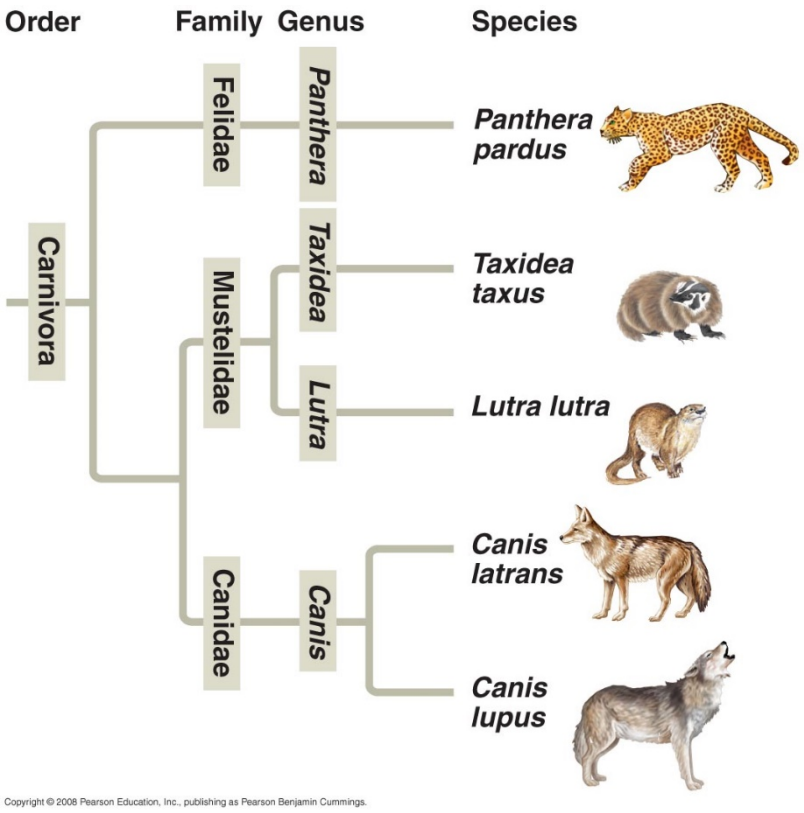


6. Object-Oriented Programming, II

Programming and Algorithms II

Degree in Bioinformatics

Fall 2018

Inheritance

```
class Carnivora(Animal):
    …

class Canid(Carnivora):
    …

class Dog(Canid):
    …

class Husky(Dog):
    …

class Fox(Canid):
    …

class Cat(Carnivora):
    …
```

subclass or child class

superclass or parent class

Dog “inherits” all the properties from Canid, etc.

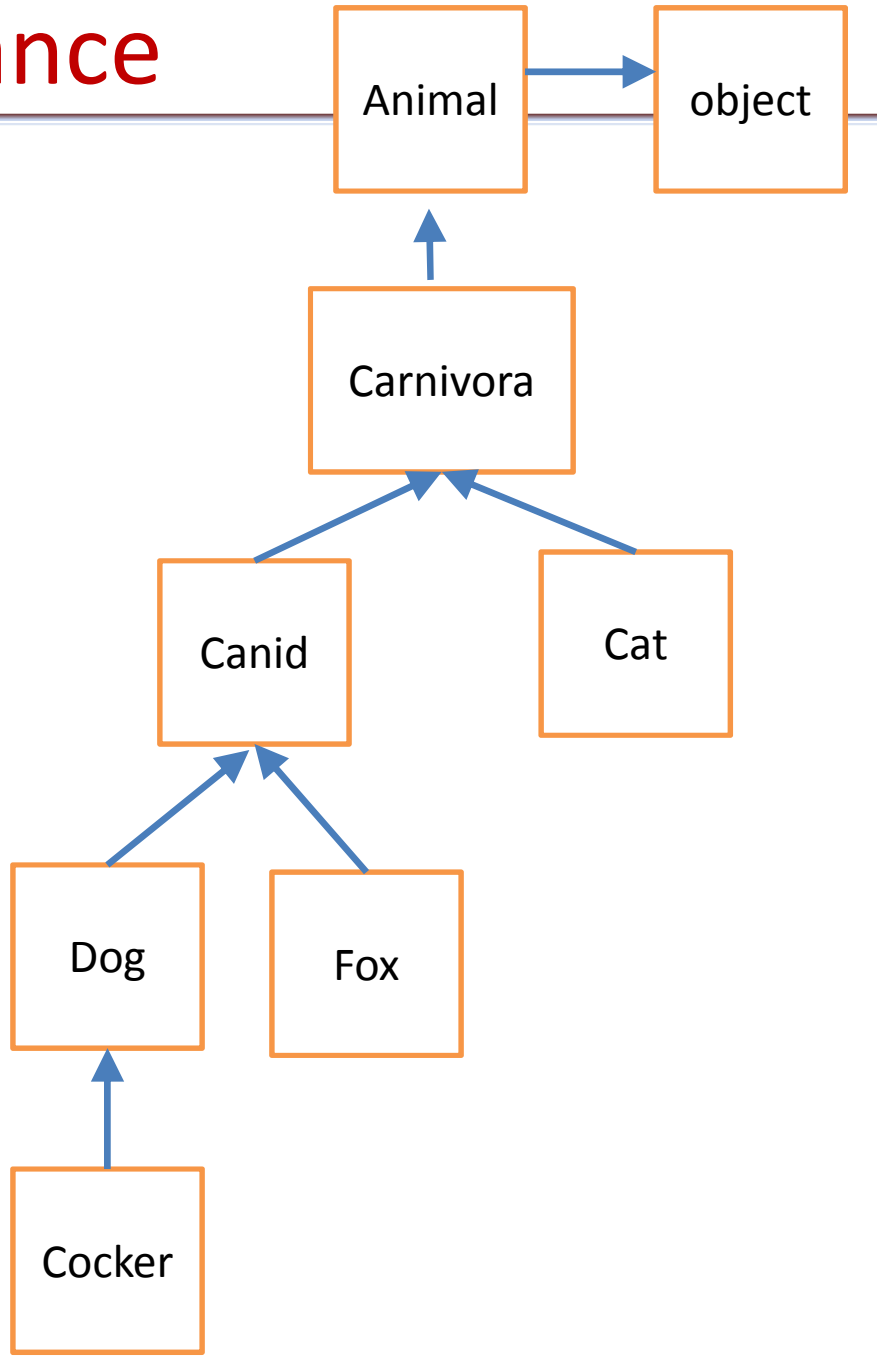
in Python, every class is a subclass of object

(in Python 3, you can omit writing it)

class diagram = hierarchy

Inheritance

```
class Carnivora(Animal):
    …

class Canid(Carnivora):
    …

class Dog(Canid):
    …

class Husky(Dog):
    …

class Fox(Canid):
    …

class Cat(Carnivora):
    …

class Cocker(Dog):
```

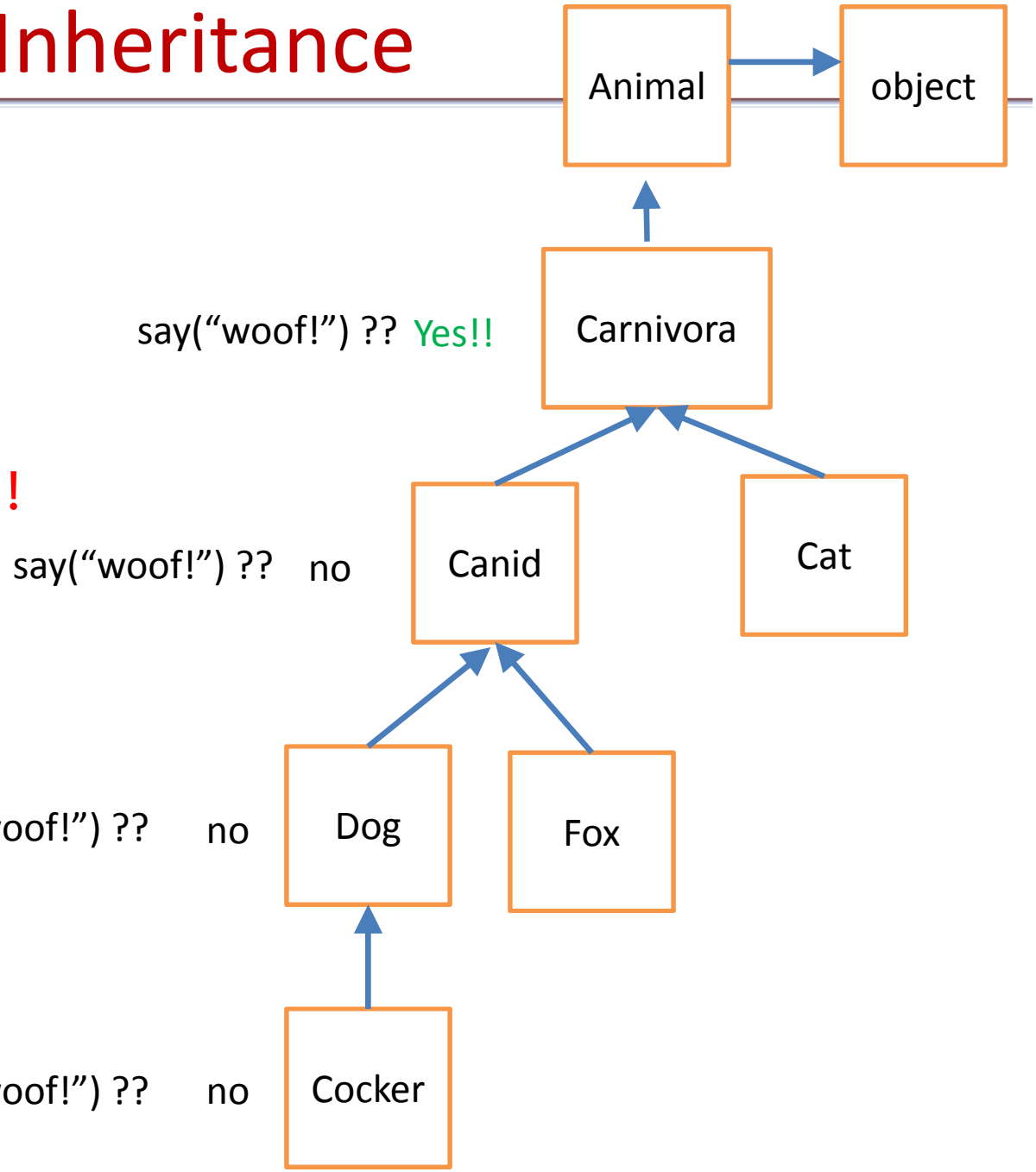


Inheritance

```
c = Cocker("pluto")
…
c.say("woof!")
```

```
>>> Pluto says woof!
```

```
class Carnivora(Animal):
    …
    def say(self,what):
        print(self.name,"says ",what)
```



How does inheritance work

x.f(parameters)

- Say x was defined using C(…) (x has type C)
- Then if C defines method f(…), that code is executed
- Else if the parent of C defines method f(…), that code is executed
- Else if the parent of the parent of C defines method f(…), that code is executed
- …
- Else if object defines method f(…), that code is executed
- Else, error

How does inheritance work

x.attr

Either x has an attribute called attr, or it doesn't

Error if not defined at this moment

It may have been defined / modified in the class of x or in any of its superclasses

Method is chosen by the object

```
class Animal(object):
    def __init__(self,name):
        self.name = name
    def talk(self):
        print("Error:",self.name,
              "can't talk!")
class Cat(Animal):
    def talk(self):
        print(self.name,"says: Meeooww!")

class Dog(Animal):
    def talk(self):
        print(self.name,"says: Woof!")

class Person(Animal):
    def talk(self):
        print(self.name,"says:
              My name is",self.name)
```

Note that only Animal has __init__ operation

Whenever a Cat, a Dog, or a Person is created, Python calls __init__ for the new object

The __init__ code executed is always that of Animal, as there is no other

Cat, Dog, Person could override __init__, as is done here with talk

Method is chosen by the object

```
class Animal:
    def __init__(self,name):
        self.name = name
    def talk(self):
        print("Error:",self.name,
              "can't talk!")
class Cat(Animal):
    def talk(self):
        print(self.name,"says: Meeooww!")

class Dog(Animal):
    def talk(self):
        print(self.name,"says: Wooof!")

class Person(Animal):
    def talk(self):
        print(self.name,"says:
              My name is",self.name)
```

```
microbe = Animal(“Joe Amoeba")
microbe.talk()

my_cat = Cat("Garfield")
my_cat.talk()

my_dog = Dog("Rintintin")
my_dog.talk()

agent_007 = Person("Bond, James Bond")
agent_007.talk()
```

Output:

```
Error: Joe Amoeba can't talk!
Garfield says: Meeooww!
Rintintin says: Wooof!
Bond, James Bond says: 'My name is
Bond, James Bond'
```

Method is chosen by the object

```
class Animal(object):
    def __init__(self,name):
        self.name = name
    def talk(self):
        print("Error:",self.name,
              "can't talk!")
class Cat(Animal):
    def talk(self):
        print(self.name,"says: Meeooww!")

class Dog(Animal):
    def talk(self):
        print(self.name,"says: Wooof!")

class Person(Animal):
    def talk(self):
        print(self.name,
              "says: My name is",
              self.name)
```

```
class Super_Dog(Dog):
    def talk(self):
        print(self.name,"says:
         'My name is",self.name,
        "and I am a genetically
        engineered dog'")

milou = Super_Dog("Milou")
milou.talk()
```

Output:

```
Milou says: ‘My name is Milou and
I am a genetically engineered dog’
```

Calling parent class method

```
class A(object):
    def __init__(self):
        print("init A fields…")

class B(A):
    def __init__(self):
        print("init B fields…")

b = B()  # does NOT initialize A fields
```

Calling parent class method

```
class A(object):
    def __init__(self):
        print("init A fields…")

class B(A):
    def __init__(self):
        A.__init__(self)  # class method -> self
        print("init B fields…")

b = B()  # initializes A fields, then B fields
```

Calling parent class method

```
class A(object):
    def __init__(self):
        print("init A fields…")

class B(A):
    def __init__(self):
        super().__init__()  # no self
        print("init B fields…")

b = B()  # initializes A fields, then B fields
```

super(): parent class of current object class

(more technically, the scope of the current class; that's why no "self" needed)

(better option in case you decide to change A's name, or add a class C in between)

(also allows wizards to do fun stuff… see mro() later)

Useful methods

dir(x) lists all the methods of x

```
>>> p = Point(0,1)
>>> dir(p)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '__weakref__', 'angle', 'angle_to_origin',
'distance_to_origin', 'mod', 'x', 'y']
```

Useful methods

dir(x) lists all the methods of x

```
>>> x = 5
>>> dir(x)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__',
'__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__',
'__floor__', '__floordiv__', '__format__', '__ge__', '__getattribute__',
'__getnewargs__', '__gt__', '__hash__', '__index__', '__init__',
'__init_subclass__', '__int__', '__invert__', '__le__', '__lshift__', '__lt__',
'__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__', '__pos__',
'__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__',
'__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__',
'__rmul__', '__ror__', '__round__', '__rpow__', '__rrshift__', '__rshift__',
'__rsub__', '__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__',
'__sub__', '__subclasshook__', '__truediv__', '__trunc__', '__xor__',
'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag', 'numerator',
'real', 'to_bytes']
```

Useful methods

isinstance(object,classname)

```
>>>   isinstance(1,int)
True
```

issubclass(B,A): True if B subclass of A

hasattr(o,'name'): True if o has a property called 'name'

```
>>> hasattr(p,'mod')
True
```

super() or classname.super(): parent class of an object or a class (scope of…)

Method Resolution Order (MRO)

```
class A(object):
    pass
class B(A):
    pass
class C(B):
    pass
class D(C):
    pass

print(D.mro())
```

```
[<class '__main__.D'>, <class '__main__.C'>, <class
'__main__.B'>, <class '__main__.A'>, <class 'object'>]
```

mro can be *modified* to interesting effects. Only for wizards

Multiple inheritance

```
class A(object):
    pass
class B(A):
    pass
class C(A):
    pass
class D(B,C):
    pass

print(D.mro())
```

```
[<class '__main__.D'>, <class '__main__.B'>, <class
'__main__.C'>, <class '__main__.A'>, <class 'object'>]
```

Multiple inheritance

```
class C(A,B):
        …
```

C inherits both from A and B

Occasionally useful; can also lead to bad designs

OK when A, B have no methods in common

Or two super classes of C define same name

Common names get resolved according to mro()

Example: Saving an object

```
class A(object):
    def save(self):
        print("save A attributes")
class B(A):
    def save(self):
        super().save(self)
        print("save B attributes")
class C(A):
    def save(self):
        super().save(self)
        print("save C attributes")
class D(C):
    def save(self):
        super().save(self)
        print("save D attributes")
```

Multiple inheritance

```
class A(object):
    def save(self):
        print("save A attributes")
class B(A):
    def save(self):
        super().save() # save A attributes
        print("then save B attributes")
class C(A):
    def save(self):
        super().save() # save A attributes
        print("then save C attributes")
class D(B,C):
    def save(self):
        B.save(self) # save A then B attributes
        C.save(self) # save A then C attributes  # A saved twice!!
        print("then save D attributes")
```

Delegation

```
class C(A,B):
   …
   def f(…):
      self.method_from_B(…)
```

Alternative to multiple inheritance. A must in languages that don't support it

C objects are not B object

→

```
class C(A):
    def __init__(self):
       my_B_object = B()

    def f(…):
       self.my_B_object.method_from_B(…)
```

Rather, C objects contain a B object that handles calls to C with methods from B

Polymorphism

"len", "print" are polymorphic operations

(different types implement it; parameter decides which implementation is executed)

- len(lst): length of list lst
- len(d): number of pairs in dictionary d
- len(i), with i an integer, is an error

- len(o) actually calls o.__len(o)__
- so it is *int's* fault, not *len's* fault

- same with print

Conclusion

- Inheritance is central to OO
- Leave it to each object to choose the code to execute for one method call
- Class diagrams
- Overriding
- Calling parent's method
- Method resolution order
- Multiple inheritance

- But why? Design advantages? Labs & Soft. Eng. course
- Open-Closed principle