MASTER IN ENTREPRENEURSHIP
INNOVATION MANAGEMENT
IN COLLABORATION WITH MIT SLOAN

IN COLLABORATION WITH
MIT MANAGEMENT SLOAN SCHOOL

UNIVERSITÀ DEGLI STUDI DI NAPOLI PARTHENOPE

MASTER MEIM 2021-2022

Python Programming Course Lesson 6

Object Oriented Programming

Lesson given by prof. Mariacarla Staffa

Prof. Of Computer Science at the University of Naples Parthenope

Object-Oriented Programming

AIM

- using classes to organize programs around modules and data abstractions

LEARNING OUTCOMES

At the end of the lesson, you are expected:

- To understand the concepts of classes, objects and encapsulation
- To implement instance variables, methods and constructors
- To be able to design, implement, and test your own classes
- To understand the behavior of object references

Object-Oriented Programming

- You have learned structured programming
  - Breaking tasks into subtasks
  - Writing re-usable methods to handle tasks
- We will now study Objects and Classes
  - To build larger and more complex programs
  - To model objects we use in the world

A class describes objects with the same behavior.
For example, a Car class describes all passenger vehicles that have a certain capacity and shape.

Objects and Programs

- You have learned how to structure your programs by decomposing tasks into functions
  - Experience shows that it does not go far enough
  - It is difficult to understand and update a program that consists of a large collection of functions
- To overcome this problem, computer scientists invented **object-oriented programming**, a programming style in which tasks are solved by collaborating objects
- Each object has its own set of data, together with a set of methods that act upon the data

Objects and Programs

- You have already experienced this programming style when you used strings, lists, and file objects. Each of these objects has a set of methods
- For example, you can use the `insert()` or `remove()` methods to operate on list objects

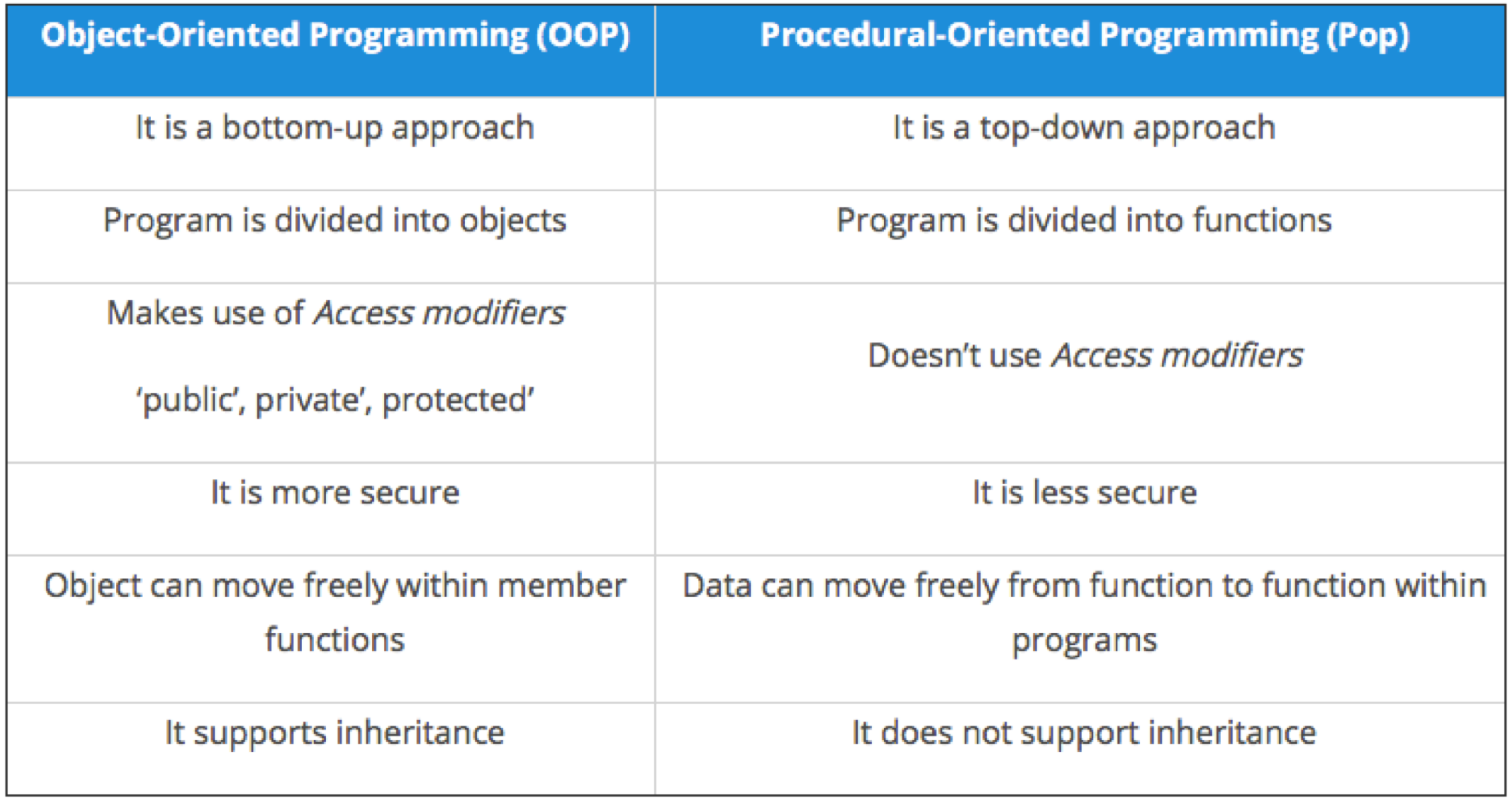
INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING IN PYTHON

Object Oriented Programming is a way of computer programming using the idea of "objects" to represents data and methods.

It is also, an approach used for creating neat and reusable code instead of a redundant one.

The program is divided into self-contained objects or several mini-programs. Every Individual object represents a different part of the application having its own logic and data to communicate within themselves.

Difference between Object-Oriented and Procedural Oriented Programming



Object-Oriented Programming (OOP)	Procedural-Oriented Programming (Pop)
It is a bottom-up approach	It is a top-down approach
Program is divided into objects	Program is divided into functions
Makes use of *Access modifiers* ‘public’, private’, protected’	Doesn’t use *Access modifiers*
It is more secure	It is less secure
Object can move freely within member functions	Data can move freely from function to function within programs
It supports inheritance	It does not support inheritance

Object-Oriented Programming methodologies:

- Inheritance
- Polymorphism
- Encapsulation
- Abstraction

Inheritance

- From the Programming aspect ‘inheritance’ means “inheriting or transfer of characteristics from parent to child class without any modification”.
- The new class is called the **derived/child** class and the one from which it is derived is called a **parent/base** class.

Polimorphism

- You all must have used GPS for navigating the route, Isn't it amazing how many different routes you come across for the same destination depending on the traffic, from a programming point of view this is called 'polymorphism'. It is one such OOP methodology where one task can be performed in several different ways. To put it in simple words, *it is a property of an object which allows it to take multiple forms.*

Two types of Polimorphism

Run-time Polymorphism: A run-time Polymorphism is also, called as dynamic polymorphism where it gets resolved into the run time. One common example of Run-time polymorphism is "method overriding"

Compile-time Polymorphism: A compile-time polymorphism also called as static polymorphism which gets resolved during the compilation time of the program. One common example is "method overloading"

Encapsulation

- In a raw form, encapsulation basically means binding up of data in a single class.
- Python does not have any private keyword, unlike Java.
- A class shouldn’t be directly accessed but be prefixed in an underscore.

Abstraction

- Suppose you booked a movie ticket from sky using net banking or any other process. You don't know the procedure of how the pin is generated or how the verification is done. This is called 'abstraction' from the programming aspect, it basically means you only show the implementation details of a particular process and hide the details from the user. It is used to simplify complex problems by modeling classes appropriate to the problem.
- An abstract class cannot be instantiated which simply means you cannot create objects for this type of class.
- It can only be used for inheriting the functionalities.

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?

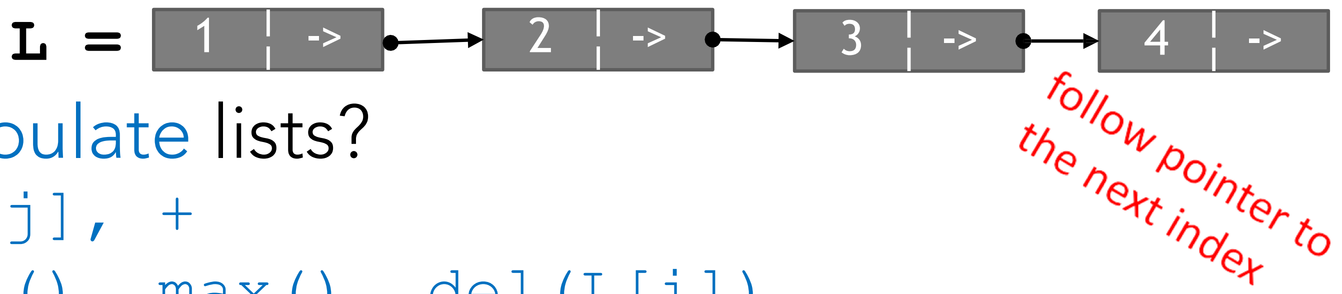
Objects are a data abstraction that captures…

1. an internal representation
   - through data attributes

2. an interface for interacting with object
   - through methods (aka procedures/functions)
   - defines behaviors but hides implementation

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

- howare 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

name/type

class parent

class definition

```
class Coordinate(object):
    #define attributes here
```

- 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)
  - `Coordinate` is a subclass of `object`
  - `object` is 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
```

special method to create an instance is double __ underscore

what data initializes a Coordinate object

parameter to refer to an instance of the class

two data attributes for every Coordinate object

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

```
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

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

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__.Coordinateobject at 0x7fa918510488
```

- uninformative print representation by defaul
- define a __str__method for a class
- Python calls the __str__ method when used with print on 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__(s elf):
       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 manyothers

https://docs.python.org/3/reference/datamodel.html#basic-customization

- like print, can override these to work with your class
- define them with double underscoresbefore/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
  - 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 will be presented during Lab lessons

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

2nd Part Inheritance

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

1 year old
b/w

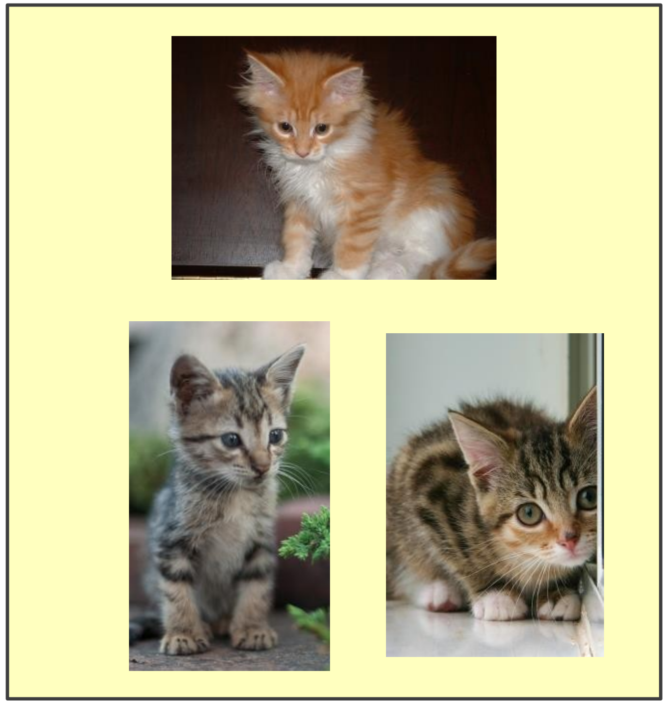
Bean
0 years old
black

Tiger
2 years old
brown

2 years old
white

WHY USE OOP AND CLASSES OF OBJECTS?

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



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 points*
  - *for an animal: make a sound*

HOW TO DEFINE A CLASS (RECAP)

```
class Animal(object):
    def __init__(self, age):
        self.age = age
        self.name = None

myanimal = Animal(3)
```

class definition

name

class parent

variable to refer to an instance of the class

what data initializes an `Animal` type

special method to create an instance

`name` is a data attribute even though an instance is not initialized with it as a param

one instance

mapped to `self.age` in class def

GETTER AND SETTER METHODS

```
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

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

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
```

```
a.get_age()
```

- access data attribute
- allowed, but not recommended

- access method
- best to use getters and setters

INFORMATION HIDING

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

replaced age data attribute by years

```
class Animal(object):
    def_init_(self, age):

        self.years = age
    def get_age(self):
        return self.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())
```

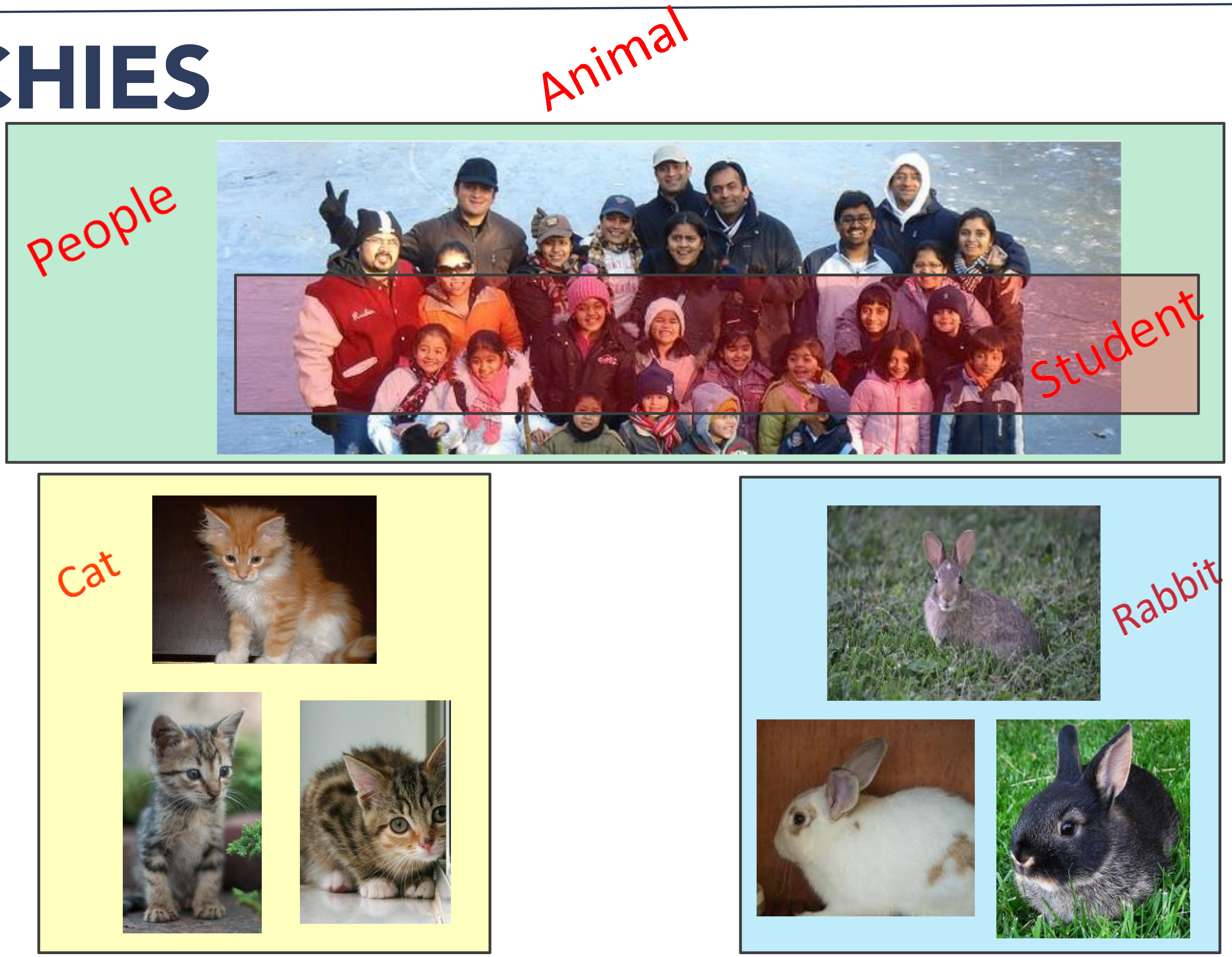
prints ""

- argument passed in is used here

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

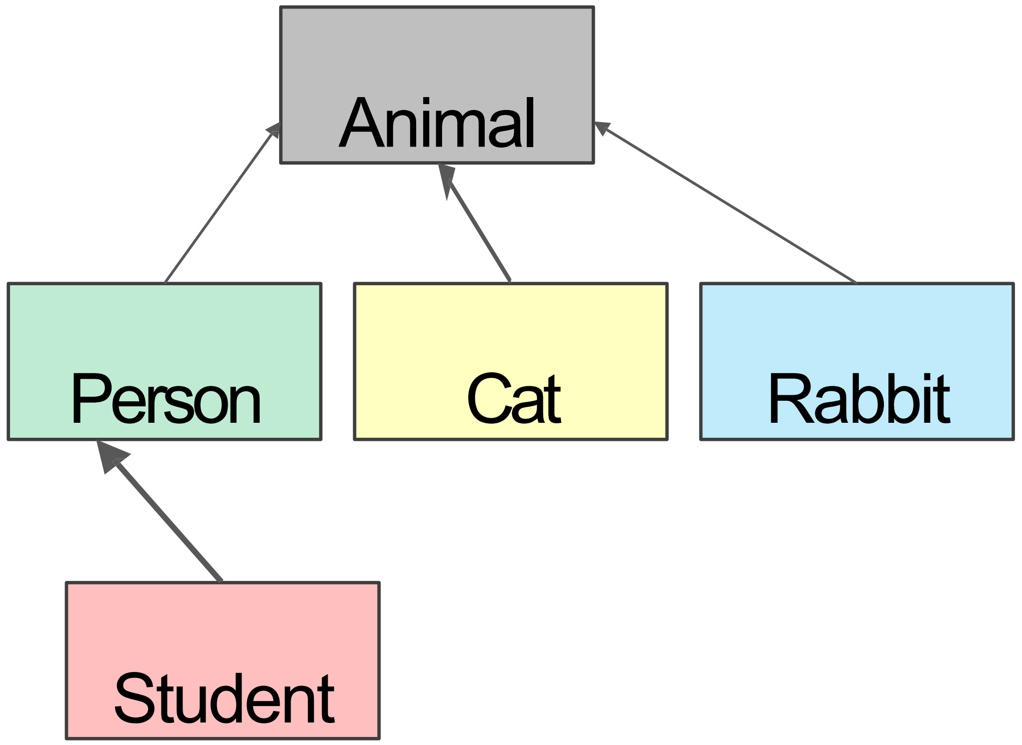
prints "fluffy"

HIERARCHIES



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

```
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__

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

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

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 soon)
- 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 r2are `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

MASTER MEIM 2021-2022

Thank you for your attention