SI100B Introduction to Information Science and Technology
Python Programming

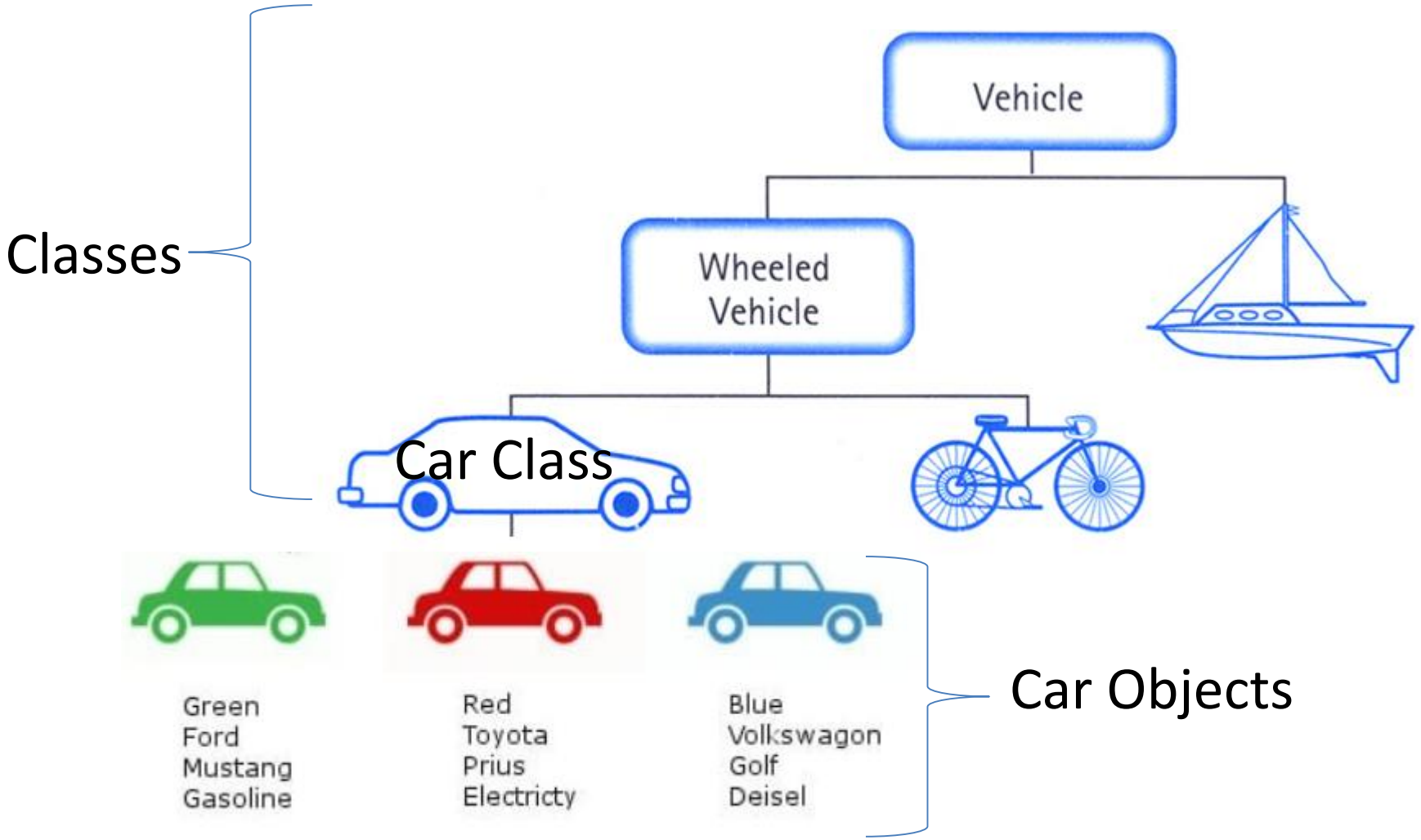
张海鹏 Haipeng Zhang
School of Information Science and Technology
ShanghaiTech University

Tutorial

- 8PM Friday, TC201
- **Exception Handling, Class, Iterator & Generator**
- **Bring your laptops, will give hands-on examples**
- Some related to HW2

Learning Objectives

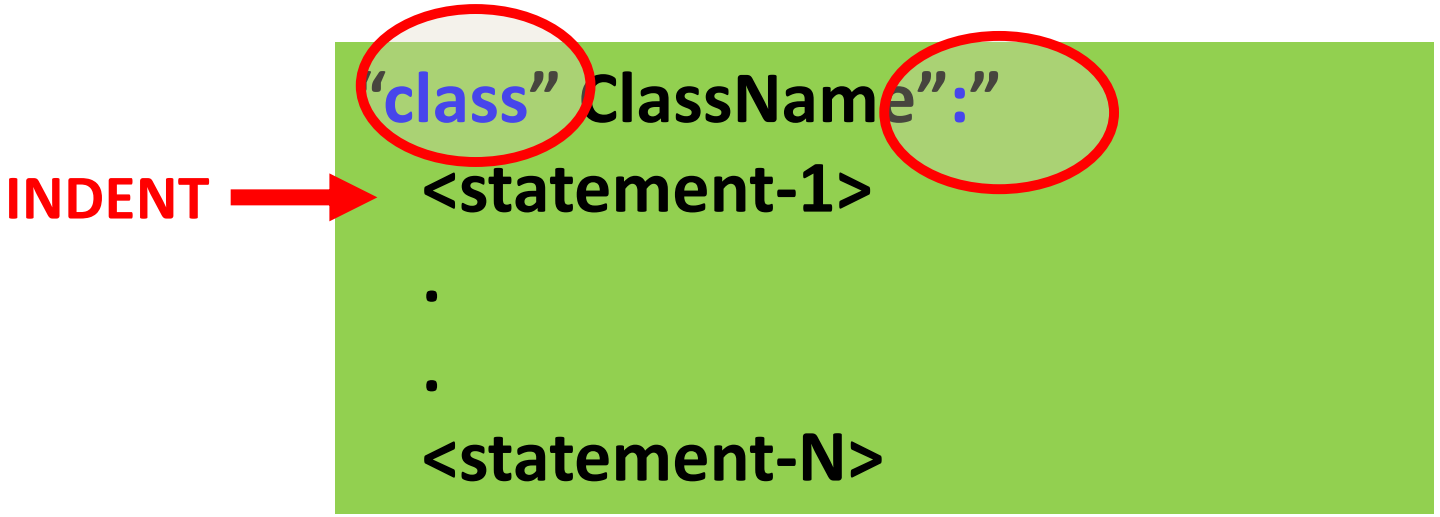
- Classes and objects
  - Instance variables
  - Class variables
  - Instance methods
  - Class methods
  - Access
  - Private and public attributes
  - Special method names
- Inheritance


Classes
Vehicle
Wheeled Vehicle
Car Class
Green
Ford
Mustang
Gasoline
Red
Toyota
Prius
Electricty
Blue
Volkswagon
Golf
Deisel
Car Objects

Object-Oriented Programming

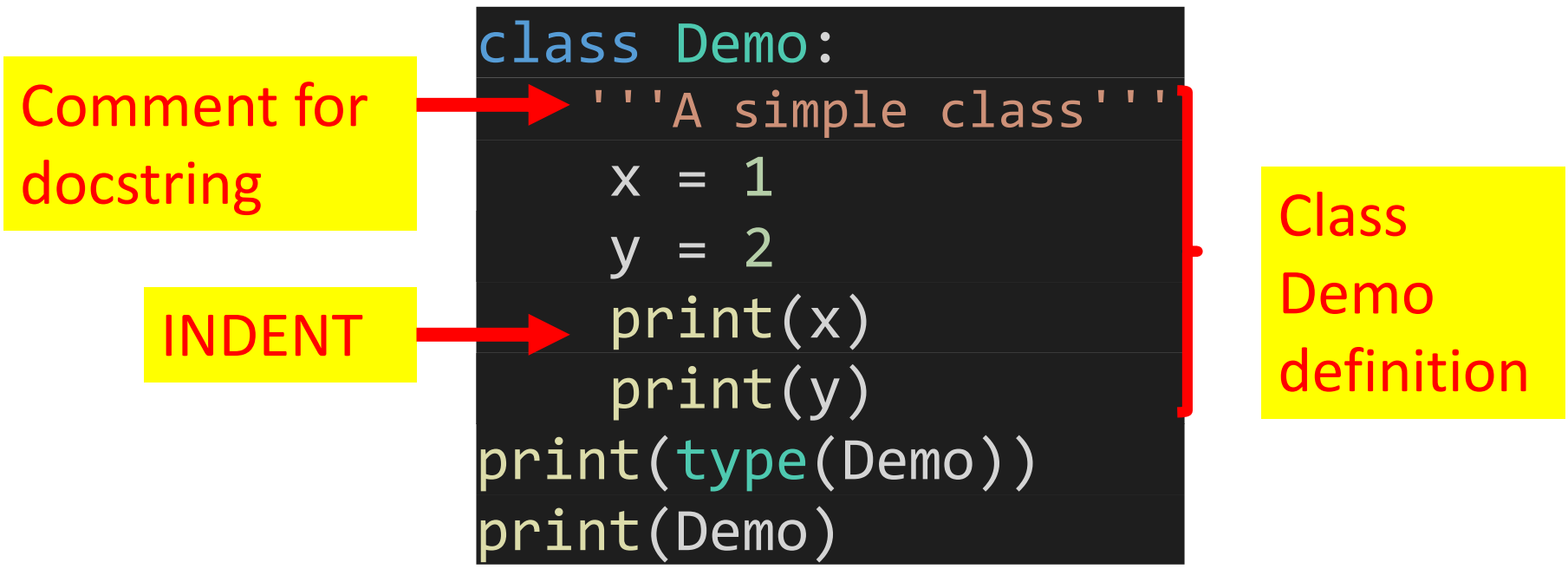
- **In OOP, code and data are combined into a single entity called a class**
  - **each instance of a given class is an object of that class type**
  - **a class is like an object constructor, or a "blueprint" for creating objects.**
- **Principles of Object-Oriented Programming**
  - **Encapsulation**
    - hides the implementation details of a class from other objects
  - **Inheritance**
    - form new classes using classes that have already been defined, and keep some characteristics
  - **Polymorphism**
    - using an operator or function in different ways for different data input
- **Python is object-oriented**
  - **Everything in Python is an object (excluding keywords)**

Class Definition



- A class definition starts with the keyword class
- The first character of the class name is usually UPPERCASED
- Then, the colon :
- The class body consists of a sequence of statements and/or function definitions, organized via INDENT

The Smallest Class



The Smallest Class

```
class Demo:
    '''A simple class'''
    x = 1
    y = 2
    print(x)
    print(y)
print(type(Demo))
print(Demo)
```

Output

```
1
2
<class 'type'>
<class '__main__.Demo'>
```

- **Four statements are executed when entering class Demo**
- **Demo is an object/instance of the class type**
- **The object Demo is in the global scope called __main__**

Instance objects

Class *instantiation* uses function notation:

Obj = ClassName(parameters)

```
print(type(Demo))
print(Demo)
d = Demo()
print(d)
print(type(d))
print(d.y)
print(d.x)
```

Output

```
<class 'type'>
<class '__main__.Demo'>
<__main__.Demo object at 0x02CA8490>
<class '__main__.Demo'>
2
1
```

Instance objects

We can check whether an object is an instance of a class

isinstance(object, class)

```
class Demo:
    pass
d = Demo()
print(isinstance(d,Demo))
print(isinstance(Demo,type))
```

Output

```
Yes
Yes
```

Demo is a class object vs d is an instance object

Constructor __init__

- All the classes have an implicit instance method __init__ as constructor (inherited from the class object)
- The __init__() function is called automatically every time the class is being used to create a new instance object
- It is called after the instance has been created, but before it is returned to the caller
- The arguments are those passed to the class constructor expression
- The first parameter of __init__ is the instance object
- One can override __init__ in user-defined classes for initialization

Constructor __init__

- Create a class named Person, use the __init__() function to assign values for name and age:

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

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

Output

```
John
36
```

self Parameter

- The first parameter of all instance methods is bound to the instance object
- The name of the first parameter can be any identifier
- But, we usually use self

```
class A:
    def __init__(self,v=0):
        self.value = v
a = A(1)
print(a.value)
```

self Parameter

```
class A:
    def __init__(self,v=0):
        self.value = v
a = A(1)
```

They are same

```
class A:
    def __init__(x,v=0):
        x.value = v
a = A(1)
```

Attributes

- Attributes of an instance
  - instance variables: are for data unique to each instance
  - instance methods: are for manipulation of instance data
- Attributes of a class
  - class variables: are for data shared by all instances of the class
  - class methods: are for manipulation of class data
- All can be dynamically added/removed in Python
- It is better to use different names for class attributes and instance attributes

Instance variables

– All variables defined via

obj.var = expr

are instance variables var of the object obj

```
class A:
    def __init__(self,v=0):
        self.value = v
a = A(1)
```

value is an instance variable of the object bound to the name self

Instance variables

```
class Car:
    def __init__(self, c):
        self.color = c

car1 = Car("Red")
car2 = Car("Blue")
car1.name = "QQ"
car2.name = "BYD"
print(car1.color,car1.name)
print(car2.color,car2.name)
```

Instance variables are dynamically added into objects

Output

```
Red QQ
Blue BYD
>>>
```

Instance variables

```
class Car:
    def __init__(self, c):
        self.color = c

car1 = Car("Red")
car2 = Car("Blue")
car1.name = "QQ"
print(car1.color,car1.name)
print(car2.color,car2.name)
```

Added instance variable is specific to the object

Output

```
Red QQ
Traceback (most recent call last):
  File "D:\Test\fib.py", line 9, in <module>
    print(car2.color,car2.name)
AttributeError: 'Car' object has no attribute 'name'
```

Instance variables

```
class Car:
    def __init__(self, c):
        self.color = c

car1 = Car("Red")
car2 = Car("Blue")
car1.name = "QQ"
car2.name = "BYD"
del car2.color
print(car1.color,car1.name)
print(car2.color,car2.name)
```

An instance variable is deleted from the object

Output

```
Red QQ
Traceback (most recent call last):
  .....
AttributeError: 'Car' object has no attribute 'color'
```

Instance variables

It is recommended to initialize all the instance variables in the constructor __init__

Class variables

– All variables defined via

var = expr

in the class definition are class variables of the class object, shared by all its instances.

```
class A:
    value = "classvariable"
    def __init__(self):
        pass
a = A()
print(a.value)
print(A.value)
```

```
classvariable
classvariable
```

Class variables

– All variables defined via

var = expr

in the class definition are class variables of the class object, shared by all its instances.

```
class A:
    value = "classvariable"
    def __init__(self):
        self.value = "instancevariable"
a = A()
print(a.value)
print(A.value)
```

instancevariable

classvariable

Class variables

```
class A:
    value = "classvariable"
    def __init__(self):
        self.value = "instancevariable"
    value2 = "classvariable2"

A.value3 = "AddValue"
print(A.value)
print(A.value2)
print(A.value3)
```

Defined in different places

A class variable is dynamically added

Output

```
classvariable
classvariable2
AddValue
>>>
```

Class variables

```
class A:
    value = "classvariable"
    def __init__(self):
        self.value = "instancevariable"

print(A.value)
del A.value
print(A.value)
```

A class variable is dynamically deleted

Output

```
classvariable
Traceback (most recent call last):
  ....
AttributeError: type object 'A' has no attribute 'value'
```

Class variables

- It is recommended to initialize all the class variables at the beginning of the class definition

```
class A:
    value = "Good"
    def __init__(self):
        self.value = "instancevariable"
    value2 = "Bad"

A.value3 = "Worse"
```

Learning Objectives

- Classes and objects
  - Instance variables
  - Class variables
  - Instance methods
  - Class methods
  - Access
  - Private and public attributes
  - Special method names
- Inheritance

Instance methods

– The first parameter of instance methods are the instance object, i.e., self (you may use other names)

```
class A:
    def __init__(self,v=0):
        self.value = v
    def GetValue(self):
        return self.value
a = A(1)
print(a.GetValue())
```

__init__ and GetValue are instance methods

Instance methods

SetValue is a normal function, not an instance method

```
class A:
    def __init__(self,v=0):
        self.value = v
    def GetValue(self):
        return self.value
def SetValue(self,v):
    self.value = v
a = A(1)
a.SetValue = SetValue
print(a.SetValue)
a.SetValue(a,2)
```

a.SetValue is an instance variable, not an instance method

Called with the object as first argument

Output

```
<function SetValue at 0x03032150>
>>>
```

Instance methods

```
import types
class A:
    def __init__(self,v=0):
        self.value = v
    def GetValue(self):
        return self.value
def SetValue(self,v):
    self.value = v
a = A(1)
a.SetValue = types.MethodType(SetValue, a)
print(a.SetValue)
a.SetValue(2)
```

import module types

Dynamically add **SetValue** as an instance method of the object **a**

Direct call SetValue via the object a

Output

```
<bound method SetValue of
<__main__.A object at 0x02FE8470>>
```

Instance methods

```
class A:
    def __init__(self,v=0):
        self.value = v
    def GetValue(self):
        return self.value
a = A(1)
print(a.GetValue)
del A.GetValue
print(a.GetValue)
```

An instance method is dynamically deleted

Output

```
<bound method GetValue of <__main__.A object at
0x035984F0>>
Traceback (most recent call last):
.....
AttributeError: 'A' object has no attribute ‘GetValue'
```

Instance methods

- It is recommended to define all the instance methods in class definition

Learning Objectives

- Classes and objects
  - Instance variables
  - Class variables
  - Instance methods
  - Class methods
  - Access
  - Private and public attributes
  - Special method names
- Inheritance

Class methods

- The first parameter of class methods is the class object, i.e., cls

```
class A:
    ClassValue = 1
    def __init__(self,v=0):
        self.value = v
    @classmethod
    def GetClassValue(cls):
        return cls.ClassValue

print(A.GetClassValue)
```

@classmethod is a Decorator claiming that the function defined following this is a class method

Output

```
<bound method A.GetClassValue of
<class '__main__.A'>>
```

cls Parameter

- The first parameter of all class methods is bound to the class object
- The name of the first parameter can be any identifier
- But, we usually use cls

Class methods

```
import types
class A:
    ClassValue = 1
    def __init__(self,v=0):
        self.value = v
    @classmethod
    def GetClassValue(cls):
        return cls.ClassValue
def SetClassValue(cls,v):
        cls.ClassValue = v
A.SetClassValue = types.MethodType(SetClassValue,A)
print(A.SetClassValue)
```

Dynamically add **SetClassValue** as a class method of the class **A**

Output

```
<bound method SetClassValue of
<class '__main__.A'>>
```

Class methods

```
class A:
    ClassValue = 1
    def __init__(self,v=0):
        self.value = v
    @classmethod
    def GetClassValue(cls):
        return cls.ClassValue
print(A.GetClassValue)
del A.GetClassValue
print(A.GetClassValue)
```

GetClassValue of the class **A** is deleted

Output

```
<bound method A.GetClassValue of <class
'__main__.A'>>
Traceback (most recent call last):
  ...
AttributeError: type object 'A' has no attribute 'GetClassValue'
```

Learning Objectives

- Classes and objects
  - Instance variables
  - Class variables
  - Instance methods
  - Class methods
  - Access
  - Private and public attributes
  - Special method names
- Inheritance

Access

- **Instance variables: are accessed via**

  object.var

- **Instance methods: are accessed via**

  object.f(p_1,…,p_n)

- **Class variables: are accessed via**

  class.var

- **Class methods: are accessed via**

  class.f(p_1,…,p_n)

It is better to use these forms

Access

- **Instance variables: are accessed via**

  object.var

- **Instance methods: are accessed via**

  object.f(p_1,…,p_n) or class.f(object,p_1,…,p_n)

- **Class variables: are accessed via**

  class.var or object.var

- **Class methods: are accessed via**

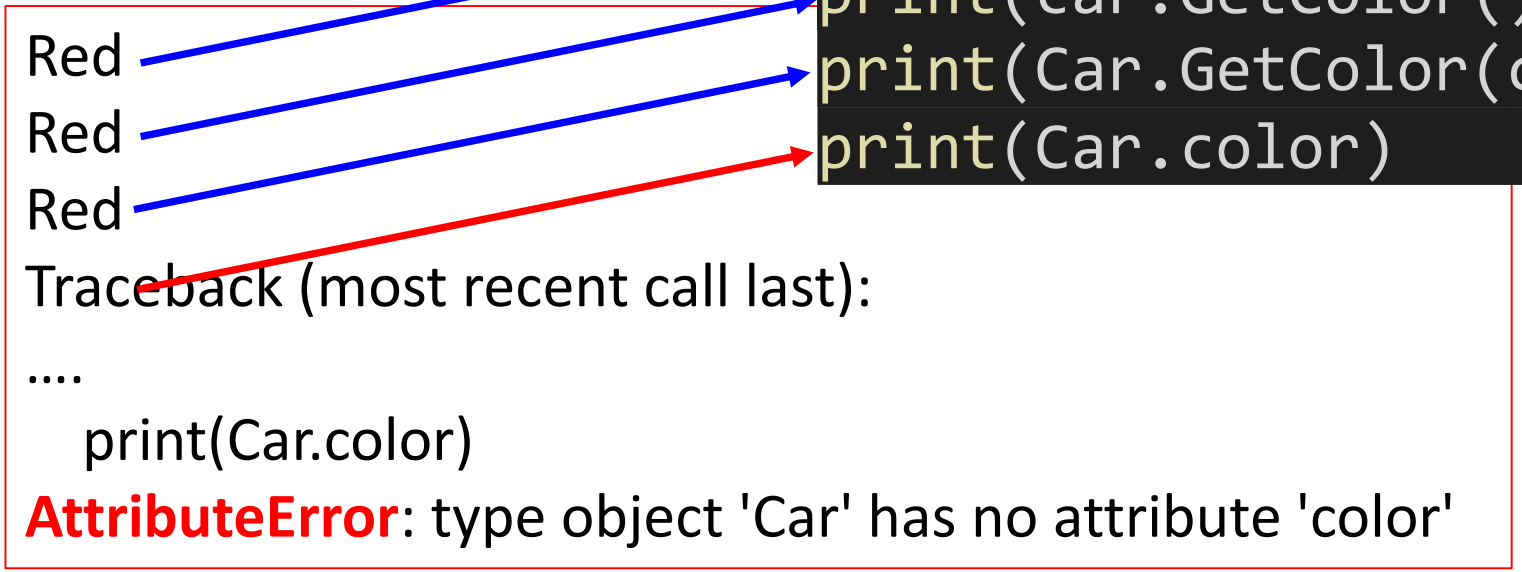
  class.f(p_1,…,p_n) or object.f(p_1,…,p_n)

Assuming all attributes are distinct

Access Instance attributes

```
class Car:
    def __init__(self, c):
        self.color = c
    def GetColor(self):
        return self.color
car = Car("Red")
print(car.color)
print(car.GetColor())
print(Car.GetColor(car))
print(Car.color)
```

Output

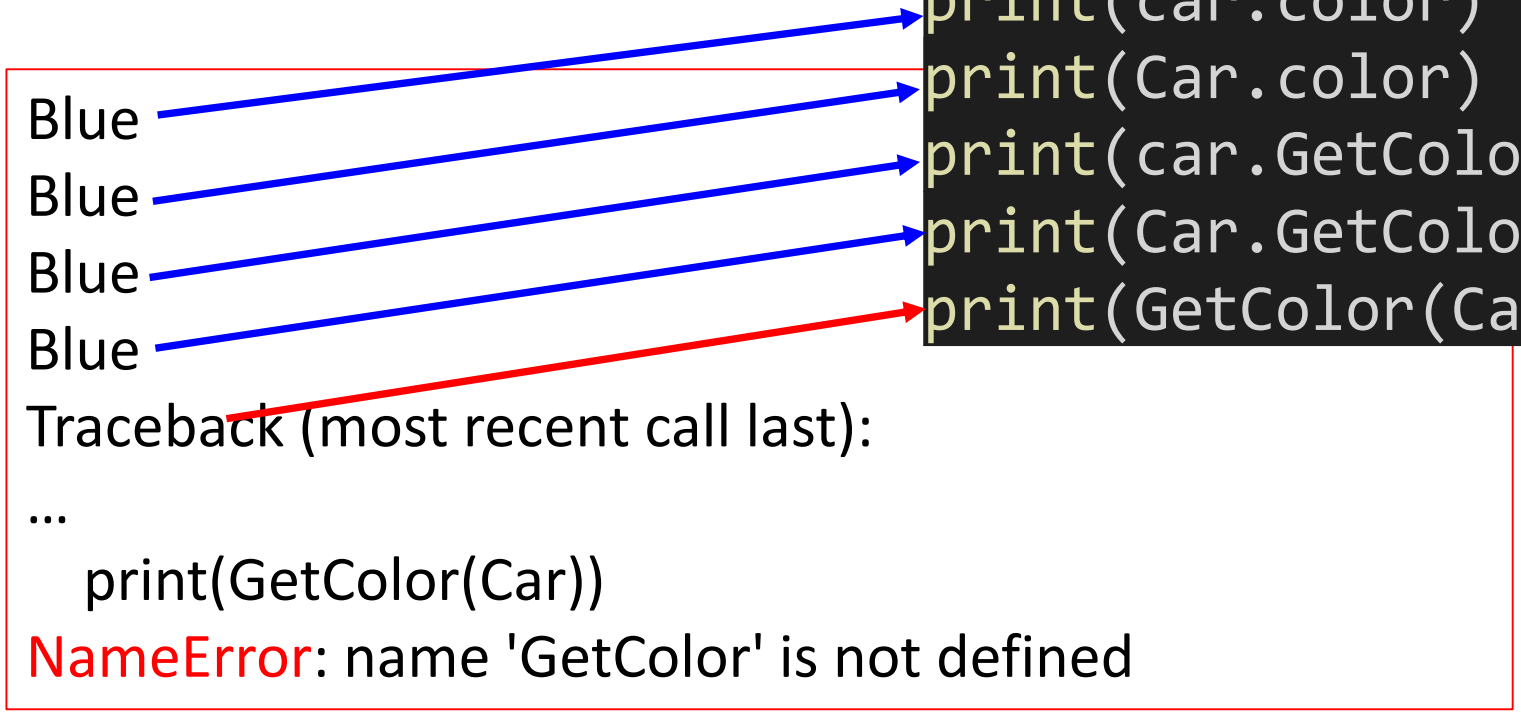


Access Class attributes

```
class Car:
    color = "Blue"
    @classmethod
    def GetColor(cls):
        return cls.color
car = Car()
print(car.color)
print(Car.color)
print(car.GetColor())
print(Car.GetColor())
print(GetColor(Car))
```

Output

```
Blue
Blue
Blue
Blue
Traceback (most recent call last):
...
   print(GetColor(Car))
NameError: name 'GetColor' is not defined
```



Learning Objectives

- Classes and objects
  - Instance variables
  - Class variables
  - Instance methods
  - Class methods
  - Access
  - Private and public attributes
  - Special method names
- Inheritance

Private and Public Attributes

- Python uses **underscore** to define special attributes
  - ✓ _xxx: denotes protected attribute xxx which cannot be imported using 'from module import *'
  - ✓ __xxx__: system defined attribute xxx, e.g., __init__
  - ✓ __xxx: private attribute xxx, which should be accessed via instance methods, cannot be accessed via object.__xxx outside of the class, or instance methods of its subclasses (we can still access via "object._class__xxx")

Private and Public Attributes

```
class Car:
    def __init__(self, c):
        self.__color = c
    def GetColor(self):
        return self.__color

car = Car("Red")
print(car.GetColor())
print(Car.GetColor(car))
print(car.__color)
```

__init__: special name method

__color: intended to be private instance attribute

__color: can be accessed in instance methods

AttributeError

Output

```
Red
Red
Traceback (most recent call last):
AttributeError: 'Car' object has no attribute '__color'
```

Private and Public Attributes

```
class Car:
    def __init__(self, c):
        self.__color = c
    def __GetColor(self):
        return self.__color

car = Car("Red")
print(car.__GetColor())
```

__color and __GetColor: intended to be private attribute

AttributeError

Output

```
Traceback (most recent call last):
  File "C:\Users\Desktop\hello.py", line 8, in <module>
    print(car.__GetColor())
AttributeError: 'Car' object has no attribute '__GetColor'
```

Private and Public Attributes

```
class Car:
    def __init__(self, c):
        self.__color = c
    def __GetColor(self):
        return self.__color

car = Car("Red")
print(car._Car__color)
print(car._Car__GetColor())
```

__color and __GetColor: intended to be private attribute

can be accessed via special way

Output

```
Red
Red
>>>
```

Private and Public Attributes

```
class Car:
    __color ="Blue"
    @classmethod
    def GetColor(cls):
        return cls.__color

print(Car.GetColor())
print(Car.__color)
```

__color: intended to be private class attribute

__color: can be accessed in this class

AttributeError

Output

```
Blue
Traceback (most recent call last):
  ...
AttributeError: type object 'Car' has no attribute '__color'
```

Private and Public Attributes

```
class Car:
    __color ="Blue"
    @classmethod
    def __GetColor(cls):
        return cls.__color

print(Car.__GetColor())
```

__GetColor: intended to be private class attribute

AttributeError

Output

```
Traceback (most recent call last):
  File "C:\Users\Desktop\hello.py", line 7, in <module>
    print(Car.__GetColor())
AttributeError: type object 'Car' has no attribute '__GetColor'
```

Private and Public Attributes

```
class Car:
    __color ="Blue"
    @classmethod
    def __GetColor(cls):
        return cls.__color
print(Car._Car__color)
print(Car._Car__GetColor())
```

can be accessed via special way

Output

```
Blue
Blue
<<<
```

Learning Objectives

- Classes and objects
  - Instance variables
  - Class variables
  - Instance methods
  - Class methods
  - Access
  - Private and public attributes
  - Special method names
- Inheritance

Special method names

- A class can implement certain operations that are invoked by special syntax (such as constructor, destructor) by defining methods with special names
- This is Python's approach to operator overloading, allowing classes to define their own behavior with respect to language operators
- The methods can be overridden if needed
- However, the methods should not be called explicitly in program
- They are called by the interpreter

Constructor and Destructor

- **Creator __new__(cls,…):** is called to create a new instance of class cls, e.g.,

  $$\mathbf{obj=cls(p_1,\ldots,p_n)}$$

  - ✓ __new__ takes the class of which an instance was requested as its first argument. The other arguments are those passed to __init__(self,…)
  - ✓ __new__ returns the new object instance before __init__ is invoked

- **Constructor __init__(self,…):** is called after the instance has been created to initialize instance variables

- **Destructor __del__(self):** is called when the instance is about to be destroyed, the object is destroyed when the reference count of the object reaches zero

Example

```
class Account:
    NumofAccounts = 0

    def __init__(self,idNum,v = 0):
        assert v>=0
        self.idNum = idNum
        self.balance = v
        Account.NumofAccounts +=1

    def Deposit(self,v):
        assert v>0
        self.balance += v

    def Withdraw(self,v):
        assert 0<v<=self.balance
        self.balance -= v
```

Example

```
    def __del__(self):
        assert Account.NumofAccounts>=1
        Account.NumofAccounts -=1

    def GetBalance(self):
        print("Balance of ",
              self.idNum, "is:",self.balance)

    @classmethod
    def GetNumofAccounts(cls):
        print("Number of accounts is:",
              cls.NumofAccounts)
```

Example

```
a = Account(1,10)
b = Account(2,20)
c = Account(3,30)
a.GetBalance()
b.GetBalance()
c.GetBalance()
Account.GetNumofAccounts()
a = None
Account.GetNumofAccounts()
b = None
Account.GetNumofAccounts()
```

Output

```
Balance of  1 is: 10
Balance of  2 is: 20
Balance of  3 is: 30
Number of accounts is: 3
Number of accounts is: 2
Number of accounts is: 1
```

Common special method names

Method	Description
__new__()	Create a new object instance
__init__()	Constructor
__del__()	Destructor
__add__()	+
__sub__()	-
__mul__()	*
__truediv__()	/
__floordiv__()	//
__mod__()	%
__pow__()	**
__eq__()、 __ne__()、 __lt__()、 __le__()、 __gt__()、 __ge__()	==、 !=、<、 <=、>、 >=
__lshift__()、__rshift__()	<<、>>
__and__()、__or__()、 __invert__()、__xor__()	&、|、~、^
__str__()	string representation of an object

Example

Implement a class for rational number

- Support

  ‘+’, ‘-’ ,’*’, ‘/’ ,’pow’,….,

- Does not support

  ‘and’, ‘or’,….

Example

```
def gcd ( a, b ):
    '''Return the greatest common divisor
       of a and b
    '''
    if b == 0:
        return a
    else:
        return gcd(b, a%b)
```

Euclidean Algorithm: gcd(a,b) = gcd(b,a%b)

Example

```
class Rational:
    """An instance represents a
       rational number."""

    def __init__(self,n=0, d=1):
        """Constructor for Rational."""
        assert d!=0, "d cannot be zero."
        g = gcd (n, d)
        self.n = int(n/g)
        self.d = int(d/g)
        __and__ = None
        __or__ = None
        # list non-supported methods
```

It is better to assign non-supported methods by None

Example

```
    def __add__(self, other):
        """Add two rational numbers."""
        return Rational(self.n * other.d +
                        other.n * self.d,
                        self.d * other.d )

    def __sub__(self,other):
        """Return self minus other."""
        return Rational (self.n * other.d -
                         other.n * self.d,
                         self.d * other.d )
    def __str__ (self):
        """Display self as a string."""
        return str(self.n)+"/"+str(self.d)
```

Example

```
r1 = Rational(2,4)
r2 = Rational(1,4)
print(r1)
print(r1-r2)
print(r1+r2)
print(r1&r2)
```

Output

1/2
1/4
3/4
Traceback (most recent call last):
…
TypeError: unsupported operand type(s) for &:
'Rational' and 'Rational'

Learning Objectives

- Classes and objects
  - Instance variables
  - Class variables
  - Instance methods
  - Class methods
  - Access
  - Private and public attributes
  - Special method names
- Inheritance

Class revisit

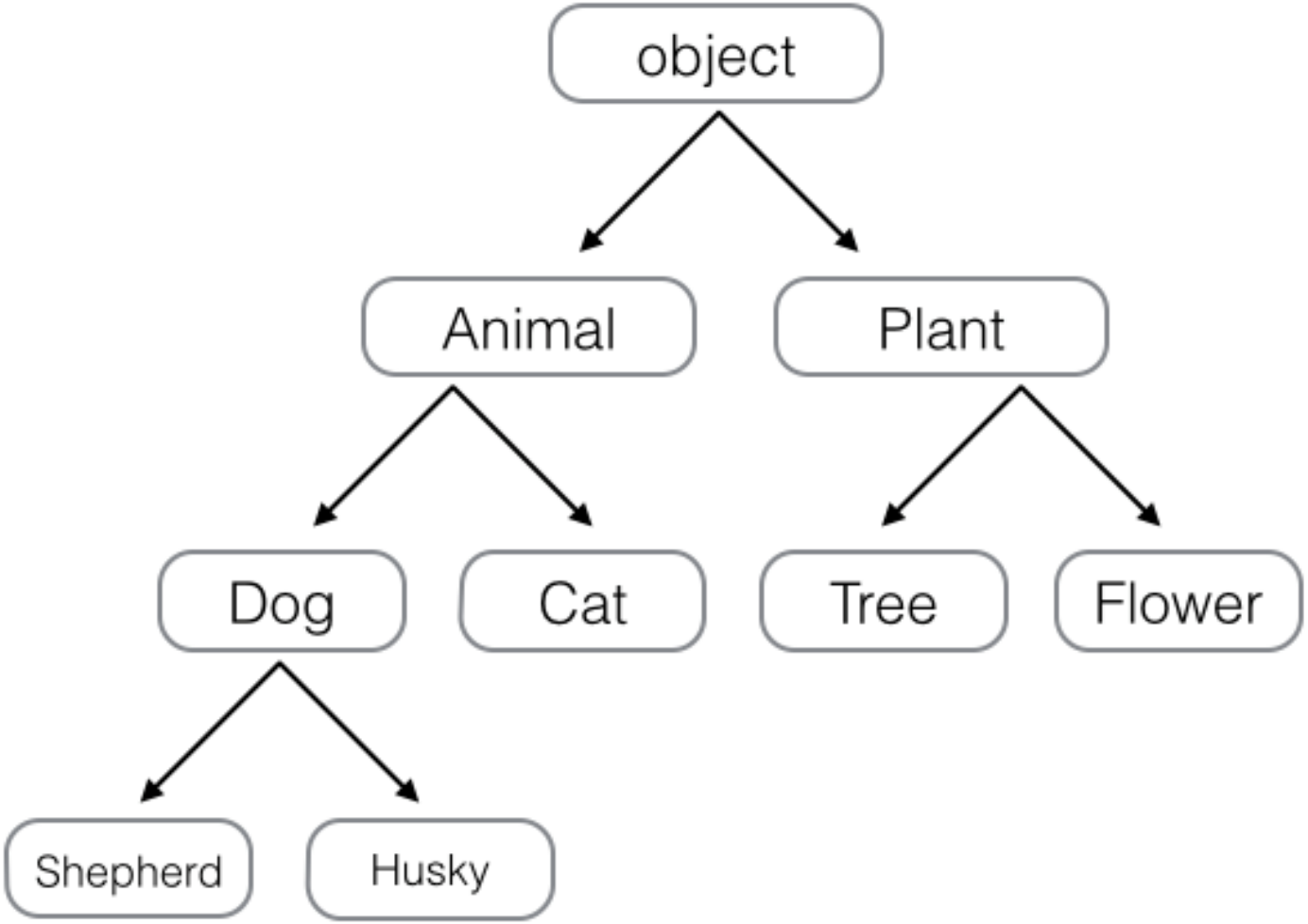
```
“class” ClassName”:”
    <statement-1>
    .
    .
    <statement-N>
```

- Class object vs instance object
- Class attribute vs instance attribute
- Private attribute vs public attribute

Inheritance

- Inheritance is yet another way to reuse code
- Other ways:
  - Functions
  - Classes
  - Modules

Inheritance



Inheritance

```
class Animal(object):
    def run(self):
        print('Animal is running...')

class Dog(Animal):
    pass

class Cat(Animal):
    pass
```

```
dog = Dog()
dog.run()

cat = Cat()
cat.run()
```

Output

```
Animal is running...
Animal is running...
```

Inheritance

```
class Animal(object):
    def run(self):
        print('Animal is running...')

class Dog(Animal):
    def eat(self):
        print('Eating meat...')
    def run(self):
        print('Dog is running...')

class Cat(Animal):
    def run(self):
        print('Cat is running...')

dog = Dog()
dog.eat()
dog.run()

cat = Cat()
cat.run()
```

Output

```
Eating meat...
Dog is running...
Cat is running...
```

Inheritance

```
a = dict()
b = Animal()
c = Dog()

print(isinstance(a, dict))
print(isinstance(b, Animal))
print(isinstance(c, Dog))
print(isinstance(c, Animal))
print(isinstance(b, Dog))
```

Output

```
True
True
True
True
False
```

Inheritance (polymorphism)

```
def run_twice(animal):
    animal.run()
    animal.run()

run_twice(Animal())
run_twice(Dog())
run_twice(Cat())
```

Output

```
Animal is running...
Animal is running...
Dog is running...
Dog is running...
Cat is running...
Cat is running...
```

```
class Tortoise(Animal):
    def run(self):
        print('Tortoise is running slowly...')

run_twice(Tortoise())
```

Output

```
Tortoise is running slowly...
Tortoise is running slowly...
```

Inheritance

```
"class" SubClass "(" BaseClass ")" ":"
    <statement-1>
    .
    .
    <statement-N>
```

- SubClass is meant to be more specialized than BaseClass
  - adding new attributes (variables and methods)
- SubClass inherits some attributes of BaseClass
- SubClass can override inherited methods

Inheritance

- Sub class inherits all public class attributes of the Base class,
- But, does not inherit any private class attributes of Base classes

```
class SubClass(BaseClass):
    y = "SubClass Y"

    @classmethod
    def __private(cls):
        print("Method-1")

    @classmethod
    def public(cls):
        cls.__private()
        BaseClass.public()
        print(cls.__x)

print(SubClass.x)
print(SubClass.y)
SubClass.public()
```

```
class BaseClass:
    x = "BaseClass X"
    y = "BaseClass Y"
    __x = "Private X"

    @classmethod
    def __private(cls):
        print("Method-2")

    @classmethod
    def public(cls):
        print("Method-3")
```

```
BaseClass X  # inherited
SubClass Y   # overridden
Method-1
Method-3   # call in base
AttributeError  # private
```

Inheritance

- Sub class inherits all public class attributes of the Base class,
- But, does not inherit any private class attributes of Base classes
- Sub class inherits all public instance methods of the Base class
- But, sub class inherits all public instance variables of the Base class, only if one of the following condition holds

1. The sub class does not override __init__ method of the base class, (meaning __init__ of the base class is implicitly invoked)
2. The sub class explicitly invokes __init__ method of the base class in its own __init__ method

```
class BaseClass:
    def __init__(self):
        self.x = "BaseClass X"
        self.y = "BaseClass Y"
        self.__x = "Private X"
    def __private(self):
        print("Method-2")

    def public(self):
        print("Method-3")

class SubClass(BaseClass):
    def __private(self):
        print("Method-1")
    def public(self):
        self.__private()
        BaseClass.public(self)
        print(self.__x)
```

```
s = SubClass()
print(s.x)
print(s.y)
s.public()
```

BaseClass X # inherited
BaseClass Y # inherited
Method-1
Method-3
AttributeError

```
class BaseClass:
    def __init__(self):
        self.x = "BaseClass X"

class SubClass(BaseClass):
    def __init__(self):
        self.z = "Subclass Z"
        BaseClass.__init__(self)


s = SubClass()
print(s.z)
print(s.x)
```

Subclass Z
BaseClass X

```
class BaseClass:
    def __init__(self):
        self.x = "BaseClass X"

class SubClass(BaseClass):
    def __init__(self):
        self.z = "Subclass Z"

s = SubClass()
print(s.z)
print(s.x)
```

Subclass Z

AttributeError: 'SubClass' object has no attribute 'x'

Inheritance

- Sub class inherits all public class attributes of the Base class,
- But, does not inherit any private class attributes of Base classes
- Sub class inherit all public instance methods of the Base class
- But, sub class inherit all public instance variables of the Base class, only if one of the following condition holds
  1. The sub class does not override __init__ method of the base class, (meaning __init__ of the base class is implicitly invoked)
  2. The sub class explicitly invokes __init__ method of the base class in its own __init__ method
- New/overridden method cannot access private attributes of the base class
- But, inherited methods can access private attributes of the base class

```
class BaseClass:
    def __init__(self):
        self.__x = "Private X“

    def __private(self):
        print("Private 1")
        print(self.__x)

class SubClass(BaseClass):
    def __private(self):
        print("Private 2")

    def public(self):
        self.__private()
        print(self.__x)
s = SubClass()
s.public()
```

Private 2
AttributeError: 'SubClass' object has no attribute '_SubClass__x

New/overridden method cannot access private attributes of the base class

```
class BaseClass:
    def __init__(self):
        self.__x = "Private X"

    def __private(self):
        print("Private 1")
        print(self.__x)

    def public(self):
        self.__private()
        print(self.__x)

class SubClass(BaseClass):
    def __private(self):
        print("Private 2")
s = SubClass()
s.public()
```

```
Private 1
Private X
Private X
```

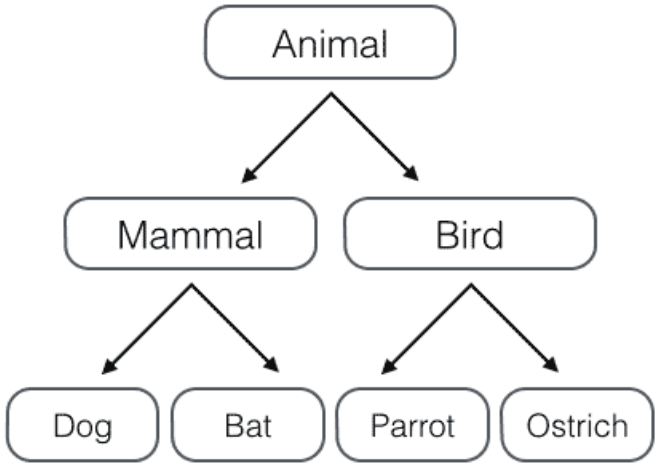
inherited methods can access private attributes of the base class

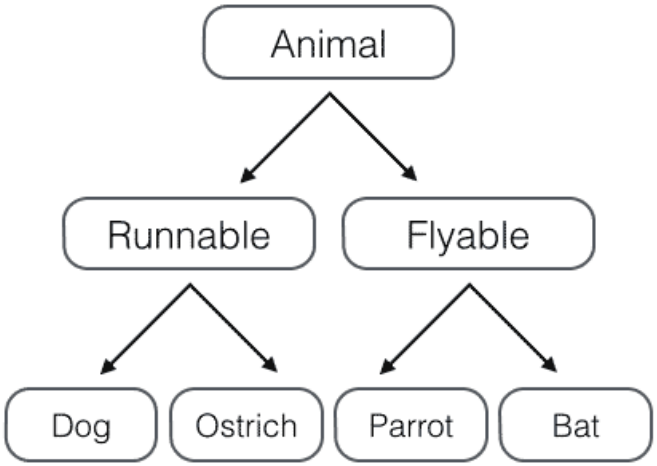
```
class BaseClass:
    def __init__(self):
        self.__x = "Private X“

    def private(self):
        print("Private 1")
        print(self.__x)

    def public(self):
        self.private()
        print(self.__x)

class SubClass(BaseClass):
    def private(self):
        print("Private 2")
s = SubClass()
s.public()
```

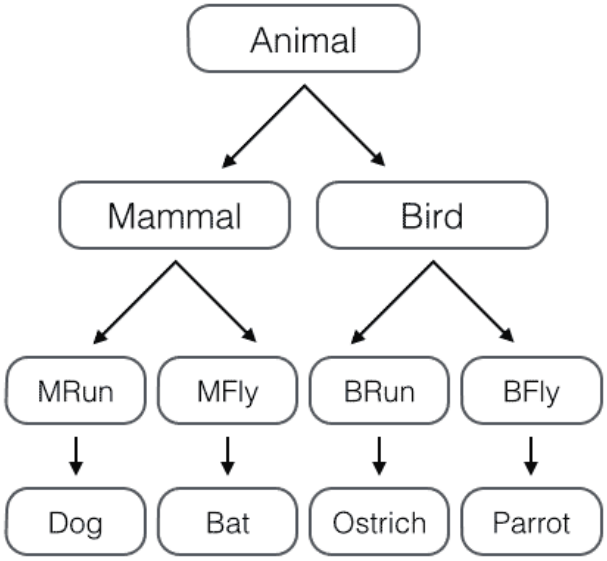
Private 2
Private X

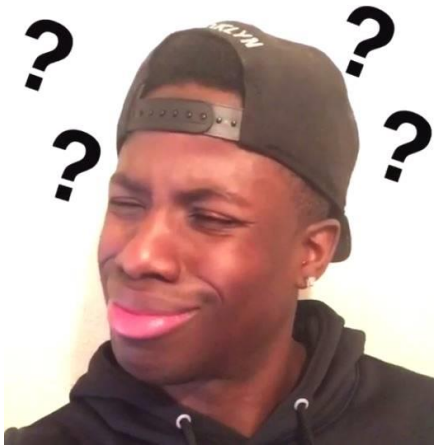
inherited methods will first search in sub class

Multiple Inheritance









Multiple Inheritance

```
class Animal(object):
    pass

class Mammal(Animal):
    pass

class Bird(Animal):
    pass

class Dog(Mammal):
    pass

class Bat(Mammal):
    pass

class Parrot(Bird):
    pass

class Ostrich(Bird):
    pass
```

```
class Runnable(object):
    def run(self):
        print('Running...')

class Flyable(object):
    def fly(self):
        print('Flying...')

class Dog(Mammal, Runnable):
    pass

class Bat(Mammal, Flyable):
    pass
```

Multiple Inheritance

```
"class" SubClass "(" Base1, Base2,…,Basen ")" ":"
    <statement-1>
    .
    .
    <statement-N>
```

- SubClass is a specialization of all base classes
- Method Resolution Order (MRO)
  - When subclass calls a method defined in multiple base classes
  - ClassName.mro()

Readings (recommended)

- The Python Tutorial
  - 9. Classes

Recap

- Classes and objects
  - Instance methods
  - Class methods
  - Access
  - Private and public attributes
  - Special method names
- Inheritance