Lecture1: Python-OOP

(Object Oriented Programming)

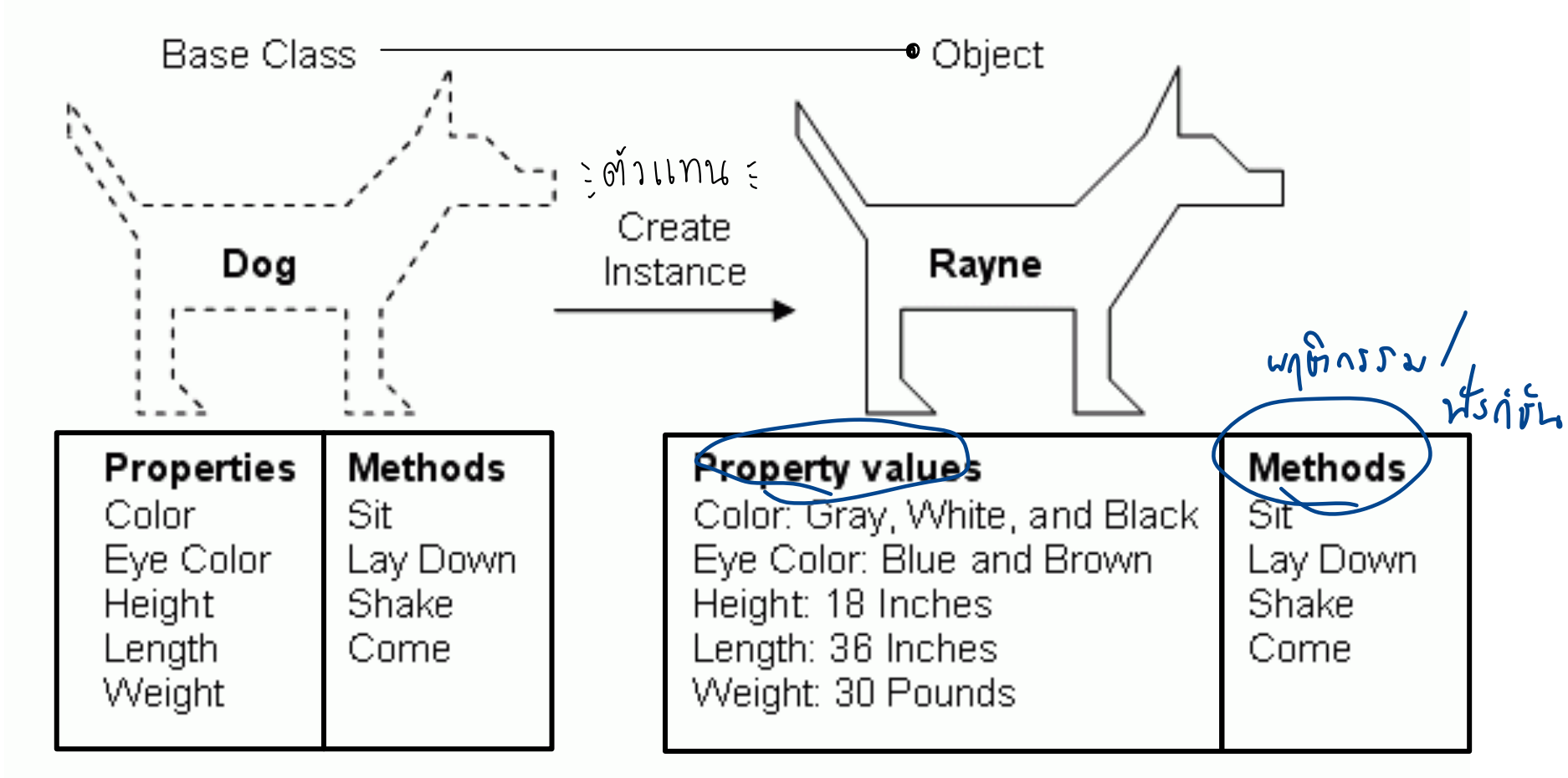
Dr. Prapong Prechaprapranwong

Computer Engineering

King Mongkut's University of Technology Thonburi

What is an **Object**?

Each object is an instance of a class



Why OOP

Object Oriented Programming

error

Robustness: = มั่นคง / เสถียร

capable of handing unexpected inputs that are not explicitly defined for its application

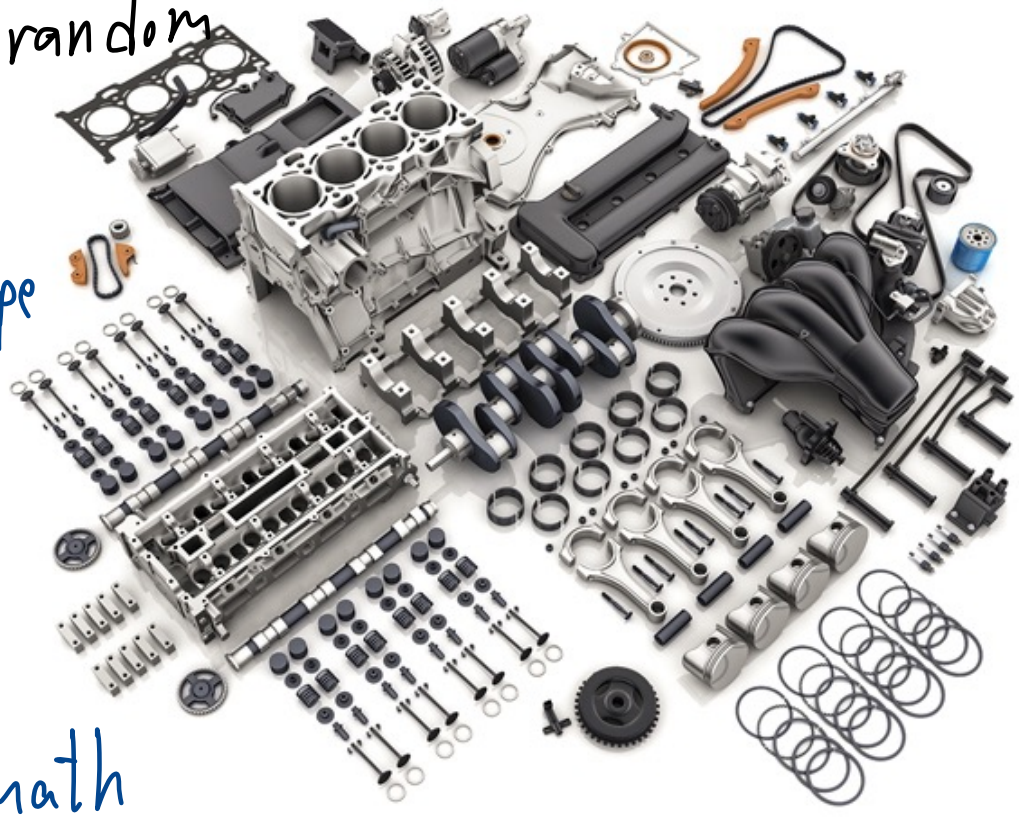
Adaptability (also called evolvability or portability):
the ability of software to run with minimal change on different hardware and OS platforms

Reusability:

the same code should be usable as a component of different system in various application

Object-Oriented Design Principles

Modularity refers to an organizing principle in which different components of a software system are divided into separate functional units.



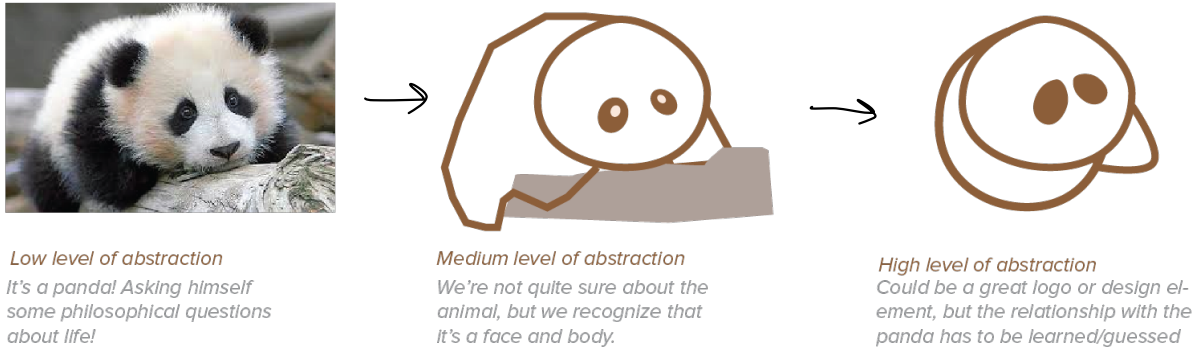
Module is a collection of closely related functions and classes that are defined together ~~in a single~~ file of source code

e.g. *math* module

Object-Oriented Design Principles

Abstraction

is to refine a complicated system down to its most fundamental parts



– describing the parts of a system involves naming them and explaining their functionality –

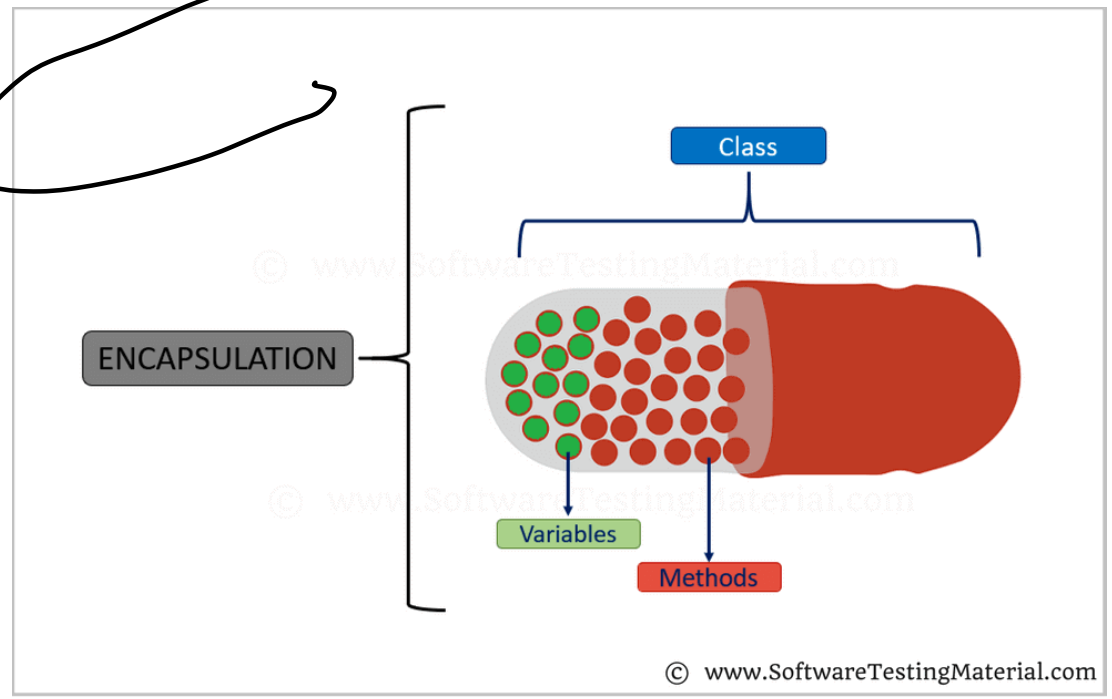
This is [Design Abstraction]

Object-Oriented Design Principles

ยัดไส้ detail แล้วห่อหุ้มด้วย class

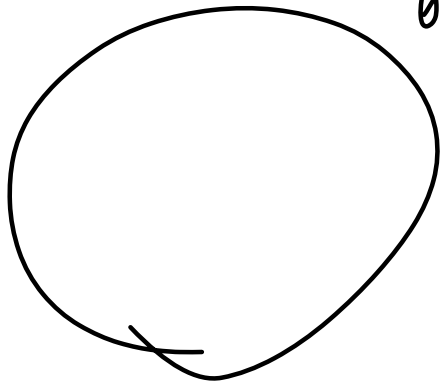
Encapsulation different components of a software system should not reveal the internal details of their respective implementations.

One of the main advantages of encapsulation is that it gives one programmer freedom to implement the details of a component, without concern that other programmers' details



Class objects support two kinds of operations:

- Instantiation
- Attribute references.



```
class ClassName:
    <statement-1>
    ...
    <statement-N>
```

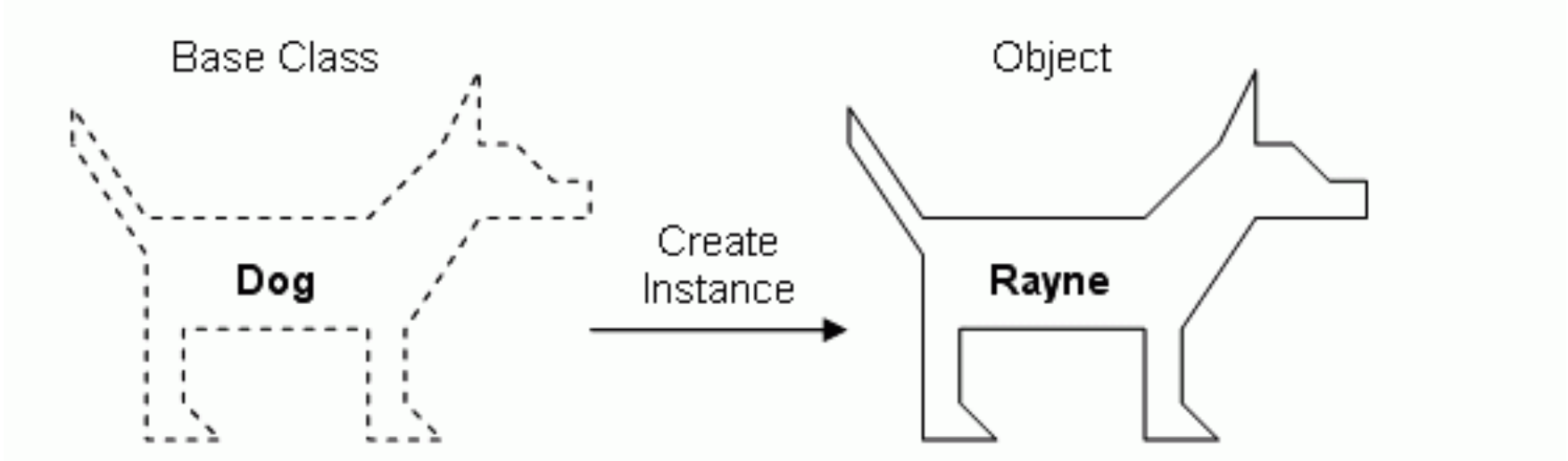
Class instantiation uses function notation.
Just pretend that the class object is a function that returns a new instance of the class

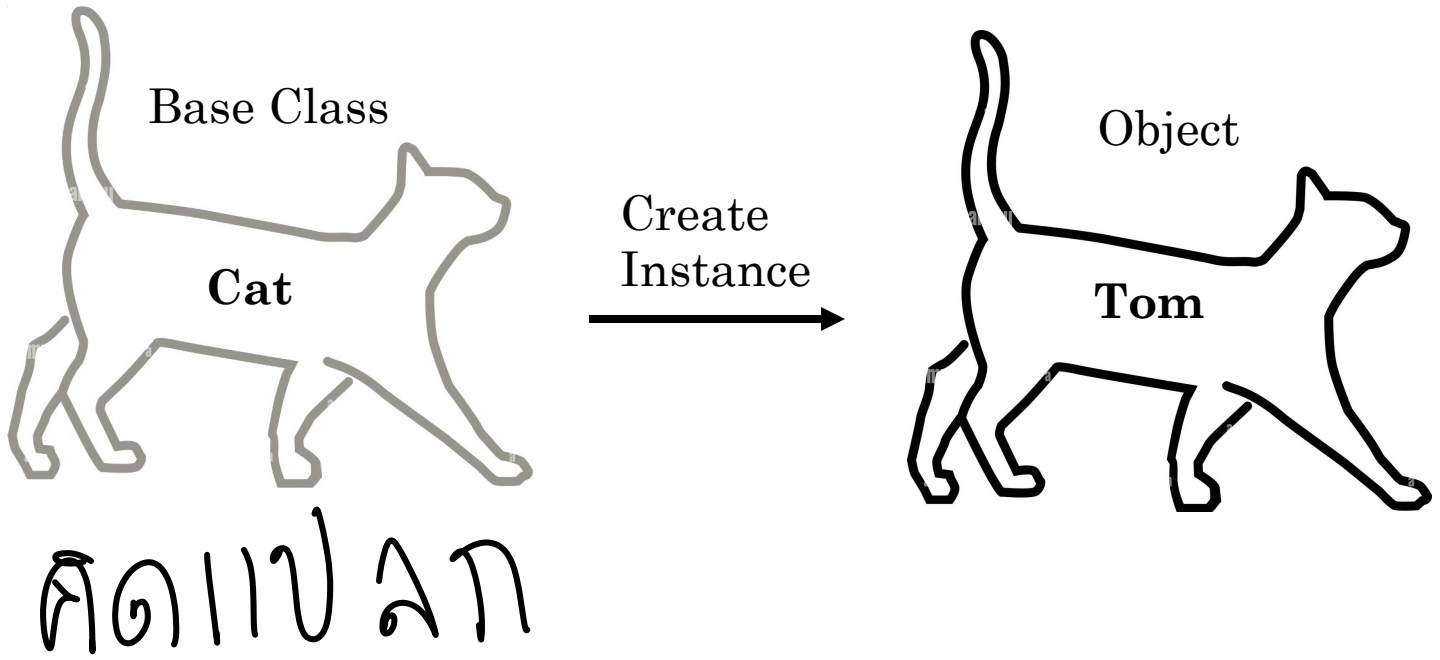
Example:

```
cpe111 = Mycourse( )
```

e.g.

```
class Mycourse:
    """ A simple example class """
    name = "Programming with Data structure"
    def showdetail(self):
        self.course_id = "CPE111"
        print("Course_ID: {}".format(self.course_id))
```


Base Class
Dog
Create
Instance
Object
Rayne

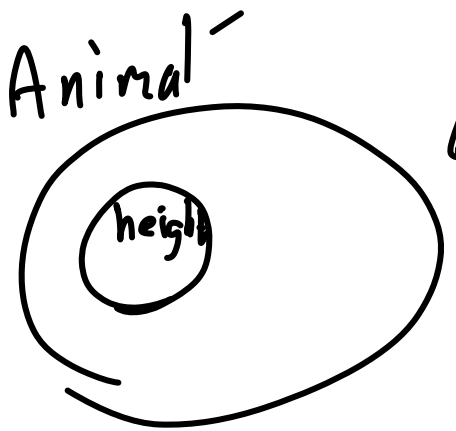

Base Class
Cat
Create
Instance
Object
Tom

ศัตแปลก

Attribute references

use ~~the standard~~ syntax used for all attribute references in ~~Python~~

e.g. ~~object.name~~



Example:

```
cpe111.showDetail()
cpe111.course_id
```

```
e.g.
class Mycourse:
    """A simple example class"""
    name = "Programming with Data structure"
    def showdetail(self):
        self.course_id = "CPE111"
        print("Course_ID: {}".format(self.course_id))
```

The operations understood by instance objects are attribute references
There are two kinds of valid attribute names:

- **Data attributes** → คุณลักษณะ
- **Methods** → พฤติกรรม

class Kaew: attribute →

cute = "True"

Class Animal():

id = "100"

Animal

elephant tiger

id=100 id=110

Data attributes
- Class variables
- Instant variables

can be refer from class directly without instantiation

class Kaew:

```
def __init__(self,bf):
    self.bfname = bf
```

A **method** is a function that "belongs to" an object.

e.g.

```
class Mycourse:
    """A simple example class """
    name = "Programming with Data structure"
    def showdetail(self):
        self.course_id = "CPE111"
        print("Course_ID: {}".format(self.course_id))
```

bfname = "Folk"

age = 20

Test = Kaew("F")

Test

bfname = F

Test.bfname → F

Initialization of a Class Instance

we use a special method called __init__( )

Double underscore

method ลับ ๆ

```
def __init__(self):
    self.data = []
```

e.g.

```
class Mycourse:
    """ A simple example class """
    def __init__(self, id, name):
        self.course_id = id
        self.course_name = name

    def showdetail(self):
        print("Course ID: {}".format(self.course_id))
        print("Course Name: {}".format(self.course_name))
```

Initialization of a Class's Instance

Question:

```
class Complex:
    def __init__(self,realpart,imgpart):
        self.r = realpart
        self.i = imgpart
```

```
>>> x = Complex(3.0,-4.5)
>>> x.r , x.i
```

answer

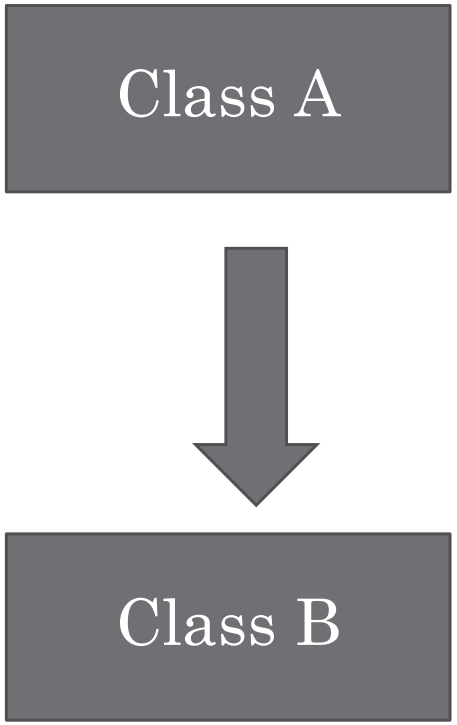
a) (3.0,-4.5)

b) 3.0,-4.5

c) realpart, imgpart

d) (realpart,imgpart)

Inheritance การสืบทอด



Base Class
ต้นฉบับ

Derived Class
ปลายทาง

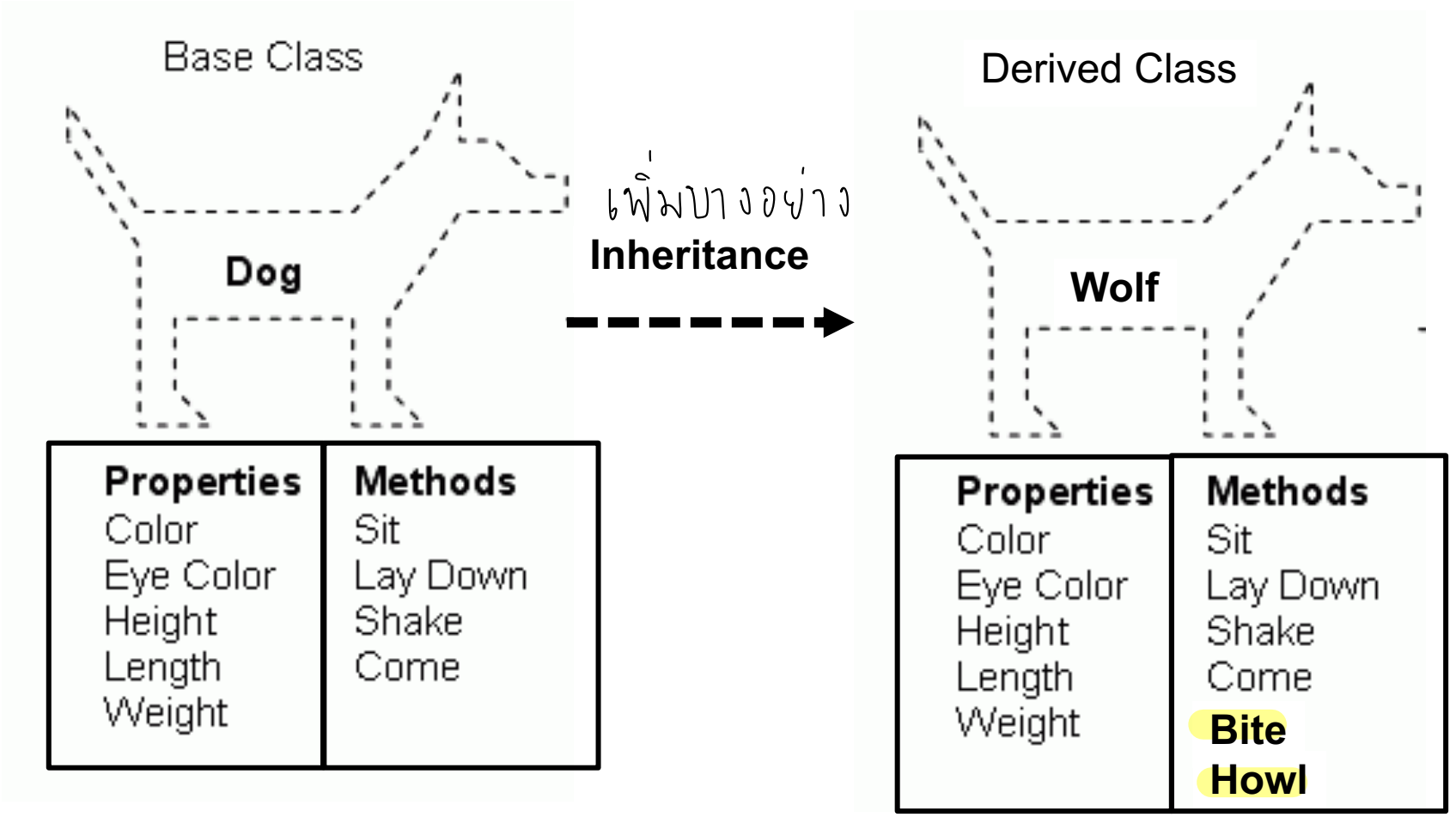
```
class DerivedClassName(BaseClassName):
    <statement-1>
    ...
    <statement-N>
```

overloading vs overriding

- overloading = same name but different parameters
- overriding = same name and same parameters

Overriding เขียนทับ

Derived classes may override methods of their base classes.



Python has two built-in functions that work with inheritance:

- **isinstance()** check an instance's type: **isinstance(obj, type)**
- **issubclass()** check class inheritance: **issubclass(obj, subclass)**

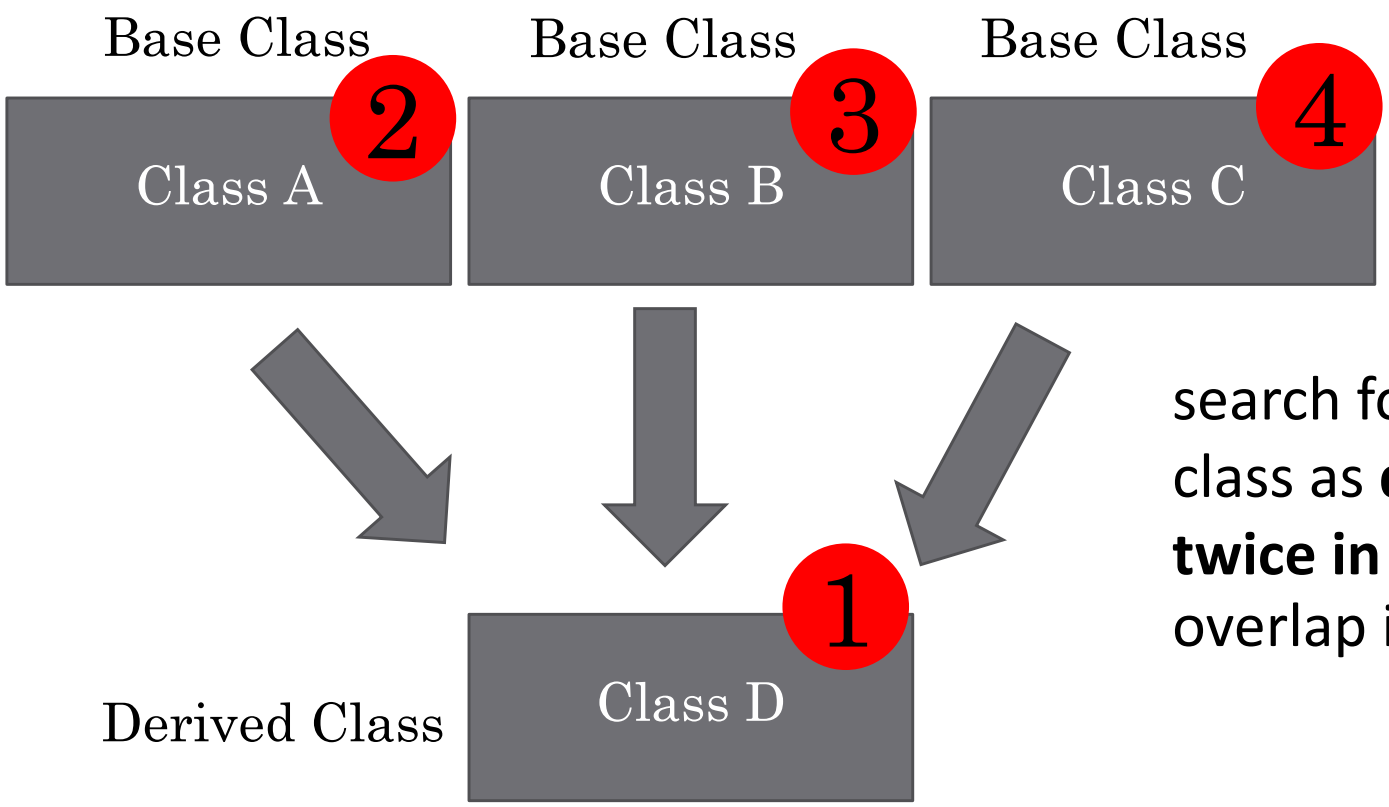
Inheritance

```
e.g.
class Mycourse:
    """ A simple example class """
    def __init__(self, id, name):
        self.course_id = id
        self.course_name = name

    def showdetail(self):
        print("Course ID: {}".format(self.course_id))
        print("Course Name: {}".format(self.course_name))
```

```
class MyOBEM(Mycourse):
    """ A simple example subclass """
```

Multiple Inheritance



search for attributes inherited from a parent class as **depth-first**, **left-to-right**, **not searching twice in the same class** where there is an overlap in the hierarchy.

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    ...
    <statement-N>
```

Access modifier: Public, Protected, Private

- **Public variables /methods** are accessible from outside the class
- The object of the same class is required to invoke a public method
- This arrangement of private instance variables and public methods ensures the principle of data encapsulation.

```
Class Myclass:
    def __init__(self, id, name, num_student ):
        self.course_id = id
        self.course_name = name
        self.num_student = num_student

    def showDetail(self):
        print("Course ID: {}".format(self.course_id))
        print("Course Name: {}".format(self.course_name))
        print("Number of students: {}".format(self.num_student))
```

- **Protected variables/methods** of a class are accessible from within the class and are also available to its sub-classes
- No other environment is permitted access to it
- This enables specific resources of the parent class to be inherited by the child class

a name prefixed with an underscore

e.g. _spam, _data, _name

should be treated as a non-public part of the API (whether it is a function, a method or a data member).

Example:

```
Class Myclass:
    def __init__(self, id, name, num_student ):
        # public
        self.course_id = id
        self.course_name = name
        # protected
        self._num_student = num_student

    def showDetail(self):
        print("Course ID: {}".format(self.course_id))
        print("Course Name: {}".format(self.course_name))
        print("Number of students: {}".format(self._num_student))
```

- Python doesn't have any mechanism that effectively restricts access to any instance variable or method
- Python prescribes a convention of prefixing the name of the variable/method with a **single** or **double underscore** to emulate the behavior of protected and private access specifiers

Private Variables/Methods

- The double underscore __ prefixed to a variable makes it private
- strong suggestion not to touch it from outside the class
- Any attempt to do so will result in an **AttributeError:**

Example:

```
Class Myclass:
    def __init__(self, id, name, num_student, lecture_name ):
        # public
        self.course_id = id
        self.course_name = name
        # protected
        self._num_student = num_student
        # private
        self.__lecture_name = lecture_name

    def showDetail(self):
        print("Course ID: {}".format(self.course_id))
        print("Course Name: {}".format(self.course_name))
        print("Number of students: {}".format(self._num_student))
        print("Lecture Name: {}".format(self.__lecture_name))
```

Setter, Getter Method

Setter Method is method for set the values to data attribute

Example:

```
def setValue(self, value):
    self.__data = value
```

Getter Method is method for get the value from data attribute

Example:

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

Summary: Difference between _, __ and __xx__ in Python

- **One underline in the beginning (Single pre underscore)**

  Python doesn't have real protected methods, so one underline in the beginning of a method or attribute means you shouldn't access this method, because it's not part of the API.
  e.g. _spam, _name, _data

  - ***single pre underscore*** doesn't stop you from accessing the *single pre underscore* variable
  - But ***single pre underscore*** effects the names that are imported from the module

Example:

```
## filename:- my_functions.py
def func ( ):
    return "datacamp"
def _private_func ( ):
    return 7
```

Test Code example:

```
>>> from my_functions import *
>>> func ( )
'data camp'
>>> _private_func ( )
Trackback(most recent call last):
   File "<stdin>", line 1, in <module>
NameError: name '_private_func' is not defined
```

!! Avoid error by importing the module normally

```
>>> import my_functions
>>> my_functions.func( )
'datacamp'
>>> my_functions._private_func( )
7
```


Ref: https://www.datacamp.com/community/tutorials/role-underscore-python#SLU

Summary: Difference between _, __ and __xx__ in Python

- **Two underlines in the beginning (Double pre underscore)**
  use for avoid the method to be overridden by a subclass.
  e.g. __methodA( ), __bite( )

Double Pre Underscores are used for the name mangling

Example: Two underlines in the beginning

Question:

```
class A(object):
    def __method(self):
        print ('I am a method in A')

    def method(self):
        self.__method( )

>>> a = A( )
>>> a.method( )
```

I'm a method in A

```
class B(A):
    def __method(self):
        print ('I am a method in B')

>>> b = B( )
>>> b.method( )
```

answer:
a) I am a method in A
b) I am a method in B

Summary: Difference between _, __ and __xx__ in Python

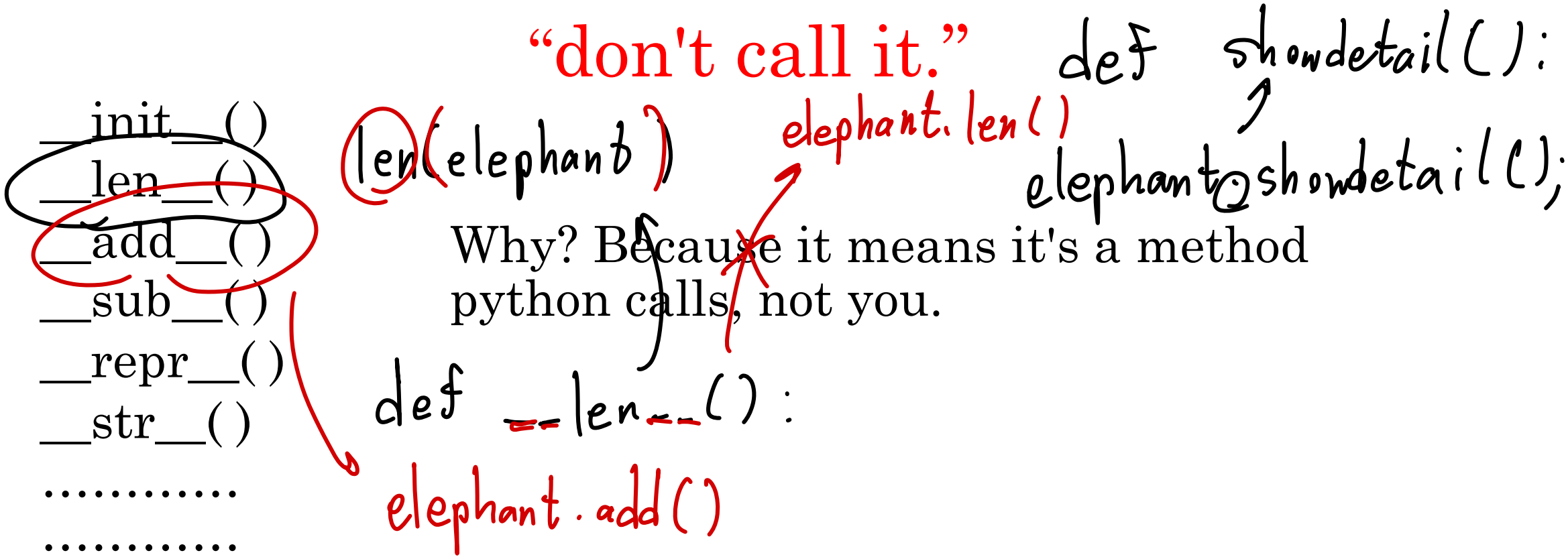
- **Two underlines in the beginning and in the end**

  When you see a method like __this__, the rule is simple:

  "don't call it."

  - __init__()
  - __len__()
  - __add__()
  - __sub__()
  - __repr__()
  - __str__()
  - .....
  - .....

  Why? Because it means it's a method python calls, not you.



http://igorsobreira.com/2010/09/16/difference-between-one-underline-and-two-underlines-in-python.html

Iterators

An iterator is an object that contains a countable number of values.

To create an object/class as an iterator you must implement the methods __iter__() and __next__() to your object.

like __init__() for iterator

An **iterator** is an object that implements **next**, which is expected to return the next element of the iterable object that returned it, and raise a **StopIteration** exception when no more elements are available

Example of iterable objects (not iterator): list, tuple, dictionary, str, file

```
for element in [1, 2, 3]:
    print(element)

for element in (1, 2, 3):
    print(element)

for key in {'one':1, 'two':2}:
    print(key)

for char in "123":
    print(char)

for line in open("myfile.txt"):
    print(line, end='')
```

Behind the scenes

- The for statement calls iter() on the container object.
- The function returns an iterator object that defines the method __next__() which accesses elements in the container one at a time.
- When there are no more elements, __next__() raises a StopIteration exception which tells the for loop to terminate.

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
```

```
>>> next(it)
```

answer:

a) a
b) b
c) c
d) 'a'
e) 'b'
f) 'c'

Generators

Generators is subset of Iterators

Generators are a simple and powerful tool for creating iterators. They are written like regular functions but use the **yield** statement whenever they want to return data.

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]
```

```
>>> data = [ 1,2,3,4,5]
>>> rdata = reverse(data)
>>> next(rdata)
```

answer:

a) 5
b) 4
c) 3
d) 2
e) 1