HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Chapter 8:
Object-oriented Programming

Contents

- Objects
- Create your own type
- Methods and attributes
- Getter and Setter methods
- Inheritance

Objects

- Python supports many different kinds of data

```
1234  3.14159    "Hello"    [1, 5, 7, 11, 13]
{"CA": "California", "MA": "Massachusetts"}
```

- each is an object, and every object has:
  - ✓a type
  - ✓an internal data representation (primitive or composite)
  - ✓a set of procedures for interaction with the object
- an object is an instance of a type
  - ✓1234 is an instance of an int
  - ✓"hello" is an instance of a string

Object oriented programming (oop)

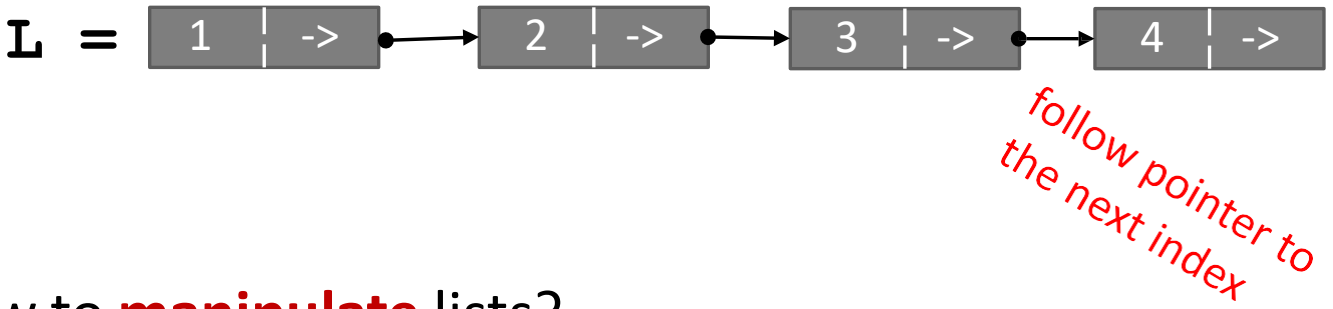
- EVERYTHING IN PYTHON IS AN OBJECT (and has a type)
- can create new objects of some type
- can manipulate objects
- can destroy objects
  - ✓explicitly using `del` or just “forget” about them
  - ✓python system will reclaim destroyed or inaccessible objects – called “garbage collection”

What are objects?

- objects are a data abstraction
- that captures…
- an internal representation
  - ✓through data attributes
- an interface for interacting with object
  - ✓through methods
- (aka procedures/functions)
  - ✓defines behaviors but hides implementation

Example: [1,2,3,4] has type list

- how are lists **represented internally**? linked list of cells



- how to **manipulate** lists?

```
L[i], L[i:j], +
len(), min(), max(), del(L[i])
L.append(),L.extend(),L.count(),L.index(),
L.insert(),L.pop(),L.remove(),L.reverse(), L.sort()
```

- internal representation should be private
- correct behavior may be compromised if you manipulate internal representation directly

Advantages of OOP

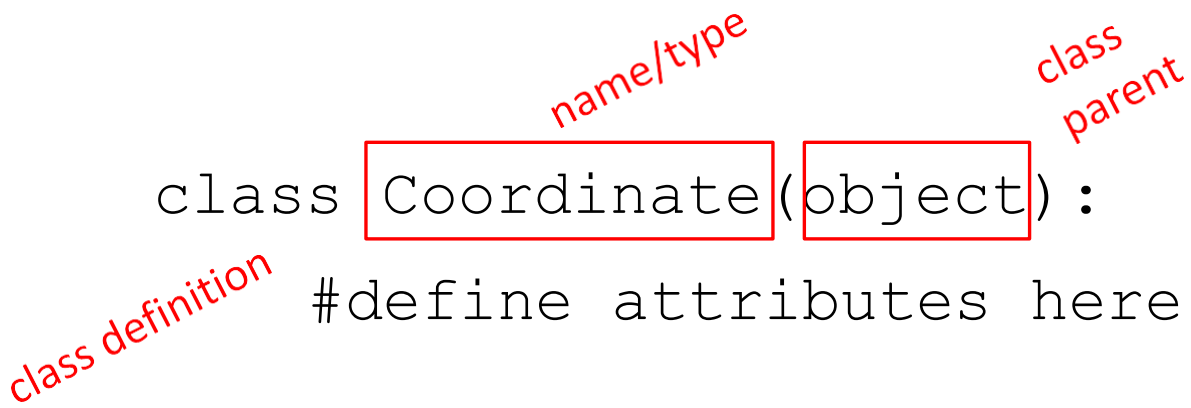
- bundle data into packages together with procedures that work on them through well-defined interfaces
- divide-and-conquer development
  - ✓implement and test behavior of each class separately
  - ✓increased modularity reduces complexity
- classes make it easy to reuse code
  - ✓many Python modules define new classes
  - ✓each class has a separate environment (no collision on function names)
  - ✓inheritance allows subclasses to redefine or extend a selected subset of a superclass' behavior

Creating and using your own types with classes

- make a distinction between creating a class and using an instance of the class
- creating the class involves
  - ✓defining the class name
  - ✓defining class attributes
  - ✓for example, someone wrote code to implement a list class
- using the class involves
  - ✓creating new instances of objects
  - ✓doing operations on the instances
  - ✓for example, L=[1,2] and len(L)

Define your own types

- use the `class` keyword to define a new type



- similar to `def`, indent code to indicate which statements are part of the **class definition**
- the word object means that Coordinate is a Python object and **inherits** all its attributes (inheritance next lecture)
  - ✓ Coordinateis a subclass of object
  - ✓ objectis a superclass of Coordinate

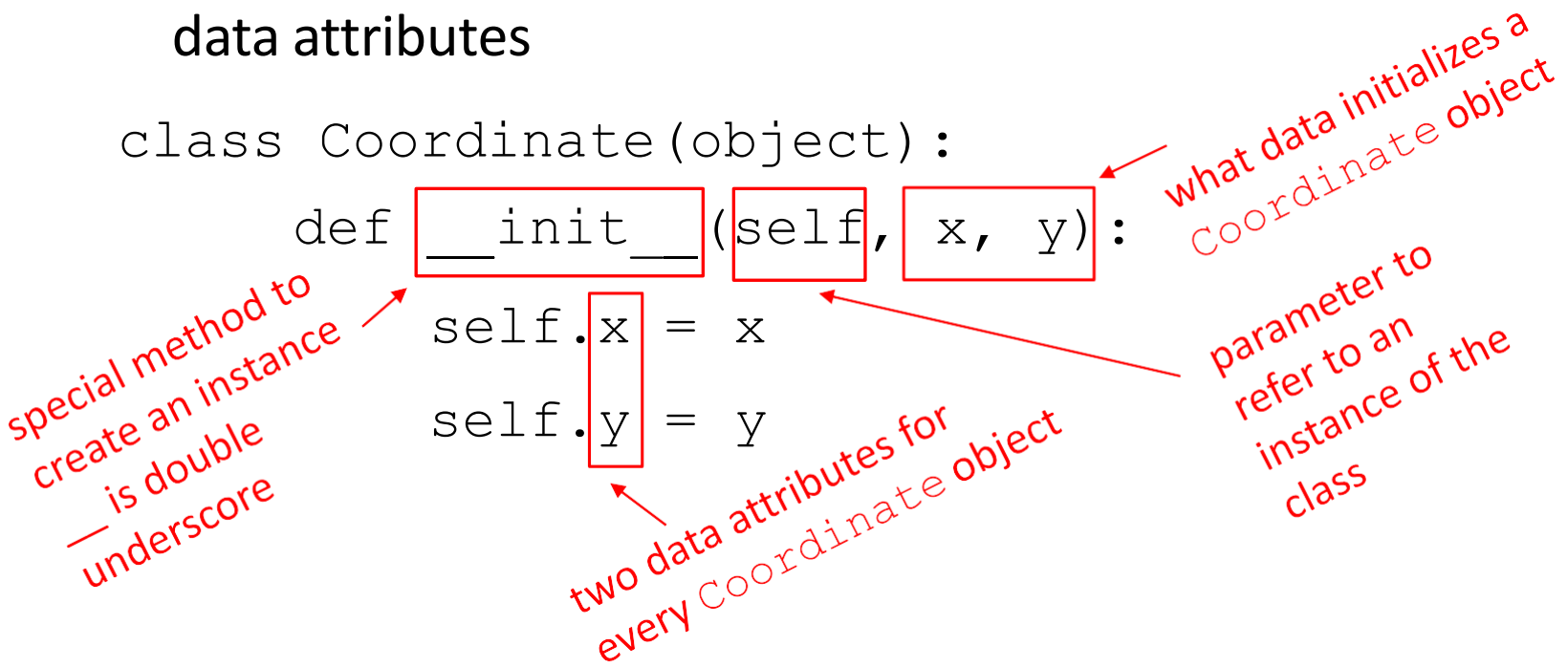
What are attributes?

- data and procedures that "belong" to the class
- data attributes
  - ✓think of data as other objects that make up the class
  - ✓for example, a coordinate is made up of two numbers
- methods (procedural attributes)
  - ✓think of methods as functions that only work with this class
  - ✓how to interact with the object
  - ✓for example you can define a distance between two coordinate objects but there is no meaning to a distance between two list objects

Defining how to create an instance of a class

- first have to define **how to create an instance** of object
- use a **special method called `__init__`** to initialize some data attributes

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```



Actually creating an instance of a class

```
c = Coordinate(3,4)
origin = Coordinate(0,0)
print(c.x)
print(origin.x)
```

create a new object of type `Coordinate` and pass in 3 and 4 to the `__init__`

use the dot to access an attribute of instance `c`

- data attributes of an instance are called **instance variables**
- don't provide argument for `self`, Python does this automatically

What is a method?

- procedural attribute, like a function that works only with this class
- Python always passes the object as the first argument
  - ✓ convention is to use self as the name of the first argument of all methods
- the "." operator is used to access any attribute
  - ✓a data attribute of an object
  - ✓a method of an object

Define a method for the Coordinate class

- other than `self` and dot notation, methods behave just like functions (take params, do operations, return)

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def distance(self, other):
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5
```

use it to refer to any instance

another parameter to method

dot notation to access data

How to use a method

```
def distance(self, other):
    # code here
```

method def

Using the class:

- conventional way

```
c = Coordinate(3,4)
zero = Coordinate(0,0)
print(c.distance(zero))
```

object to call method on

name of method

parameters not including `self` (`self` is implied to be `c`)

- equivalent to

```
c = Coordinate(3,4)
zero = Coordinate(0,0)
print(Coordinate.distance(c, zero))
```

name of class

name of method

parameters, including an object to call the method on, representing `self`

Print representation of an object

```
>>> c = Coordinate(3,4)
>>> print(c)
<_main_.Coordinate object at 0x7fa918510488>
```

- **uninformative** print representation by default
- define a **`__str__` method** for a class
- Python calls the `__str__` method when used with printon your class object
- you choose what it does! Say that when we print a `Coordinate` object, want to show

```
>>> print(c)
<3,4>
```

Defining your own print method

```
class Coordinate(object):
    def  __init__ (self, x, y):
        self.x = x
        self.y = y
    def distance(self, other):
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5
    def __str__ (self):
        return "<"+str(self.x)+","+str(self.y)+">"
```

name of special method

must return a string

Wrapping your head around types and classes

- can ask for the type of an object instance

```
>>> c = Coordinate(3,4)
>>> print(c)
<3,4>
>>> print(type(c))
<class __main_.Coordinate>
```

return of the __str__ method

the type of object c is a class Coordinate

- this makes sense since

```
>>> print(Coordinate)
<class __main_.Coordinate>
>>> print(type(Coordinate))
<type 'type'>
```

a Coordinate is a class

a Coordinate class is a type of object

- use `isinstance()` to check if an object is a `Coordinate`

```
>>> print(isinstance(c, Coordinate))
True
```

Special operators

- +, -, ==, <, >, len(), print, and many others
- https://docs.python.org/3/reference/datamodel.html#basic-customization
- like print, can override these to work with your class
- define them with double underscores before/after

```
__add__(self, other)    →    self + other
__sub__(self, other)    →    self - other
__eq__(self, other)     →    self == other
__lt__(self, other)     →    self < other
__len__(self)           →    len(self)
__str__(self)           →    print self
```

... and others

Example: fractions

- create a **new type** to represent a number as a fraction
- **internal representation** is two integers: numerator and denominator
- **interface** a.k.a. **methods** a.k.a **how to interact** with
- Fraction objects
  - ✓ add, subtract
  - ✓ print representation, convert to a float
  - ✓ invert the fraction
- the code for this is in the handout, check it out!

The power of OOP

- bundle together objects that share
  - common attributes and
  - procedures that operate on those attributes
- use abstraction to make a distinction between how to implement an object vs how to use the object
- build layers of object abstractions that inherit behaviors from other classes of objects
- create our own classes of objects on top of Python's basic classes

Implementing the class vs using the class

- write code from two different perspectives

implementing a new object type with a class

- ✓ **define** the class
- ✓ define **data attributes** (WHAT IS the object)
- ✓ define **methods** (HOW TO use the object)

using the new object type in code

- ✓ create **instances** of the object type
- ✓ do **operations** with them

Class definition of an object type vs instance of a class

- class name is the type

```
class Coordinate(object)
```

- class is defined generically
  - ✓use self to refer to some instance while defining the class

```
(self.x - self.y)**2
```

  - ✓self is a parameter to methods in class definition
- class defines data and methods common across all instances

- instance is one specific object

```
coord = Coordinate(1,2)
```

- data attribute values vary between instances

```
c1 = Coordinate(1,2)
c2 = Coordinate(3,4)
```

  - ✓ `c1` and `c2` have different data attribute values `c1.x` and `c2.x` because they are different objects
- instance has the structure of the class

Why use OOP and classes of objects?

- mimic real life
- group different objects part of the same type

Jelly
1 year old
brown

5 years old
brown

Tiger
2 years old
brown

Bean
0 years old
black

2 years old
white

1 year old
b/w

Image Credits, clockwise from top: Image Courtesy Harald Wehner, in the public Domain. Image Courtesy MTSOfan, CC-BY-NC-SA. Image Courtesy Carlos Solana, license CC-BY-NC-SA. Image Courtesy Rosemarie Banghart-Kovic, license CC-BY-NC-SA. Image Courtesy Paul Reynolds, license CC-BY. Image Courtesy Kenny Louie, License CC-BY

Why use OOP and classes of objects?

- mimic real life
- group different objects part of the same type

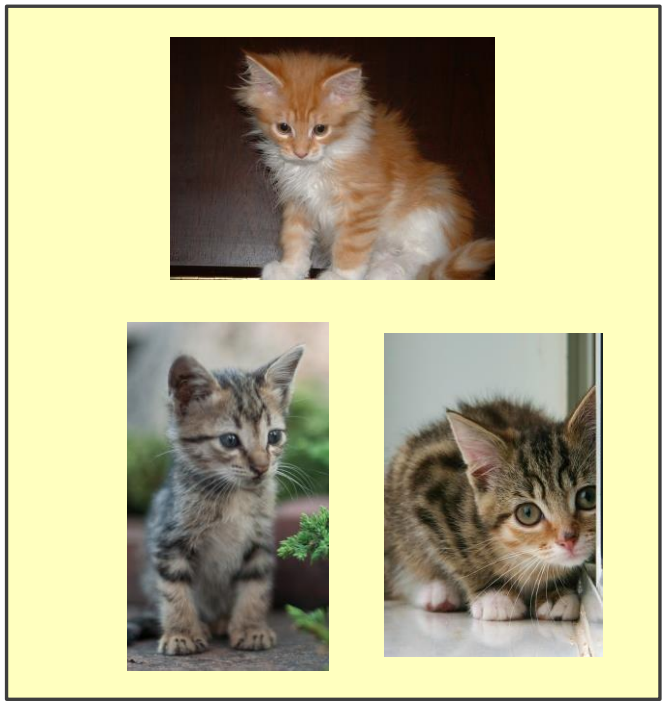
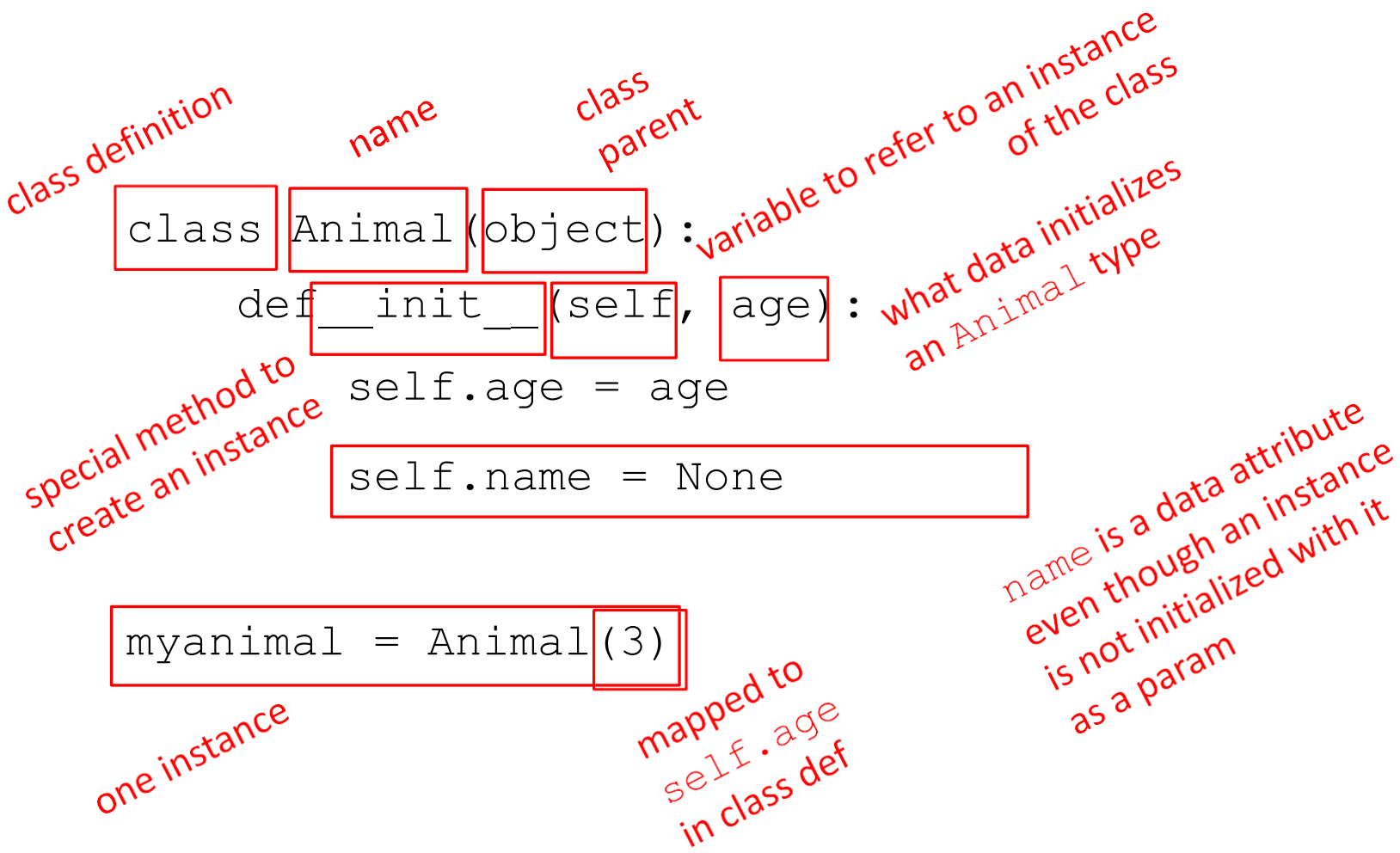


Image Credits, clockwise from top: Image Courtesy Harald Wehner, in the public Domain. Image Courtesy MTSOfan, CC-BY-NC-SA. Image Courtesy Carlos Solana, license CC-BY-NC-SA. Image Courtesy Rosemarie Banghart-Kovic, license CC-BY-NC-SA. Image Courtesy Paul Reynolds, license CC-BY. Image Courtesy Kenny Louie, License CC-BY

Groups of objects have attributes (recap)

- data attributes
  - how can you represent your object with data?
  - what it is
  - for a coordinate: x and y values
  - for an animal: age, name
- procedural attributes (behavior/operations/methods)
  - how can someone interact with the object?
  - what it does
  - for a coordinate: find distance between two
  - for an animal: make a sound

How to define a class (recap)



Getter and setter methods

- **getters and setters** should be used outside of class to access data attributes

```
class Animal(object):
    def __init__ (self, age):
        self.age = age
        self.name = None
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name
    def set_age(self, newage):
        self.age = newage
    def set_name(self, newname=""):
        self.name = newname
    def str (self):
        return "animal:"+str(self.name)+":"+str(self.age)
```

getter

setter

An instance and Dot notation (recap)

- instantiation creates an instance of an object

```
a = Animal(3)
```

- **dot notation** used to access attributes (data and methods) though it is better to use getters and setters to access data attributes

```
a.age
```

- access data attribute
- allowed, but not recommended

```
a.get_age()
```

- access method
- best to use getters and setters

Information hiding

- author of class definition may **change data attribute** variable names

```
class Animal(object):
    def __init__(self, age):
        self.years = age
    def get_age(self):
        return self.years
```

replaced age data attribute by years

- if you are **accessing data attributes** outside the class and class **definition changes**, may get errors

- outside of class, use getters and setters instead use `a.get_age()` NOT `a.age`
  - ✓ good style
  - ✓ easy to maintain code
  - ✓ prevents bugs

Python not great at information hiding

- allows you to access data from outside class definition

```
print(a.age)
```

- allows you to write to data from outside class definition

```
a.age = 'infinite'
```

- allows you to create data attributes for an instance from outside class definition

```
a.size = "tiny"
```

- it's not good style to do any of these!

Default arguments

- **default arguments** for formal parameters are used if no actual argument is given

```
def set_name(self, newname=""):
    self.name = newname
```

- default argument used here

```
a = Animal(3)
a.set_name()
print(a.get_name())
```

prints ""

- argument passed in is used here

```
a = Animal(3)
a.set_name("fluffy")
print(a.get_name())
```

prints "fluffy"

Hierarchies

Animal

People

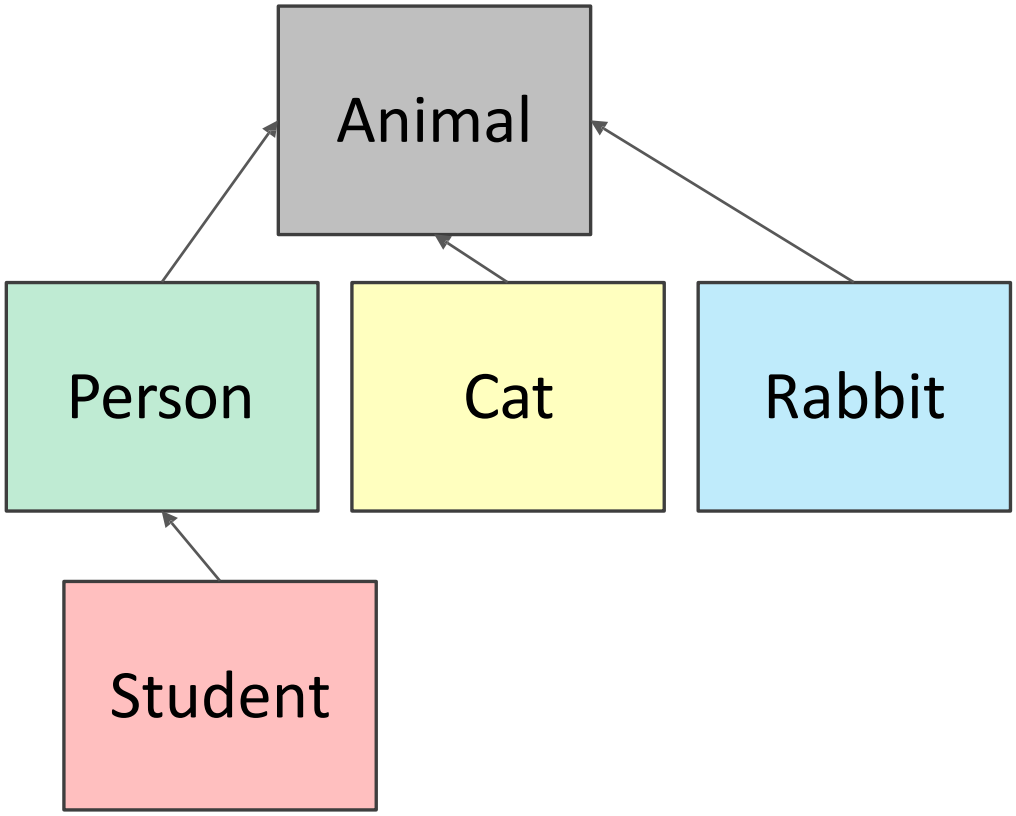
Student

Cat

Rabbit

Hierarchies

- **parent class** (superclass)
- **child class** (subclass)
  - ✓ **inherits** all data and behaviors of parent class
  - ✓ **add** more **info**
  - ✓ **add** more **behavior**
  - ✓ **override** behavior



Inheritance: parent class

```
class Animal(object):
    def __init__ (self, age):
        self.age = age
        self.name = None
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name
    def set_age(self, newage):
        self.age = newage
    def set_name(self, newname=""):
        self.name = newname
    def  __str__ (self):
        return "animal:"+str(self.name)+":"+str(self.age)
```

everything is an object
- class object
implements basic
operations in Python, like
binding variables, etc

Inheritance: subclass

inherits all attributes of Animal:
__init__()
age, name
get_age(), get_name()
set_age(), set_name()
__str__()

```
class Cat(Animal):
    def speak(self):
        print("meow")
    Def __str__(self):
        return "cat:"+str(self.name)+":"+str(self.age)
```

add new functionality via speak method

overrides __str__

- add new functionality with speak()
  - ✓ instance of type Cat can be called with new methods
  - ✓ instance of type Animalthrows error if called with Cat's new method
- __init__ is not missing, uses the Animalversion

Which method to use?

- subclass can have methods with same name as superclass
- for an instance of a class, look for a method name in current class definition
- if not found, look for method name up the hierarchy (in parent, then grandparent, and so on)
- use first method up the hierarchy that you found with that method name

```
class Person(Animal):
    def __init__(self, name, age):
        Animal.  init  (self, age)
        self.set_name(name)
        self.friends = []
    def get_friends(self):
        return self.friends
    def add_friend(self, fname):
        if fname not in self.friends:
            self.friends.append(fname)
    def speak(self):
        print("hello")
    def age_diff(self, other):
        diff = self.age - other.age
        print(abs(diff), "year difference")
    Def __str__(self):
        return "person:"+str(self.name)+":"+str(self.age)
```

parent class is Animal

call Animal constructor

call Animal's method

add a new data attribute

new methods

override Animal's __str__ method

```
import random
class Student(Person):
    def __init__ (self, name, age, major=None):
        Person. init (self, name, age)
        self.major = major
    def change_major(self, major):
        self.major = major
    def speak(self):
        r = random.random()
        if r < 0.25:
            print("i have homework")
        elif 0.25 <= r < 0.5:
            print("i need sleep")
        elif 0.5 <= r < 0.75:
            print("i should eat")
        else:
            print("i am watching tv")
    def __str__ (self):
        return "student:"+str(self.name)+":"+str(self.age)+":"+str(self.major)
```

bring in methods from random class

inherits Person and Animal attributes

adds new data

- I looked up how to use the random class in the python docs
- random() method gives back float in [0, 1)

Class variables and the Rabbit subclass

- **class variables** and their values are shared between all instances of a class

```
class Rabbit(Animal):
    tag = 1
    def __init__(self, age, parent1=None, parent2=None):
        Animal._init_(self, age)
        self.parent1 = parent1
        self.parent2 = parent2
        self.rid = Rabbit.tag
        Rabbit.tag += 1
```

parent class

class variable

instance variable

access class variable

incrementing class variable changes it for all instances that may reference it

- `tag` used to give **unique id** to each new rabbit instance

Rabbit GETTER methods

```
class Rabbit(Animal):
    tag = 1
    def __init__(self, age, parent1=None, parent2=None):
        Animal._init_(self, age)
        self.parent1 = parent1
        self.parent2 = parent2
        self.rid = Rabbit.tag
        Rabbit.tag += 1
    def get_rid(self):
        return str(self.rid).zfill(3)
    def get_parent1(self):
        return self.parent1
    def get_parent2(self):
        return self.parent2
```

method on a string to pad the beginning with zeros for example, 001 not 1

- getter methods specific for a `Rabbit` class
- there are also getters `get_name` and `get_age` inherited from `Animal`

Working with your own types

```
def __add__ (self, other):
    # returning object of same type as this class
    return Rabbit(0, self, other)
```

recall Rabbit's init (self, age, parent1=None, parent2=None)

- define + operator between two `Rabbit` instances
  - define what something like this does: `r4 = r1 + r2`
- where `r1` and `r2` are `Rabbit` instances
  - `r4` is a new `Rabbit` instance with age 0
  - `r4` has self as one parent and `other` as the other parent
  - in `__init__`, parent1 and parent2 are of type Rabbit

Special method to compare two Rabbits

- decide that two rabbits are equal if they have the **same two parents**

```
def __eq__(self, other):
    parents_same = self.parent1.rid == other.parent1.rid \
                   and self.parent2.rid == other.parent2.rid
    parents_opposite = self.parent2.rid == other.parent1.rid \
                       and self.parent1.rid == other.parent2.rid
    return parents_same or parents_opposite
```

booleans

- compare ids of parents since **ids are unique** (due to class var)
- note you can't compare objects directly
  - for ex. with `self.parent1 == other.parent1`
  - this calls the `__eq__` method over and over until call it on `None` and gives an `AttributeError` when it tries to do `None.parent1`

Object oriented programming

- create your own **collections of data**
- **organize** information
- **division** of work
- access information in a **consistent** manner
- add **layers** of complexity
- like functions, classes are a mechanism for **decomposition** and **abstraction** in programming

References

1. MIT Introduction to Computer Science and Programming in Python

ĐẠI HỌC BÁCH KHOA

25 YEARS ANNIVERSARY SOICT

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Thank you for your attention!

soict.hust.edu.vn/ fb.com/groups/soict