7. Object-Oriented Programming, II
Programming and Algorithms II
Degree in Bioinformatics
Fall 2017
Inheritance

class Carnivora(Animal):
    ...

subclass or child class

class Canid(Carnivora):
    ...

superclass or parent class

Dog “inherits” all the

properties from Canid, etc.

class Dog(Canid):
    ...

in Python, every class is a

subclass of object

(in Python 3, you can omit

writing it)

class Husky(Dog):
    ...

class Fox(Canid):
    ...

class Cat(Carnivora):
    ...

class diagram = hierarchy
class Carnivora(Animal):
    ...

class Canid(Carnivora):
    ...

class Dog(Canid):
    ...

class Husky(Dog):
    ...

class Fox(Canid):
    ...

class Cat(Carnivora):
    ...

Inheritance

Animal
Carnivora
Canid
Dog
Fox
Husky
Cat
object

Dog
Fox
Husky
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

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

class Cat(Animal):
    def say_something(self):
        print(self.name, "says: Meeooww!")

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

class Person(Animal):
    def say_something(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 `say_something`.
Method is chosen by the object

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

class Cat(Animal):
    def say_something(self):
        print(self.name, "says: Meeooww!")

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

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

microbe = Animal("Joe Amoeba")
microbe.say_something()

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

my_dog = Cat("Rintintin")
my_dog.say_something()

agent_007 = Person("Bond, James Bond")
agent_007.say_something()

Output:

Error: Joe Amoeba can't talk!
Garfield says: Meeooww!
Rintintin says: Meeooww!
Bond, James Bond says: 'My name is Bond, James Bond'
Method is chosen by the object

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

class Cat(Animal):
    def say_something(self):
        print(self.name, "says: Meeooww!")

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

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

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

milou = Super_Dog("Milou")
milou.say_something()

Output:

Milou says: My name is Milou and I am a genetically engineered dog
class A:
    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:
    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:
    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 or x

```python
>>> 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 or x

```python
>>> 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)`

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

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

`hasattr(o,’name’):` True if o has a property called ‘name’

```python
>>> hasattr(p,’mod’)
True
```

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

14
Method Resolution Order (MRO)

class A:
    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:
    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()
class A:
    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:
    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 (??)
        print("then save D attributes")
class C(A,B):
    ...
    def f(...):
        self.method_from_B(...)

→

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

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

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

C objects are not B object

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