# Special Topics in Computer Science- CSC 4992

Introduction to Programmer-Defined Classes

# Objects, Classes, and Methods

- Every data value in Python is an *object*
- Every object is an instance of a *class*
- Built in classes include **`int`**, **`float`**, **`str`**, **`tuple`**, **`list`**, **`dict`**
- A class includes operations (*methods*) for manipulating objects of that class (**`append`**, **`pop`**, **`sort`**, **`find`**, etc.)
- Operators (**`==`**, **`[]`**, **`in`**, etc.) are “syntactic sugar” for methods

# Python Classes

- We have already explored a number of classes that are provided by Python.
- Some of these, such as int, bool, and float, are called primitive classes because they represent only a single value.
- For example, the integer (object) 5 is an instance of the class int.
- Likewise, the object True is an instance of the class bool.

# Python Classes (cont'd.)

- Similarly, we have explored classes such as str and list that describe string and list objects.
- These so-called collection classes provide a structure that allows objects to be grouped together.
- For example the list [23, 66, True] is a collection of three objects: two integers and a Boolean.
- The list structure provides a number of methods that we can use to manipulate these collections.
- Indexing allows us to "ask" the list for one of its objects using the the index.
- The reverse method reverses the order of the items in the list.
- Note that we used the familiar dot operator to have the list object invoke the reverse method.

```
>>> mylist = [23,66,True]
>>> mylist[1]
66
>>> mylist.reverse()
>>> mylist
[True, 66, 23]
```

# Python Classes (cont'd.)

- Although it may not be obvious, even integers use methods to perform basic arithmetic.
- Integers use a special method called __*add* __ *()* to perform addition.
- This method is defined in the *int* class and returns the result of adding the value of the object to the value of the parameter.

```
>>> count = 1
>>> count = count + 1
>>> count
2
>>> count = count.__add__(1)
>>> count
3
```

# What Do Objects and Classes Do for Us?

- An object bundles together data and operations on those data

- A computational object can model practically any object in the real (natural or artificial) world

- Some classes come with a programming language

- Any others must be defined by the programmer

# Programmer-Defined Classes

- The **`EasyFrame`** class is used to create GUI windows that are easy to set up
- The **`Image`** class is used to load, process, and save images
- Like the built-in classes, these classes include operations to run with their instances

# Other Examples

- A **`Student`** class represents information about a student and her test scores
- A **`Rational`** class represents rational numbers and their operations
- A **`Die`** class represents dice used in games
- **`SavingsAccount`**, **`CheckingAccount`**, **`Bank`**, and **`ATM`** are used to model a banking system
- **`Proton`**, **`Neutron`**, **`Electron`**, and **`Positron`** model subatomic particles in nuclear physics

# The **`Die`** Class: Its Interface and Use

Interface

```
die.py                    # The module for the Die class

Die()                     # Returns a new Die object

roll()                    # Resets the die's value

getValue()                # Returns the die's value
```

# The `Die` Class: Its Interface and Use

Interface

```
die.py                    # The module for the Die class

Die()                     # Returns a new Die object

roll()                    # Resets the die's value

getValue()                # Returns the die's value
```

Use

```
from die import Die

d = Die()                  # Create a new Die object

d.roll()                   # Roll it

print(d.getValue())        # Display its value

help(Die)                  # Look up the documentation
```

# Specifying an Interface

- The user of a class is only concerned with learning the information included in the headers of the class's methods
- This information includes the method name and parameters
- Collectively, this information comprises the class's *interface*
- Docstrings describe what the methods do

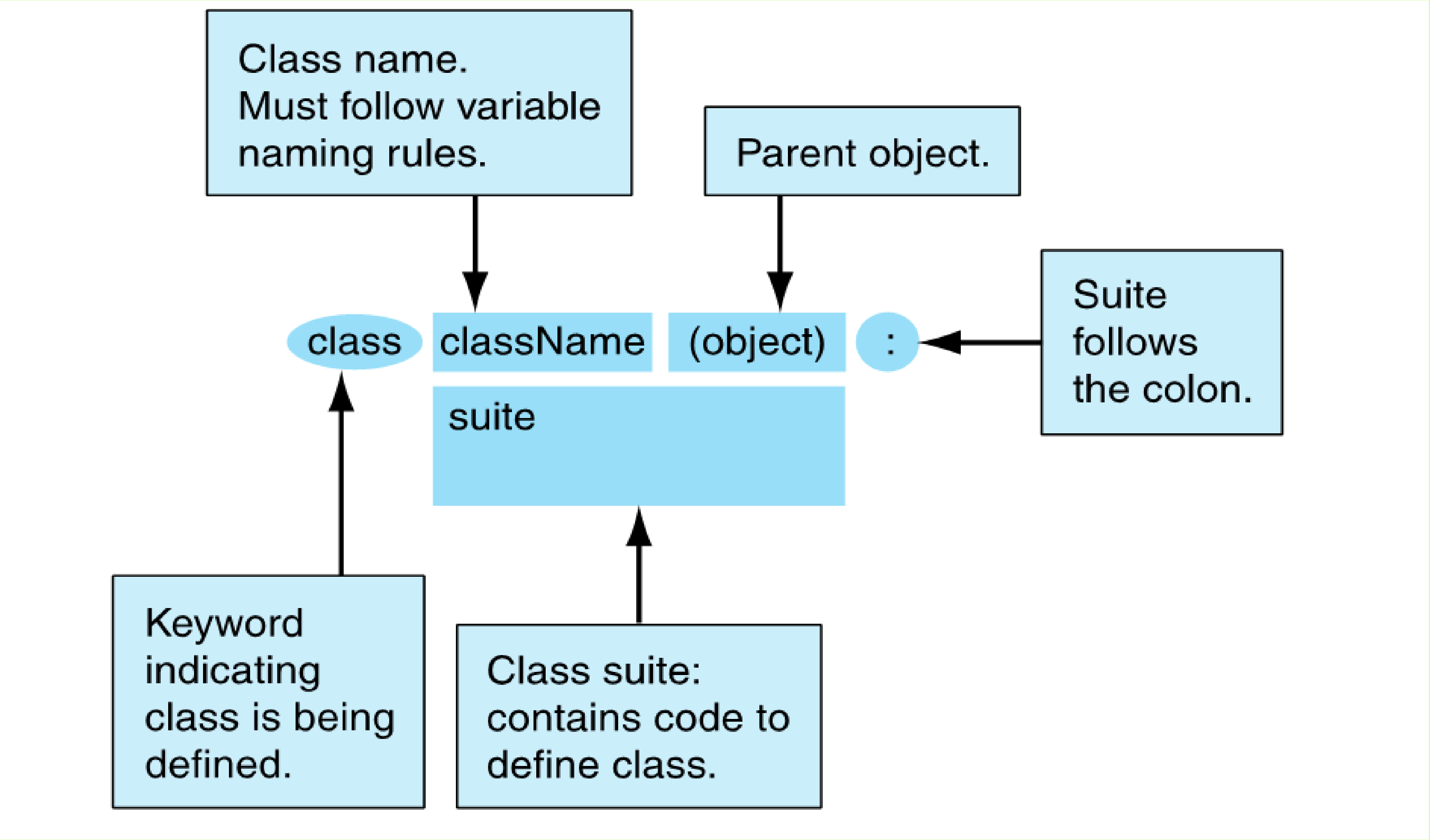
# Defining (Implementing) a Class

- The *definition* or *implementation* of a class includes completed descriptions of an object's data and the methods for accessing and modifying those data

- The data are contained in *instance variables* and the methods are called *instance methods*

- Related class definitions often occur in the same module

# Syntax Template for a Simple Class Definition



# Syntax Template for a Simple Class Definition

```
<docstring for the module>

<imports of other modules used>

class <name>(<parent class name>):
    <docstring for the class>

    <method definitions>
```

Basically a header followed by several method definitions

# Defining the `Die` Class

```
from random import randint

class <name>(<parent class name>):
    <docstring for the class>

    <method definitions>
```

We'll use **`random.randint`** to roll the die

# The Class Header

```
from random import randint

class Die(object):
    <docstring for the class>

    <method definitions>
```

By convention, programmer-defined class names are capitalized in Python

Built-in class names, like **`str`**, **`list`**, and **`object`**, are not

Like built-in function names, built-in class names appear in purple

# The Class Header

```
from random import randint

class Die(object):
    <docstring for the class>

    <method definitions>
```

All Python classes are *subclasses* of the **`object`** class

A class can *inherit* behavior from its parent class

# The Class Docstring

```
from random import randint

class Die(object):
    """This class represents a six-sided die."""

    <method definitions>
```

A class' s *docstring* describes the purpose of the class

# Setting the Initial State

```
from random import randint

class Die(object):
    """This class represents a six-sided die."""

    def __init__(self):
        self._value = 1
```

A method definition looks a bit like a function definition

The **`__init__`** method (also called a *constructor*) is automatically run when an object is instantiated; this method usually sets the object's initial state (**`d = Die()`**)

# The **`self`** Parameter

```
from random import randint

class Die(object):
    """This class represents a six-sided die."""

    def __init__(self):
        self._value = 1
```

The name **`self`** must appear as the first parameter in each instance method definition

Python uses this parameter to refer to the object on which the method is called

# Instance Variables

```
from random import randint

class Die(object):
    """This class represents a six-sided die."""

    def __init__(self):
        self._value = 1
```

**`self`** must also be used with all instance method calls and instance variable references within the defining class

**`self`** refers to the current object (a die)

# Using Instance Variables

```
from random import randint

class Die(object):
    """This class represents a six-sided die."""

    def __init__(self):
        self._value = 1

    def roll(self):
        """Resets the die's value."""
        self._value = randint(1, 6)

    def getValue(self):
        return self._value
```

`self._value` refers to this object's *instance variable*

# Where Are Classes Defined?

- Like everything else, in a module
- Define the **`Die`** class in a **`die`** module
- Related classes usually go in the same module (**`SavingsAccount`** and **`Bank`** the **`bank`** module)

# The Interface of the `SavingsAccount` Class

```
SavingsAccount(name, pin, bal)  # Returns a new object

getBalance()                   # Returns the current balance

deposit(amount)                # Makes a deposit

withdraw(amount)               # Makes a withdrawal

computeInterest()              # Computes the interest and
                               # deposits it
```

# Defining the **SavingsAccount** Class

```
class SavingsAccount(object):
    """This class represents a savings account."""

    def __init__(self, name, pin, balance = 0.0):
        self._name = name
        self._pin = pin
        self._balance = balance

    # Other methods go here
```

Note that **name** is a method's parameter, whereas **self._name** is an object's instance variable

# The Lifetime of a Variable

```
class SavingsAccount(object):
    """This class represents a savings account."""

    def __init__(self, name, pin, balance = 0.0):
        self._name = name
        self._pin = pin
        self._balance = balance

    # Other methods go here
```

Parameters exist only during the lifetime of a method call, whereas instance variables exist for the lifetime of an object

# Parameters or Instance Variables?

- Use a parameter to send information through a method to an object
- Use an instance variable to retain information in an object
- An object's *state* is defined by the current values of all of its instance variables
- References to instance variables must include the qualifier **`self`**

# The Scope of a Variable

- The *scope* of a variable is the area of program text within which its value is visible
- The scope of a parameter is the text of its enclosing function or method
- The scope of an instance variable is the text of the enclosing class definition (perhaps many methods)

# The Scope of a Variable

```python
class SavingsAccount(object):
    """This class represents a savings account."""

    def __init__(self, name, pin, balance = 0.0):
        self._name = name
        self._pin = pin
        self._balance = balance

    def deposit(self, amount):
        self._balance += amount

    def withdraw(self, amount):
        self._balance -= amount
```

`self._balance` always refers to the same storage area (for one object)

`amount` refers to a different storage area for each method call

# Student class

```python
class Student(object):
    def __init__(self, first='', last='', id=0):
        # print 'In the __init__ method'
        self.first_name_str = first
        self.last_name_str = last
        self.id_int = id

    def update(self, first='', last='', id=0):
        if first:
            self.first_name_str = first
        if last:
            self.last_name_str = last
        if id:
            self.id_int = id
```

- self is bound to the default instance as it is being made
- If we want to add an attribute to that instance, we modify the attribute associated with self.

# Example

```
s1 = Student()
print(s1.last_name_str)

s2 = Student(last='Python',first='Monty')
print(s1.last_name_str)

Python
```

# Private Variables in an Instance

- many OOP approaches allow you to make a variable or function in an instance ***private***
- private means not accessible by the class user, only the class developer.
- Use __ (double underlines) in front of any variable

```
class NewClass (object):
    def __init__(self, attribute='default', name='Instance'):
        self.name = name                    # public attribute
        self.__attribute = attribute   # a "private" attribute
```

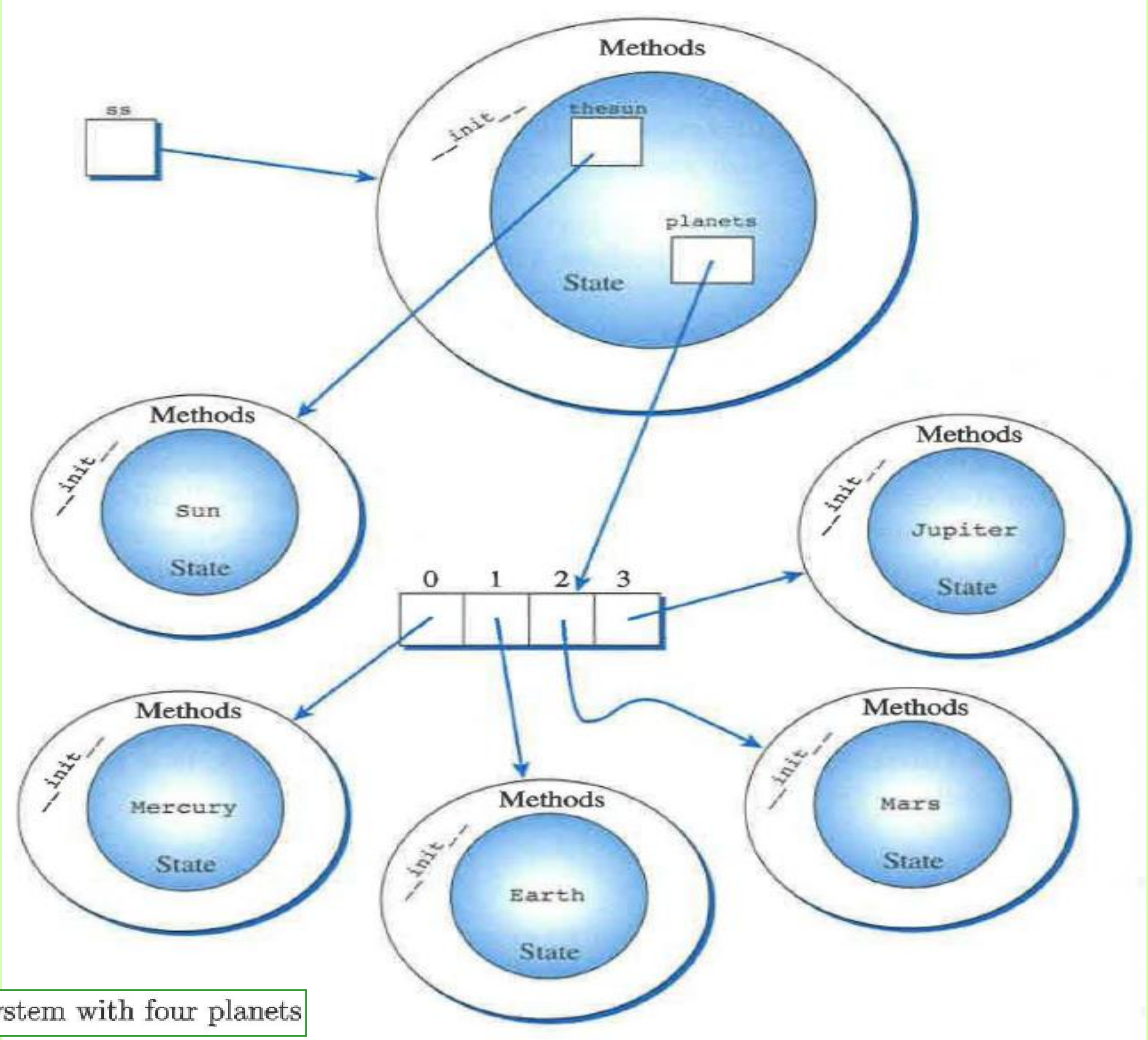
```
>>> inst1 = NewClass(name='Monty', attribute='Python')
>>> print(inst1.name)
Monty
>>> print(inst1.__attribute)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    print(inst1.__attribute)
AttributeError: 'newClass' object has no attribute '__attribute'
```

# User Classes

```
class Person:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    def set_name(self, name):
        self.__name = name


    def set_age(self, age):
        self.__age = age


    def get_name(self):
        return self.__name


    def get_age(self):
        return self.__age


mySelf = Person("John", 25)
print ( mySelf.get_name())
print ( mySelf.get_age() )

yourSelf = Person("Jack", 30)
print ( yourSelf.get_name())
print ( yourSelf.get_age() )
```

# Designing and Implementing Solar System

- We now turn our attention to solving the problem of building a model of the solar system.
- To do so will require that we consider the data that will be present.
- However, even with the rich set of built-in classes provided by Python, it is often preferable to describe our problem and solution in terms of classes that are specifically designed to represent the objects present in the problem.




A solar system with four planets

# Designing and Implementing a Planet Class

- We begin by building a simple representation of a planet and then we will design and implement a Planet class.
- To design a class to represent the idea of a planet, it is necessary to consider the data the planet objects will need to know about themselves.
- The values of the instance data will help to differentiate the individual planet objects.
- We assume that each planet has a name.
- Each planet also has size information such as the radius and the mass.
- We also want each planet to know how far it is from the sun.

# Designing and Implementing a Planet Class (cont'd.)

- In addition to data, the class will provide methods that a planet can perform.
- Some of these might be simple, such as returning the name of the planet.
- Other methods may require more computation.

# Constructor Method

- The first method that all classes should provide is the constructor, which can be defined as the way data objects are created.
- In Python, the constructor is always called __ *init* __ . (Note that there are two underscores before and after init.)
- To create a Planet object, we will need the four pieces of information listed previously as parameters: (1) name, (2) radius, (3) mass, and (4) distance.
- The constructor will then create *instance variables* to hold these values.
- Each instance variable holds a reference to an object.

# Constructor Method (cont'd.)

- Notice that even though we stated that four pieces of information would be necessary to construct a planet, there are *five* formal parameters.
- The extra parameter, self, is a special parameter that will always refer to the object that is being constructed.
- It must always be the first parameter in the list.
- Python automatically adds an actual parameter corresponding to self when you call the constructor. This means you should not explicitly pass a parameter corresponding to self.

# Constructor Method (cont'd.)

- The *self.name*, appearing on the left side of an assignment statement in the constructor defines an *instance variable*.
- Since self refers to the object being constructed, *self.name* refers to a variable in the object.

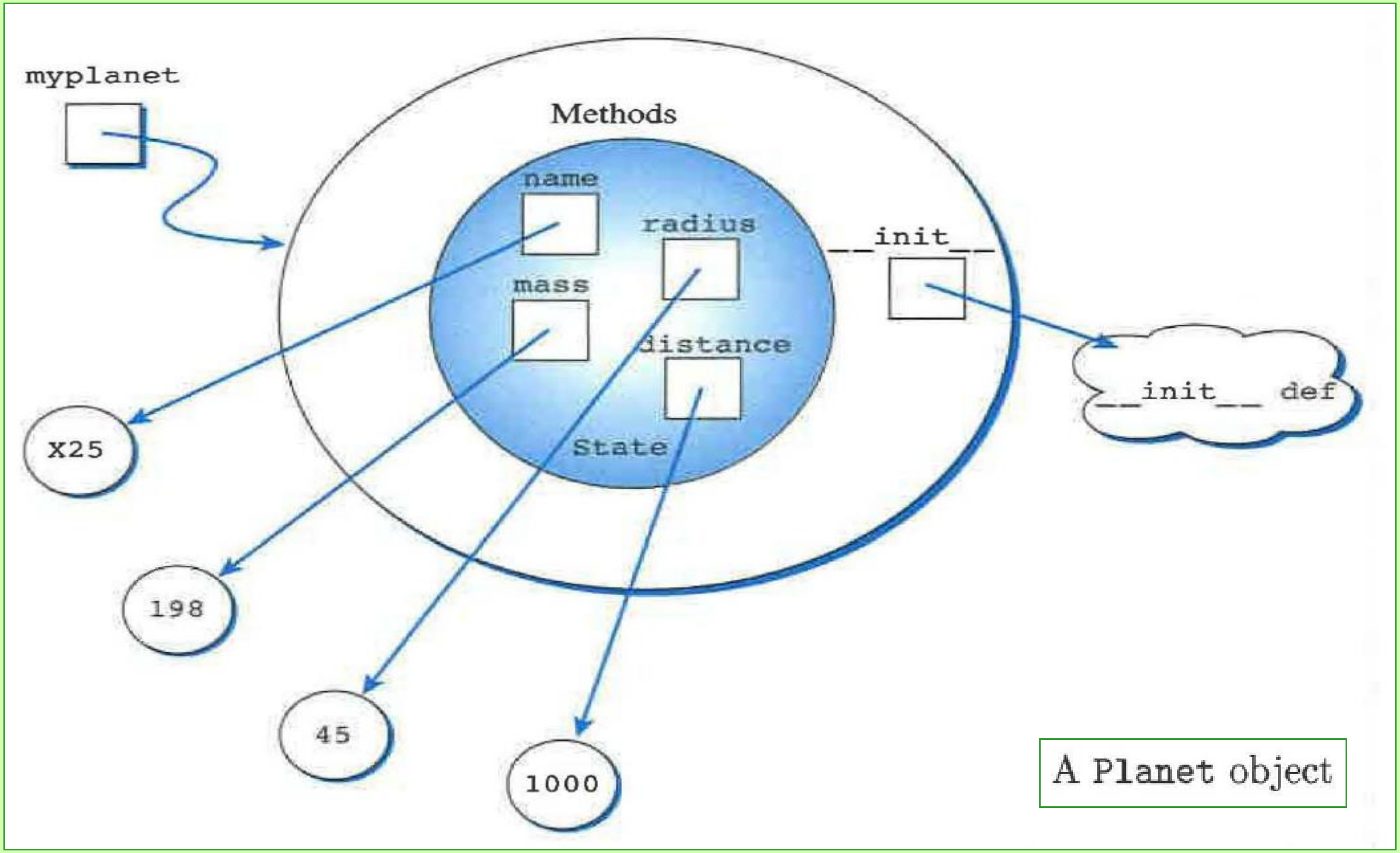
```
class Planet:

    def __init__(self, iname, irad, im, idist):
        self.name = iname
        self.radius = irad
        self.mass = im
        self.distance = idist
```

# Constructor Method (cont'd.)

- We use the constructor to create an *instance of the class* by using the name of the class (we do not call_jnit _ directly).
- The call to the Planet constructor requires only four parameters even though it is defined to have five.
- The first parameter, self, never receives an explicit value as it always refers implicitly back to the object being constructed.
- Note that evaluating the reference myplanet shows that it is an instance of the Planet class. The value Ox58530 is the actual address in memory where the object is stored.

```
>>> myplanet = Planet("X25", 45, 198, 1000)
>>> myplanet
<__main__.Planet instance at 0x58530>
```

# Designing and Implementing a Planet Class (cont'd.)



# Designing and Implementing a Planet Class (cont'd.)

- The newly created object myplanet is an instance of the Planet class with a name of "X25", a radius of 45, a mass of 198, and a distance of 1000. Note that we have separated the object into two distinct layers.
- The inner layer, which we call the state, contains the instance variable names. The outer layer contains the names of the methods.
- In both cases the names are simply references to the actual objects.
- There is a strong relationship between the methods of an object and its instance data.

# Accessor Methods

- The next methods that we will write are commonly called *accessor methods* as they allow us to access the instance variables of the object.
- It is typical that each instance variable might have an associated accessor method.
- For example, four accessor methods associated with the instance variables name, radius, mass, and distance from the Planet class.

# Accessor Methods (cont'd.)

- In the get Name method we referred to the name instance variable as *self.name*, where *self* is a synonym for *myplanet* (self and myplanet both reference the same object).

```
def getName(self):
    return self.name

def getRadius(self):
    return self.radius

def getMass(self):
    return self.mass

def getDistance(self):
    return self.distance
```

```
>>> myplanet.getName()
'X25'
>>> myplanet.getMass()
198
>>> myplanet.name
'X25'
```

# Accessor Methods (cont'd.)

- In order to see this more clearly, you can access the instance variable directly using an expression such as *myplanet.name*.
- Although either of these two techniques will allow us to access instance variables within the object, at this point it is preferable to use the accessor methods.
- Using the accessor methods provides a more formal and controlled access to the object.
- On a large software project, it is common for the internal representation of an instance variable to change.
- An accessor method hides those internal changes from the user. The common term for this practice is *information hiding*.

# Designing and Implementing a Planet Class (cont'd.)

- It is also possible to create accessor methods that return computed results based upon values of instance variables.
- For example, if we assume that a planet is a sphere, we can write an accessor method that will return the volume of the planet since the radius is already an instance variable.
- We can also include methods that return the surface area as well as the density of the planet.

# Designing and Implementing a Planet Class (cont'd.)

```
def getVolume(self):
    v = 4.0/3 * math.pi * self.radius**3
    return v

def getSurfaceArea(self):
    sa = 4.0 * math.pi * self.radius**2
    return sa

def getDensity(self):
    d = self.mass / self.getVolume()
    return d
```

```
>>> myplanet.getVolume()
381703.50741115981
>>> myplanet.getSurfaceArea()
25446.900494077323
>>> myplanet.getDensity()
0.00051872722219291404
>>>
```

# Mutator Methods

- *Mutator methods* are procedures that mutate or change an object in some way.
- Changes to the object involve changes to one or more of the instance variables.
- Recall that each object from a particular class has the same instance variables but the values of those variables are different, therefore allowing the object to "behave" differently when asked to perform methods.
- In order to change those variable values, we provide methods.

# Mutator Methods (cont'd.)

- A mutator method will allow us to change the name of a planet in our example.
- Instead of using a cryptic name, such as "X25," we can change the name to something more meaningful.
- To do this, we need a method that takes the new name as a parameter and modifies the value of the name instance variable.
- It is important to note that mutator methods modify the state of an object but do not return any value to the caller.

```
def setName(self, newname):
    self.name = newname
```

```
>>> myplanet.getName()
'X25'
>>> myplanet.setName("Gamma Hydra")
>>> myplanet.getName()
'Gamma Hydra'
```

# Special Methods

- There are two ways that we can provide better printing capability for the Planet class.
- One is to define a method called *show* that will allow any Planet object to show itself in a form by printing the individual instance variable items.
- For this example, we will simply print the name of the planet.
- Now if we create a Planet object, we can ask it to show itself. The object will respond by printing its name
- However, we still cannot print the object directly.

```
def  show(self):
     print(self.name)
```

```
>>> myhome = Planet("Earth",6371,5.97e24,152097701)
>>> myhome.show()
Earth
>>> print(myhome)
<__main__.Planet object at 0x590d0>
```

# Special Methods (cont'd.)

- There are a number of other special methods that we can define within a class.
- One of these, __ *str* __ , is used to provide a string representation for an object.
- The method does not require any other additional information other than the self parameter.
- The method in turn returns the string that represent the object.

```
>>> myhome = Planet("Earth",6371,5.97e24,152097701)
>>> print(myhome)
Earth
>>> myhome.__str__()
'Earth'
>>> str(myhome)
'Earth'
```

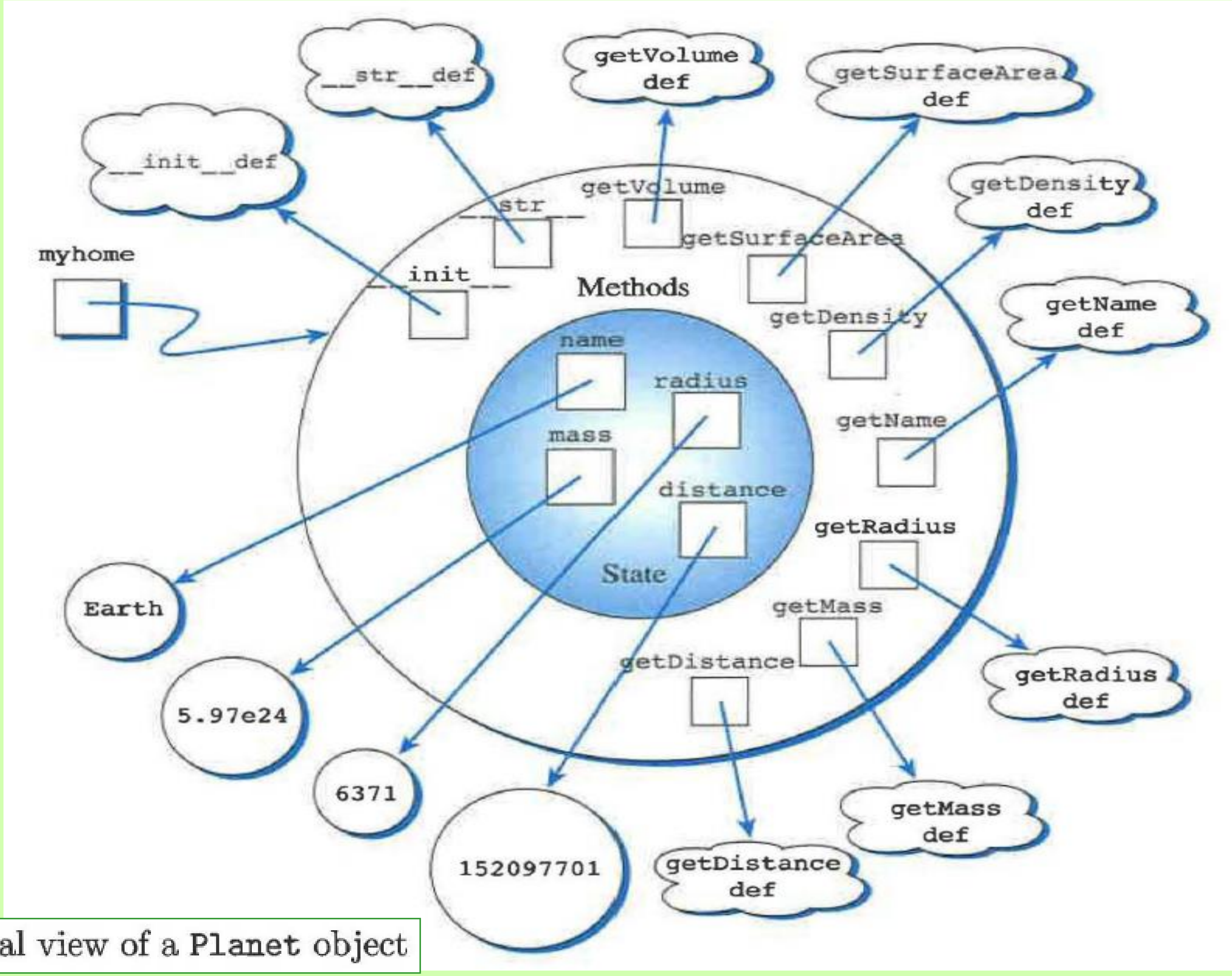
```
def __str__(self):
    return self.name
```

# The Planet Class

```python
import math

class Planet:
    def __init__(self, iname, irad, im, idist):
        self.name = iname
        self.radius = irad
        self.mass = im
        self.distance = idist

    def getName(self):
        return self.name

    def getRadius(self):
        return self.radius

    def getMass(self):
        return self.mass

    def getDistance(self):
        return self.distance

    def getVolume(self):
        v = 4.0/3 * math.pi * self.radius**3
        return v

    def getSurfaceArea(self):
        sa = 4.0 * math.pi * self.radius**2
        return sa

    def getDensity(self):
        d = self.mass / self.getVolume()
        return d

    def __str__(self):
        return self.name
```

```python
myplanet = Planet("X25",45,198,1000)
print(myplanet.getVolume())
print(myplanet.getDensity())

print(myplanet)
```

```
>>>
381703.5074111598
0.0005187272219291404
X25
```




Logical view of a Planet object

# Designing and Implementing a Sun Class

- Our task in this chapter is to construct a software model of a planetary system.
- This means that we need to consider not only the planets that might be present but also the most important member, the sun.
- We will consider the sun to be similar to a planet, in that it is a large, round, celestial body. It will certainly have some of the same characteristics as planets, including name, mass, and radius.
- However, since the sun is at the center of the solar system, it will not have any distance measure.
- In addition, the sun provides heat and light that can be characterized by the temperature on the surface.

# Designing and Implementing a Sun Class (cont'd.)

- Given that description, we can create a Sun class using the same patterns that we followed for the Planet class.

```
import math
class Sun:

    def __init__(self, iname, irad, im, itemp):
        self.name = iname
        self.radius = irad
        self.mass = im
        self.temp = itemp

    def getMass(self):
        return self.mass

    def __str__(self):
        return self.name
```

# Designing and Implementing a Solar System

- Now we are ready to build our solar system, which will consist of a sun and a collection of planets, each defined to be some distance away from the sun.
- We will assume that the sun resides at the center of the solar system.
- The SolarSystem class will be implemented in the same way as the other classes seen so far.
- We need to provide a constructor that will be responsible for defining the instance variables.
- We also define appropriate accessor and mutator methods.

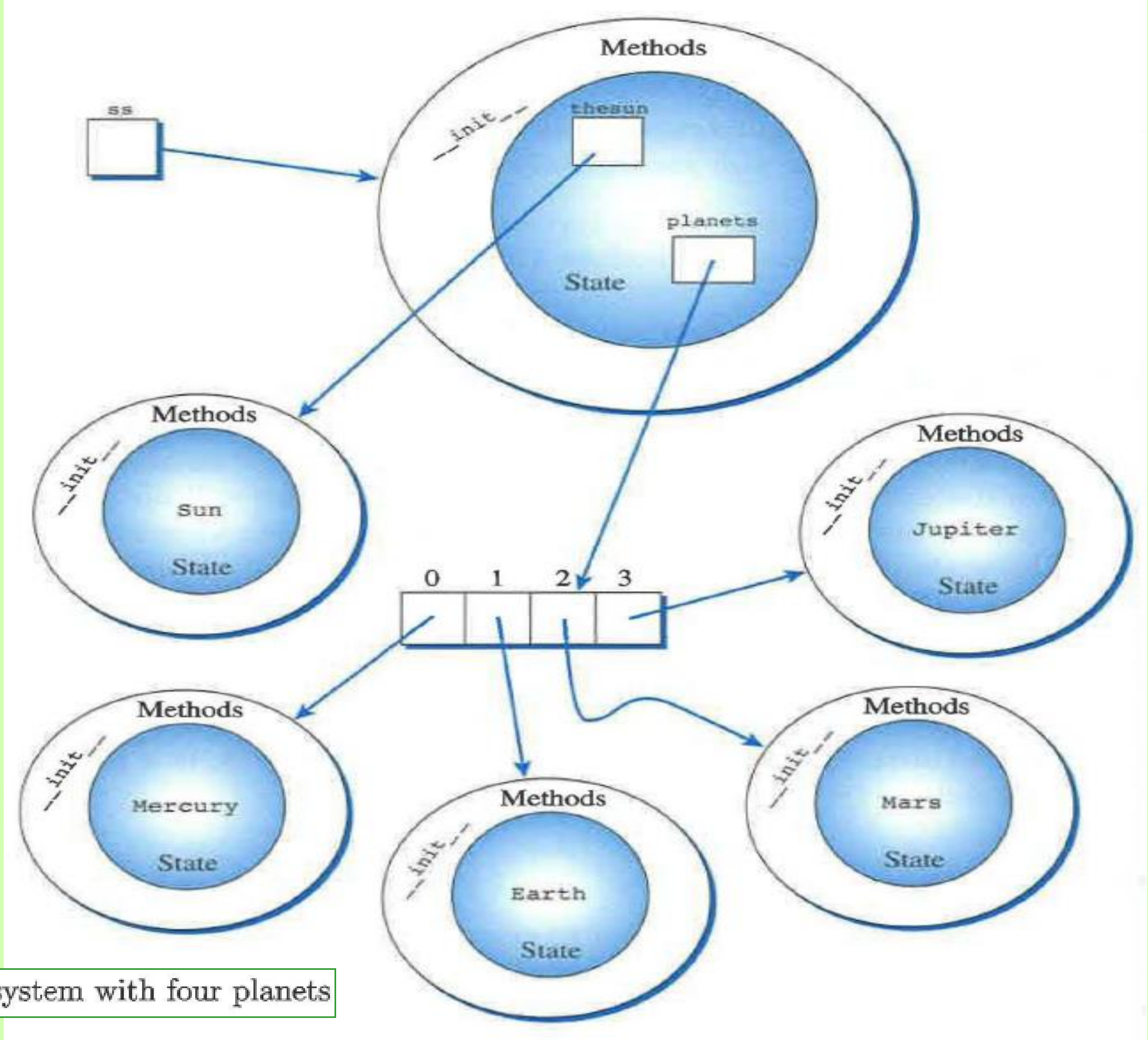
# Designing and Implementing a Solar System (cont'd.)

- Our constructor will assume that a basic SolarSystem object must have a Sun object at its center.
- This means that the constructor will expect to receive a Sun object as a parameter but will assume an empty collection of planets.
- We will implement the planet collection as a list.

```
class SolarSystem:

    def __init__(self, asun):
        self.thesun = asun
        self.planets = []

    def addPlanet(self, aplanet):
        self.planets.append(aplanet)

    def showPlanets(self):
        for aplanet in self.planets:
            print(aplanet)
```

# Designing and Implementing a Solar System (cont'd.)

- In order to add a Planet to the SolarSystem, we include a mutator method called addPlanet that can modify the collection of planets.
- This method receives a Planet object as a parameter and adds the object to the collection of planets.
- Since the collection is a list, the modification will simply use the append method.
- Finally, a simple accessor method called showPlanets will show all of the planets in the solar system.
- This can be implemented by iterating through the list of planets and printing each one of them.
- Recall that the Planet class implements the __ *str* __ method that returns the name of the planet.




A solar system with four planets

# Designing and Implementing a Solar System (cont'd.)

- In order to use the three classes we have implemented, we must save them as Python files.
- The classes Sun, Planet, and SolarSystem will be saved in the files sun.py, planet.py, and solarsystem.py.
- When we store a class in a file like this, we have created a module that can be used by other programs.
- To use our Planet, Sun, or SolarSystem classes, we simply import the module that contains them.

```
>>>from sun import *
>>>from planet import *
>>>from solarsystem import *
>>>
>>>sun = Sun("SUN", 5000, 1000, 5800)
>>>ss = SolarSystem(sun)
>>>
>>>p = Planet("MERCURY", 19, 10, 25)
>>>ss.addPlanet(p)
>>>
>>>p = Planet("EARTH", 50, 60, 30)
>>>ss.addPlanet(p)
>>>
>>>p = Planet("MARS", 47, 50, 35)
>>>ss.addPlanet(p)
>>>
>>>m = Planet("JUPITER", 75, 100, 50)
>>>ss.addPlanet(p)
>>>
>>>ss.showPlanets()
MERCURY
EARTH
MARS
JUPITER
```

# Python overload ops

- Python provides a set of operators that can be overloaded. You can't overload all the operators.
- Like all the special class operations, they use the two underlines before and after
- They come in three general classes:
  - numeric type operations (+,-,<,>,print etc.)
  - container operations ([], len, etc.)
  - general operations (printing, construction)

| Math-like Operators | | |
|---|---|---|
| **Expression** | **Method name** | **Description** |
| x + y | `__add__()` | Addition |
| x − y | `__sub__()` | Subtraction |
| x * y | `__mul__()` | Multiplication |
| x / y | `__div__()` | Division |
| x == y | `__eq__()` | Equality |
| x > y | `__gt__()` | Greater than |
| x >= y | `__ge__()` | Greater than or equal |
| x < y | `__lt__()` | Less than |
| x <= y | `__le__()` | Less than or equal |
| x != y | `__ne__()` | Not equal |
| **Sequence Operators** | | |
| len(x) | `__len__()` | Length of the sequence |
| x in y | `__contains__()` | Does the sequence y contain x? |
| x[key] | `__getitem__()` | Access element *key* of sequence x |
| x[key]=y | `__setitem__()` | Set element *key* of sequence x to value y |
| **General Class Operations** | | |
| x=myClass() | `__init__()` | Constructor |
| print (x), str(x) | `__str__()` | Convert to a readable string |
| | `__repr__()` | Print a Representation of x |
| | `__del__()` | Finalizer, called when x is garbage collected |

Python Special Method Names

```python
class MyClass(object):
        def __init__(self, param1=0):
            ''' constructor, sets attribute value to
            param1, default is 0'''
            print('in constructor')
            self.value = param1

        def __str__(self):
            ''' Convert val attribute to string.'''
            print('in str')
            return 'Val is: {}'.format(str(self.value))

        def __add__(self,param2):
            ''' Perform addition with param2, a MyClass instance.
          Return a new MyClass instance with sum as value attribute'''
            print('in add')
            result = self.value + param2.value
            return MyClass(result)
```

```
>>> a = MyClass(5)
in constructor
>>> b = MyClass(5)
in constructor
>>> print (a + b)
in add
in constructor
in str
Val is: 10
```