INHERITANCE

(download slides and .py files to follow along)

6.100L Lecture 19

Ana Bell

WHY USE OOP AND CLASSES OF OBJECTS?

- Mimic real life
- Group different objects part of the same type

Instance:
Jelly
1 year old
brown

Instance:
5 years old
brown

Instance:
Tiger
2 years old
brown

Instance:
Bean
0 years old
black

Instance:
2 years old
white

Instance:
1 year old
b/w

WHY USE OOP AND CLASSES OF OBJECTS?

- Mimic real life
- Group different objects part of the same type

All instances of a type have the same data abstraction and behaviors

GROUPS OF OBJECTS HAVE ATTRIBUTES (RECAP)

- **Data attributes**
  - How can you represent your object with data?
  - **What it is**

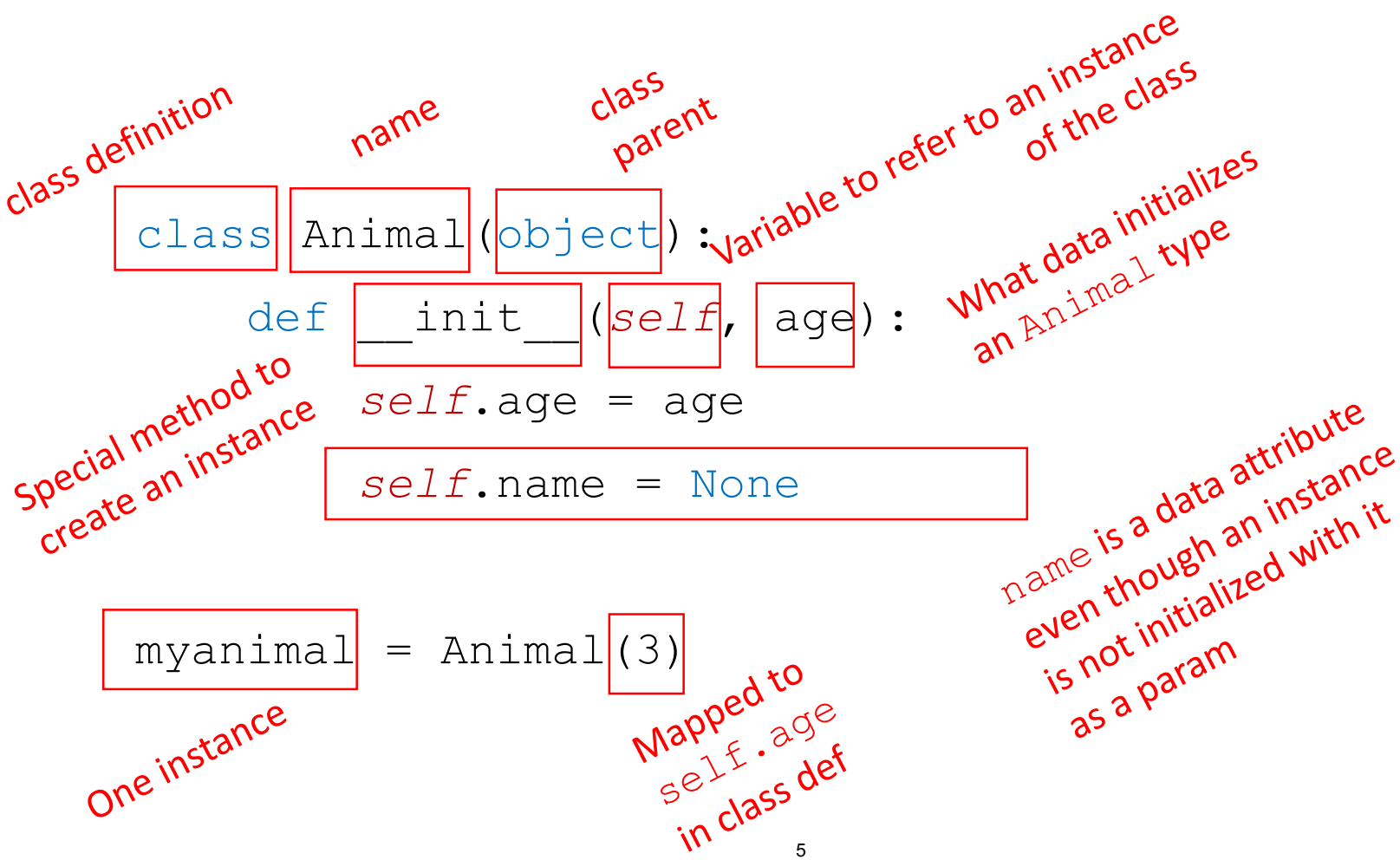
    for a coordinate: x and y values

    for an animal: age
- **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: print how long it's been alive

HOW TO DEFINE A CLASS (RECAP)



GETTER AND SETTER METHODS

```
class Animal(object):
    def __init__(self, age):
        self.age = age
        self.name = None
    def __str__(self):
        return "animal:"+str(self.name)+":"+str(self.age)
```

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

GETTER AND SETTER METHODS

```
class Animal(object):
    def __init__(self, age):
        self.age = age
        self.name = None
    def __str__(self):
        return "animal:"+str(self.name)+":"+str(self.age)
    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
```

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
```

- 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

CHANGING INTERNAL REPRESENTATION

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

Change internal rep from self.age = age

```
a.get_age()    # works
a.age          # error
```

Accessing methods works correctly, but accessing data attributes no longer works.

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

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!

USE OUR NEW CLASS

```
def animal_dict(L):
    """ L is a list
    Returns a dict, d, mappping an int to an Animal object.
    A key in d is all non-negative ints, n, in L. A value
    corresponding to a key is an Animal object with n as its age. """
    d = {}
    for n in L:
        if type(n) == int and n >= 0:
            d[n] = Animal(n)
    return d

L = [2,5,'a',-5,0]
```

Invoke the name of the class with parameter n (i.e. the age of that animal)

USE OUR NEW CLASS

- Python doesn't know how to call print recursively

```
def animal_dict(L):
    """ L is a list
    Returns a dict, d, mappping an int to an Animal object.
    A key in d is all non-negative ints n L. A value corresponding
    to a key is an Animal object with n as its age. """
    d = {}
    for n in L:
        if type(n) == int and n >= 0:
            d[n] = Animal(n)
    return d

L = [2,5,'a',-5,0]

animals = animal_dict(L)
print(animals)
```

Return is a dict mapping int:Animal

```
{2: <__main__.Animal object at 0x00000199AFF350A07',
 5: <__main__.Animal object at 0x00000199AFF35A307',
 0: <__main__.Animal object at 0x00000199AFF35D007}
```

USE OUR NEW CLASS

```
def animal_dict(L):
    """ L is a list
    Returns a dict, d, mappping an int to an Animal object.
    A key in d is all non-negative ints n L. A value corresponding
    to a key is an Animal object with n as its age. """
    d = {}
    for n in L:
        if type(n) == int and n >= 0:
            d[n] = Animal(n)
    return d

L = [2,5,'a',-5,0]

animals = animal_dict(L)
for n,a in animals.items():
    print(f'key {n} with val {a}')
```

Manually loop over animal objects and access their data attr through getter methods

```
key 2 with val animal:None:2
key 5 with val animal:None:5
key 0 with val animal:None:0
```

YOU TRY IT!

- Write a function that meets this spec.

```
def make_animals(L1, L2):
    """ L1 is a list of ints and L2 is a list of str
        L1 and L2 have the same length
    Creates a list of Animals the same length as L1 and L2.
    An animal object at index i has the age and name
    corresponding to the same index in L1 and L2, respectively. """

#For example:
L1 = [2,5,1]
L2 = ["blobfish", "crazyant", "parafox"]
animals = make_animals(L1, L2)
print(animals)     # note this prints a list of animal objects
for i in animals:  # this loop prints the individual animals
    print(i)
```

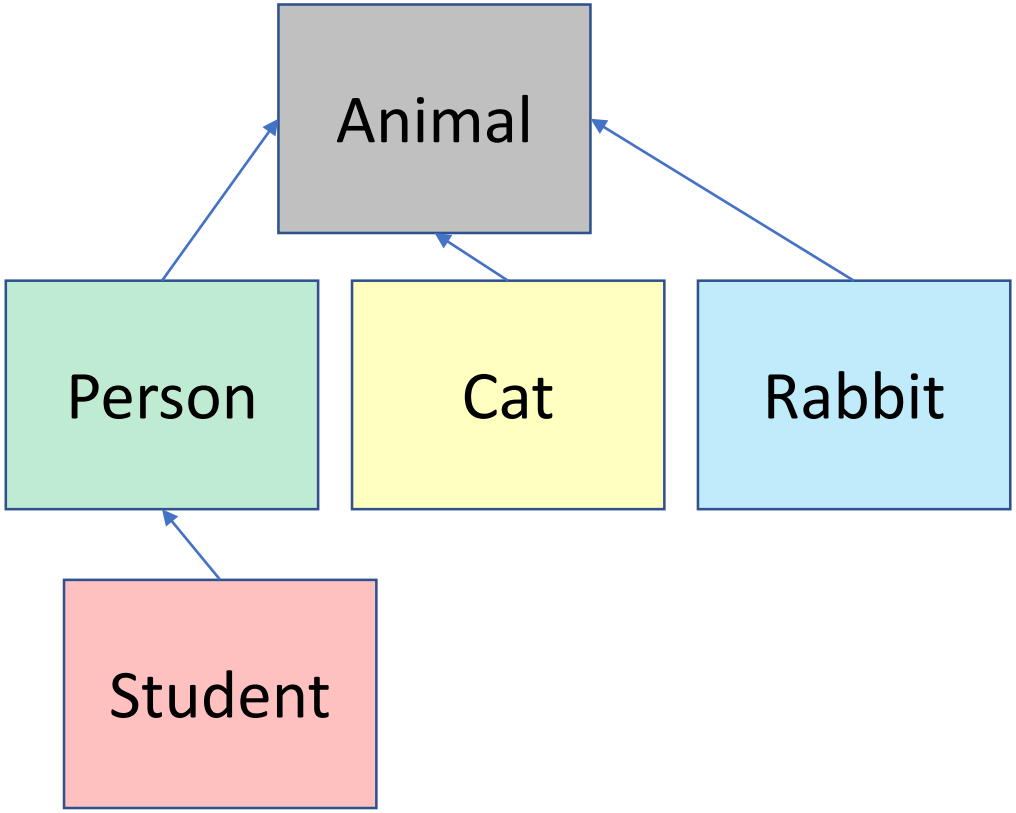
BIG IDEA

Access data attributes
(stuff defined by self.xxx)
through methods – it's better style.

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

SUBCLASS CAT

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

SUBCLASS PERSON

```
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.copy()
    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 to run lines of code in Animal's init

Call Animal's method

Add a new data attribute

New methods

Override Animal's __str__ method

YOU TRY IT!

- Write a function according to this spec.

```
def make_pets(d):
    """ d is a dict mapping a Person obj to a Cat obj
    Prints, on each line, the name of a person, a colon, and the
    name of that person's cat """
    pass

p1 = Person("ana", 86)
p2 = Person("james", 7)
c1 = Cat(1)
c1.set_name("furball")
c2 = Cat(1)
c2.set_name("fluffsphere")

d = {p1:c1, p2:c2}
make_pets(d)  # prints ana:furball
              #        james:fluffsphere
```

BIG IDEA

A subclass can
use a parent's attributes,
override a parent's attributes, or
define new attributes.

Attributes are either data or methods.

SUBCLASS STUDENT

```
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'm still zooming")
    def __str__(self):
        return "student:"+str(self.name)+":"+str(self.age)+":"+str(self.major)
```

Bring in functions from random library

Inherits Person and Animal attributes

Person __init__ takes care of all initializations

Adds new data

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

SUBCLASS RABBIT

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

Shared class variable

instance variable

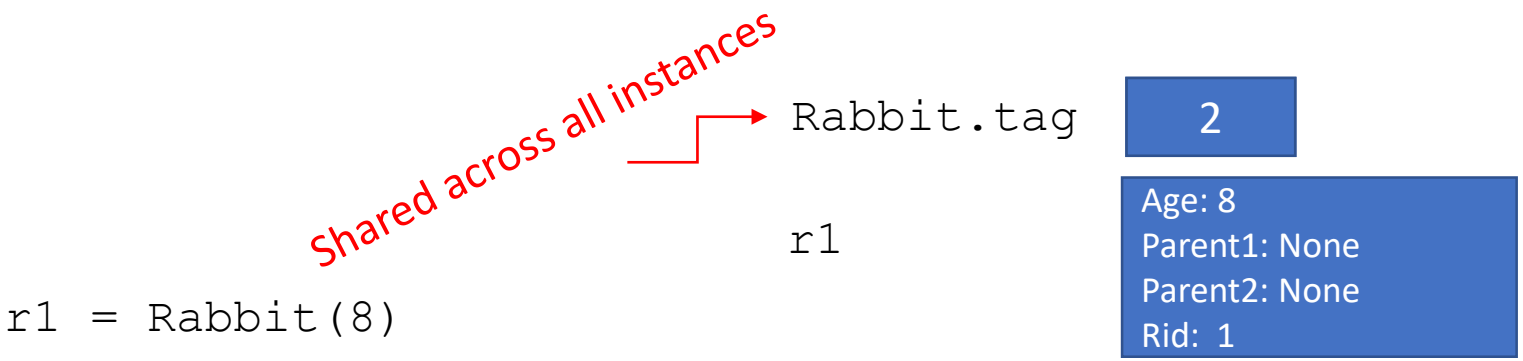
Access shared class variable

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

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

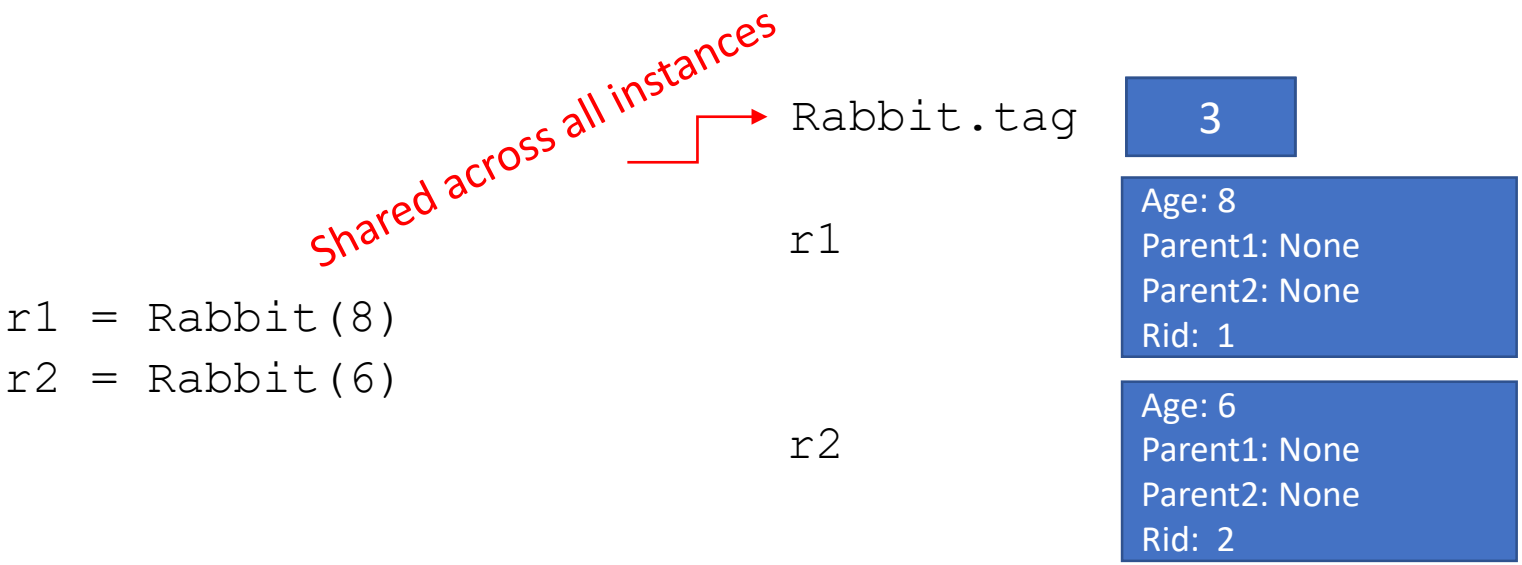
RECALL THE `__init__` OF `Rabbit`

```
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
```



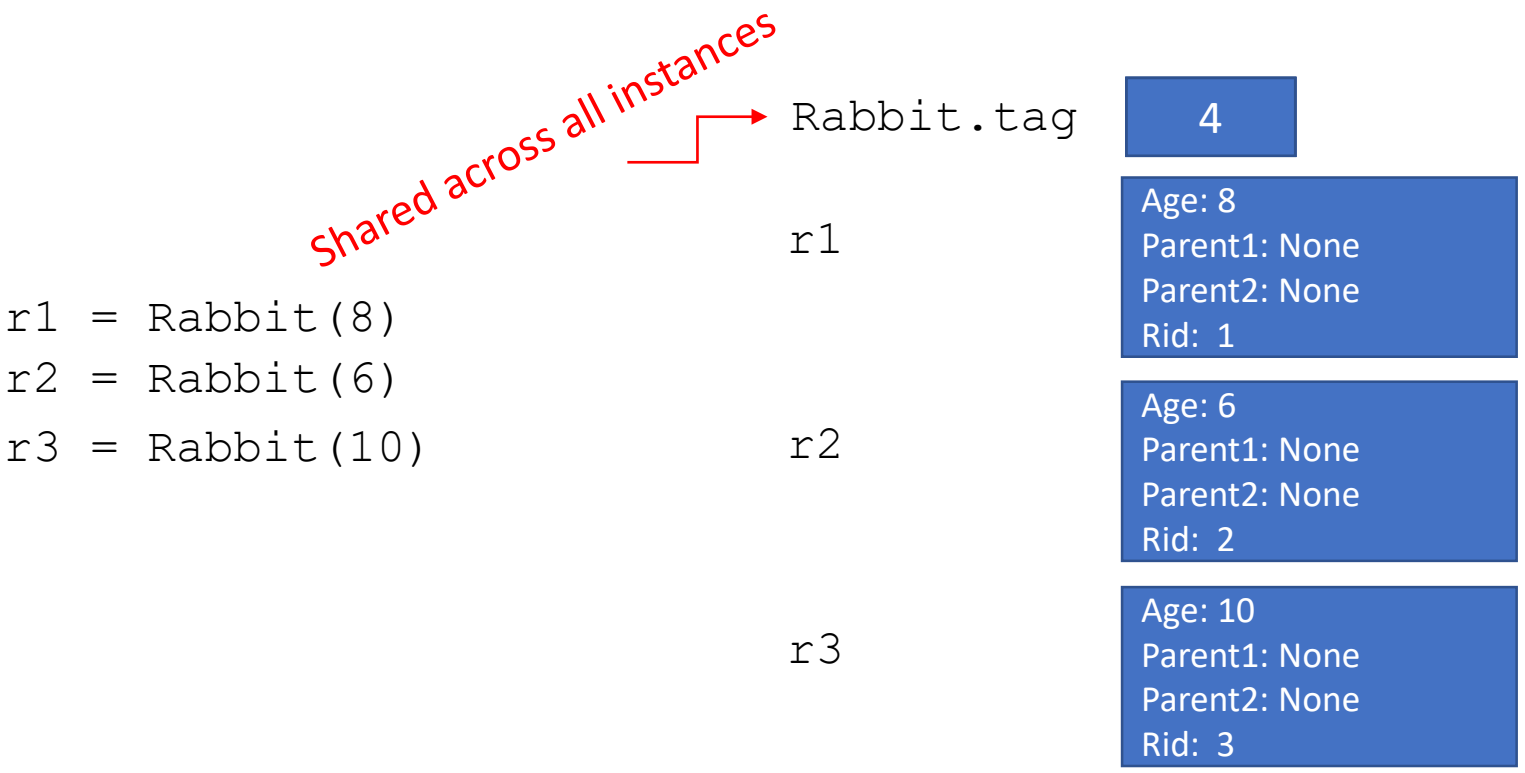
RECALL THE `__init__` OF `Rabbit`

```
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
```



RECALL THE `__init__` OF `Rabbit`

```
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
```



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(5)
    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, 00001 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`**

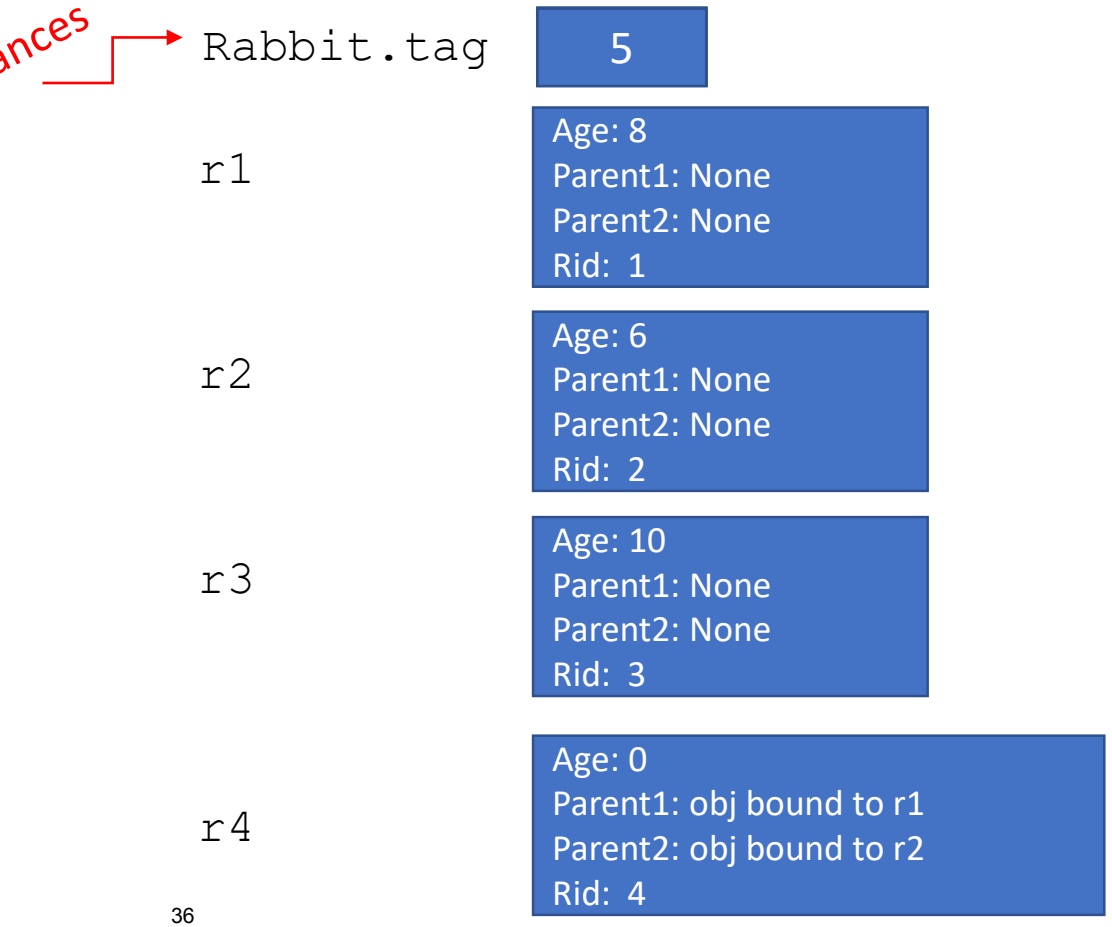
RECALL THE `__init__` OF `Rabbit`

```
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
```

Shared across all instances

```
r1 = Rabbit(8)
r2 = Rabbit(6)
r3 = Rabbit(10)
r4 = r1 + r2
```

Rabbit(0,r1, r2)



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.p1.rid == oth.p1.rid and self.p2.rid == oth.p2.rid)
    parents_opp = (self.p2.rid == oth.p1.rid and self.p1.rid == oth.p2.rid)
    return parents_same or parents_opp
```

Booleans checking r1+r2 or r2+r1

- 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`

BIG IDEA

Class variables are shared between all instances.

If one instance changes it, it's changed for every instance.

OBJECT ORIENTED PROGRAMMING

- Create your own **collections of data**
- **Organize** information
- **Division** of work
- Access information in a **consistent** manner

- Add **layers** of complexity
  - Hierarchies
  - Child classes inherit data and methods from parent classes
- Like functions, classes are a mechanism for **decomposition** and **abstraction** in programming

6.100L Introduction to Computer Science and Programming Using Python
Fall 2022