Object-Oriented Design with Python

CSCI 5448: Object – Oriented A & D Presentation

Yang Li

Summary

- This presentation assumes audience have the knowledge of Object-Oriented A & D and emphasize on OOP programming with python
- Introduces Python's special methods to realize class definition, inheritance, multiple inheritance, accessibility, polymorphism, encapsulation.
- This presentation indicates the difference of how to realize OOP method between python and other OOP language
- Compare Python's OOP methods with other OOP languages. Analyze their advantages and disadvantages.

What's Python?

- Python is a general-purpose, interpreted high-level programming language.
- Its syntax is clear and emphasize readability.
- Python has a large and comprehensive standard library.
- Python supports multiple programming paradigms, primarily but not limited to object-oriented, imperative and, to a lesser extent, functional programming styles.
- It features a fully dynamic type system and automatic memory management

Advantages of Python

- Simple
- Easy to study
- Free and open source
- High-level programming language
- Portability
- Expansibility
- Embedability
- Large and comprehensive standard libraries
- Canonical code

A Example of Python Class

```
class Person:

    def __init__(self,name):
        self.name = name

    def Sayhello(self):
        print 'Hello, my name is', self.name

    def __del__(self):
        print '%s says bye.' % self.name

A = Person('Yang Li')
```

This example includes
class definition, **constructor function**, **destructor function**, **attributes and methods definition** and **object definition.**
These definitions and uses will be introduced specifically in the following.

Class Definition and Object Instantiation

- Class definition syntax:

```
class subclass[(superclass)]:
        [attributes and methods]
```

- Object instantiation syntax:

```
object = class()
```

- Attributes and methods invoke:

```
object.attribute
object.method()
```

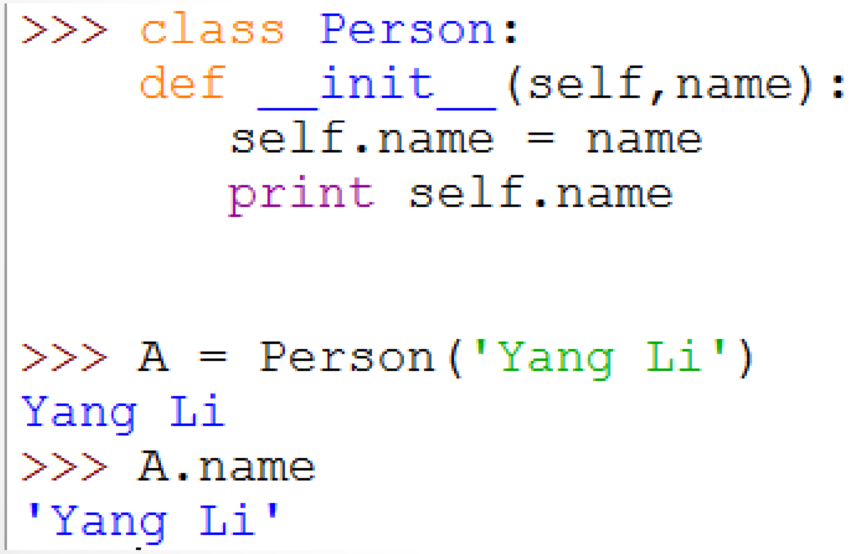
Special Class Attributes in Python

- 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

Constructor: __init__()

- The __init__ method is run as soon as an object of a class is instantiated. Its aim is to initialize the object.



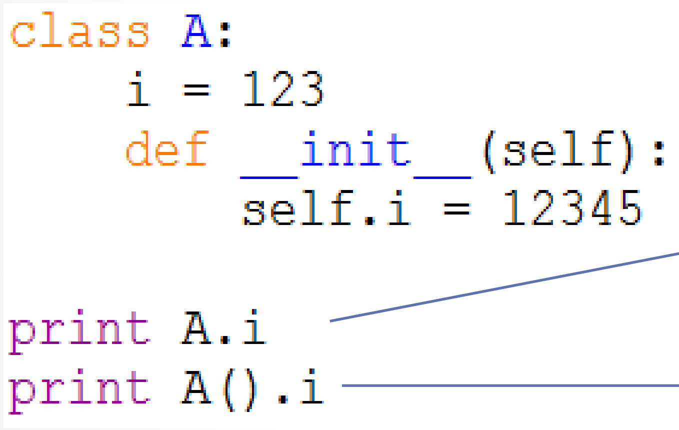
```
>>> class Person:
    def __init__(self,name):
        self.name = name
        print self.name


>>> A = Person('Yang Li')
Yang Li
>>> A.name
'Yang Li'
```

From the code , we can see that after instantiate object, it automatically invokes __init__()

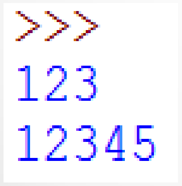
As a result, it runs
self.name = 'Yang Li',
and
print self.name

Form and Object for Class

- Class includes two members: form and object.
- The example in the following can reflect what is the difference between object and form for class.



```
class A:
    i = 123
    def __init__(self):
        self.i = 12345

print A.i
print A().i
```

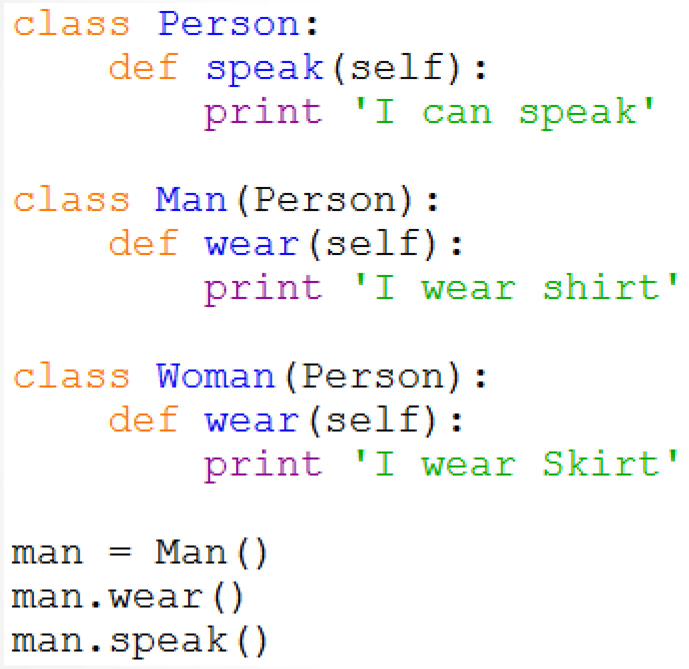


```
>>>
123
12345
```

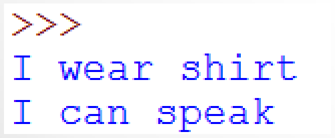
Invoke form: just invoke data or method in the class, so i=123

Invoke object: instantialize object Firstly, and then invoke data or Methods.
Here it experienced __init__(),
i=12345

Inheritance



```python
class Person:
    def speak(self):
        print 'I can speak'

class Man(Person):
    def wear(self):
        print 'I wear shirt'

class Woman(Person):
    def wear(self):
        print 'I wear Skirt'

man = Man()
man.wear()
man.speak()
```



```
>>>
I wear shirt
I can speak
```

Inheritance in Python is simple,
Just like JAVA, subclass can invoke
Attributes and methods in superclass.

From the example, **Class Man** inherits **Class Person**, and invoke **speak()** method In **Class Person**

Inherit Syntax:

```
class subclass(superclass):
        …
        …
```

In Python, it supports multiple inheritance,
In the next slide, it will be introduced.

Multiple Inheritance

- Python supports a limited form of multiple inheritance.
- A class definition with multiple base classes looks as follows:

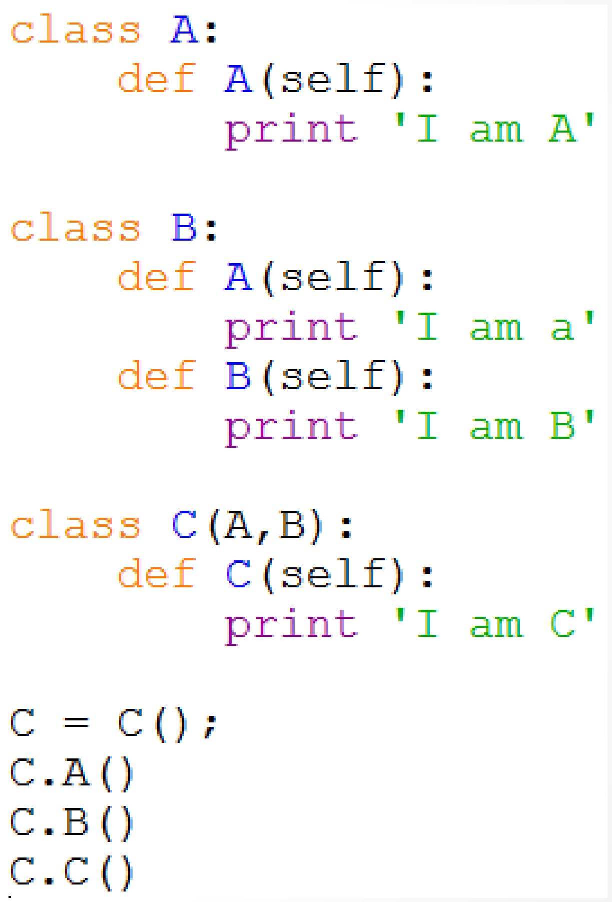
```
class DerivedClass(Base1, Base2, Base3 …)
        <statement-1>
        <statement-2>
        …
```

- The only rule necessary to explain the semantics is the resolution rule used for class attribute references. This is depth-first, left-to-right. Thus, if an attribute is not found in DerivedClass, it is searched in Base1, then recursively in the classes of Base1, and only if it is not found there, it is searched in Base2, and so on.

An Example of Multiple Inheritance

C multiple-inherit A and B, but since A is in the left of B, so C inherit A and invoke A.A() according to the left-to-right sequence.

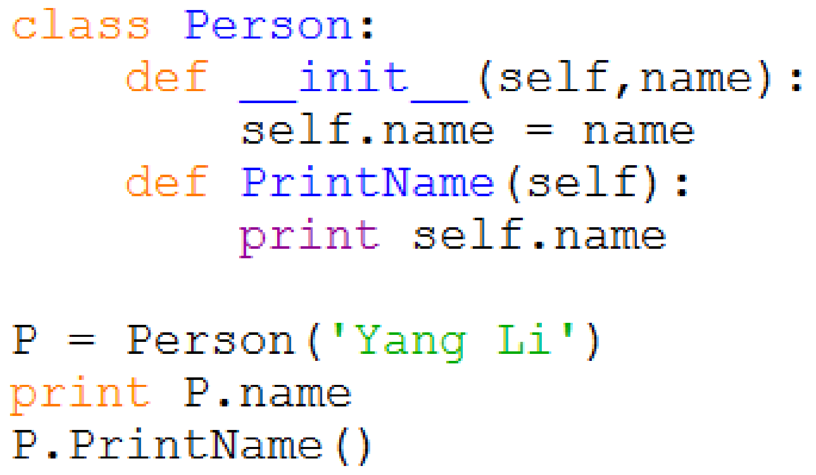
To implement C.B(), class A does not have B() method, so C inherit B for the second priority. So C.B() actually invokes B() in class B.



```
class A:
    def A(self):
        print 'I am A'

class B:
    def A(self):
        print 'I am a'
    def B(self):
        print 'I am B'

class C(A,B):
    def C(self):
        print 'I am C'

c = C();
c.A()
c.B()
c.C()
```

"Self"

- "Self" in Python is like the pointer "this" in C++. In Python, functions in class access data via "self".



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

P = Person('Yang Li')
print P.name
P.PrintName()
```

- "Self" in Python works as a variable of function but it won't invoke data.

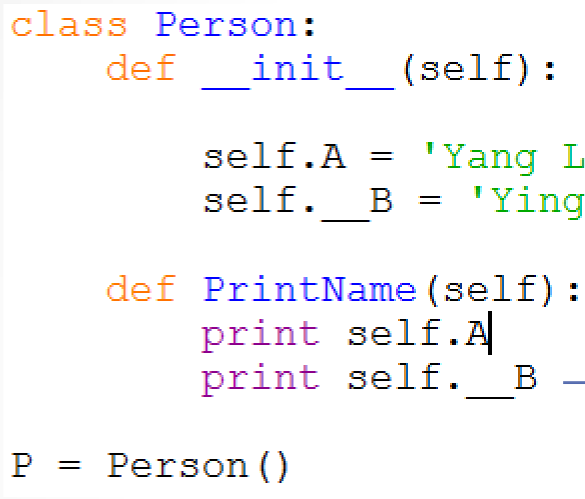
Encapsulation – Accessibility (1)

- In Python, there is no keywords like ‘public’, ‘protected’ and ‘private’ to define the accessibility. In other words, In Python, it acquiesce that all attributes are public.

- But there is a method in Python to define Private:

  Add “__” in front of the variable and function name can hide them when accessing them from out of class.

An Example of Private

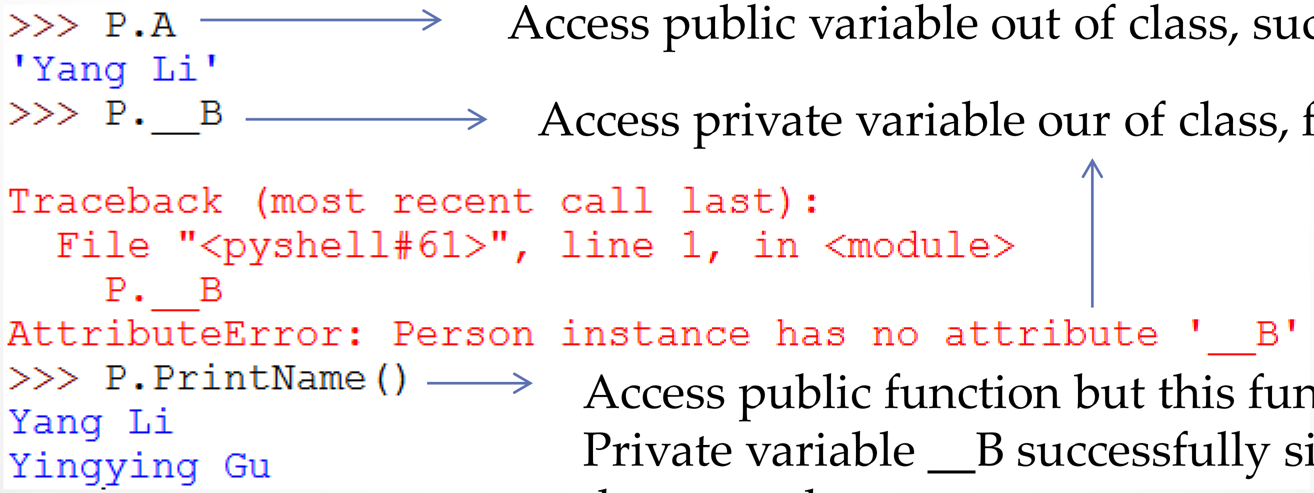


```
class Person:
    def __init__(self):

        self.A = 'Yang Li'
        self.__B = 'Yingying Gu'

    def PrintName(self):
        print self.A
        print self.__B

P = Person()
```

Public variable

Private variable

Invoke private variable in class



```
>>> P.A
'Yang Li'
>>> P.__B

Traceback (most recent call last):
  File "<pyshell#61>", line 1, in <module>
    P.__B
AttributeError: Person instance has no attribute '__B'
>>> P.PrintName()
Yang Li
Yingying Gu
```

Access public variable out of class, succeed

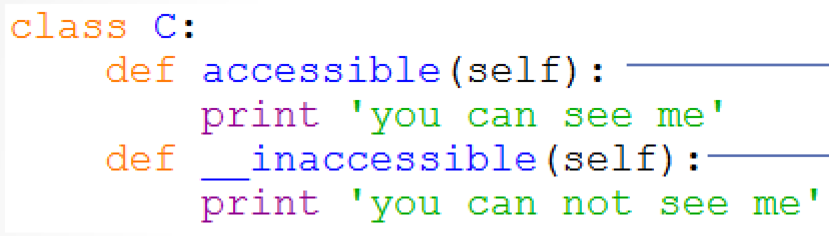
Access private variable our of class, fail

Access public function but this function access Private variable __B successfully since they are in the same class.

Encapsulation – Accessibility (2)

- Actually, the private accessibility method is just a rule, not the limitation of compiler.
- Its fact is to change name of private name like __variable or __function() to _ClassName__variable or _ClassName__function(). So we can't access them because of wrong names.
- We even can use the special syntax to access the private data or methods. The syntax is actually its changed name. Refer to the following example in the next slide.

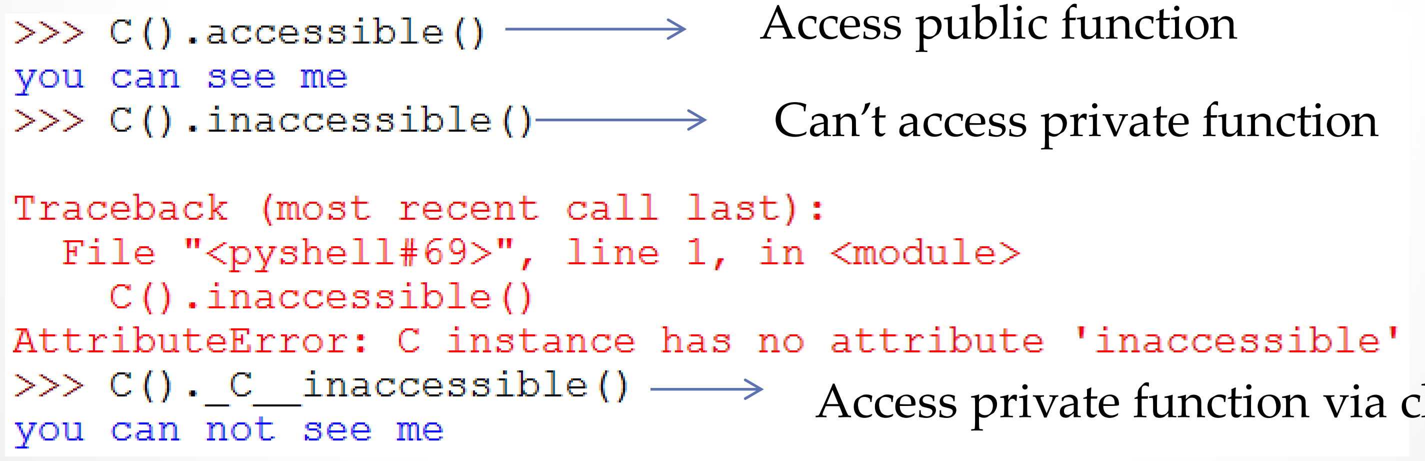
An example of Accessing Private



```
class C:
    def accessible(self):
        print 'you can see me'
    def __inaccessible(self):
        print 'you can not see me'
```

Define public function

Define private function



```
>>> C().accessible()
you can see me
>>> C().inaccessible()

Traceback (most recent call last):
  File "<pyshell#69>", line 1, in <module>
    C().inaccessible()
AttributeError: C instance has no attribute 'inaccessible'
>>> C()._C__inaccessible()
you can not see me
```

Access public function

Can’t access private function

Access private function via changed name

Polymorphism

- Polymorphism is an important definition in OOP. Absolutely, we can realize polymorphism in Python just like in JAVA. I call it “traditional polymorphism”
- In the next slide, there is an example of polymorphism in Python.
- But in Python,

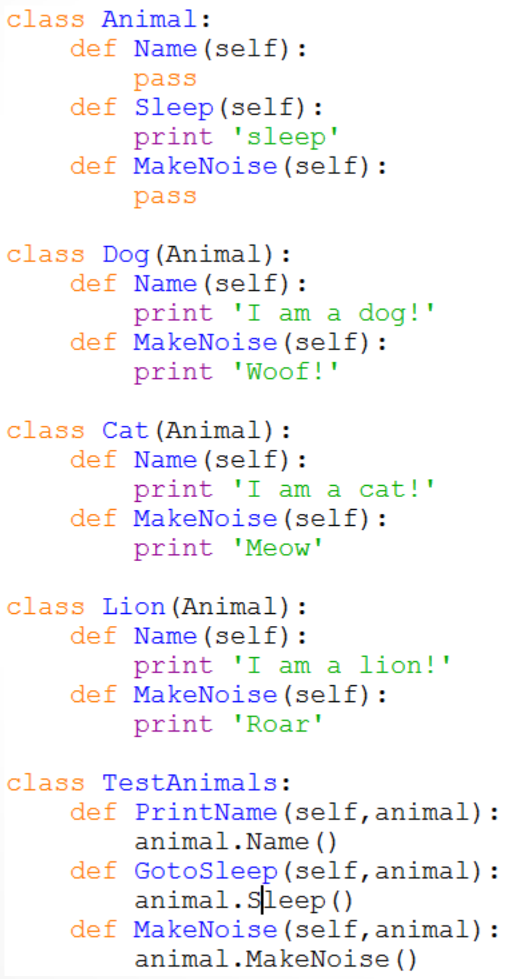
Only traditional polymorphism exist?

NO!

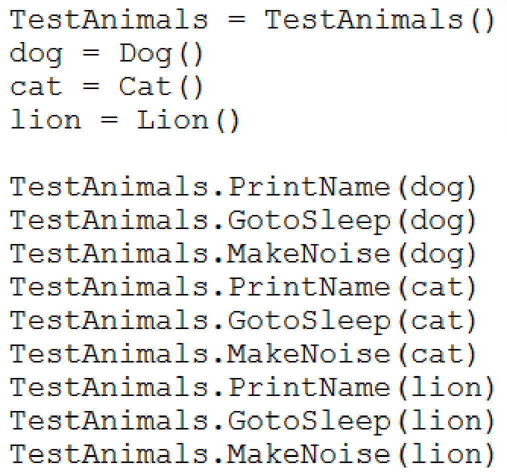
Compare Accessibility of Python and Java

- Java is a static and strong type definition language. Java has strict definition of accessibility type with keywords.
- While Python is a dynamic and weak type definition language. Python acquiesces all the accessibility types are public except for supporting a method to realize private accessibility virtually.
- Someone think Python violates the requirement of encapsulation. But its aim is to make language simple.

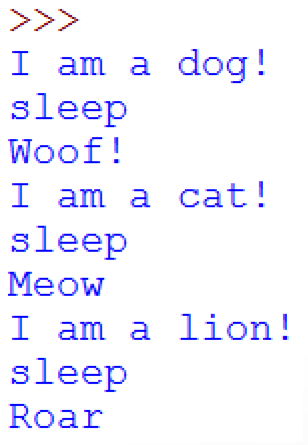
Traditional Polymorphism Example



```
class Animal:
    def Name(self):
        pass
    def Sleep(self):
        print 'sleep'
    def MakeNoise(self):
        pass

class Dog(Animal):
    def Name(self):
        print 'I am a dog!'
    def MakeNoise(self):
        print 'Woof!'

class Cat(Animal):
    def Name(self):
        print 'I am a cat!'
    def MakeNoise(self):
        print 'Meow'

class Lion(Animal):
    def Name(self):
        print 'I am a lion!'
    def MakeNoise(self):
        print 'Roar'

class TestAnimals:
    def PrintName(self,animal):
        animal.Name()
    def GotoSleep(self,animal):
        animal.Sleep()
    def MakeNoise(self,animal):
        animal.MakeNoise()
```



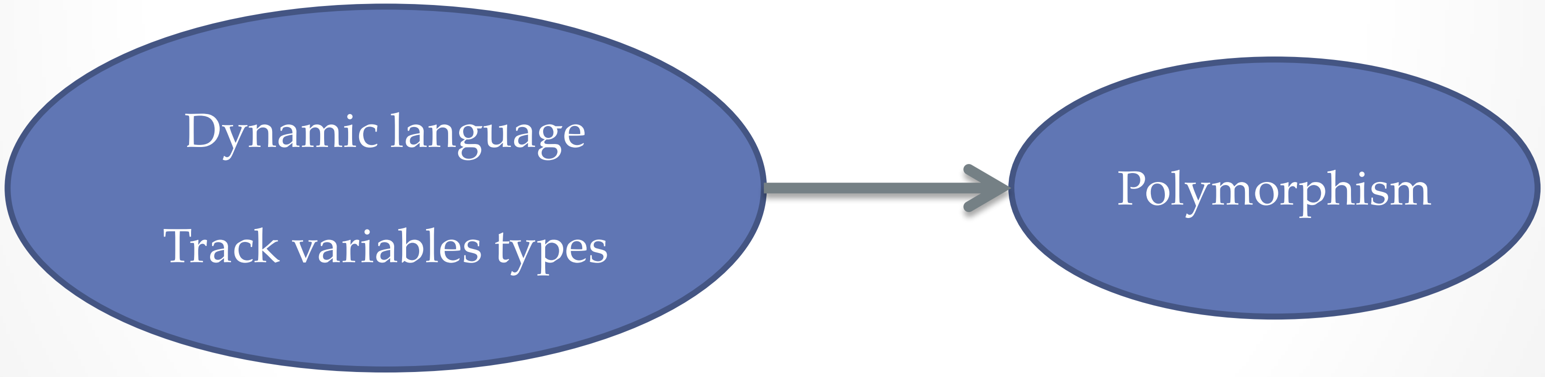
```
TestAnimals = TestAnimals()
dog = Dog()
cat = Cat()
lion = Lion()

TestAnimals.PrintName(dog)
TestAnimals.GotoSleep(dog)
TestAnimals.MakeNoise(dog)
TestAnimals.PrintName(cat)
TestAnimals.GotoSleep(cat)
TestAnimals.MakeNoise(cat)
TestAnimals.PrintName(lion)
TestAnimals.GotoSleep(lion)
TestAnimals.MakeNoise(lion)
```



```
>>>
I am a dog!
sleep
Woof!
I am a cat!
sleep
Meow
I am a lion!
sleep
Roar
```

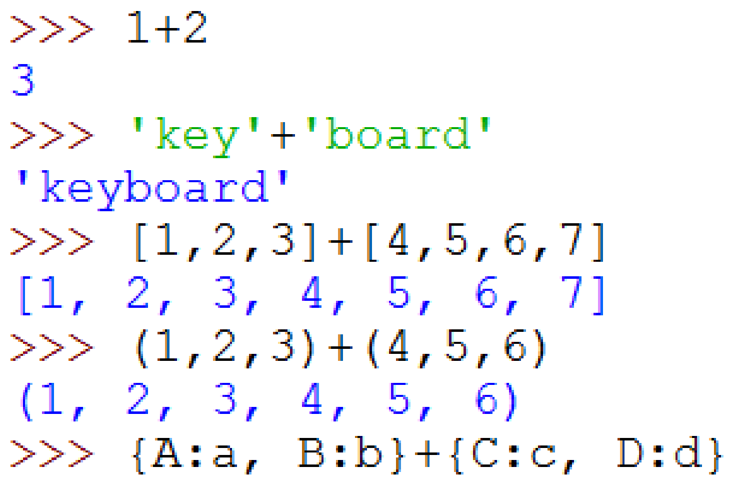
Everywhere is polymorphism in Python (1)

- Since Python is a dynamic programming language, it means Python is strongly typed as the interpreter keeps track of all variables types. It reflects the polymorphism character in Python.



Everywhere is polymorphism in Python (2)

- So, in Python, many operators have the property of polymorphism. Like the following example:

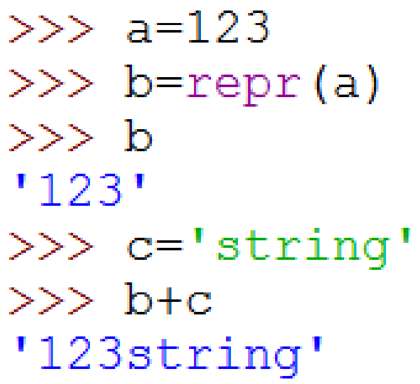


```
>>> 1+2
3
>>> 'key'+'board'
'keyboard'
>>> [1,2,3]+[4,5,6,7]
[1, 2, 3, 4, 5, 6, 7]
>>> (1,2,3)+(4,5,6)
(1, 2, 3, 4, 5, 6)
>>> {A:a, B:b}+{C:c, D:d}
```

- Looks stupid, but the key is that variables can support any objects which support ‘add’ operation. Not only integer but also string, list, tuple and dictionary can realize their relative ‘add’ operation.

Everywhere is polymorphism in Python (3)

- Some methods in Python also have polymorphism character like ‘repr’ function.



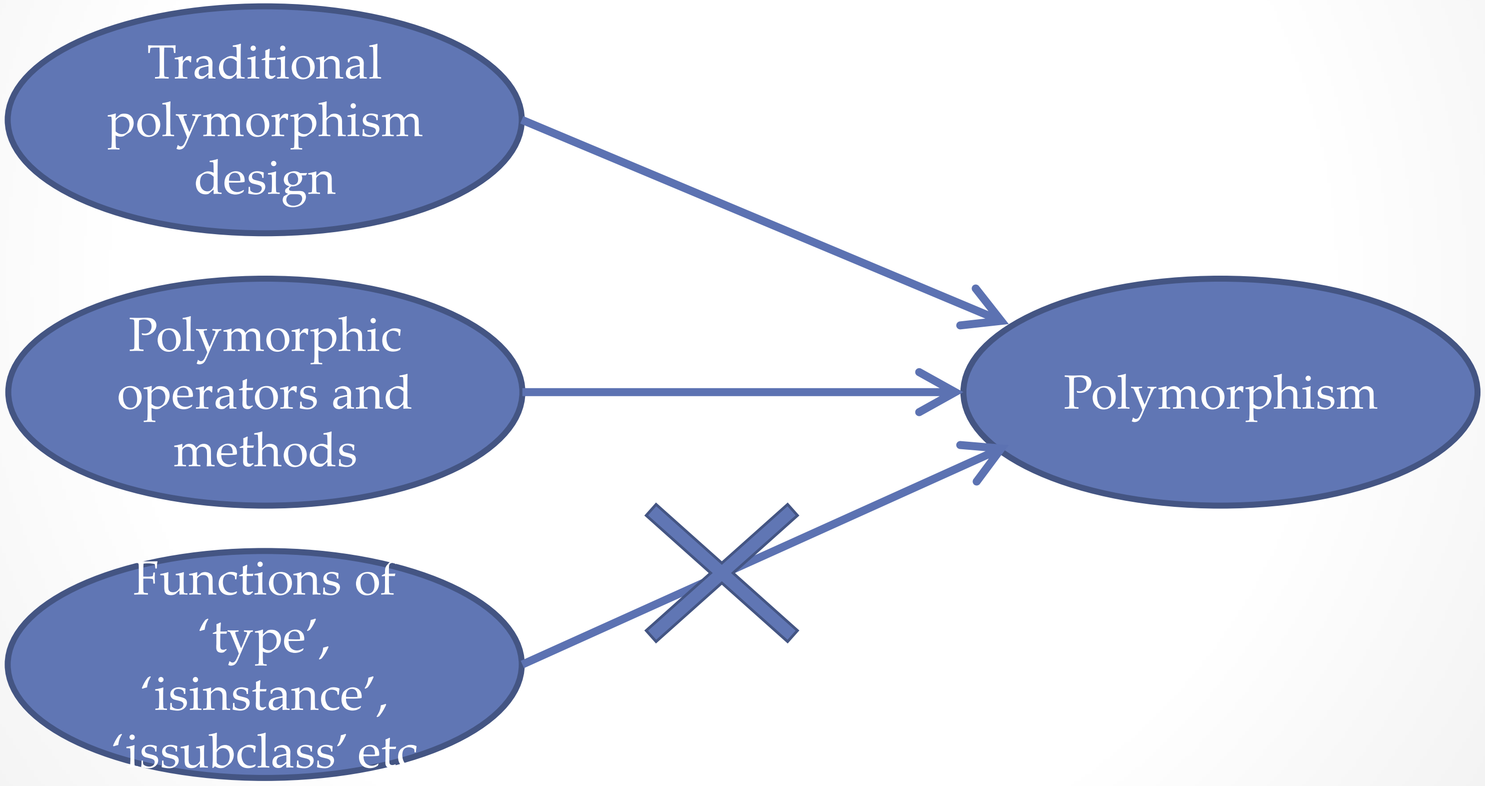
```
>>> a=123
>>> b=repr(a)
>>> b
'123'
>>> c='string'
>>> b+c
'123string'
```

- For ‘repr’ method, it can transfer any kinds of data to string type. In the above example, it converts integer 123 to string ‘123’ and it can even added to string c ‘string’ to get ‘123string’.

Avoid Destroying Polymorphism!

- Many operators and functions in Python are polymorphic. So as long as you use the polymorphic operators and functions, polymorphism will exist even if you don't have this purpose.
- The only way to destroy polymorphism is check types by using functions like 'type', 'isinstance' and 'issubclass' etc.
- So in the programming, we should avoid using these methods which might destroy polymorphism except for compiler design.
- The most important thing is to let objects to work according to your requirements rather than mind if they have right types

How to Affect Polymorphism


Traditional polymorphism design
Polymorphic operators and methods
Functions of 'type', 'isinstance', 'issubclass' etc
Polymorphism

Conclusion

- As a OOP language, Python has its special advantages but also has its disadvantages.
- Python can support operator overloading and multiple inheritance etc. advanced definition that some OOP languages don't have.
- The advantages for Python to use design pattern is that it supports dynamic type binding. In other words, an object is rarely only one instance of a class, it can be dynamically changed at runtime.
- But Python might ignore a basic regulation of OOP: data and methods hiding. It doesn't have the keywords of 'private', 'public' and 'protected' to better support encapsulation. But I think it aims to guarantee syntax simple. As the inventor of Python, Guido Van Rossum said: "abundant syntax bring more burden than help".