Object Oriented Programming

Objects

- Python supports many different kinds of data
  `1234  3.14159  "Hello"  [1, 5, 7, 11, 13]`
  `{"RJ": "Rajasthan", "IN": "India"}`
- each is an instance of an object, and 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 an instance of an `int`
  - `a = "hello"`
    a is an instance of a string

Object Oriented Programming

- everything in Python is an object and has a type
- objects are a data abstraction that capture:
  - 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 destroy objects
  - explicitly using `del` or just "forget" about them
  - Python system will reclaim destroyed or inaccessible objects – called "garbage collection"

Standard data Objects

- some object types built in to Python
  - lists – `[1, 2, 3, 4]`
  - tuples – `(1, 2, 3, 4)`
  - strings – `'abcd'`
- want to explore ability to create our own data object types

Example: [1, 2, 3, 4]

- [1,2,3,4] is of type list
- how are lists represented internally? linked list of cells

- how to manipulate lists?
  - `L[i], L[i:j], L[i,j,k], +`
  - `len(), min(), max(), del(L[i])`
  - `L.append(),L.extend(),L.count(),L.index(), L.insert(),L.pop(),L.remove(),L.reverse(), L.sort()`
- internal representation should be private
- correct behavior may be compromised if you manipulate internal representation directly – use defined interfaces

Creating Objects and Classes

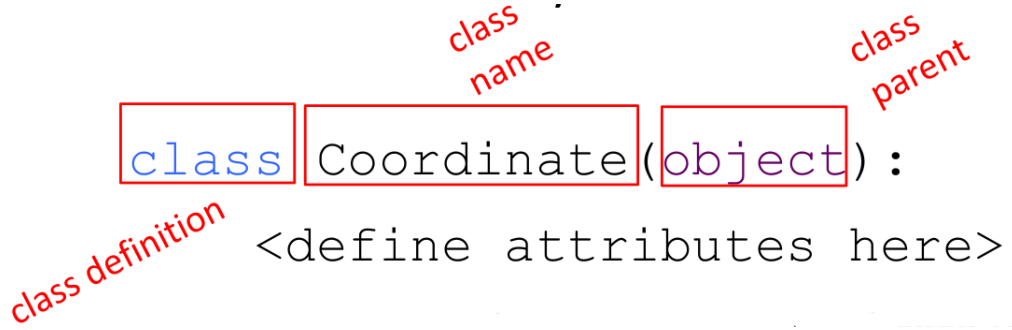
- make a distinction between creating a class and using an instance of the class
- creating the class involves
  - defining the class name
  - defining class attributes
  - for example, someone wrote code to implement a list class
- using the class involves
  - creating new instances of objects
  - doing operations on the instances
  - for example, `L=[1,2]` and `len(L)`

Advantages of OOP

- bundle data into packages together with procedures 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

Define Own Types

- use the `class` keyword to define a new type



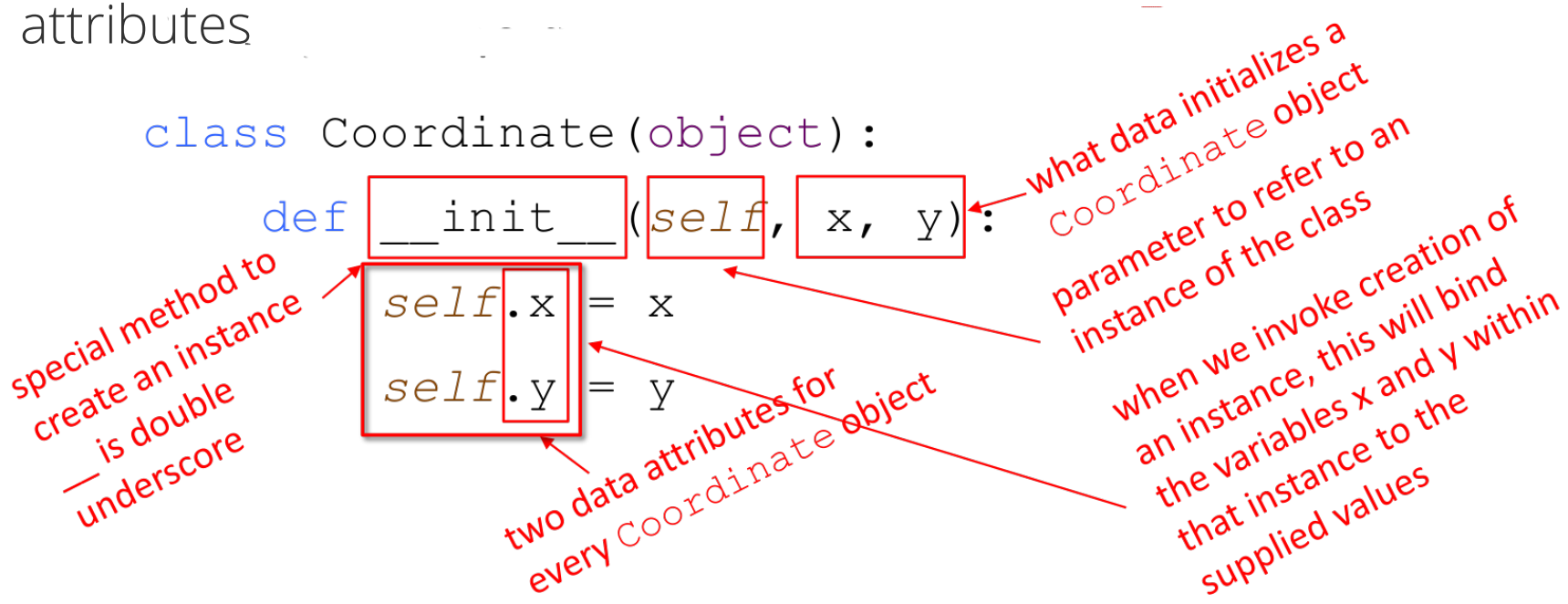
- similar to `def`, indent code to indicate which statements are part of the class definition
- the word `object` means that `Coordinate` is a Python object and inherits all its attributes
  - `Coordinate` is a subclass of `object`
  - `object` is a superclass of `Coordinate`

Attributes

- data and procedures that "belong" to the class
- data attributes
  - think of data as other objects that make up the class
  - for example, a coordinate is made up of two numbers
- procedural attributes (methods)
  - think of methods as functions that only work with this class
  - for example you can define a distance between two coordinate objects but there is no meaning to a distance between two list objects

Defining creation of instance of a Class

- first have to define how to create an instance of object
- use a special method called **__init__** to initialize some data attributes



Actual Creation of instance of a class

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

create a new object of type `Coordinate` and pass in 3 and 4 to the `__init__` method

note that argument for self is automatically supplied by Python!

use the dot to access an attribute of instance `c`

- data attributes of an instance are called instance variables
- don't provide argument for `self`, Python does this automatically

Actual Creation of instance of a class

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

- think of c as pointing to a frame (like we saw with function calls)
  - 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 associate with x within that frame (thus the specific value for this instance)

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

Define method for Coordinate class

```
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
```

use it to refer to any instance

another parameter to method

dot notation to access data

- other than self and dot notation, methods behave just like functions (take params, do operations, return value)

HOW TO USE A METHOD FROM THE Coordinate CLASS

```
def distance(self, other)
```

- Conventional way

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

object on which to call method

name of method

parameters not including `self` (`self` is implied to be `c`)

- Equivalent to

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

name of class

name of method

parameters, including an object on which to call the method, representing `self`

HOW TO USE A METHOD FROM THE Coordinate CLASS

```
def distance(self, other)
```

- Conventional way

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

- Equivalent to

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

- think of Coordinate as pointing to a frame
  - within the scope of that frame we created methods
  - Coordinate.distance gets the value of Coordinate (a frame), then looks up the value associated with distance (a procedure), then invokes it (which requires two arguments)
  - c.distance inherits the distance from the class definition, an automatically uses c as the first argument

Print representation of an Object

```
In  [1]: c = Coordinate(3,4)
In  [2]: print(c)

<__main__.Coordinate object at 0x7fa918510488>
```

- uninformative print representation by default
- define a **__str__** method for a class
- Python calls the __str__ method when used with print on your class object
- you choose what it does! Say that when we print a Coordinate object, want to show

```
In  [3]: print(c)
<3,4>
```

Defining own print method

```
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) + ">"
```

name of special method

must return a string

Types & Classes

- can ask for the type of an object instance

```
In  [4]: c = Coordinate(3,4)
In  [5]: print(c)
<3,4>
In  [6]: print(type(c))
<class __main__.Coordinate>
```

- this makes sense since

```
In  [7]: print(Coordinate, type(Coordinate))
<class __main__.Coordinate> <type 'type'>
```

- use `isinstance()` to check if an object is a Coordinate

```
In  [8]: print(isinstance(c, Coordinate))
True
```

Special Operators

- +, -, ==, <, >, len(), print, and many others
- like print, can override these to work with your class
- define them with double underscores before/after

```
__add__(self, other)    →    self + other
__sub__(self, other)    →    self - other
__eq__(self, other)     →    self == other
__lt__(self, other)     →    self < other
__len__(self)           →    len(self)
__str__(self)           →    print(self)
```

Example: Fractions

- create a new type to represent a number as a fraction
- internal representation is two integers
  - numerator
  - denominator
- interface a.k.a. methods a.k.a how to interact with `Fraction` objects
  - print representation
  - add, subtract
  - convert to a float

Initial Fraction class

```
class fraction(object):
    def __init__(self, numer, denom):
        self.numer = numer
        self.denom = denom
    def __str__(self):
        return str(self.numer) + ' / ' + str(self.denom)
```

```
In  [9]: oneHalf = fraction(1,2)

In [10]: twoThirds = fraction(2,3)

In [11]: print(oneHalf)
1 / 2

In [12]: print(twoThirds)
2 / 3
```

Accessing data attributes

```
class fraction(object):
    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
```

Accessing data attributes

```
In  [9]: oneHalf = fraction(1,2)

In [10]: twoThirds = fraction(2,3)

In [13]: oneHalf.getNumer()
Out[13]: 1

In [14]: fraction.getDenom(twoThirds)
Out[14]: 3
```

this is a procedure, so must invoke by passing in arguments (zero in this case)

Adding Methods

```
class fraction(object):
    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.getDenon() * 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)
```

Adding Methods

```
In  [9]: oneHalf = fraction(1,2)

In [10]: twoThirds = fraction(2,3)

In [15]: new = oneHalf + twoThirds

In [16]: print(new)
7 / 6
```

Adding more methods

```
class fraction(object):
    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.getDenon() * 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()
```

Adding more methods

```
In  [9]: oneHalf = fraction(1,2)

In [10]: twoThirds = fraction(2,3)

In [17]: new = oneHalf + twoThirds

In [18]: new.convert()
Out[18]: 1.1666666666666667
```

Example: a set of integers

- create a new type to represent a collection of integers
  - initially the set is empty
  - a particular integer appears only once in a set: representational invariant enforced by the code
- internal data representation
  - use a list to store the elements of a set
- interface
  - insert(e) – insert integer e into set if not there
  - member(e) – return True if integer e is in set, False else
  - remove(e) – remove integer e from set, error if not present

Integer set class

```
class intSet(object):
    def __init__(self):
        self.vals = []
    def insert(self, e):
        if not e in self.vals:
            self.vals.append(e)
    def member(self, e):
        return e in self.vals
    def remove(self, e):
        try:
            self.vals.remove(e)
        except:
            raise ValueError(str(e) + ' not found')
    def __str__(self):
        self.vals.sort()
        result = ''
        for e in self.vals:
            result = result + str(e) + ','
        return '{' + result[:-1] + '}'
```

using properties of lists; ensuring that element only appears once

using property that list is an iterable

can use exception to catch attempt to remove nonexistent element

Using Integers sets

```
In [19]: s = intSet()

In [20]: print(s)
{}

In [21]: s.insert(3)
In [22]: s.insert(4)
In [23]: s.insert(3)
In [24]: print(s)
{3, 4}
```

Using Integer sets

```
In [19]: s = intSet()
In [21]: s.insert(3)
In [22]: s.insert(4)
In [23]: s.insert(3)

In [25]: s.member(3)
True

In [26]: s.member(6)
False
```

Using Integer sets

```
In [19]: s = intSet()
In [21]: s.insert(3)
In [22]: s.insert(4)
In [23]: s.insert(3)

In [27]: s.remove(3)
In [28]: s.insert(6)

In [29]: print(s)
{4,6}
```

Using Integer sets

```
In [19]: s = intSet()
In [21]: s.insert(3)
In [22]: s.insert(4)
In [23]: s.insert(3)
In [27]: s.remove(3)
In [28]: s.insert(6)

In [30]: s.remove(3)
ValueError: 3 not found
```

Power of OOP

- bundle together objects that share
  - common attributes and
  - procedures that operate 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

Implementing Class vs. Using the class

- implementing a new object type with a class
  - define the class
  - • define data attributes (what IS the object)
  - define methods (HOW to use the object)
- using the new object type in code
  - create instances of the object type
  - do operations with them
- write code from two different perspectives
- all class examples we saw so far were numerical

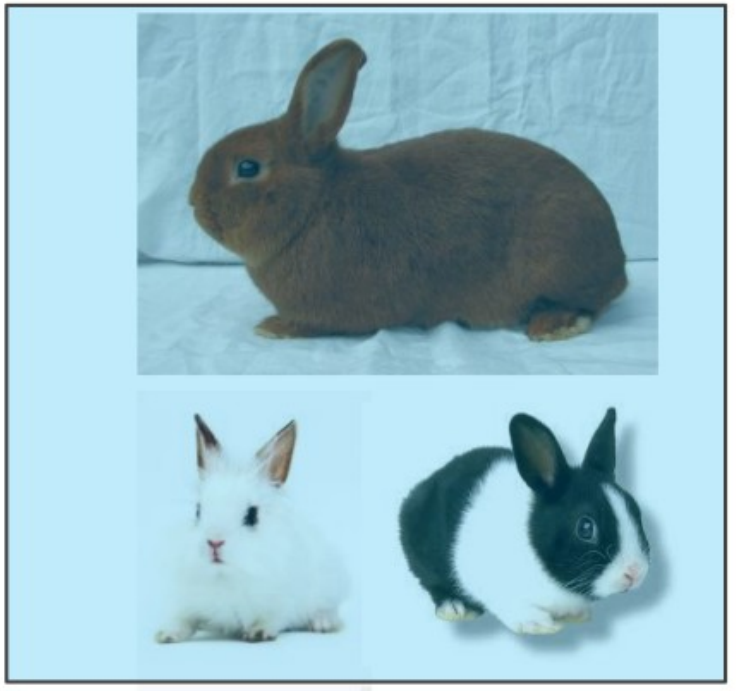
Class definition of an Object type vs. Instance of a class

- class is the type
  - a Coordinate type
  - class Coordinate(object):
- class is defined generically
  - use self to refer to any instance while defining the class
- class defines data and methods common across all instances

- instance is one particular object
  - mycoo = Coordinate(1,2)
- data values vary between instances
  - c1 = Coordinate(1,2)
  - c2 = Coordinate(3,4)
  - c1 and c2 have different data values because they are different objects
- instance has the structure of the class

Why Use OOP and Classes of Objects ?

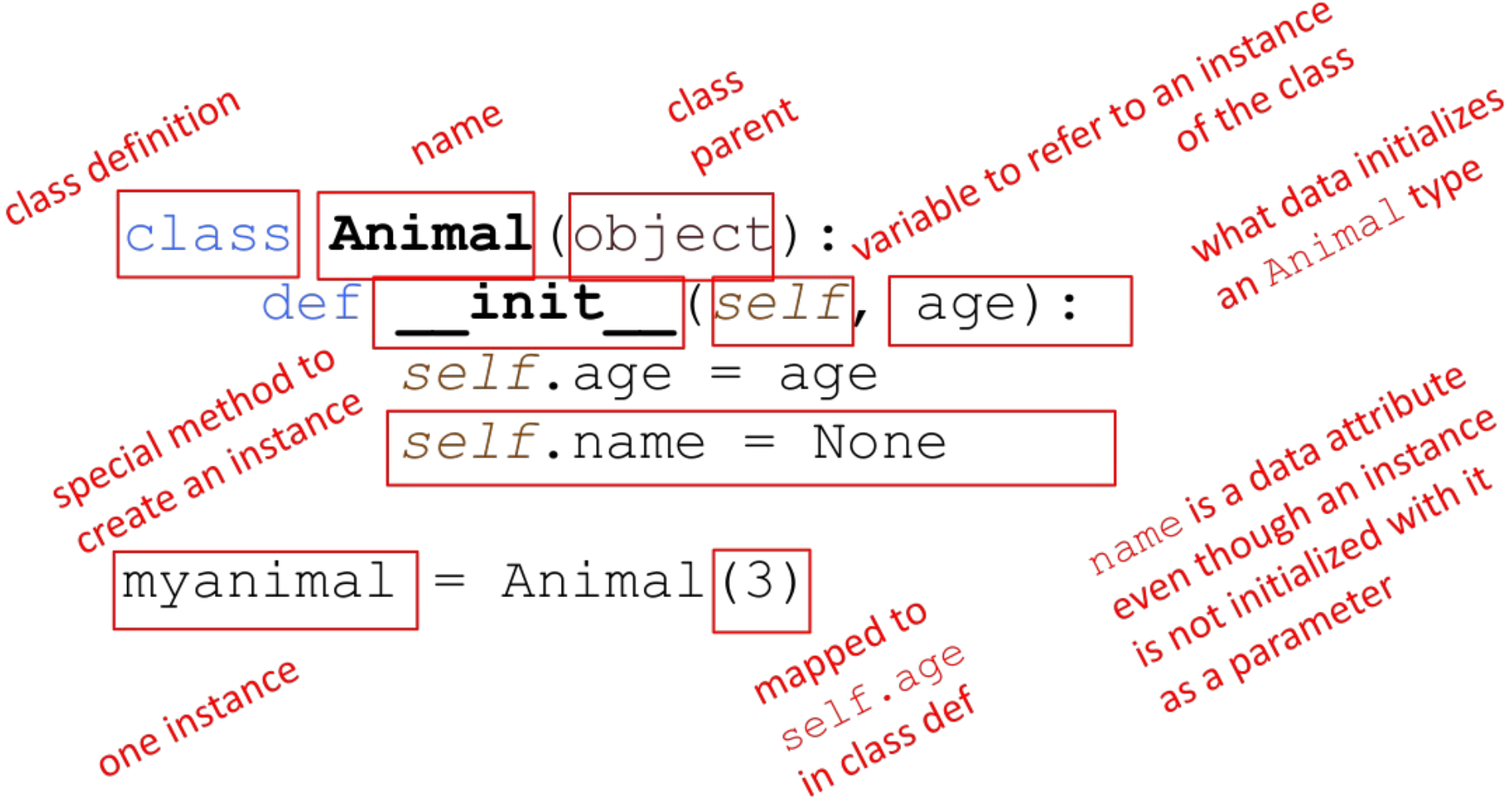
- mimic real life
- group different objects as part of the same type



Group of Objects have Attributes

- data attributes
  - how can you represent your object with data?
  - what it is
  - <for a coordinate: x and y values>
  - <for an animal: age, name>
- procedural attributes (behavior/operations/methods)
  - what kinds of things can you do with the object?
  - what it does
  - <for a coordinate: find distance between two>
  - <for an animal: make a sound>

Defining a Class



Getter and Setter Methods

- getters and setters should be used outside of class to access data attributes

getter

```
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)
```

setter

An instance and dot notation

- instantiation creates an instance of an object
  a = Animal(3)
- dot notation used to access attributes (data and methods) though it is better to use getters and setters to access data attributes

```
a.age
```

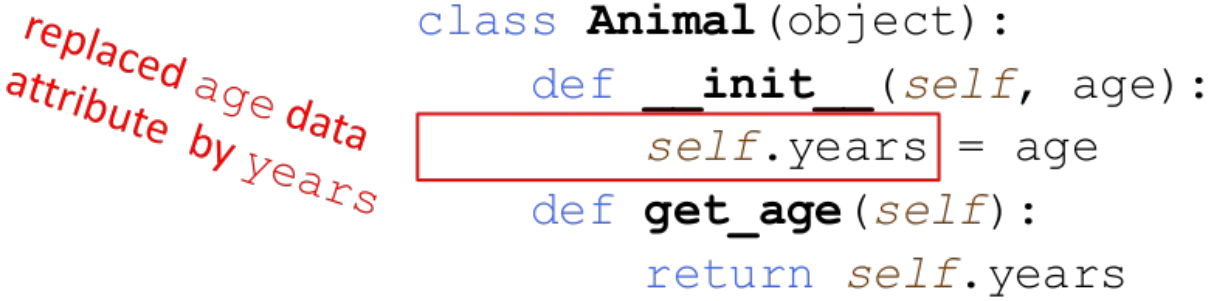
```
a.get_age()
```

access data attribute

access and **use** method; if just call `a.get_age`, get method but don't invoke

Information Hiding

- author of class definition may change data attribute variable names



- if you are accessing data attributes outside the class and class definition changes, may get errors
- outside of class, use getters and setters instead

  use a.get_age() NOT a.age

  - good style
  - easy to maintain code
  - prevents bugs

Python not great at information hiding

- allows you to access data from outside class definition
  `print(a.age)`
- allows you to write to data from outside class definition
  `a.age = 'infinite'`
- allows you to create data attributes for an instance from outside class definition

  `a.size = "tiny"`
- it's not good style to do any of these!

self and other Arguments

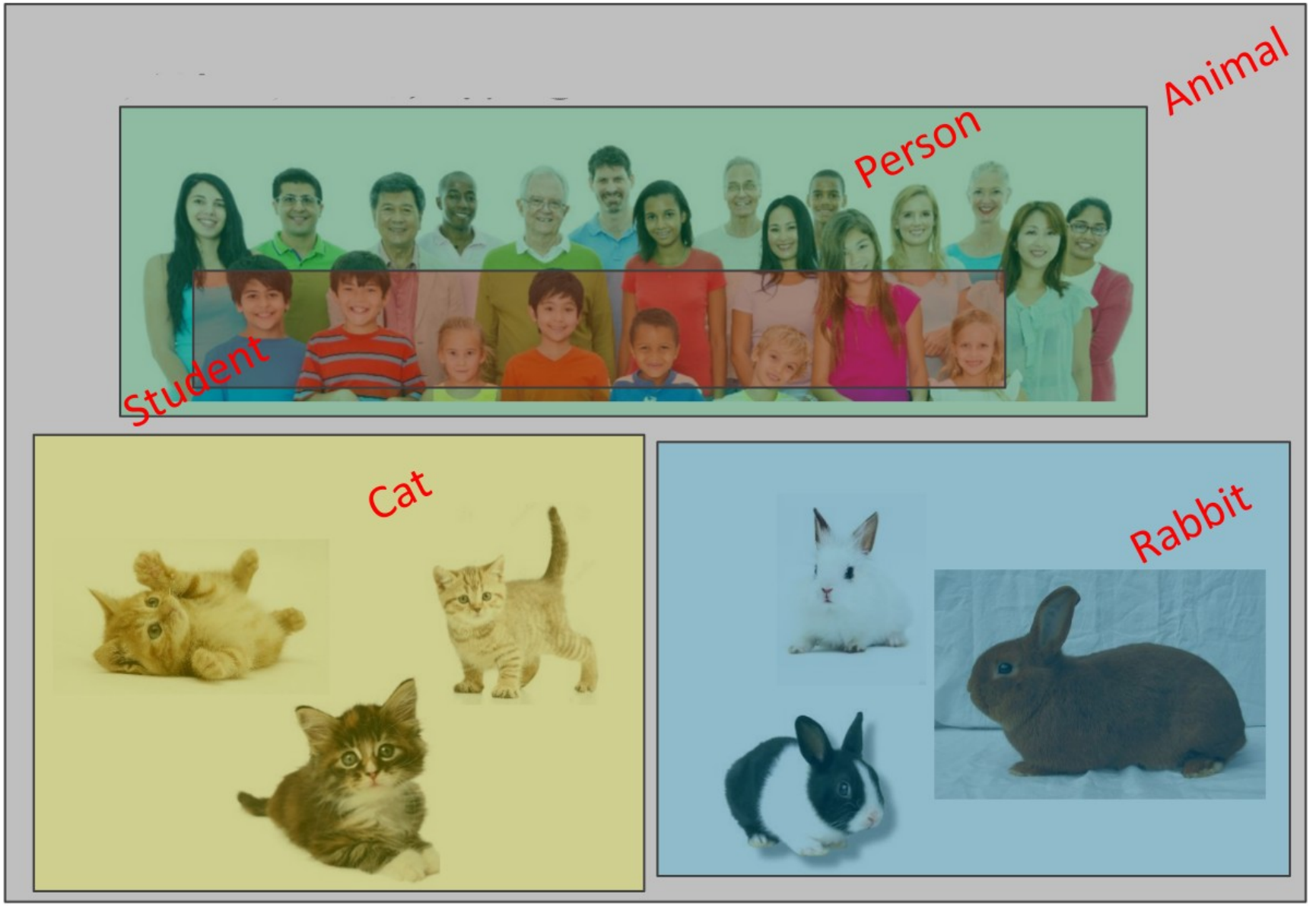
- self determined from instance, passed in as argument
  - for the method: `def __init__(self, age)`
    - creates self, passes it in automatically as argument
  - `a = Animal(3)`
  - for the method: def get_age(self)
    - call method with `a.get_age()`
    - or an alternate way `Animal.get_age(a)`

self and other Arguments

- default arguments for formal parameters are used if no actual argument is given
  - for the method:
    ```
    def set_name(self, newname="")
    ```
    - default argument used here
    - argument passed is used here
      ```
      a.set_name("fluffy")
      ```

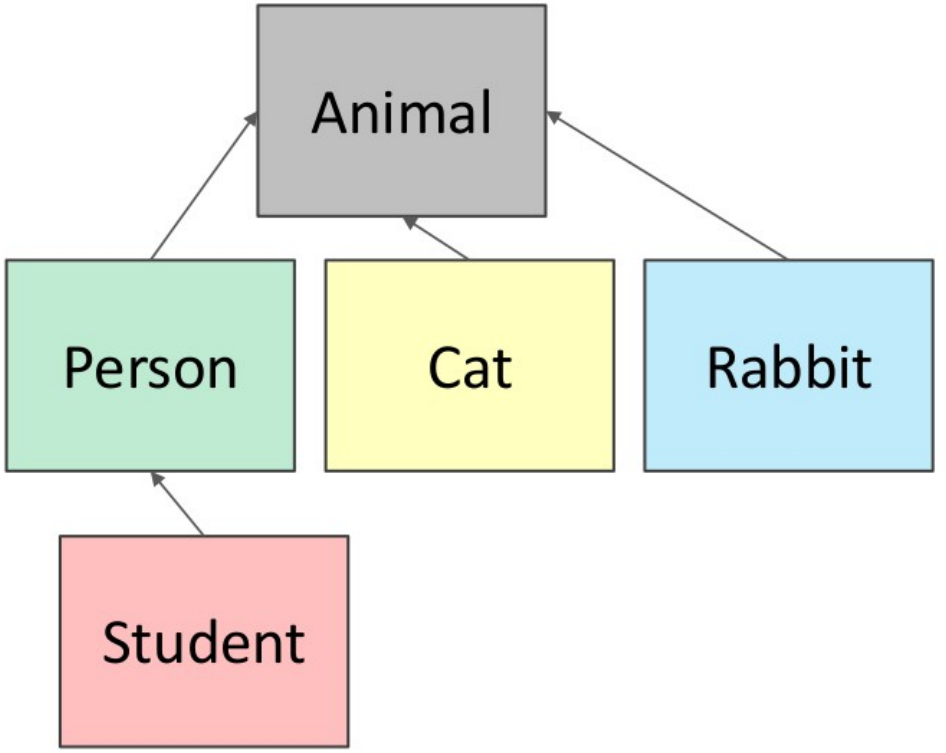
```
a.set_name()
```

Hierarchies



Hierarchies

- parent class (superclass)
- child class (subclass)
  - inherits all data and behaviors of parent class
  - add more info
  - add more behavior
  - Override behavior



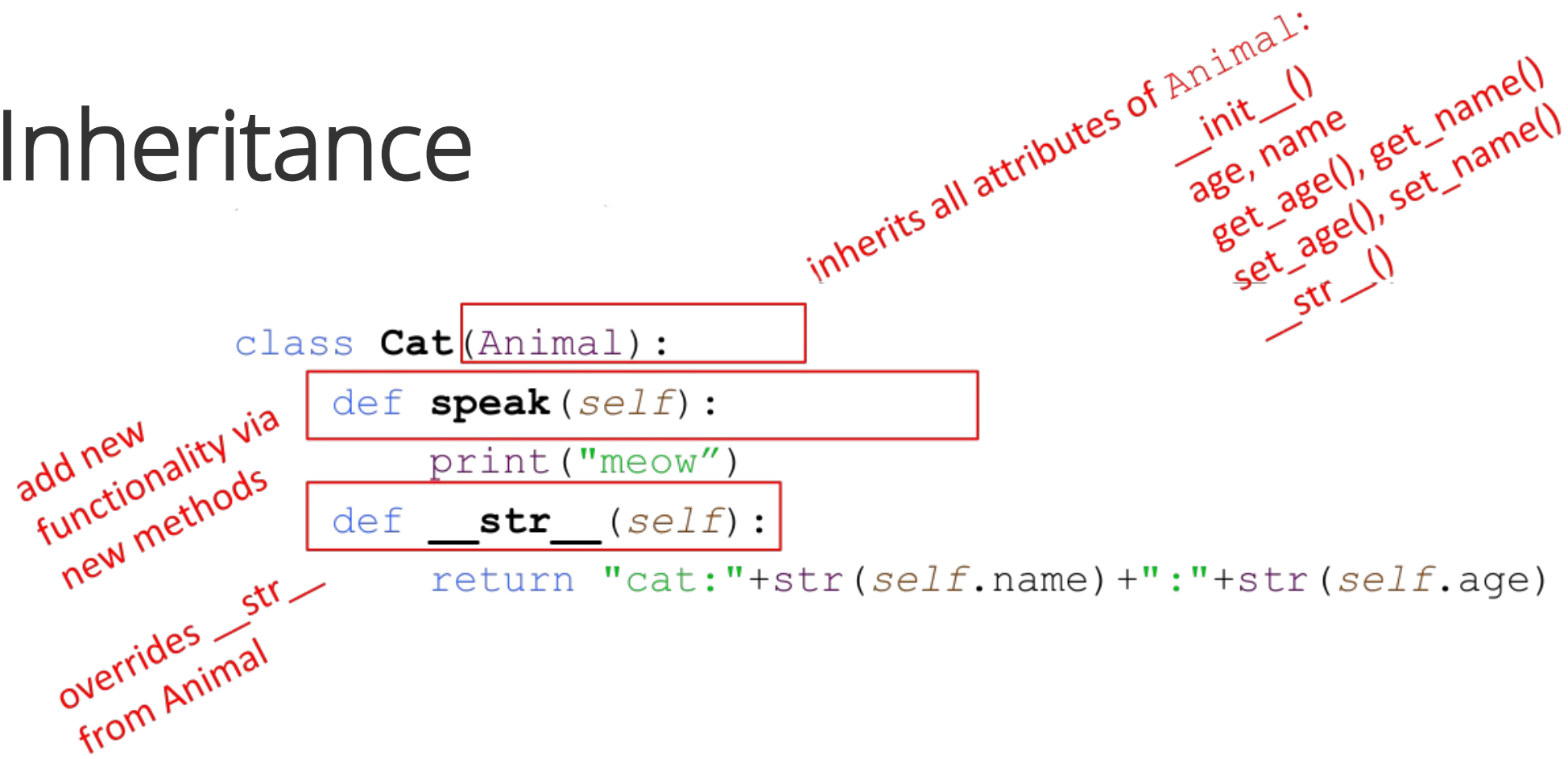
Inheritance

```
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)
```

- everything is an object
- class `object` implements basic operations in Python, like binding variables, etc

- new object class inherits properties of underlying Python `object` class

Inheritance



- add new functionality with speak()
  - instance of type Cat can be called with new methods
  - instance of type Animal throws error if called with new methods
- __init__ is not missing, uses the Animal version

Using the Hierarchy

```
In [31]: jelly = Cat(1)
In [32]: jelly.set_name('JellyBelly')
In [33]: print(jelly)
cat:JellyBelly:1

In [34]: print(Animal.__str__(jelly)
Animal:JellyBelly:1

In [35]: blob = Animal(1)
In [36]: print(blob)
animal:None:1

In [37]: blob.set_name()
In [38]: print(blob)
animal::1
```

inherits method from `Animal`

`cat __str__` method shadows method from `Animal`

could explicitly recover underlying `Animal` method

can change values of attributes of an instance

Inheritance

```
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)
```

Using the Hierarchy

```
In [31]: jelly = Cat(1)
In [34]: blob = Animal(1)
In [38]: peter = Rabbit(5)
In [39]: jelly.speak()
meow

In [40]: peter.speak()
meep

In [41]: blob.speak()
AttributeError: 'Animal' object has no
attribute 'speak'
```

uses method from `Cat`

uses method from `Rabbit`

tries to find method in `Animal`

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 method name up the hierarchy (in parent, then grandparent, and so on)
- use first method up the hierarchy that you found with that method name

```
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)
```

parent class is Animal

call Animal constructor

call Animal's method

add a new data attribute

new method to give age diff in a user friendly way

override Animal's __str__ method

Using the hierarchy

```
In [42]: eric = Person('Eric', 45)
In [43]: john = Person('John'  55)
In [44]: eric.speak()
Hello

In [45]: eric.age_diff(john)
Eric is 10 years younger than John

In [46]: Person.age_diff(john,eric)
John is 10 years older than Eric
```

uses method from `Person`

uses method associated with instance

can use class attribute to find method

```
import random

class Student(Person):
    def __init__(self, name, age, major=None):
        Person.__init__(self, name, age)
        self.major = major
    def change_major(self, major):
        self= major
    def speak(self):
        r = random.random()
        if r < 0.25:
            print("i have homework")
        elif 0.25 <= r < 0.5:
            print("i need sleep")
        elif 0.5 <= r < 0.75:
            print("i should eat")
        else:
            print("i am watching tv")
    def __str__(self):
        return "student:"+str(self.name)+":"+str(self.age)+":"+str(self.major)
```

bring in methods from `random` class

inherits `Person` and `Animal` attributes

adds new data

- look up how to use the `random` class in the python docs
- `random()` method gives back float in [0, 1)

override `Person`'s `__str__` method

Using the hierarchy

```
In [42]: eric = Person('Eric', 45)
In [47]: fred = Student('Fred', 18, 'Course VI')
In [48]: print(fred)
student:Fred:18:Course VI

In [49]: fred.speak()
i have homework
In [50]: fred.speak()
i have homework
In [51]: fred.speak()
i am watching tv
In [52]: fred.speak()
i should eat
```

uses method from `Student`

uses method from `Student`

if called multiple times, may get different behavior because of random

Instance Variables vs. Class Variables

- we have seen instance variables so far in code
- specific to an instance
- created for each instance, belongs to an instance
- used the generic variable name self within the class definition

```
self.variable_name
```

- introduce class variables that belong to the class
- defined inside class but outside any class methods, outside `__init__`
- shared among all objects/instances of that class

Animal Class

```
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)
```

Class Variables & Rabbit subclass

- subclasses inherit all data attributes and methods of the parent class

```
class Rabbit(Animal):
    tag = 1
    def __init__(self, age, parent1=None, parent2=None):
        Animal.__init__(self, age)
        self.parent1 = parent1
        self.parent2 = parent2
        self.rid = Rabbit.tag
        Rabbit.tag += 1
```

parent class

class variable

instance variable

access class variable

incrementing class variable changes it for all instances that may reference it

- tag used to give unique id to each new rabbit instance

Rabbit Getter Methods

```
class Rabbit(Animal):
    tag = 1
    def __init__(self, age, parent1=None, parent2=None):
        Animal.__init__(self, age)
        self.parent1 = parent1
        self.parent2 = parent2
        self.rid = Rabbit.tag
        Rabbit.tag += 1
    def get_rid(self):
        return str(self.rid).zfill(3)
    def get_parent1(self):
        return self.parent1
    def get_parent2(self):
        return self.parent2
```

method on a string to pad the beginning with zeros for example, 001 not 1

- getter methods specific for a `Rabbit` class
- there are also getters `get_name` and `get_age` inherited from `Animal`

Example Usage

```
In [53]: peter = Rabbit(2)
In [54]: peter.set_name('Peter')
In [55]: hopsy = Rabbit(3)
In [56]: hopsy.set_name('Hopsy')
In [57]: cotton = Rabbit(1, peter, hopsy)
In [58]: cotton.set_name('Cottontail')

In [59]: print(cotton)
animal:Cottontail:1

In [60]: print(cotton.get_parent1())
animal:Peter:2
```

uses method from `Animal`

providing explicit arguments

calling method returns instance; print then calls that instance's method

Working with own types

- define + operator between two Rabbit instances
  - define what something like this does: r4 = r1 + r2 where r1 and r2 are Rabbit instances
  - r4 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

```
def __add__(self, other):
        # returning object of same type as this class
        return Rabbit(0, self, other)
```

recall Rabbit's `__init__(self, age, parent1=None, parent2=None)`

Example Usage

```
In [53]: peter = Rabbit(2)
In [54]: peter.set_name('Peter')
In [55]: hopsy = Rabbit(3)
In [56]: hopsy.set_name('Hopsy')
In [61]: mopsy = peter + hopsy
In [62]: mopsy.set_name('Mopsy')
In [63]: print(mopsy.get_parent1())
animal:Peter:2

In [64]: print(mopsy.get_parent2())
animal:Hopsy:3
```

Method to compare two Rabbits

- decide that two rabbits are equal if they have the same two parents

```
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
```

booleans

- comparing ids of parents since ids are unique (due to class var)
- note that comparing objects (self.parent1==other.parent1) will call the __eq__ method over and over until call it on None (will get AttributeError)

Example Usage

```
In [53]: peter = Rabbit(2)
In [54]: peter.set_name('Peter')
In [55]: hopsy = Rabbit(3)
In [56]: hopsy.set_name('Hopsy')
In [57]: cotton = Rabbit(1, peter, hopsy)
In [58]: cotton.set_name('Cottontail')
In [61]: mopsy = peter + hopsy
In [62]: mopsy.set_name('Mopsy')

In [65]: print(mopsy == cotton)
True
```

Summary

- bundle together objects that share
  - common attributes and
  - procedures that operate 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