PYTHON CLASSES and INHERITANCE

(download slides and .py files • • • •follow along!)

6.0001 LECTURE 9

LAST TIME

- abstract data types through classes
- `Coordinate` example
- `Fraction` example

TODAY

- more on classes
  - getters and setters
  - information hiding
  - class variables
- 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 name.

- 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)
```

Coord instance of class

- data attribute values vary between instances

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

CLR

  - `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

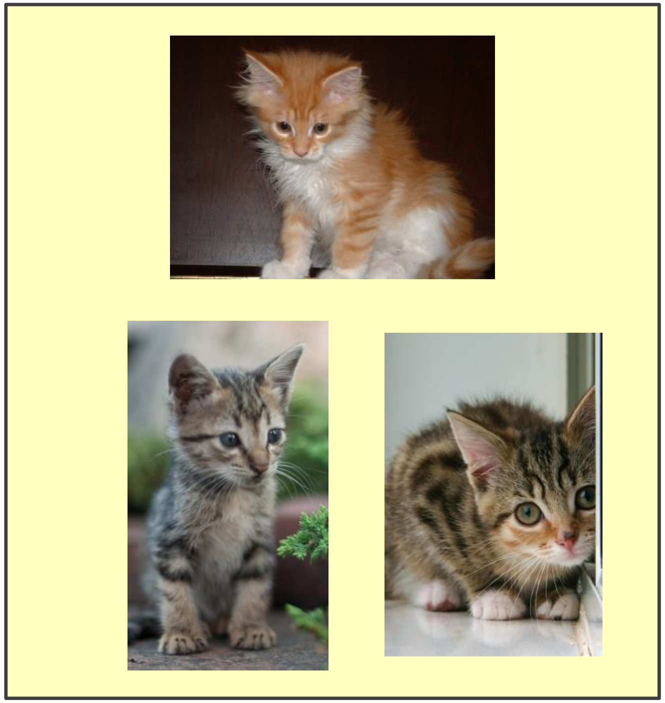


Image Credits, clockwise from top: Image Courtesy Harald Wehner, in the public Domain. Image Courtesy MTSOfan, CC-BY-NC-SA. Image Courtesy Carlos Solana, license CC-BY-NC-SA. Image Courtesy Rosemarie Banghart-Kovic, license CC-BY-NC-SA. Image Courtesy Paul Reynolds, license CC-BY. Image Courtesy Kenny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

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 of class

mapped to `self.age` in class def

GETTER AND SETTER METHODS

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

define class

getter

```
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name
```

setter

```
    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)
```

- **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

instance of age.

class name.

```
a.age
```

- access data attribute
- allowed, but not recommended

```
a.get_age()
```

function call

- 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
```

changing binding.

- 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)
```

class name

<=> ~~Must local variable~~

- 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())
```

function call.

prints ""

- argument passed in is used here

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

Assign. argument.

prints "fluffy"

HIERARCHIES

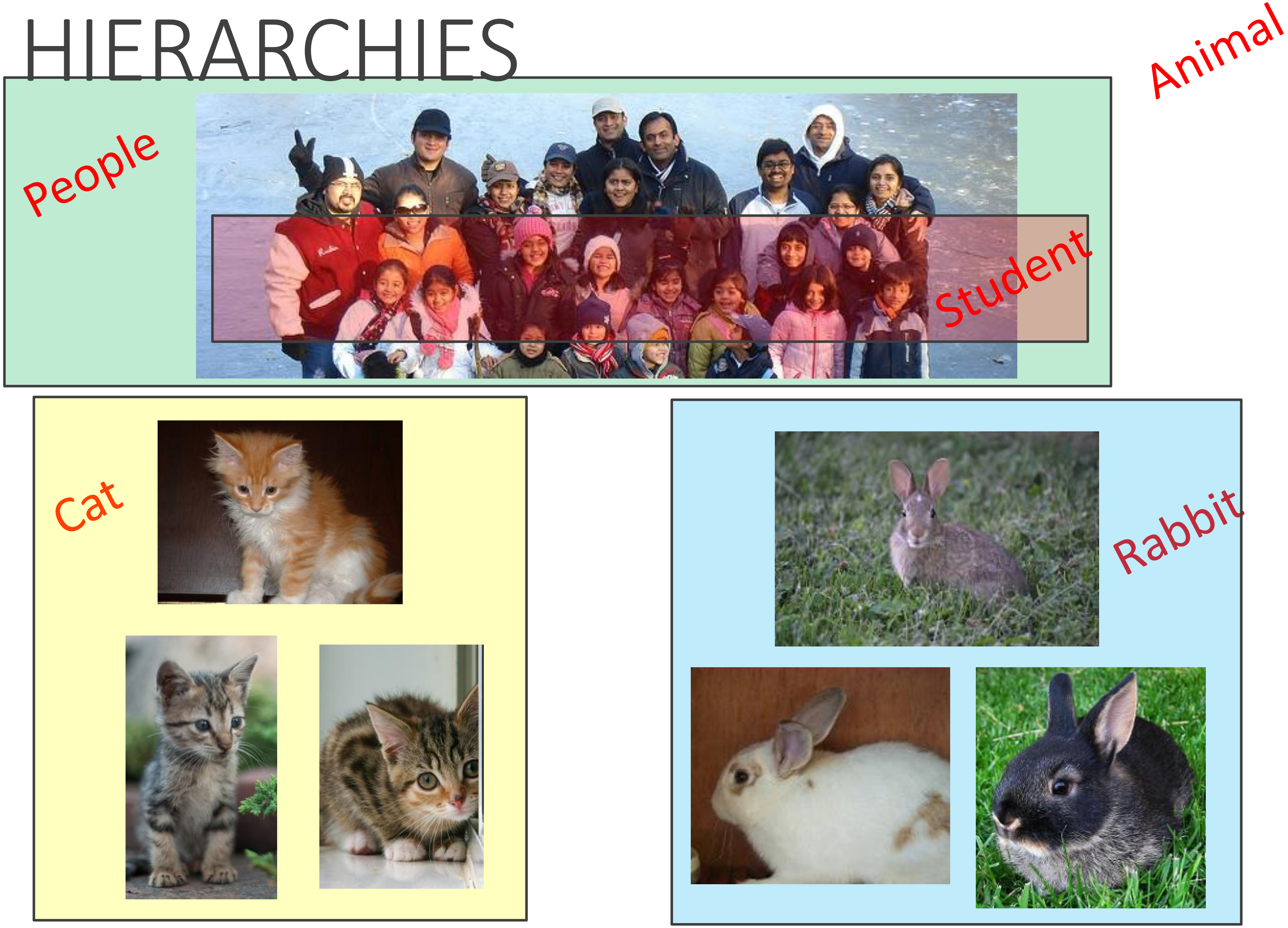
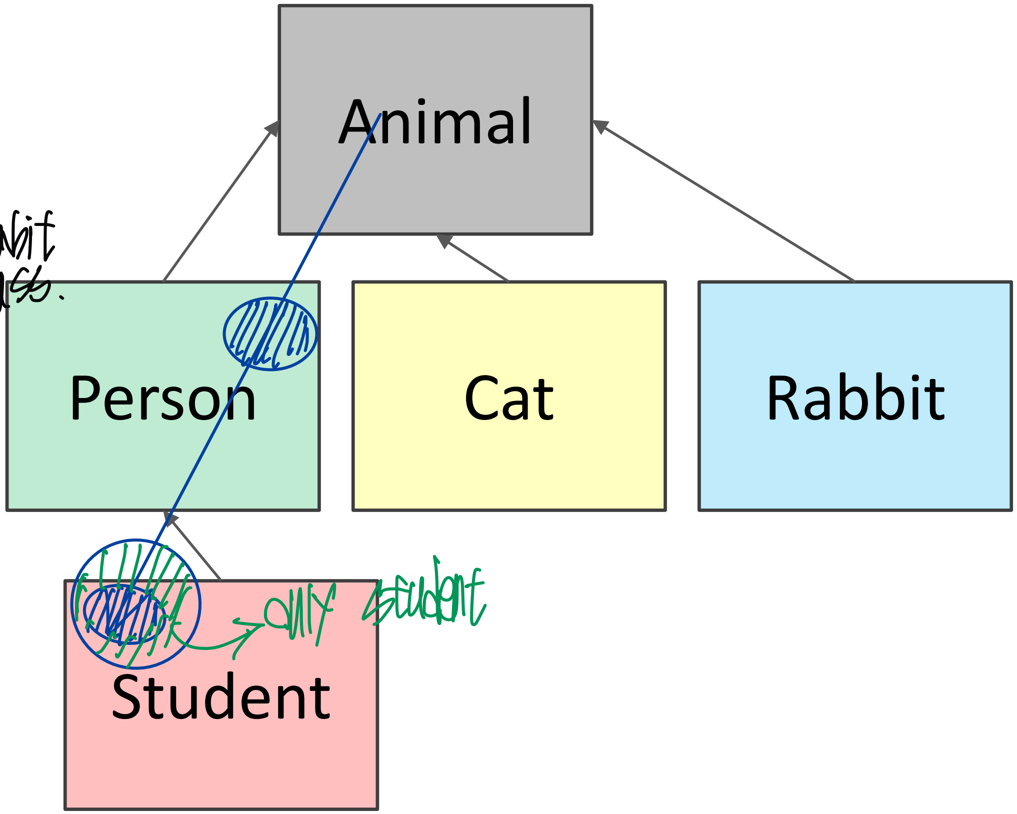


Image Credits, clockwise from top: Image Courtesy Deeeep, CC-BY-NC. Image Image Courtesy MTSOfan, CC-BY-NC-SA. Image Courtesy Carlos Solana, license CC-BY-NC-SA. Image Courtesy Rosemarie Banghart-Kovic, license CC-BY-NC-SA. Image Courtesy Paul Reynolds, license CC-BY. Image Courtesy Kenny Louie, License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

HIERARCHIES

- **parent class** - - - Animal class.
  (superclass)

- **child class** - - - Person class - cat class - rabbit 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 `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 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

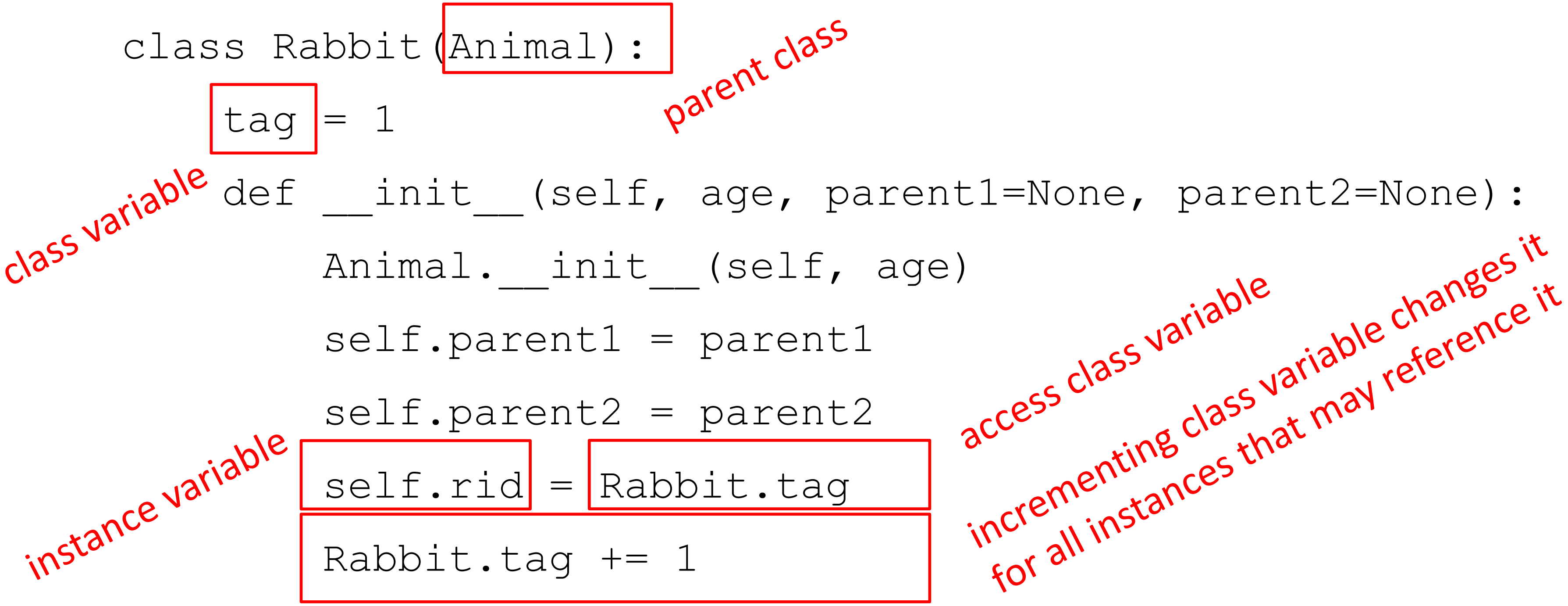
inherits `Person` and `Animal` attributes

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

access class variable

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

instance variable

- `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