Python Object-Oriented Program with Libraries

Unit 1: Object-Oriented Programming

CHAPTER 1: CLASS AND OBJECTS

DR. ERIC CHOU

IEEE SENIOR MEMBER

Class and Objects

LECTURE 1

Introduction

We've seen Python useful for
- Simple Scripts
- Numerical Programming

This Chapter discusses Object Oriented Programming
- Better program design
- Better modularization

What is an Object?

- An object is a collection of data fields and their associated methods.

- What is a class? A class is a kind of data type, just like a string, integer or list. When we create an object of that data type, we call it an instance (object) of a class.

- A class is the design template of the instances (objects).

What are Objects?

- An **object** is a location in memory having a value and possibly referenced by an identifier.

- In Python, every piece of data you see or come into contact with is represented by an object.

- Each of these objects has three components:
  I.   **Identity**
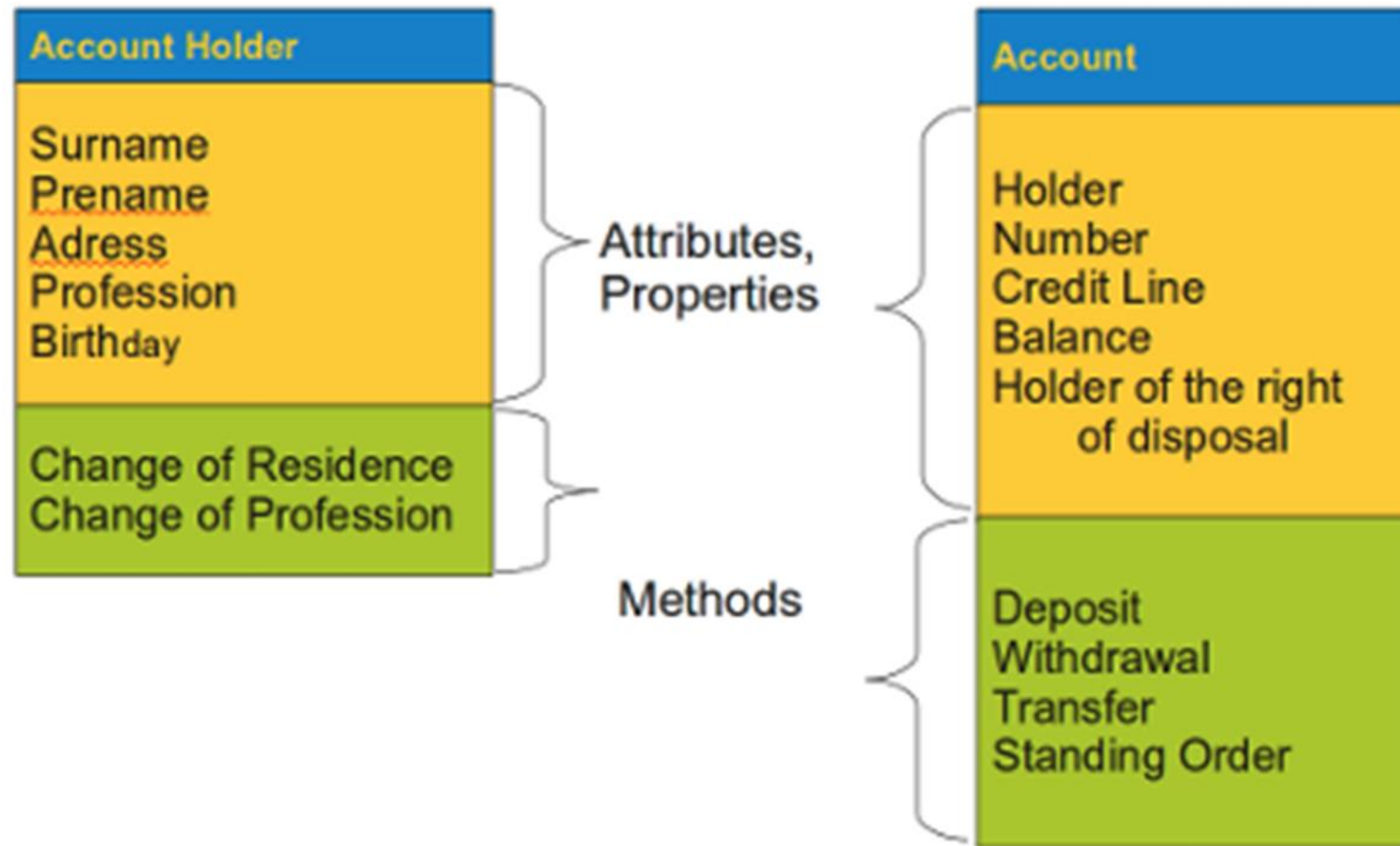 II.   **Type**
  III. **Value**

```
>>>str = "This is a String"
>>>dir(str)
.....
>>>str.lower()
'this is a string'
>>>str.upper()
'THIS IS A STRING'
```
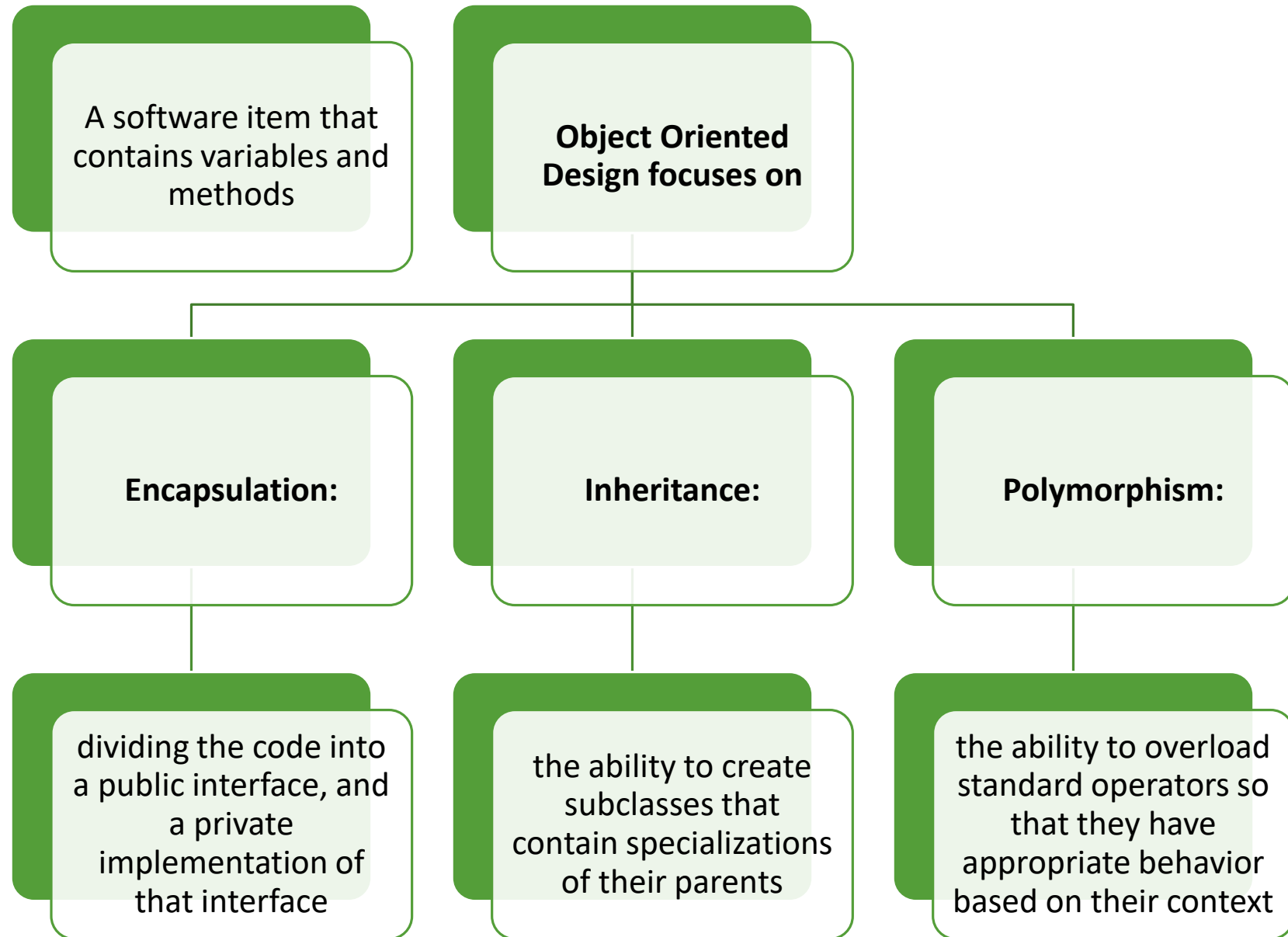
Class and Types

- Classes and types are themselves objects, and they are of type **type**. You can find out the type of any object using the type function:

    type(any_object)

- The data values which we store inside an object are called **attributes**, and

- the functions which are associated with the object are called **methods**.

- We have already used the methods of some built-in objects, like strings and lists.

Python Objects

What is an Object-Oriented Design?

A software item that contains variables and methods

Object Oriented Design focuses on

Encapsulation:

Inheritance:

Polymorphism:

dividing the code into a public interface, and a private implementation of that interface

the ability to create subclasses that contain specializations of their parents

the ability to overload standard operators so that they have appropriate behavior based on their context

Class Definition

LECTURE 2

Defining a Class

Python's class mechanism adds classes with a minimum of new syntax and semantics.

It is a mixture of the class mechanisms found in C++ and Modula-3.

As C++, Python class members are public and have Virtual Methods.

- A basic class consists only of the $class$ keyword.
- Give a suitable name to class.
- Now You can create class members such as data members and member function.

The simplest form of class definition looks like this:

$$class\ className:$$
$$<statement\ 1>$$
$$<statement\ 2>$$
$$.$$
$$.$$
$$.$$
$$<statement\ N>$$

Defining a Class

- As we know how to create a Function.

- Create a function in class MyClass named func().

- Save this file with extension .py

- You can create object by invoking class name.

- Python doesn't have new keyword

- Python don't have new keyword because everything in python is an object.

```
class MyClass:
    """A simple Example of class"""
    i=12
    def func(self):
        return "Hello World
```

```
>>>obj = MyClass(   )
>>> obj.func()
 'Hello World'
```

Defining a Class

```python
1.  class Employee:
2.     "Common base class for all employees"
3.     empCount = 0

4.     def __init__(self, name, salary):
5.         self.name = name
6.         self.salary = salary
7.         Employee.empCount += 1

8.     def displayCount(self):
9.         print("Total Employee %d" % Employee.empCount)

10.    def displayEmployee(self):
11.        print("Name : ", self.name, ", Salary: ", self.salary)
```

Defining a Class
self is similar to this in Java

- The first method __*init*__() is a special method, which is called class constructor or initialization method that Python calls when you create a new instance of this class.

- Other class methods declared as normal functions with the exception that the first argument to each method is self.

- Self: This is a Python convention. There's nothing magic about the word self.

- The first argument in __init__() and other function gets is used to refer to the instance object, and by convention, that argument is called self.

Creating instance objects

"This would create first object of Employee class"
emp1 = Employee("Zara", 2000)
"This would create second object of Employee class"
emp2 = Employee("Manni", 5000)

- To create instances of a class, you call the class using class name and pass in whatever arguments its __*init*__ method accepts.
- During creating instance of class. Python adds the *self* argument to the list for you. You don't need to include it when you call the methods

```
emp1.displayEmployee()
emp2.displayEmployee()
print ("Total Employee %d" % Employee.empCount)
```

Output

Name : Zara ,Salary: 2000
Name : Manni ,Salary: 5000
Total Employee 2

Special Class Attributes in Python

Except for self-defined class attributes in Python, class has some special attributes. They are provided by object module.

Attributes Name	Description
__dict__	Dict variable of class name space
__doc__	Document reference string of class
__name__	Class Name
__module__	Module Name consisting of class
__bases__	The tuple including all the superclasses

Destroying Objects (Garbage Collection):

- Python deletes unneeded objects (built-in types or class instances) automatically to free memory space.

- An object's reference count increases when it's assigned a new name or placed in a container (list, tuple or dictionary).

- The object's reference count decreases when it's deleted with *del*, its reference is reassigned, or its reference goes out of scope.

- You normally won't notice when the garbage collector destroys an orphaned instance and reclaims its space

- But a class can implement the special method *__del__()*, called a destructor

- This method might be used to clean up any non-memory resources used by an instance.

Destroying Objects (Garbage Collection):

```python
a = 40 # Create object <40>
b = a # Increase ref. count of <40>
c = [b] # Increase ref. count of <40>

del a # Decrease ref. count of <40>
b = 100 # Decrease ref. count of <40>
c[0] = -1 # Decrease ref. count of <40>
```

Atom and MoleCule Example Classes

LECTURE 3

Python Classes

Python contains classes that define objects
- Objects are instances of classes

__init__ is the default constructor

```python
class atom:

    def __init__(self, atno, x, y, z):

        self.atno = atno

        self.position = (x,y,z)
```

self refers to the object itself,
like *this* in Java.

Example: Atom class

```python
class atom:
    def __init__(self,atno,x,y,z):
        self.atno = atno
        self.position = (x,y,z)
    def symbol(self):   # a instance method
        return Atno_to_Symbol[self.atno]
    def __repr__(self): # overloads printing
        return "%d %10.4f %10.4f %10.4f" % \
            (self.atno, self.position[0], \
            self.position[1], self.position[2])
```

Python Execution:

```
>>> at = atom(6,0.0,1.0,2.0)
>>> print at
6  0.0000  1.0000 2.0000
>>> at.symbol()
'C'
```

Atom class

- Overloaded the default constructor

- Defined class variables (atno,position) that are persistent and local to the atom object

- Good way to manage shared memory:
  - instead of passing long lists of arguments, encapsulate some of this data into an object, and pass the object.
  - much cleaner programs result

- Overloaded the print operator

We now want to use the atom class to build molecules...

Molecule Class

```python
class molecule:
    def __init__(self,name='Generic'):
        self.name = name
        self.atomlist = []
    def addatom(self,atom):
        self.atomlist.append(atom)
    def __repr__(self):
        str = "This is a molecule named %s\n" % self.name
        str = str+"It has %d atoms\n" % len(self.atomlist)
        for atom in self.atomlist:
            str = str + atom.symbol() + " "+ atom.__repr__()  + '\n'
        return str
```

Using Molecule Class

```
>>> mol = molecule("Water")
>>> at = atom(8,0.,0.,0.)
>>> mol.addatom(at)
>>> mol.addatom(atom(1,0.,0.,1.))
>>> mol.addatom(atom(1,0.,1.,0.))
>>> print (mol)
This is a molecule named Water
It has 3 atoms
O 8  0.000 0.000 0.000
H 1  0.000 0.000 1.000
H 1  0.000 1.000 0.000
```

Note that the print function calls the atoms print function

- Code reuse: only have to type the code that prints an atom once; this means that if you change the atom specification, you only have one place to update.

Demo Program:
MoleCules Project (atom.py+molecule.py)

Go PyCharm!!!

Classes and Their Use

LECTURE 4

Class as Namespaces

A namespace is a module in Python

- At the simplest level, classes are simply namespaces

```python
class myfunctions:

    def exp():

        return 0
```

```
>>> math.exp(1)

2.71828...

>>> myfunctions.exp(1)

0
```

- It can sometimes be useful to put groups of functions in their own namespace to differentiate these functions from other similarly named ones.
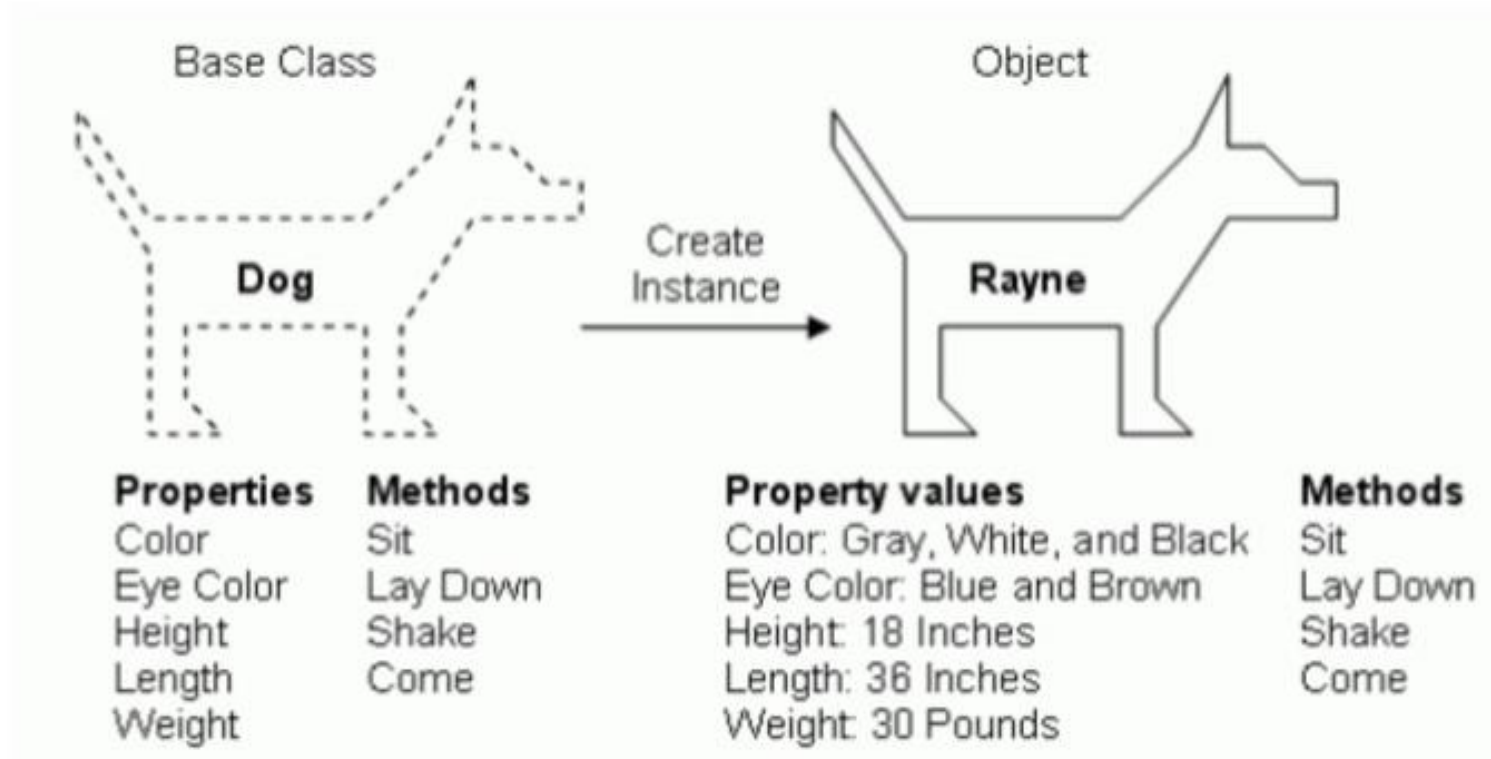
Class as Utility Class

A utility class is a collection of constant variables and utility functions. A famous utility class is **math** class. **random** is another utility class.
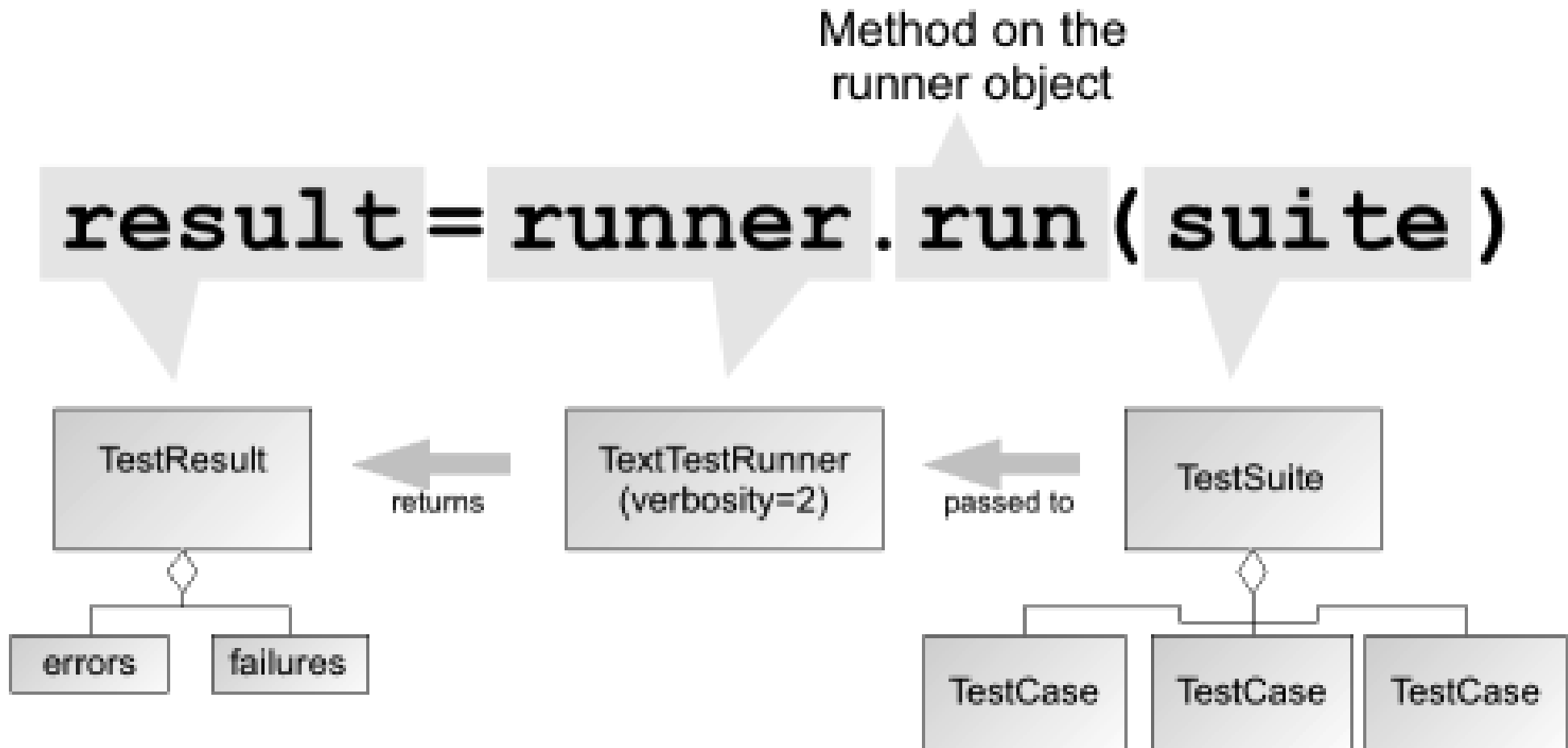
 from utility import some_func

- Best practice is, if you're using only static functions, then just put them in the global namespace of a separate module, it will make things easier.

- Store good utility functions in the class in a separate module (.py)

Defining a Class

Base Class

Dog

Properties
Color
Eye Color
Height
Length
Weight

Methods
Sit
Lay Down
Shake
Come

Create Instance →

Object

Rayne

Property values
Color: Gray, White, and Black
Eye Color: Blue and Brown
Height: 18 Inches
Length: 36 Inches
Weight: 30 Pounds

Methods
Sit
Lay Down
Shake
Come

Class as Data Class

Python **unittest** Module for Test Class

Class Variables and Instance Variables

LECTURE 5

Classes

- **Python classes support a full OO programming model**
  - **Class- and instance-level methods**
  - **Class- and instance-level attributes**
  - **Multiple inheritance**

Class statement introduces
name and any base classes

__init__ is automatically
invoked after object creation

Convention rather than
language dictates that the
current object is called *self*

```python
class Point:
    def __init__(self, x=0, y=0):
        self.x, self.y = x, y
    def is_at_origin(self):
        return self.x == self.y == 0
```

Data vs. Class Attributes

```python
class counter:
    overall_total = 0
            # class attribute
    def __init__(self):
        self.my_total = 0
            # data attribute

    def increment(self):
        counter.overall_total = \

        counter.overall_total + 1
        self.my_total = \
        self.my_total + 1
```

```python
>>> a = counter()
>>> b = counter()
>>> a.increment()
>>> b.increment()
>>> b.increment()
>>> a.my_total
1
>>> a.__class__.overall_total
3
>>> b.my_total
2
>>> b.__class__.overall_total
3
```

Class Scope

- Another important aspect of Python classes is scope.

- The scope of a variable is the context in which it's visible to the program.

- **Global Variables:** Variables that are available everywhere. **Atno_to_Symbol**

- **Class variables:** Variables that are only available to members of a certain class (). **empCount** (shared attribute among instances)

- **Instance variables:** Variables that are only available to particular instances of a class (). self.**atno** (instance owned attributes)

```python
import datetime # we will use this for date objects
class Person:
    def __init__(self, name, surname, birthdate, address, telephone, email):
        self.name = name
        self.surname = surname
        self.birthdate = birthdate
        self.address = address
        self.telephone = telephone
        self.email = email
    def age(self):
        today = datetime.date.today()
        age = today.year - self.birthdate.year
        if today < datetime.date(today.year, self.birthdate.month, self.birthdate.day):
            age -= 1
        return age
def main():
    person = Person("Jane", "Doe",
    datetime.date(1992, 3, 12),
    "No. 12 Short Street, Greenville",
    "555 456 0987",
    "jane.doe@example.com"
    )
    print(person.name)
    print(person.email)
    print(person.age())
if __name__ == "__main__":
    main()
```

person.py

Defining and using a class
Demo Program: person.py

Exercise 1

1. Explain what the following variables refer to, and their scope:

 1. Person

 2. person

 3. surname

 4. self

 5. age (the function name)

 6. age (the variable used inside the function)
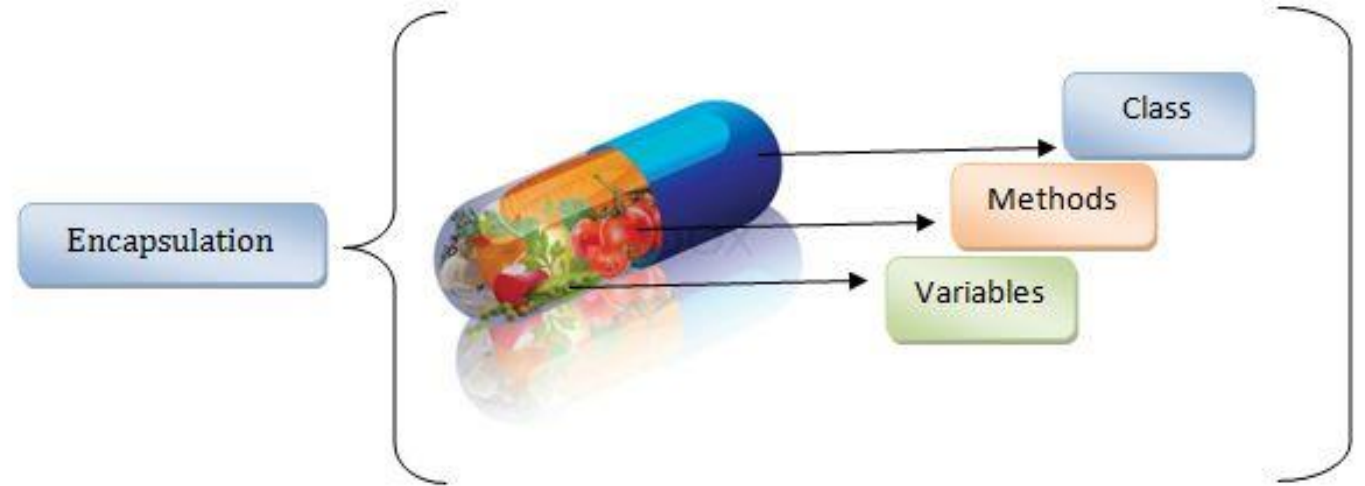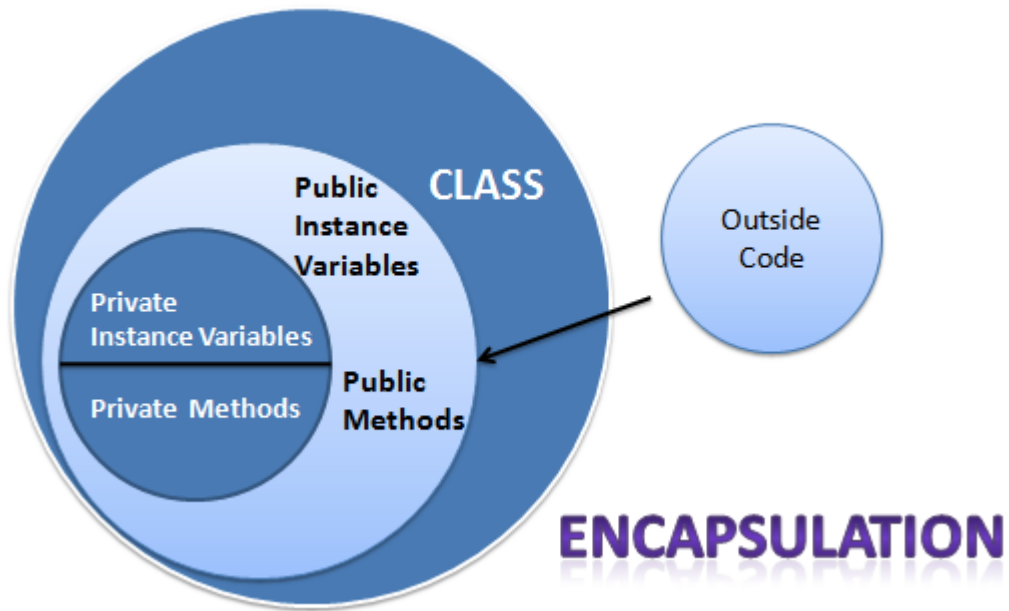
 7. self.email

 8. person.email

Exercise 1.

Ans:
1. **Person** – A global class,
2. **person** – a Person object in main function
3. **name** – an instance variable for Person objects
4. **surname** – an instance variable for Person objects
5. **self** – a default instance reference variable for an object.
6. **age** – a instance member function in a Person objects
7. **age** – a local variable in age member function
8. **self.email** – an instance variable for Person objects
9. **person.email** – an instance variable for the person object.

Instance Attributes

LECTURE 6

Data Encapsulation

```python
import datetime # we will use this for date objects
class Person:
    def __init__(self, name, surname, birthdate, address, telephone, email):
        self.name = name
        self.surname = surname
        self.birthdate = birthdate
        self.address = address
        self.telephone = telephone
        self.email = email
    def age(self):
        today = datetime.date.today()
        age = today.year - self.birthdate.year
        if today < datetime.date(today.year, self.birthdate.month, self.birthdate.day):
            age -= 1
        return age
def main():
    person = Person("Jane", "Doe",
    datetime.date(1992, 3, 12),
    "No. 12 Short Street, Greenville",
    "555 456 0987",
    "jane.doe@example.com"
    )
    print(person.name)
    print(person.email)
    print(person.age())
if __name__ == "__main__":
    main()
```

person.py

__init__ function

- It is important to note that the attributes set on the object in the __init__ function do not form an exhaustive list of all the attributes that our object is ever allowed to have.

- In some languages you must provide a list of the object's attributes in the class definition, placeholders are created for these allowed attributes when the object is created, and you may not add new attributes to the object later.

- **In Python, you can add new attributes**, and even new methods, to an object on the fly. In fact, there is nothing special about the __init__ function when it comes to setting attributes.

Private and Public Attribute

- All Python attributes are public, except that you may double use under prefix to create privacy.

- Dunder: the double under (double underscore, __) For a variable, it will also be private.

```python
class Foo():
    def __init__(self):
        self.__attr = 0

    @property
    def attr(self):
        return self.__attr

    @attr.setter
    def attr(self, value):
        self.__attr = value

    @attr.deleter
    def attr(self):
        del self.__attr
```

age function: adding attribute by function
_age: not real private in Python 3 (Demo Program: person2.py)

We could store a cached age value on the object from inside the age function:

```python
def age(self):
    if hasattr(self, "_age"):
        return self._age

    today = datetime.date.today()

    age = today.year - self.birthdate.year

    if today < datetime.date(today.year, self.birthdate.month, self.birthdate.day):
        age -= 1

    self._age = age
    return age
```

1. If existing, return right away.
2. If not existing, add the attribute to the instance when age function is called.
3. This age function is the accessor method (getter) method

ⓘ Note

Starting an attribute or method name with an underscore (_) is a convention which we use to indicate that it is a "private" internal property and should not be accessed directly. In a more realistic example, our cached value would sometimes expire and need to be recalculated – so we should always use the `age` method to make sure that we get the right value.

Adding attributes on the fly
Python allows it but it is a bad practice.

We could even add a completely unrelated attribute from outside the object:

```
person.pets = ['cat', 'cat', 'dog']
```

- It is very common for an object's methods to update the values of the object's attributes, but it is considered bad practice to create new attributes in a method without initialising them in the **__init__** method.
- Setting arbitrary properties from outside the object is frowned upon even more, since it breaks the object-oriented paradigm.
- The **__init__** method will definitely be executed before anything else when we create the object – so it's a good place to do all of our initialisation of the object's data.
- If we create a new attribute outside the **__init__** method, we run the risk that we will try to use it before it has been initialised.

Adding attributes on the fly

In the age example above we have to check if an _age attribute exists on the object before we try to use it, because if we haven't run the age method before it will not have been created yet. It would be much tidier if we called this method at least once from __init__, to make sure that _age is created as soon as we create the object.

Initialising all our attributes in __init__, even if we just set them to empty values, makes our code less error-prone. It also makes it easier to read and understand – we can see at a glance what attributes our object has.

An __init__ method doesn't have to take any parameters (except self) and it can be completely absent.

Built-in Accessor, Mutator, and Checker for Attributes (Demo Program: person3.py)

Built-in Accessor:
- **getattr(object, "attribute")**   # similar to object.getAttr() in Java

Built-in Mutator:
- **setattr(object, "attribute", new_value)**   # similar to object.setAttr() in Java

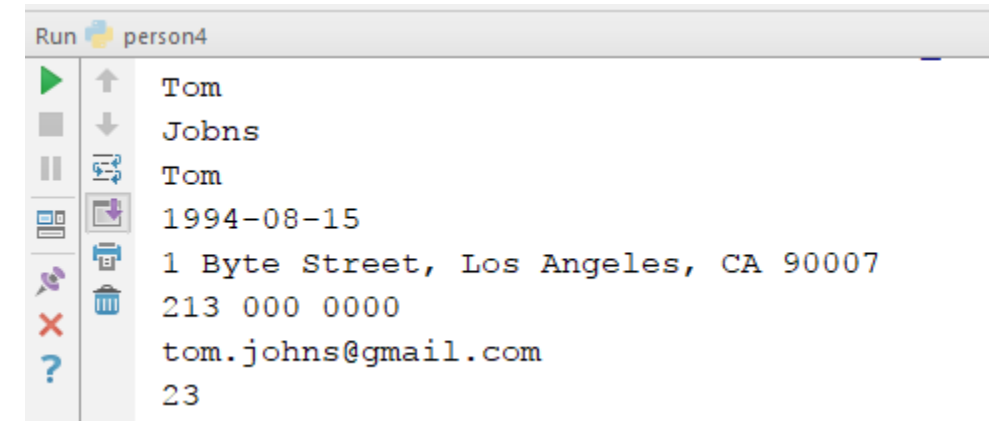**Built-in Checker for an attribute:**
- **hasattr(object, "attribute")**   # no similar function to in Java

Accessor, Mutator in OOP Format

Demo Program: person4.py

Go PyCharm!!!

Run person4

```
Tom
Jobns
Tom
1994-08-15
1 Byte Street, Los Angeles, CA 90007
213 000 0000
tom.johns@gmail.com
23
```

Learning Channel

Class Attributes

LECTURE 7

Class attributes

- All the attributes which are defined on a Person instance are instance attributes – they are added to the instance when the **__init__** method is executed.

- We can, however, also define attributes which are set on the class.

- These attributes will be shared by all instances of that class. In many ways they behave just like instance attributes, but there are some caveats that you should be aware of.

- We define class attributes in the body of a class, at the same indentation level as method definitions (one level up from the insides of methods):

```python
class Person:

    TITLES = ('Dr', 'Mr', 'Mrs', 'Ms')

    def __init__(self, title, name, surname):
        if title not in self.TITLES:
            raise ValueError("%s is not a valid title." % title)

        self.title = title
        self.name = name
        self.surname = surname
```

Class Variables are usually for three purposes:
- Shared property
- Constants (In this example, TITLES is a constant list.)
- Default value to be updated among objects.

self.TITLES uses instance access to access the class variable TITLES

No Real Constant in Python

- In Python language, there is no real language struct to create immutable constant like **final** or **const** in other language.

- You may just declare a variable and never update it. Usually use all capital letters as the variable name.

Access to Class Attributes
Class attributes can be accessed in class' way or instance's way

- As you can see, we access the class attribute **TITLES** just like we would access an instance attribute – it is made available as a property on the instance object, which we access inside the method through the self variable.

- All the **Person** objects we create will share the same **TITLES** class attribute.

- Class attributes are often used to define constants which are closely associated with a particular class. Although we can use class attributes from class instances, we can also use them from class objects, without creating an instance:

```
# we can access a class attribute from an instance
person.TITLES

# but we can also access it from the class
Person.TITLES
```

Note that the class object doesn't have access to any instance attributes – those are only created when an instance is created!

```
# This will give us an error
Person.name
Person.surname
```

Instance Variable Overriding Class Variables

- When we set an attribute on an instance which has the same name as a class attribute, we are overriding the class attribute with an instance attribute, which will take precedence over it.
- If we create two Person objects and call the **mark_as_deceased** method on one of them, we will not affect the other one.
- We should, however, be careful when a class attribute is of a mutable type – because if we modify it in-place, we will affect all objects of that class at the same time.

Class variable used as constant and default value. It shouldn't be changed by any instance.

```python
class Person:
    deceased = False

    def mark_as_deceased(self):
        self.deceased = True
```

person.deceased will access the class copy if mark_as deceased has not been called. If it has been called, person.deceased will access the instance copy which overriding the class copy.

Demo Program: smith.py

Go PyCharm!!!

```python
class Smith:
    surname = "Smith"
    profession = "smith"

    def __init__(self, name, profession=None):
        self.name = name
        if profession is not None:
            self.profession = profession


def main():
    print(Smith.surname)
    print(Smith.profession)
    sm = Smith("Eric", "eric")
    print("Name="+sm.name)               # instance variable
    print("Surname="+sm.surname)         # class varibale default value
    print("Profession="+sm.profession)   # instance variable overriding class variable


if __name__ == "__main__":
    main()
```

Avoid Side-Effects
Demo Program: person5.py

```python
class Person1:
    pets = []

    def add_pet(self, pet):
        self.pets.append(pet)

def main1():
    jane = Person1()
    bob = Person1()
    jane.add_pet("cat")
    print(jane.pets)
    print(bob.pets)  # oops!
```

```python
class Person2:

    def __init__(self):
        self.pets = []
    def add_pet(self, pet):
        self.pets.append(pet)

def main2():
    jane = Person2()
    bob = Person2()

    jane.add_pet("cat")
    print(jane.pets)
    print(bob.pets)
```

```
Run       person5
    C:\Python\Python36\python.exe "C:/Eric_Chou/Pytho
    Using Class Variable for pets list...
    ['cat']
    ['cat']
    Using Instance Variable for pets list...
    ['cat']
    []

    Process finished with exit code 0
```

Class variable always exists. It can be used as default value for constructor and functional parameter default value. Default value is assigned if the parameter is not provided by the caller.

```python
class Person:
    TITLES = ('Dr', 'Mr', 'Mrs', 'Ms')

    def __init__(self, title, name, surname, allowed_titles=TITLES):
        if title not in allowed_titles:
            raise ValueError("%s is not a valid title." % title)

        self.title = title
        self.name = name
        self.surname = surname
```

ValueError Exception raised.

Class and Static Methods

Static Methods

- Simple functions with **no self argument**.

- Nested inside class.

- Work on **class attributes**; not on instance attributes.

- Can be called through both class and instance.

- The built-in function `staticmethod()` is used to create them.

How to create?

```
Class MyClass:
        def my_static_method():

                ...............................................

                ...............................................

        --------Rest of the code------------

        ------------------------------------------------


my_static_method = staticmethod(my_static method)
```

Example

```python
class Spam(object):
    numInstances = 0
    def __init__(self):
        Spam.numInstances = Spam.numInstances + 1
    def printNumInstances():
        print("Number of instances created: ", Spam.numInstances)

    printNumInstances = staticmethod(printNumInstances)

X = Spam()
Y = Spam()
Spam.printNumInstances()
X.printNumInstances()
```

The output will be –

```
'Number of instances created: ', 2
'Number of instances created: ', 2
```

Benefits of Static Methods.

- It localizes the function name in the class scope (so it won't clash with other names in the module).

- It moves the function code closer to where it is used (inside the class statement).

- It allows subclasses to *customize* the static method with inheritance.

- Classes can inherit the static method without redefining it

Class Methods

- Functions that have first argument as **class** name.

- Can be called through both class and instance.

- These are created with `classmethod` inbuilt function.

- These always receive the lowest class in an instance's tree. (See next example)

How to create?

```
Class MyClass:
        def my_class_method(class_var):

                ....................................................

                ....................................................

        --------Rest of the code-----------

        ............................................

        my_class_method = classmethod(my_class_method)
```

Example

```python
class A:
    hi="Hi! I am in class "
    def my_method(cls):
        print(A.hi,cls)
    my_method = classmethod(my_method)

class B(A):
    pass

A.my_method()
B.my_method()
```

Output

```
('Hi! I am in class ', <class __main__.A at 0x004A81B8>)
('Hi! I am in class ', <class __main__.B at 0x00540030>)
```

This behavior of class method make them suitable for data that differs in each Class hierarchy.

Example

```
class Spam:
    numInstances = 0
    def count(cls):  # Per-class instance counters
        cls.numInstances += 1 # cls is lowest class above instance
    def __init__(self):
        self.count() # Passes self.__class__ to count
    count = classmethod(count)

class Sub(Spam):
    numInstances = 0

class Other(Spam):  # Inherits __init__
    numInstances = 0

x = Spam()
y1, y2 = Sub(), Sub()
z1, z2, z3 = Other(), Other(), Other()
print x.numInstances, y1.numInstances, z1.numInstances
print Spam.numInstances, Sub.numInstances, Other.numInstances
```

Output

1 2 3
1 2 3

Static Method and Class Method

- Static Method is for Class object and all instance objects, A simple method written inside class body which doesn't pass any extra object(instance or class) automatically.

- Class Method is to work on class specific data, A method written inside class body which pass class object as its first argument.

	@classmethod	@staticmethod

Similarity	**Signature & Implementation:**
	
	"Can be called via **Instance & class itself**
	e.g. class C:
	    @classmethod
	    def f(cls, arg1, arg2, …): …
	
	Then you can call
	C.f() or C().f()"
		Signature & Implementation
		
		"Can be called via **Instance & class itself**
		e.g. class C:
		    @staticmethod
		    def f(arg1, arg2, …): …
		
		Then you can call
		C.f() or C().f()"

Dissimilarity	**Inheritance support:** It's definition is mutable via inheritance, its definition follows Subclass, not Parent class, via inheritance, can be overridden by subclass	**Inheritance support:** It's definition is immutable via inheritance
	Object argument passing: Implicit object argument passing (cls)-it's obligatory	**Object argument passing:** N/A- neither self (the object instance) nor cls (the class) is implicitly passed
	Form inside class: N/A	**Form inside class:** Only way to write method in a class without (cls,self) implicitly passing the object is being called.
	Similarity to other language: N/A	**Similarity to other language:** Static methods in Python are similar to those found in Java or C++

Instance Method:
Used by a single instance.

Static Method:
Used by a class/all instances under a certain class.

Class Method:
Used by all instance of the same class.

Demo Program:
classNstaticMethod.py

Go PyCharm!!!

```python
class Base:
    attr = "class attribute"
    def static_method():
        print("static method written inside class")
    def class_method(cls):
        print("class method written inside class")
        print(cls.attr)
    #Using staticmethod builtin function
    static_method = staticmethod(static_method)
    #Using classmethod builtin function
    class_method = classmethod(class_method)
```
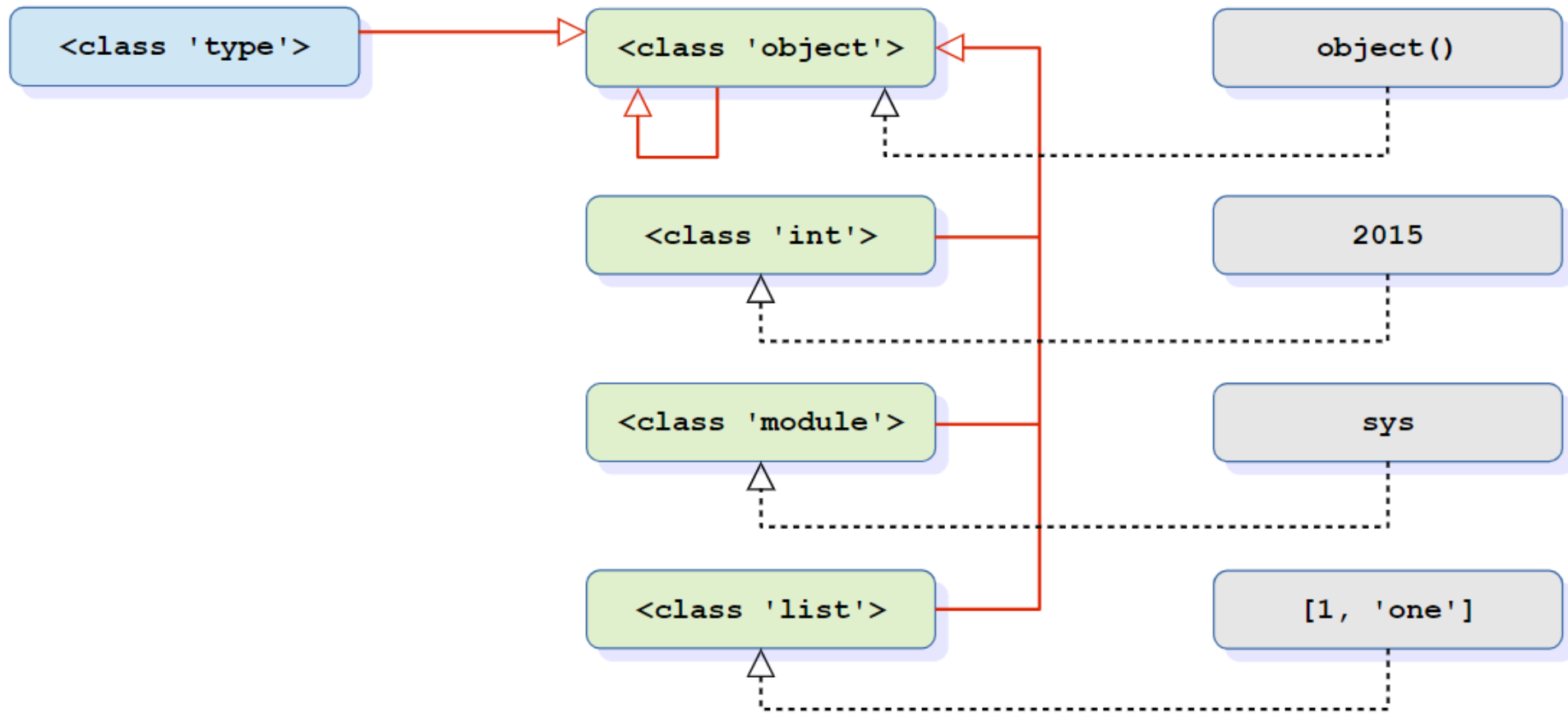
object class – Inspecting objects

object Class is the Top Level Class for Python

- In **Python 2** we have to inherit from object explicitly, otherwise our class will be almost completely empty except for our own custom properties. Classes which don't inherit from object are called "old-style classes", and using them is not recommended. If we were to write the person class in **Python 2** we would write the first line as **class Person(object):**.

- In **Python 3**, **object** is automatically assigned as the top level class. All classes inherits directly or indirectly from **object (metaclass, type)** class. Note: this topic will be explained in Inheritance Chapter.

Every Class Is A Subclass Of object

That's why `isinstance(MyClass(), object)` is always `True`

Inspecting an object

To inspect properties are defined on an object using the **dir** function:

```python
class Person:
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname


    def fullname(self):
        return "%s %s" % (self.name, self.surname)


jane = Person("Jane", "Smith")


print(dir(jane))
```

Now we can see our attributes and our method – but what's all that other stuff?

We will discuss inheritance in the next chapter, but for now all you need to know is that any class that you define has object as its parent class even if you don't explicitly say so – so your class will have a lot of default attributes and methods that any Python **object** has.

Note: dir is just like dir in DOS prompt. This command is used to list all properties.

From object Class

- **Default Constructor**: This is why you can just leave out the **__init__** method out of your class if you don't have any initialisation to do – the default that you inherited from object (which does nothing) will be used instead.

- **Overriding**: If you do write your own **__init__** method, it will override the default method..

- Many default methods and attributes that are found in built-in Python objects have names which begin and end in double underscores, like **__init__** or **__str__**.

- These names indicate that these properties have a special meaning – you shouldn't create your own methods or attributes with the same names unless you mean to override them.

- These properties are usually methods, and they are sometimes called **magic methods**.

- We can use **dir** on any object. You can try to use it on all kinds of objects which we have already seen before, like numbers, lists, strings and functions, to see what built-in properties these objects have in common.

Object Properties
Data Field and Methods

Here are some examples of special object properties:

- **__init__**: the initialisation method of an object, which is called when the object is created.

- **__str__**: the string representation method of an object, which is called when you use the **str** function to convert that object to a string.

- **__class__**: an attribute which stores the the class (or type) of an object – this is what is returned when you use the **type** function on the object.

Object Properties
Data Field and Methods

- **__eq__**: a method which determines whether this object is equal to another. There are also other methods for determining if it's not equal, less than, etc.. These methods are used in object comparisons, for example when we use the equality operator **==** to check if two objects are equal.

- **__add__**: is a method which allows this object to be added to another object. There are equivalent methods for all the other arithmetic operators. Not all objects support all arithemtic operations – numbers have all of these methods defined, but other objects may only have a subset.

- **__iter__**: is a method which returns an iterator over the object – we will find it on strings, lists and other iterables. It is executed when we use the **iter** function on the object.

Object Properties
Data Field and Methods

- **__len__**: a method which calculates the length of an object – we will find it on sequences. It is executed when we use the **len** function of an object.

- **__dict__:** is a dictionary which contains all the instance attributes of an object, with their names as keys. It can be useful if we want to iterate over all the attributes of an object. **__dict__** does not include any methods, class attributes or special default attributes like **__class__**.

Python object Methods

Python Magic Methods	Java Equivalent Methods
__str__(self) # representation	**public String toString()** // representation
__cmp__(self, other) # compare objects (Supports operator overloading for >, <, etc.)	**public int compareTo(that)** // compare objects (Supports implementing Comparable interface)
__add__(self, other) # and sub, mul, div, etc (Supports operator overloading for +, -, *, /, etc)	Note: Java operator overloading is not supported
__eq__(self, other) # check equality	**public boolean equals(other)** // check equality
__iter__(self) # returns an iterator (Supports "for item in items" type of loop)	**public Iterator<T> iterator()** // returns an interator (Supports "for (item : items)" type of loop and implementing Iterable<T> interface)
__del__(self) # clean up	**protected void finalize()** // clean up

ec Learning Channel

Class Decorators

LECTURE 10

Class decorators

- Decorators – functions which are used to modify the behavior of other functions.
- There are some built-in decorators which are often used in class definitions:
- **@classmethod** – share the method among objects of the same class. Equivalent to adding

    MyMethod = classmethod(Mymethod)

- **@staticmethod** – share the method among objects of the same inheritance tree under the subject class. Equivalent to adding

    MyMethod = classmethod(Mymethod)

- **@property** – creating a new data field: use a method to generate a property of an object dynamically, calculating it from the object's other properties.

@classmethod

- Just like we can define class attributes, which are shared between all instances of a class, we can define class methods.

- We do this by using the **@classmethod** decorator to decorate an ordinary method.

- A class method still has its calling object as the first parameter, but by convention we rename this parameter from **self** to **cls**.

- If we call the class method from an instance, this parameter will contain the instance object, but if we call it from the class it will contain the class object.

- By calling the parameter **cls** we remind ourselves that it is not guaranteed to have any instance attributes.

Use of Class Method

- What are class methods good for?

1. There are tasks associated with a class which we can perform using constants and other class attributes, without needing to create any class instances.

2. It is useful to write a class method which creates an instance of the class after processing the input so that it is in the right format to be passed to the class constructor.

Example for @classmethod

```python
class Person:

    def __init__(self, name, surname, birthdate, address, telephone, email):
        self.name = name
        # (...)

    @classmethod
    def from_text_file(cls, filename):
        # extract all the parameters from the text file
        return cls(*params) # this is the same as calling Person(*params)
```

@staticmethod

- A static method doesn't have the calling object passed into it as the first parameter.

- This means that it doesn't have access to the rest of the class or instance at all. We can call them from an instance or a class object, but they are most commonly called from class objects, like class methods.

- If we are using a class to group together related methods which don't need to access each other or any other data on the class, we may want to use this technique.

About Static Methods

- The advantage of using static methods is that we eliminate unnecessary cls or self parameters from our method definitions.

- The disadvantage is that if we do occasionally want to refer to another class method or attribute inside a static method we have to write the class name out in full, which can be much more verbose than using the cls variable which is available to us inside a class method.

```python
class Person:
    TITLES = ('Dr', 'Mr', 'Mrs', 'Ms')
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname
    def fullname(self):  # instance method
        # instance object accessible through self
        return "%s %s" % (self.name, self.surname)
    @classmethod
    def allowed_titles_starting_with(cls, startswith):  # class method
        # class or instance object accessible through cls
        return [t for t in cls.TITLES if t.startswith(startswith)]
    @staticmethod
    def allowed_titles_ending_with(endswith):  # static method
        # no parameter for class or instance object
        # we have to use Person directly
        return [t for t in Person.TITLES if t.endswith(endswith)]

def main():
    jane = Person("Jane", "Smith")
    print(jane.fullname())
    print(jane.allowed_titles_starting_with("M"))
    print(Person.allowed_titles_starting_with("M"))
    print(jane.allowed_titles_ending_with("s"))
    print(Person.allowed_titles_ending_with("s"))

if __name__ == "__main__":
    main()
```

Demo: person7.py

@property
Synthetic Attribute

Sometimes we use a method to generate a property of an object dynamically, calculating it from the object's other properties. Sometimes you can simply use a method to access a single attribute and return it. You can also use a different method to update the value of the attribute instead of accessing it directly. Methods like this are called **getter**s and **setter**s, because they **"get"** and **"set"** the values of attributes, respectively.

In some languages you are encouraged to use getters and setters for all attributes, and never to access their values directly – and there are language features which can make attributes inaccessible except through setters and getters. In Python, accessing simple attributes directly is perfectly acceptable, and writing getters and setters for all of them is considered unnecessarily verbose.

Demo Program: Undecorated Version
person8a.py

Go PyCharm!!!

```python
class Person:
    def __init__(self, height):
        self.height = height

    def get_height(self):
        return self.height

    def set_height(self, height):
        self.height = height


def main():
    print("Undecorated property version")
    jane = Person(153)  # Jane is 153cm tall
    print("When constructed=%d" % jane.get_height())
    jane.height += 1  # Jane grows by a centimetre
    print("After Increment =%d" % jane.get_height())
    jane.set_height(jane.height + 1)  # Jane grows again
    print("After setter    =%d" % jane.get_height())


if __name__ == "__main__":
    main()
```

Annotations for @property

@property   # define an attribute and its getter, as the RHS value, val = attribute
def attribute(self):
    return (self.a + self.b * self.c)   # an attribute is created.

@attribute.setter   # accept the value like a LHS value, attribute = value
def attribute(self, value):
    self.a = value
    self.b = value * 2

@attribute.deleter
def attribute(self):
    del self.name
    del self.surname

Demo Program: Decorated Version 1
person8b.py

Go PyCharm!!!

```python
class Person:
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname

    @property
    def fullname(self):
        return "%s %s" % (self.name, self.surname)


def main():
    print("@property decorated version 1: ")
    jane = Person("Jane", "Smith")
    print(jane.fullname)  # no brackets!


if __name__ == "__main__":
    main()
```

Demo Program: Decorated Version 2
person8c.py

Go PyCharm!!!

```python
class Person:
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname

    @property
    def fullname(self):
        return "%s %s" % (self.name, self.surname)

    @fullname.setter
    def fullname(self, value):
        # this is much more complicated in real life
        name, surname = value.split(" ", 1)
        self.name = name
        self.surname = surname

    @fullname.deleter
    def fullname(self):
        del self.name
        del self.surname

def main():
    print("Fully decorated version")
    jane = Person("Jane", "Smith")
    print("Jane's fulle name when constructed ="+jane.fullname)
    jane.fullname = "Jane Doe"
    print("Jane's full name after LHS setter  ="+jane.fullname)
    print("Jane's name after LHS setter       ="+jane.name)
    print("Jane's surname after LHS setter    ="+jane.surname)

if __name__ == "__main__":
    main()
```

person8c.py

Overriding object Methods

Magic Functions – Dunder functions

double underscore functions

- Magic functions are built-in Functions

- Magic functions are also called as Dunder functions

- Magic functions are member functions for object class

- Magic functions can be overridden (User-redefined function, or overloading of Dunder functions)

Example class: Length

We will demonstrate in the following Length class, how you can overload the "+" operator for your own class. To do this, we have to overload the **__add__** method. Our class contains the **__str__** and **__repr__** methods as well. The instances of the class Length contain length or distance information. The attributes of an instance are self.value and self.unit.

This class allows us to calculate expressions with mixed units like this one:

2.56 m + 3 yd + 7.8 in + 7.03 cm

```python
class Length:

    __metric = {"mm" : 0.001, "cm" : 0.01, "m" : 1, "km" : 1000,
                "in" : 0.0254, "ft" : 0.3048, "yd" : 0.9144,
                "mi" : 1609.344 }

    def __init__(self, value, unit = "m" ):
        self.value = value
        self.unit = unit

    def Converse2Metres(self):
        return self.value * Length.__metric[self.unit]

    def __add__(self, other):
        l = self.Converse2Metres() + other.Converse2Metres()
        return Length(l / Length.__metric[self.unit], self.unit )

    def __str__(self):
        return str(self.Converse2Metres())

    def __repr__(self):
        return "Length(" + str(self.value) + ", '" + self.unit + "')"

if __name__ == "__main__":
    x = Length(4)
    print(x)
    y = eval(repr(x))

    z = Length(4.5, "yd") + Length(1)
    print(repr(z))
    print(z)
```

Overloaded __add__ operator

Overridden __str__ function
(Java-like: toString() method)

Overridden __repr__ function

Demo Program:
length.py

Go PyCharm!!!

Binary Operators

Operator	Method
+	object.__add__(self, other)
-	object.__sub__(self, other)
*	object.__mul__(self, other)
//	object.__floordiv__(self, other)
/	object.__truediv__(self, other)
%	object.__mod__(self, other)
**	object.__pow__(self, other[, modulo])
<<	object.__lshift__(self, other)
>>	object.__rshift__(self, other)
&	object.__and__(self, other)
^	object.__xor__(self, other)
|	object.__or__(self, other)

Extended Assignments

Operator	Method
+=	object.__iadd__(self, other)
-=	object.__isub__(self, other)
*=	object.__imul__(self, other)
/=	object.__idiv__(self, other)
//=	object.__ifloordiv__(self, other)
%=	object.__imod__(self, other)
**=	object.__ipow__(self, other[, modulo])
<<=	object.__ilshift__(self, other)
>>=	object.__irshift__(self, other)
&=	object.__iand__(self, other)
^=	object.__ixor__(self, other)
|=	object.__ior__(self, other)

Unary Operators

Operator	Method
-	object.__neg__(self)
+	object.__pos__(self)
abs()	object.__abs__(self)
~	object.__invert__(self)
complex()	object.__complex__(self)
int()	object.__int__(self)
long()	object.__long__(self)
float()	object.__float__(self)
oct()	object.__oct__(self)
hex()	object.__hex__(self

Comparison Operators

Operator	Method
<	object.__lt__(self, other)
<=	object.__le__(self, other)
==	object.__eq__(self, other)
!=	object.__ne__(self, other)
>=	object.__ge__(self, other)
>	object.__gt__(self, other)

Magic Methods

LECTURE 12

What are Python's magic methods

- They're special methods that you can define to add "magic" to your classes.

- They're everything in object-oriented Python.

- They're always surrounded by double underscores (e.g. __init__ or __lt__).

- They're also not as well documented as they need to be.

Construction and Initialization

```
__new__(cls, [...)
__init__(self, [...)
__del__(self)
```

Customizing initialization and delete

```python
from os.path import join

class FileObject:
    '''Wrapper for file objects to make sure the file gets closed on deletion.'''

    def __init__(self, filepath='~', filename='sample.txt'):

        # open a file filename in filepath in read and write mode

        self.file = open(join(filepath, filename), 'r+')

    def __del__(self):
        self.file.close()
        del self.file
```

Comparison magic methods

```
__cmp__(self, other)
__eq__(self, other)
__ne__(self, other)
__lt__(self, other)
__gt__(self, other)
__le__(self, other)
__ge__(self, other)
```

Customizing comparison behavior

```python
class Word(str):
    '''Class for words, defining comparison based on word length.'''

    def __new__(cls, word):
        # Note that we have to use __new__. This is because str
        # is an immutable # type, so we have to initialize it early (at creation)
        if ' ' in word:
            print "Value contains spaces. Truncating to first space."
            word = word[:word.index(' ')] # Word is now all chars before first space

        return str.__new__(cls, word)
    def __gt__(self, other):
        return len(self) > len(other)
    def __lt__(self, other):
        return len(self) < len(other)
    def __ge__(self, other):
        return len(self) >= len(other)
    def __le__(self, other):
        return len(self) <= len(other)
```

Representing your Classes

```
__str__(self)

__repr__(self)

__format__(self, formatstr)

__hash__(self)

__nonzero__(self)

__dir__(self)

__sizeof__(self)
```

Controlling Attribute Access

```
__getattr__(self, name)

__setattr__(self, name, value)

__delattr__(self, name)

__getattribute__(self, name)
```

Container magic methods

```
__len__(self)
__getitem__(self, key)
__setitem__(self, key, value)
__delitem__(self, key)
__iter__(self)
__reversed__(self)
__contains__(self, item)
__missing__(self, key)
```

Other magic methods

- Unary operators and functions

- Reflected arithmetic operators

- Augmented assignment

- Type conversion magic methods

Summary of magic methods

Magic Method	When it gets invoked
__new__(cls [,...])	instance = MyClass(arg1, arg2)
__init__(self [,...])	instance = MyClass(arg1, arg2)
__cmp__(self, other)	self == other, self > other, etc.
__pos__(self)	+self
__neg__(self)	-self
__invert__(self)	~self
__index__(self)	x[self]
__nonzero__(self)	bool(self)
__getattr__(self, name)	self.name # name doesn't exist
__setattr__(self, name, val)	self.name = val
__delattr__(self, name)	del self.name

Summary of magic methods (2)

Magic Method	When it gets invoked
__getattribute__(self, name)	self.name
__getitem__(self, key)	self[key]
__setitem__(self, key, val)	self[key] = val
__delitem__(self, key)	del self[key]
__iter__(self)	for x in self
__contains__(self, value)	value in self, value not in self
__call__(self [,...])	self(args)
__enter__(self)	with self as x:
__exit__(self, exc, val, trace)	with self as x:
__getstate__(self)	pickle.dump(pkl_file, self)
__setstate__(self)	data = pickle.load(pkl_file)