                   + other.getNumer() * self.getDenom()
        denomNew = other.getDenom() * self.getDenom()
        return fraction(numerNew, denomNew)
    def __sub__(self, other):
        numerNew = other.getDenom() * self.getNumer() \
                   - other.getNumer() * self.getDenom()
        denomNew = other.getDenom() * self.getDenom()
        return fraction(numerNew, denomNew)
    def convert(self):
        return self.getNumer() / self.getDenom()

oneHalf = fraction(1,2)
twoThirds = fraction(2,3)
new = oneHalf + twoThirds
print(new)
```

```
7 / 6
```

[36]:
```
threeQuarters = fraction(3,4)
secondNew = twoThirds - threeQuarters
print(secondNew)
```

```
-1 / 12
```

[37]:
```
# Now I want to convert to a float.

class fraction(object):
    # This was already here
    def __init__(self, numer, denom):
        self.numer = numer
        self.denom = denom
    def __str__(self):
        return str(self.numer) + ' / ' + str(self.denom)
    def getNumer(self):
        return self.numer
    def getDenom(self):
        return self.denom
    def __add__(self, other):
        numerNew = other.getDenom() * self.getNumer() \
                   + other.getNumer() * self.getDenom()
        denomNew = other.getDenom() * self.getDenom()
        return fraction(numerNew, denomNew)
    def __sub__(self, other):
        numerNew = other.getDenom() * self.getNumer() \
                   - other.getNumer() * self.getDenom()
        denomNew = other.getDenom() * self.getDenom()
        return fraction(numerNew, denomNew)
    # This is new
    def convert(self):
        return self.getNumer() / self.getDenom()

oneHalf = fraction(1,2)
twoThirds = fraction(2,3)
oneHalf.convert()
```

[37]:
```
0.5
```

We are going to create yet another example: **A set of integers**

- Initially the set is empty.
- A particular integer appears only once in a set: representational invariant enforced by the code.

For this I will need:

- Internal data representation: use a list to store the elements of a set.
- Interface:
  - Insert - insert integer e into set if not there.
  - Member - return True if integer e is in set, False else.
  - Remove - remove integer e from set, error if not present.

[38]:
```
class intSet(object):
    """An intSet is a set of integers
    The value is represented by a list of ints, self.vals.
    Each int in the set occurs in self.vals exactly once."""

    def __init__(self):
        """Create an empty set of integers"""
        self.vals = []

    def insert(self, e):
        """Assumes e is an integer and inserts e into self"""
        if not e in self.vals:
            self.vals.append(e)

    def member(self, e):
        """Assumes e is an integer
        Returns True if e is in self, and False otherwise"""
        return e in self.vals

    def remove(self, e):
        """Assumes e is an integer and removes e from self
        Raises ValueError if e is not in self"""
        try:
            self.vals.remove(e)
        except:
            raise ValueError(str(e) + ' not found')

    def __str__(self):
        """Returns a string representation of self"""
        self.vals.sort()
        result = ''
        for e in self.vals:
            result = result + str(e) + ','
        return '{' + result[:-1] + '}'

s = intSet()
print(s)
s.insert(3)
s.insert(4)
s.insert(3)
print(s)
print(s.member(3))
print(s.member(5))
s.insert(6)
print(s)
s.remove(3)
print(s)
```

```
s.remove(3)
```

```
{}
{3,4}
True
False
{3,4,6}
{4,6}
```

```
      ␣
↪---------------------------------------------------------------------------
```

```
       ValueError                                Traceback (most recent call␣
↪last)

       <ipython-input-38-c470139e6128> in remove(self, e)
        23         try:
   ---> 24             self.vals.remove(e)
        25         except:

       ValueError: list.remove(x): x not in list


   During handling of the above exception, another exception occurred:

       ValueError                                Traceback (most recent call␣
↪last)

       <ipython-input-38-c470139e6128> in <module>
        46 s.remove(3)
        47 print(s)
   ---> 48 s.remove(3)

       <ipython-input-38-c470139e6128> in remove(self, e)
        24             self.vals.remove(e)
        25         except:
   ---> 26             raise ValueError(str(e) + ' not found')
        27
        28     def __str__(self):

       ValueError: 3 not found
```

5 Video 5: Why OOP

- Bundle together objects that share common attributes and procedures that operature on those attributes.
- Use **abstraction** to make a distinction between how to implement an object vs how to use the object.
- Build **layers** of object abstractions that inherit behaviors from other classes of objects.
- Create our **own classes of objects** on top of Python's basic classes.

Two kinds of attributes a group of objects have:

Data attributes

- How can you represent your object with data?
- What it is
- For a coordinate: x and y values.
- For an animal: age and name.

Procedural attributes

- What kinds of things can you do with the object?
- What it does?
- For a coordiante: find distance between two.
- For an animal: make a sound.

Let's now build an animal.

[39]:
```
class Animal(object):
    def __init__(self, age):
        # I can define other data attributes even if I don't pass them as␣
↪parameters when creating an instance of this class.
        self.age = age
        self.name = None
    # Getter methods
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name
    # Setter methods
    def set_age(self, newage):
        self.age = newage
    # Very interesting! Default value for one of the parameters.
    def set_name(self, newname=""):
```

```
        self.name = newname
    def __str__(self):
        return "animal:"+str(self.name)+":"+str(self.age)

myAnimal = Animal(3)
print(myAnimal)
```

```
animal:None:3
```

[40]:
```
myAnimal.set_name('foobar')
print(myAnimal)
```

```
animal:foobar:3
```

[41]:
```
# This is how we should access the internal representation.
print('Correct method')
print(myAnimal.get_age())
# This is another way - which we should NOT use.
print('Incorrect method')
print(myAnimal.age)
```

```
Correct method
3
Incorrect method
3
```

One should **always use the getters** instead of accesing the data attributes directly:

- Good style.
- Easy to maintain code.
- Prevents bugs.

Python is not great at information hiding

- Allows you to access data from outside class definition.
- Allows you to write to data from outside class definition.
- Allows you to create data attributes for an instance from outside class definition.

It's **not good style** to do any of these.

6 Video 6: Hierarchies

- Parent class (superclass) - Animal.
- Child class (subclass) - Person/Cat/Rabbit.
  - Inherits all data and behaviors of parent class.
  - Add more info.
  - Add more behavior.
  - Override behavior.

— Example

[42]:
```
class Animal(object):
    def __init__(self, age):
        self.age = age
        self.name = None
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name
    def set_age(self, newage):
        self.age = newage
    def set_name(self, newname=""):
        self.name = newname
    def __str__(self):
        return "animal:"+str(self.name)+":"+str(self.age)

# Inherits all attributes of Animal
# Add new functionality through new methods
# Override the __str__ method to identify an instance of this class as a Cat␣
 ↪and not just any Animal.
class Cat(Animal):
    def speak(self):
        print("meow")
    def __str__(self):
        return "cat:"+str(self.name)+":"+str(self.age)

class Rabbit(Animal):
    def speak(self):
        print("meep")
    def __str__(self):
        return "rabbit:"+str(self.name)+":"+str(self.age)

class Person(Animal):
    def __init__(self, name, age):
        Animal.__init__(self, age)
        Animal.set_name(self, name)
```

```
        self.friends = []
    def get_friends(self):
        return self.friends
    def add_friend(self, fname):
        if fname not in self.friends:
            self.friends.append(fname)
    def speak(self):
        print("hello")
    def age_diff(self, other):
        # alternate way: diff = self.age - other.age
        diff = self.get_age() - other.get_age()
        if self.age > other.age:
            print(self.name, "is", diff, "years older than", other.name)
        else:
            print(self.name, "is", -diff, "years younger than", other.name)
    def __str__(self):
        return "person:"+str(self.name)+":"+str(self.age)
```

[43]:
```
# Examples of using this new information
jelly = Cat(1)
jelly.get_name()
```

[44]:
```
jelly.set_name('JellyBelly')
jelly.get_name()
```

[44]:
```
'JellyBelly'
```

[45]:
```
print(jelly)
```

```
cat:JellyBelly:1
```

[46]:
```
print(Animal.__str__(jelly))
```

```
animal:JellyBelly:1
```

[47]:
```
# Python looks for the definition of speak inside Cat, because jelly is an␣
↪instance of Cat.
jelly.speak()
```

```
meow
```

While a subclass inherits all methods from upper parts of the hierarchy, the superclass can't access the methods from the lower parts of the hierarchy.

Which method to use?

- Subclass can have methods with same name as superclass.

- Subclass can have methods with same name as other subclasses.
- For an instance of a class, look for a method name in current class definition.
- If not found, look for a method name up the hierarchy.
- Use first method up the hierarchy that you found with that method name.

7 Video 7: Class Variables

- Different from an instance variable (which is what we normally use, or have used so far).

Let's see them with an example

[48]:
```
class Animal(object):
    # These are instance variables
    def __init__(self, age):
        self.age = age
        self.name = None
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name
    def set_age(self, newage):
        self.age = newage
    def set_name(self, newname=""):
        self.name = newname
    def __str__(self):
        return "animal:"+str(self.name)+":"+str(self.age)
# New code
class Rabbit(Animal):
    # Class variable - OUTSIDE of the init definition.
    tag = 1
    def __init__(self, age, parent1=None, parent2=None):
        Animal.__init__(self, age)
        self.parent1 = parent1
        self.parent2 = parent2
        self.rid = Rabbit.tag
        # This is the part that changes the class variable over and over.
        Rabbit.tag += 1
    def get_rid(self):
        # Technique to make sure all numbers are the same size.
        return str(self.rid).zfill(3)
    def get_parent1(self):
        return self.parent1
    def get_parent2(self):
        return self.parent2
    def __add__(self, other):
        # returning object of same type as this class
        # This is using the __init__ definition of the Rabbit class. So age,␣
 ↪parent1, parent2.
        return Rabbit(0, self, other)
    def __eq__(self, other):
        parents_same = self.parent1.rid == other.parent1.rid \
                       and self.parent2.rid == other.parent2.rid
        parents_opposite = self.parent2.rid == other.parent1.rid \
```

```
                    and self.parent1.rid == other.parent2.rid
        return parents_same or parents_opposite
```

The class is keeping track of the tag, and every time I create a new instance of a Rabbit object, it'll receive a unique ID, the latest value.

- rid is an instance variable.
- tag is a class variable.

Addition procedure

- Allows for r4 = r1 + r2, where r1 and r2 are Rabbit instances.
- f4 is a new Rabbit instance with age 0.
- r4 has self as one parent, and other as the other parent.
- In **init**, should change to check that parent1 and parent2 are of type Rabbit.

8 Video 8: Building a Class

- Explore in some detail an example of building an application that organizes info about people.

[49]:
```
# Use it to calculate age
import datetime

class Person(object):
    def __init__(self, name):
        """create a person called name"""
        self.name = name
        self.birthday = None
        # Name is a string, so split into a list of strings based on spaces,␣
 ↪then extract last element.
        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

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

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

    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
```

[50]:
```
# example usage

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]

for e in personList:
    print(e)
```

```
Mark Zuckerberg
Drew Houston
Bill Gates
Andrew Gates
Steve Wozniak
```

[51]:
```
personList.sort()
for e in personList:
    print(e)
```

```
Andrew Gates
Bill Gates
Drew Houston
Steve Wozniak
Mark Zuckerberg
```

9 Video 9: Visualizing the Hierarchy

Let's define a subclass of the Person class - an MIT person.

[52]:
```
class MITPerson(Person):
    # Class attribute
    nextIdNum = 0 # next ID number to assign

    def __init__(self, name):
        # Using the person class initialization - no need to reinvent the wheel!
        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

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

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]
print(m3)
print(m1.speak('hi there'))
```

```
Mark Zuckerberg
Gates says: hi there
```

[53]:
```
for e in MITPersonList:
    print(e)
```

```
Bill Gates
Drew Houston
Mark Zuckerberg
```

[54]:
```
# Sorting by ID
personList.sort()
```

```
for e in MITPersonList:
    print(e)
```

```
Bill Gates
Drew Houston
Mark Zuckerberg
```

[55]:
```
# MITPerson have IDs, no problem comparing them.
p1 =  MITPerson('Eric')
p2 = MITPerson('John')
p3 = MITPerson('John')
# Not an MITPerson, does not have an ID number!
p4 = Person('John')

p1<p2
```

[55]:
```
True
```

[56]:
```
p1<p4
```

```
␣
↪---------------------------------------------------------------------------

       AttributeError                            Traceback (most recent call␣
↪last)

       <ipython-input-56-e4b6153c0891> in <module>
   ----> 1 p1<p4


       <ipython-input-52-d8c6494e3657> in __lt__(self, other)
        15     # sorting MIT people uses their ID number, not name!
        16     def __lt__(self, other):
   ---> 17         return self.idNum < other.idNum
        18
        19     def speak(self, utterance):


       AttributeError: 'Person' object has no attribute 'idNum'
```

[ ]:
```
p1>p4
```

Why does p4 < p1 work but p4 > p1 doesn't?

- p4 < p1 is equivalent to p4.___lt___(p1) which means use the **lt** method associated with the

type of p4, namely a Person.

- p1 < p4 is equivalent to p1.__lt__(p4) which means we use the **lt** method associated with the type of p1, namely an MITPerson and since p4 is a Person, it does not have an IDNum.

10 Video 10: Adding another class

We keep creating classes

[57]:
```
class UG(MITPerson):
    def __init__(self, name, classYear):
        # Use the inherited method from the MITPerson class.
        MITPerson.__init__(self, name)
        self.year = classYear

    def getClass(self):
        return self.year

    def speak(self, utterance):
        # Use the same from MITPerson, but adding something!
        return MITPerson.speak(self, " Dude, " + utterance)

class Grad(MITPerson):
    pass

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

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

studentList = [s1, s2, s3, s4]

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

```
Matt Damon
2017
Damon says:  Dude, where is the quiz?
Affleck says:  Dude, I have no clue!
```

Suppose we now want to create another class of students, a transfer student. But in order to modify the isStudent method, I'd need to change some things. So let's look at this possibility.

[58]:
```
# Create a superclass that covers all students
class Student(MITPerson):
    # Pass is a special keyword
    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)

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')

studentList = [s1, s2, s3, s4, s5]

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

```
Matt Damon
2017
Damon says:  Dude, where is the quiz?
Affleck says:  Dude, I have no clue!
```

Substitution principle: Important behaviors of superclass should be supported by all subclasses.

In this case, Student was created in order to make all the subclasses (UG, grad and transfer) support the behaviors from the main Student class.

11 Video 11: Using Inherited Methods

This time we create yet another class.

[59]:
```
class Professor(MITPerson):
    def __init__(self, name, department):
        MITPerson.__init__(self, name)
        self.department = department

    def speak(self, utterance):
        # This will shadow MITPerson speak method
        newUtterance = 'In course ' + self.department + ' we say '
        return MITPerson.speak(self, newUtterance + utterance)

    def lecture(self, topic):
        # Uses own speak method, not MITPerson's
        return self.speak('it is obvious that ' + topic)

faculty = Professor('Doctor Arrogant', 'six')
```

[60]:
```
print(s1.speak('I have no idea'))
```

```
Damon says:  Dude, I have no idea
```

[61]:
```
print(m1.speak('Hey there'))
```

```
Gates says: Hey there
```

[62]:
```
# Uses Professor speak method, which uses MITPerson method
print(faculty.speak('hi there'))
```

```
Arrogant says: In course six we say hi there
```

[63]:
```
# Uses Professor speak method
print(faculty.lecture('hi there'))
```

```
Arrogant says: In course six we say it is obvious that hi there
```

Let's now change the original MITPerson class.

[64]:
```
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

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

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)

# Testing changes
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')


studentList = [s1, s2, s3, s4, s5]
print(s1)
print(s1.getClass())
print(s1.speak('where is the quiz?'))
print(s2.speak('I have no clue!'))
print(faculty.lecture('hi there'))
```

Matt Damon
2017
Matt Damon says: Dude, where is the quiz?
Ben Affleck says: Dude, I have no clue!
Doctor Arrogant says: In course six we say it is obvious that hi there

12 Video 12: Gradebook Example

Idea: create a class that includes instances of other classes within it.

[65]:
```
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

    def addGrade(self, student, grade):
        """Assumes: grade is a float
           Add grade to the list of grades for student"""
        try:
            # Grades is a dictionary. Keys = IDs, Values = list of grades.
            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')

    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

    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)

# Loading the data
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)

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

13 Video 13: Generators

An issue is raised: What if I don't (or don't want to) generate a copy of a huge list every single time I want to operate with it?

The answer is **generators**.

Any procedure or method with a 'yield' statement inside it is called a generator. 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.

[66]:
```
# Example
def genTest():
    yield 1
    yield 2

foo = genTest()
```

[67]:
```
foo.__next__()
```

[67]:
```
1
```

[68]:
```
foo.__next__()
```

[68]:
```
2
```

[69]:
```
foo.__next__()
```

```
␣
↪---------------------------------------------------------------------------

       StopIteration                             Traceback (most recent call␣
↪last)

       <ipython-input-69-5d994c17f9ca> in <module>
   ----> 1 foo.__next__()


       StopIteration:
```

Execution will proceed in body of foo, until it reaches a yield statement; then returns value associated with the statement.

How to use this structure?

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

[70]:
```
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
```

[71]:
```
fib = genFib()
fib
```

[71]:
```
<generator object genFib at 0x000001356EA7E1C8>
```

[72]:
```
fib.__next__()
```

[72]:
```
1
```

[73]:
```
fib.__next__()
```

[73]:
```
2
```

[74]:
```
fib.__next__()
```

[74]:
```
3
```

[75]:
```
fib.__next__()
```

[75]:
```
5
```

[76]:
```
fib.__next__()
```

[76]:
```
8
```

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 object 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.