Lecture 15: Inheritance & String Representation

CS 61A - Summer 2024
Charlotte Le & Cyrus Bugwadia

Announcements

Announcements

- Cats Project
  - Due **tonight** (July 15th)
- HW 04 (Iterators, Generators, Efficiency, Object-Oriented Programming)
  - Due this **Thursday** (July 18th)
- If you need help debugging, remember that you can post private questions on Ed! We respond pretty quickly before assignment deadlines.

About Cyrus (he/him/his)

- Undergrad @ Cal from 2019-2023 (B.A. in Computer Science)
- Taught 61A 3 times as a TA and 2 times as a tutor
- Industry Experience
    - Software engineering internships at Cisco, Intuit, Amazon, Tableau (Salesforce)
    - Currently full-time software engineer at Salesforce
- cyrus.bugwadia@berkeley.edu

Contents

1 Objects Review

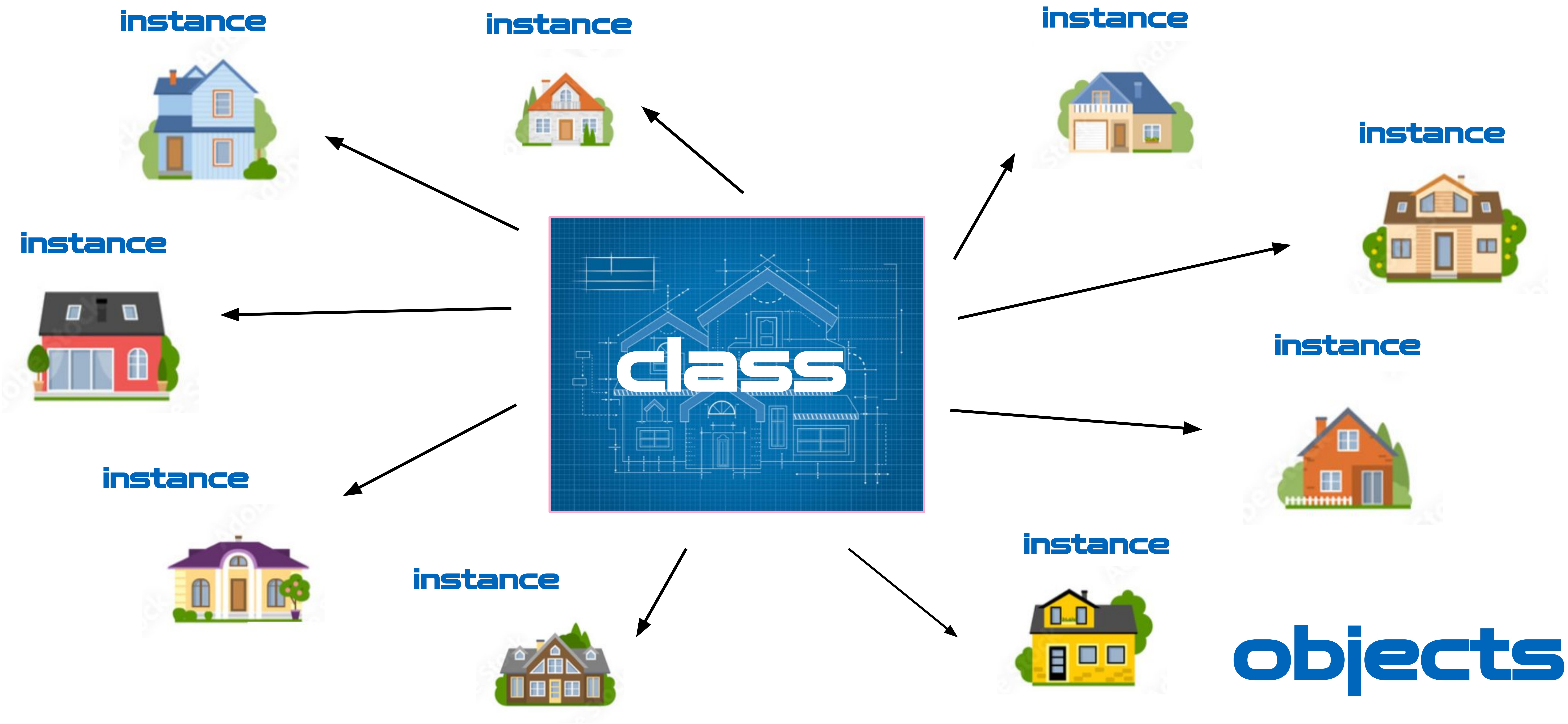
2 Inheritance

3 String Representations

4 F-strings

1) Objects Review

Objects



instance

instance

instance

instance

instance

instance

class

instance

instance

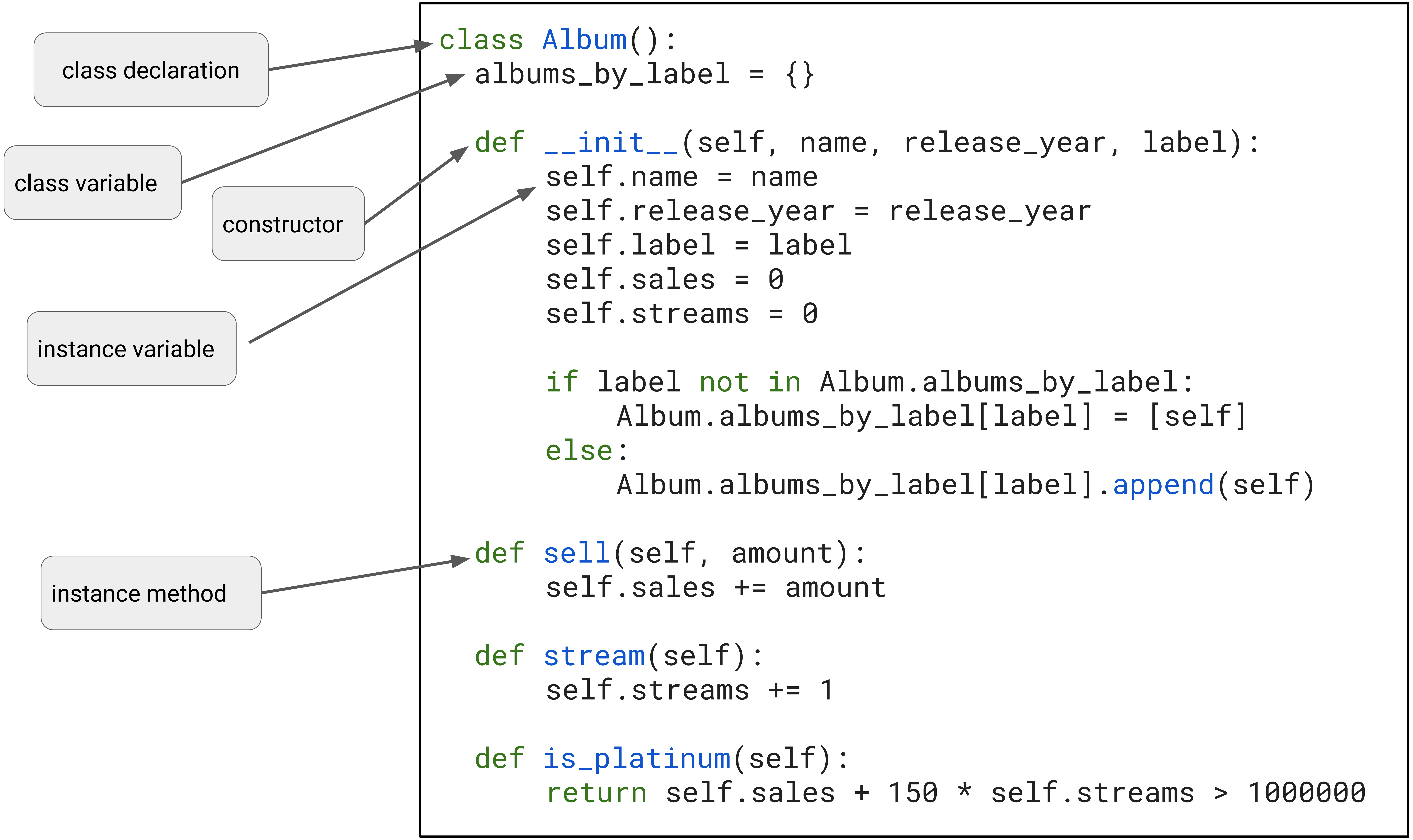
instance

objects

Definitions

- Class
  - A template for creating objects
- Instance
  - A single object created from a class
- Instance Variable
  - A data attribute of an object, specific to an instance
- Class Variable
  - A data attribute of an object, shared by all instances of a class
- Instance method
  - A function that operates on individual instances of a class
- Constructor
  - A method that specifies how to initialize an individual instance

Example of A Complete Class Definition



```
class Album():
  albums_by_label = {}

  def __init__(self, name, release_year, label):
      self.name = name
      self.release_year = release_year
      self.label = label
      self.sales = 0
      self.streams = 0

      if label not in Album.albums_by_label:
          Album.albums_by_label[label] = [self]
      else:
          Album.albums_by_label[label].append(self)

  def sell(self, amount):
      self.sales += amount

  def stream(self):
      self.streams += 1

  def is_platinum(self):
      return self.sales + 150 * self.streams > 1000000
```

class declaration

class variable

constructor

instance variable

instance method

The Album Class + The Listener Class

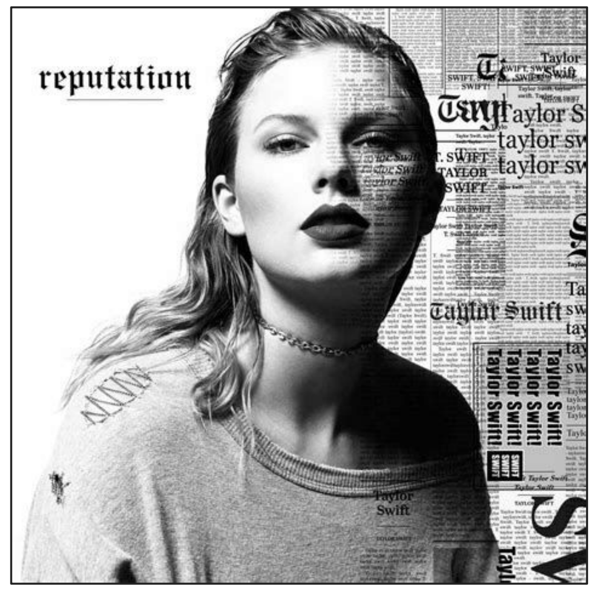
```
class Album():
  albums_by_label = {}

  def __init__(self, name, release_year, label):
      self.name = name
      self.release_year = release_year
      self.label = label
      self.sales = 0
      self.streams = 0

      if label not in Album.albums_by_label:
          Album.albums_by_label[label] = [self]
      else:
          Album.albums_by_label[label].append(self)

  def sell(self, amount):
      self.sales += amount

  def stream(self):
      self.streams += 1

  def is_platinum(self):
      return self.sales + 150 * self.streams > 1000000
```

```
class Listener():
  def __init__(self, name, favorite_album):
    self.name = name
    self.favorite_album = favorite_album
    self.albums = []
    self.buy(favorite_album)

  def buy(self, album):
    album.sell(1)
    self.albums.append(album)
```

In Action

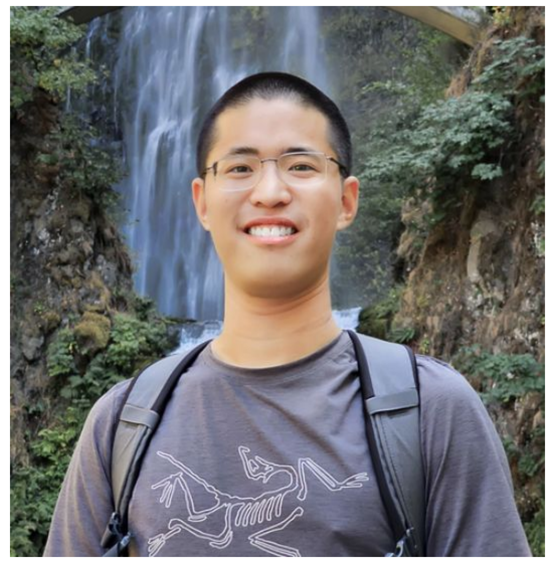
```
# Initialize albums
lover = Album("Lover", 2019, "Republic Records")
reputation = Album("Reputation", 2017, "Big Machine")
nineteen_eighty_nine = Album("1989", 2014, "Big Machine")
```

name	"Lover"
release_year	2019
label	"Republic Records"
sales	0



name	"Reputation"
release_year	2017
label	"Big Machine"
sales	0

name	"1989"
release_year	2014
label	"Big Machine"
sales	0

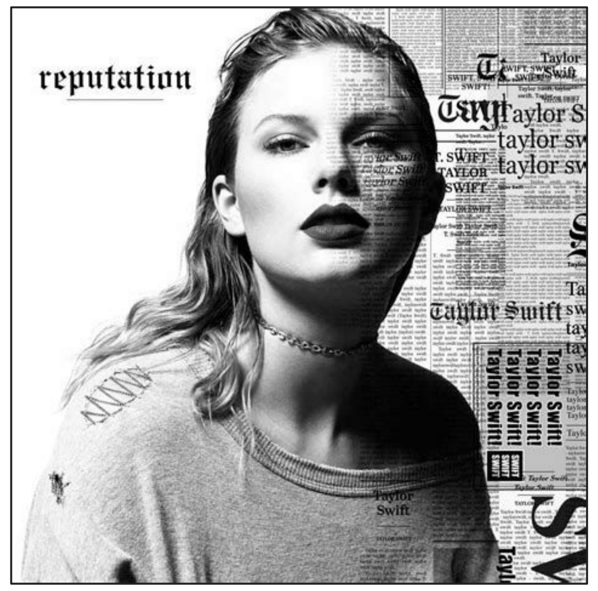


name	"Hans"
favorite_album	
albums	

```
class Listener():
 def __init__(self, name, favorite_album):
    self.name = name
    self.favorite_album = favorite_album
    self.albums = []
    self.buy(favorite_album)

 def buy(self, album):
    album.sell(1)
    self.albums.append(album)
```

```
hans = Listener("Hans", reputation)
```

name	"Lover"
release_year	2019
label	"Republic Records"
sales	0

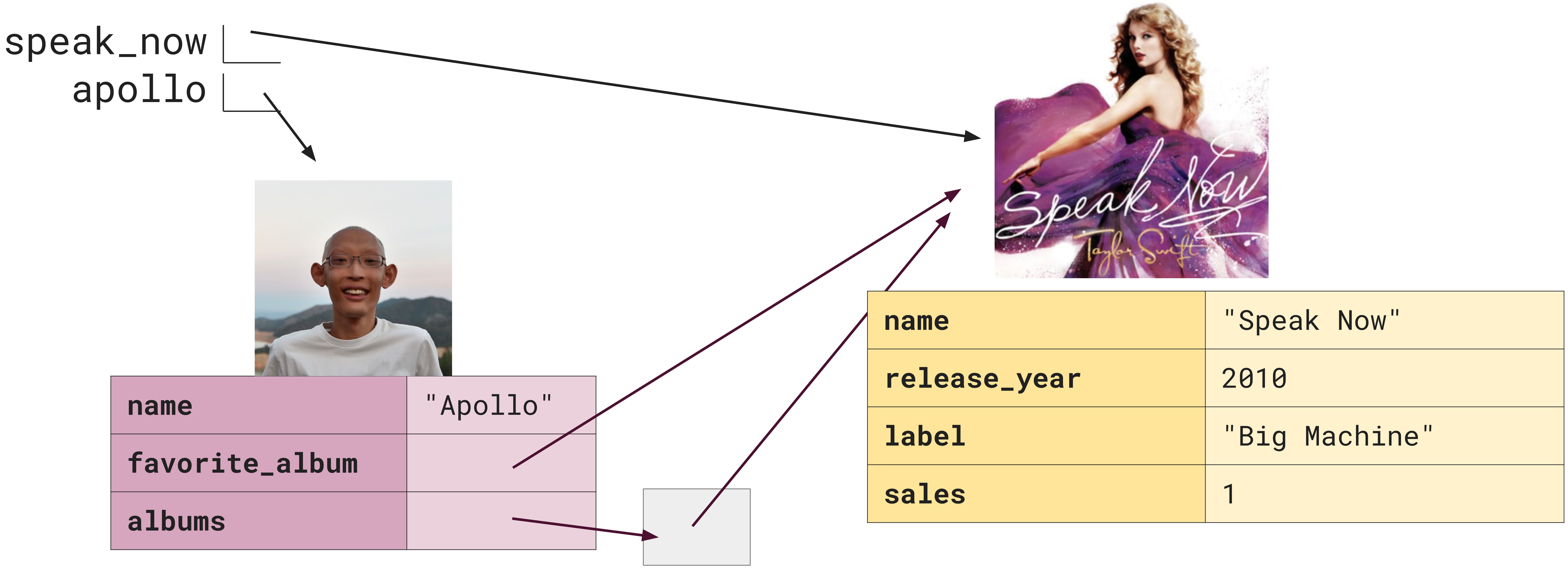


name	"Reputation"
release_year	2017
label	"Big Machine"
sales	1

name	"1989"
release_year	2014
label	"Big Machine"
sales	0

```
speak_now = Album("Speak Now", 2010, "Big Machine")

apollo = Listener("Apollo", speak_now)
speak_now.name = "Speak Now (Taylor's Version)"
speak_now.release_year = 2023
speak_now.label = "Republic Records"
```

Before

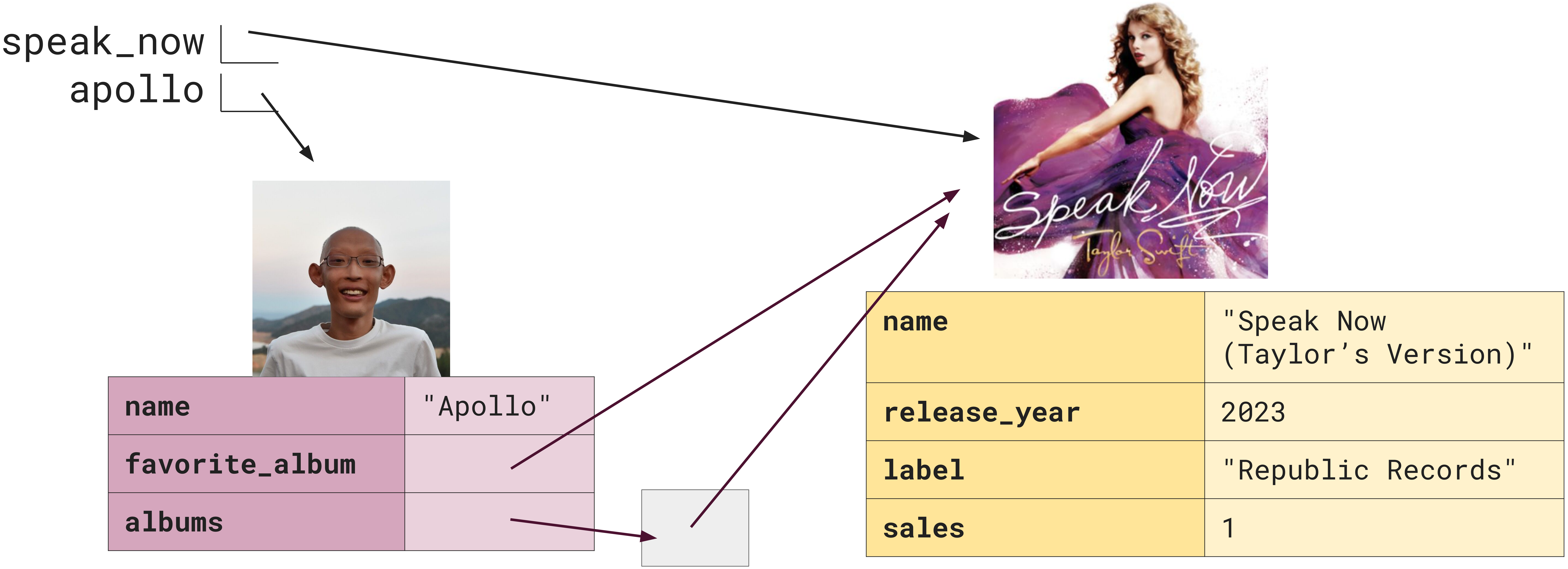


name	"Apollo"
favorite_album	
albums	

name	"Speak Now"
release_year	2010
label	"Big Machine"
sales	1

```
speak_now = Album("Speak Now", 2010, "Big Machine")

apollo = Listener("Apollo", speak_now)
speak_now.name = "Speak Now (Taylor's Version)"
speak_now.release_year = 2023
speak_now.label = "Republic Records"
```

After



name	"Apollo"
favorite_album	
albums	

name	"Speak Now (Taylor’s Version)"
release_year	2023
label	"Republic Records"
sales	1

2) Inheritance

Elephant Class

```
class Elephant:
  scientific_name = “Loxodonta africana”
  calories_needed = 16000

  def __init__(self, name, age=0):
    self.name = name
    self.age = age
    self.calories_eaten = 0
    self.happiness = 0

  def play(self, num_hours):
    self.happiness += num_hours * 4
    print(“pawoo”)

  def eat(self, food):
    self.calories_eaten += food.calories
    print(“nom nom nom yummy “ + food.name)
    if self.calories_eaten > self.calories_needed:
      self.happiness -= 1
      print(“too full, need nap”)
```

Rabbit Class

```
class Rabbit:
  scientific_name = “Oryctolagus cuniculus”
  calories_needed = 200

  def __init__(self, name, age=0):
    self.name = name
    self.age = age
    self.calories_eaten = 0
    self.happiness = 0

  def play(self, num_hours):
    self.happiness += num_hours * 10
    print(“kip kip”)

  def eat(self, food):
    self.calories_eaten += food.calories
    print(“nom nom nom yummy “ + food.name)
    if self.calories_eaten > self.calories_needed:
      self.happiness -= 1
      print(“too full, need nap”)
```

Elephant Class

```
class Elephant:
  scientific_name = “Loxodonta africana”
  calories_needed = 16000

  def __init__(self, name, age=0):
    self.name = name
    self.age = age
    self.calories_eaten = 0
    self.happiness = 0

  def play(self, num_hours):
    self.happiness += num_hours * 4
    print(“pawoo”)

  def eat(self, food):
    self.calories_eaten += food.calories
    print(“nom nom nom yummy “ + food.name)
    if self.calories_eaten > self.calories_needed:
      self.happiness -= 1
      print(“too full, need nap”)
```

Rabbit Class

```
class Rabbit:
  scientific_name = “Oryctolagus cuniculus”
  calories_needed = 200

  def __init__(self, name, age=0):
    self.name = name
    self.age = age
    self.calories_eaten = 0
    self.happiness = 0

  def play(self, num_hours):
    self.happiness += num_hours * 10
    print(“kip kip”)

  def eat(self, food):
    self.calories_eaten += food.calories
    print(“nom nom nom yummy “ + food.name)
    if self.calories_eaten > self.calories_needed:
      self.happiness -= 1
      print(“too full, need nap”)
```

Similarities

Elephant

- Class Attributes
  - scientific_name, calories_needed
- Instance Attributes
  - name, age, happiness, calories_needed
- Methods
  - eat(food), play

Rabbit

- Class Attributes
  - scientific_name, calories_needed
- Instance Attributes
  - name, age, happiness, calories_needed
- Methods
  - eat(food), play

Differences

- Methods
  - The happiness multiplier is different
  - They make different noises

- Attributes
  - The scientific name and calories needed are different and dependent on the animal

How Do We Make This Better?

- We can use **inheritance** to make a general Animal class and then have the specific animals inherit from that class!
- The Animal class can have a generic animal definition
- The specific classes will add the specific elements to it

- In this example, Animal will be the **superclass** (parent class) and Elephant, Rabbit, Panda, Dog, etc. will be the **subclasses** (child classes)

Elephant Class

```
class Elephant:
  scientific_name = “Loxodonta africana”
  calories_needed = 16000

  def __init__(self, name, age=0):
    self.name = name
    self.age = age
    self.calories_eaten = 0
    self.happiness = 0

  def play(self, num_hours):
    self.happiness += num_hours * 4
    print(“pawoo”)

  def eat(self, food):
    self.calories_eaten += food.calories
    print(“nom nom nom yummy “ + food.name)
    if self.calories_eaten > self.calories_needed:
      self.happiness -= 1
      print(“too full, need nap”)
```

Rabbit Class

```
class Rabbit:
  scientific_name = “Oryctolagus cuniculus”
  calories_needed = 200

  def __init__(self, name, age=0):
    self.name = name
    self.age = age
    self.calories_eaten = 0
    self.happiness = 0

  def play(self, num_hours):
    self.happiness += num_hours * 10
    print(“kip kip”)

  def eat(self, food):
    self.calories_eaten += food.calories
    print(“nom nom nom yummy “ + food.name)
    if self.calories_eaten > self.calories_needed:
      self.happiness -= 1
      print(“too full, need nap”)
```

Animal Class

```
class Animal:
  scientific_name = “Animalia”
  calories_needed = 100
  play_multiplier = 1
  noise = “woo”

  def __init__(self, name, age=0):
    self.name = name
    self.age = age
    self.calories_eaten = 0
    self.happiness = 0

  def play(self, num_hours):
    self.happiness += num_hours * play_multiplier
    print(noise)

  def eat(self, food):
    self.calories_eaten += food.calories
    print(“nom nom nom yummy “ + food.name)
    if self.calories_eaten > self.calories_needed:
      self.happiness -= 1
      print(“too full, need nap”)
```

Creating a Subclass

- You can create the subclass by adding the parent class into the class definition

```
class Panda(Animal):
```

- You should only write code in these subclasses that are unique to them
- Overrriding: you can redefine class variables, methods, or the constructor

Overriding Attributes (Elephant and Rabbit)

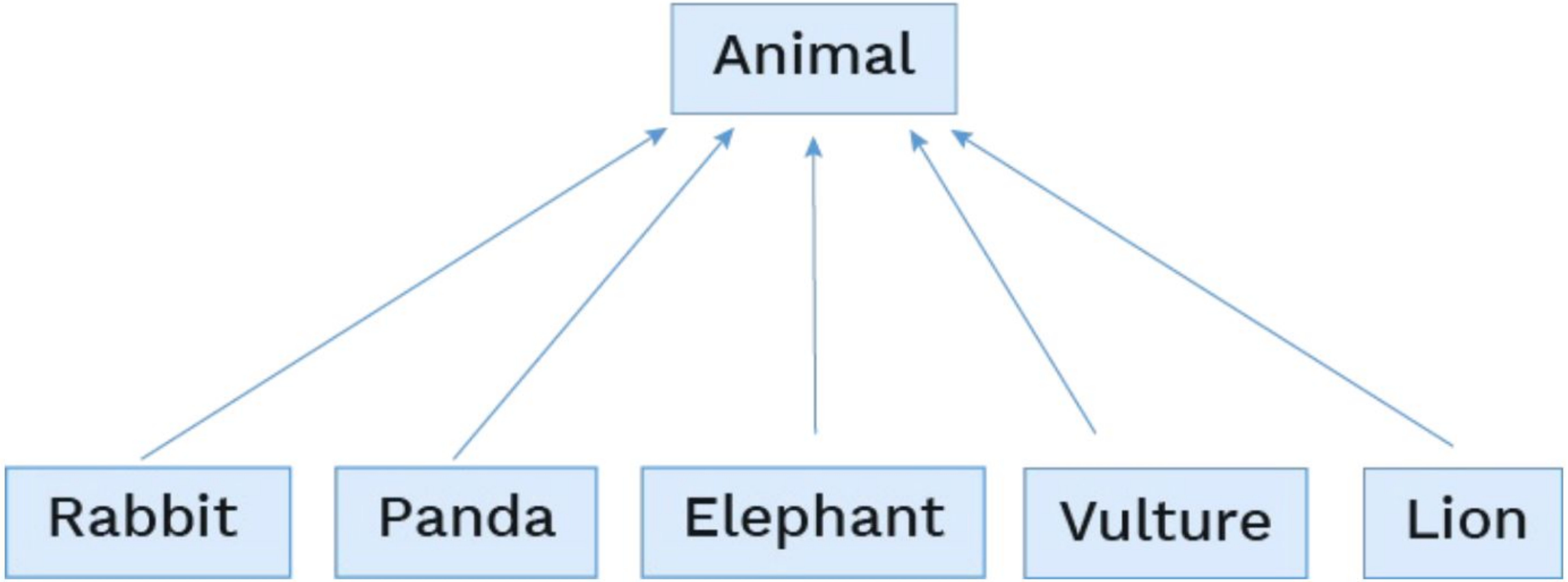
```
class Elephant(Animal):
  scientific_name = “Loxodonta africana”
  calories_needed = 16000
  play_multiplier = 4
  noise = “pawoo”

class Rabbit(Animal):
  scientific_name = “Oryctolagus cuniculus”
  calories_needed = 200
  play_multiplier = 10
  noise = “kip kip”
  hops = True
```

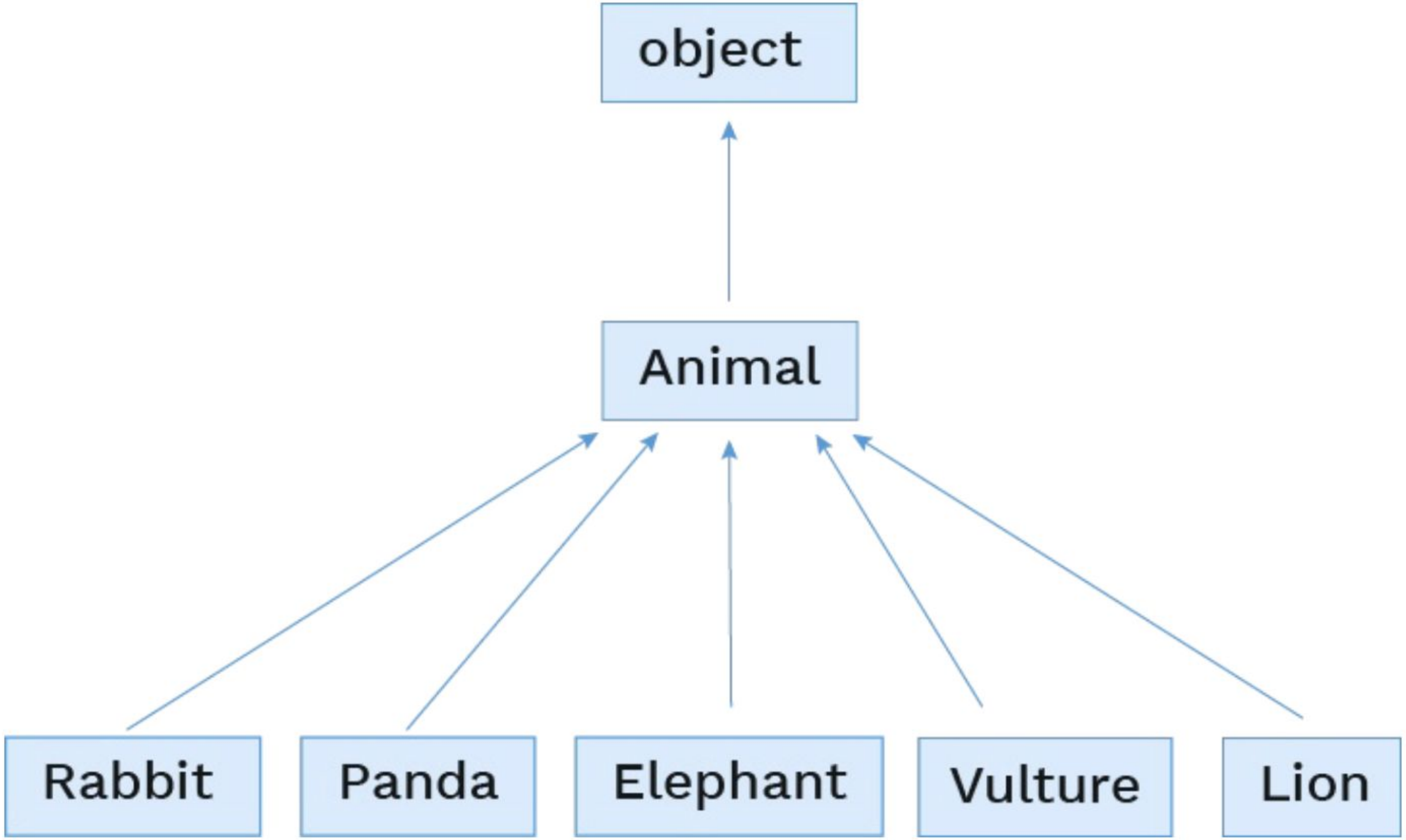
Overriding Methods

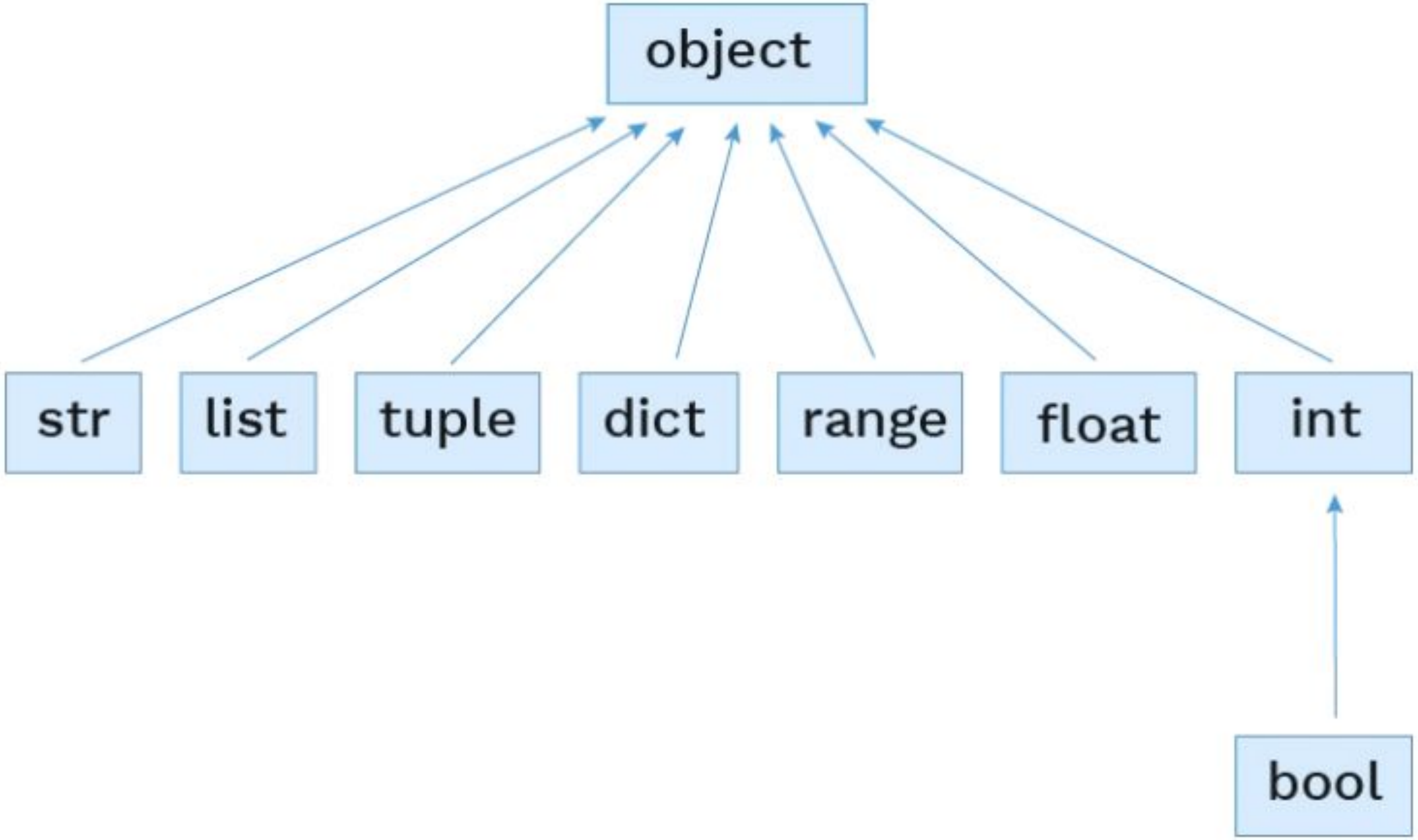
```
class Panda(Animal):
  scientific_name = “Ailuropoda melanoleuca”
  calories_needed = 6000
  play_multiplier = 5
  noise = “neeeeeh”

  def eat(self):
    self.calories_eaten += 150
    print(“nom nom bamboo is the best nom nom”)
    if self.calories_eaten > self.calories_needed:
      self.calories_eaten += 150
      self.happiness += 1
      print(“more bamboo, i love bamboo”)
```

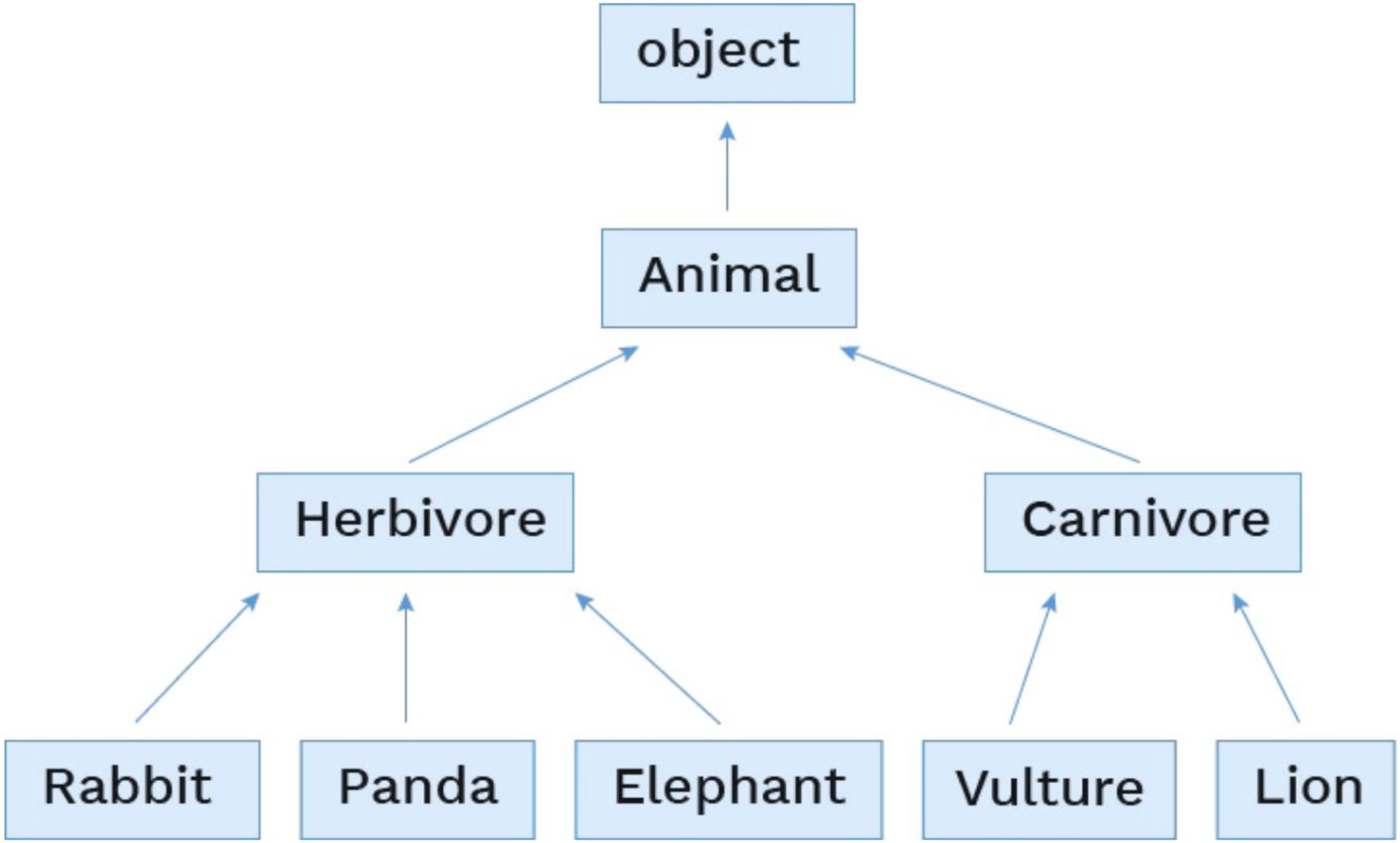
Using Methods from the Superclass

```
class Lion(Animal):

  scientific_name = “Panthera”
  calories_needed = 3000
  play_multiplier = 12
  noise = “rooooaaar”

  def eat(self, food):
    if food.type == “meat” :
      super().eat(food)        # Animal.eat(self, food)
    else:
      print(“no thank you”)
```

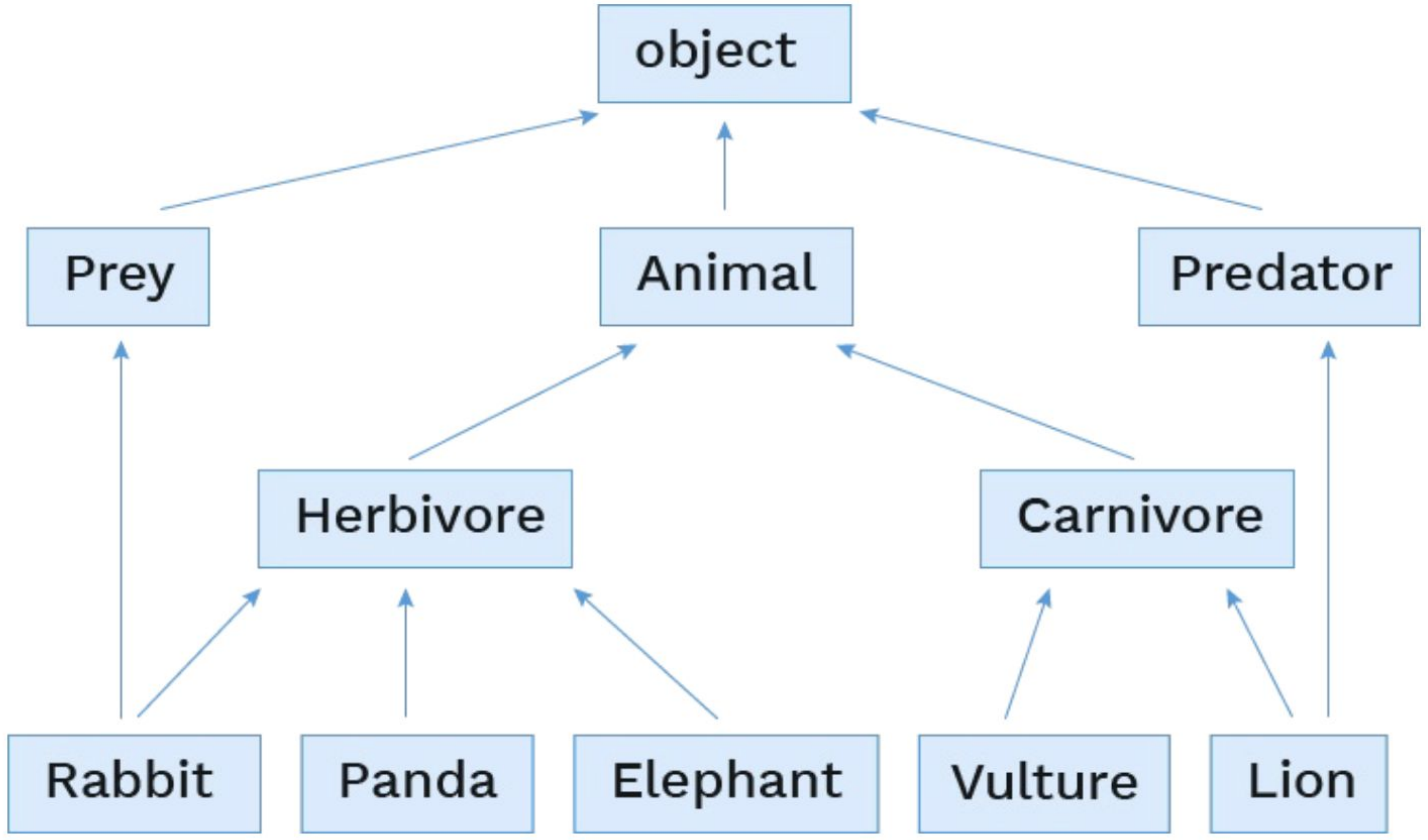

Animal
Rabbit
Panda
Elephant
Vulture
Lion


object
Animal
Rabbit
Panda
Elephant
Vulture
Lion


object
str
list
tuple
dict
range
float
int
bool



object
Animal
Herbivore
Carnivore
Rabbit
Panda
Elephant
Vulture
Lion


object
Prey
Animal
Predator
Herbivore
Carnivore
Rabbit
Panda
Elephant
Vulture
Lion

Looking Up Attribute Names with Inheritance

The base class attributes are **not** copied into the subclasses

To look up a name with inheritance:

1. If it names an attribute in the class, return the attribute value
2. Otherwise, look up the name in the super class, if it exists

Using Inheritance

- With inheritance, we want to avoid repeated ourselves if possible, we can just use the existing implementation instead
- Attributes that have been overridden are still accessible via the subclass
- Look up attributes on instances when possible

Inheritance vs Composition

OOP is best when we think about the following metaphor:

Inheritance is best for representing **is-a** relationships

- A panda *is* an animal so Panda inherits from Animal

Composition is best for representing **has-a** relationships

- An album *has* listeners
- A zoo *has* animals

Break

String Representations

String Representations

- Strings are important - they represent language and programs
- Printing strings is exactly how the interpreter communicates back to us
    - We should be able to use strings to represent objects!
- Python gives us two ways/functions to represent objects
    - **str**: returns a representation that a **human** should be able to read
        - used by the print function
    - **repr**: returns a representation that the **interpreter** should be able to read
        - used by the interpreter for printing results
    - these are usually the same, but not always!

str

- **str**: returns a representation that a **human** should be able to read
- **`print(obj)`** is actually the same thing as **`print(str(obj))`**
- What if we call str on a string?
    - the string itself is returned directly, without adding quotes
- Demo

repr

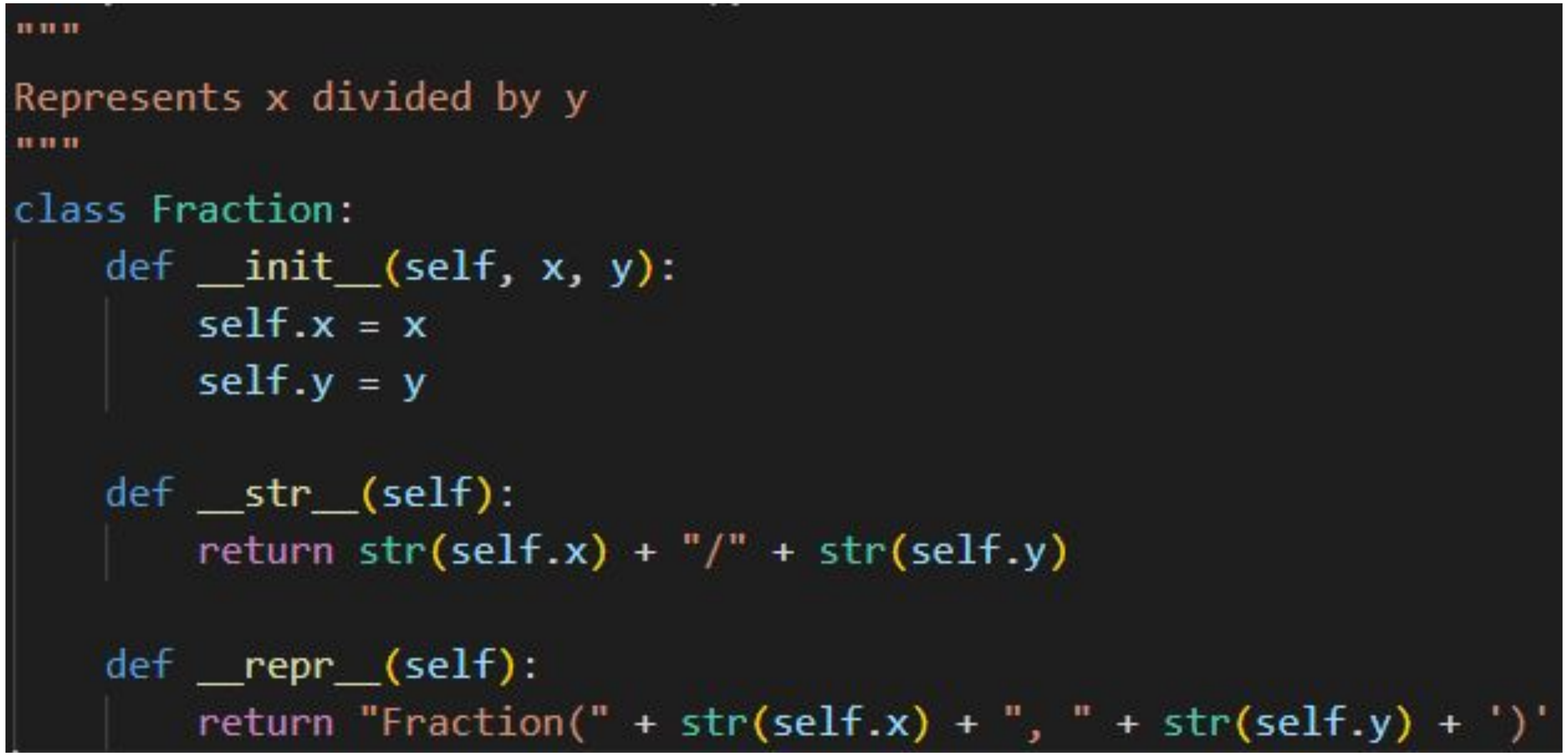
- **repr**: returns a representation that the **interpreter** should be able to read
- When typing an expression into the interpreter, Python calls repr on the result of evaluation, and prints that
    - this means copy-pasting the interpreter's output back into the interpreter should ideally give back an equivalent result
- What if we call repr on a string?
    - string is returned, but with quotes added (since this is how the interpreter identifies strings)
- Demo

__str__ and __repr__

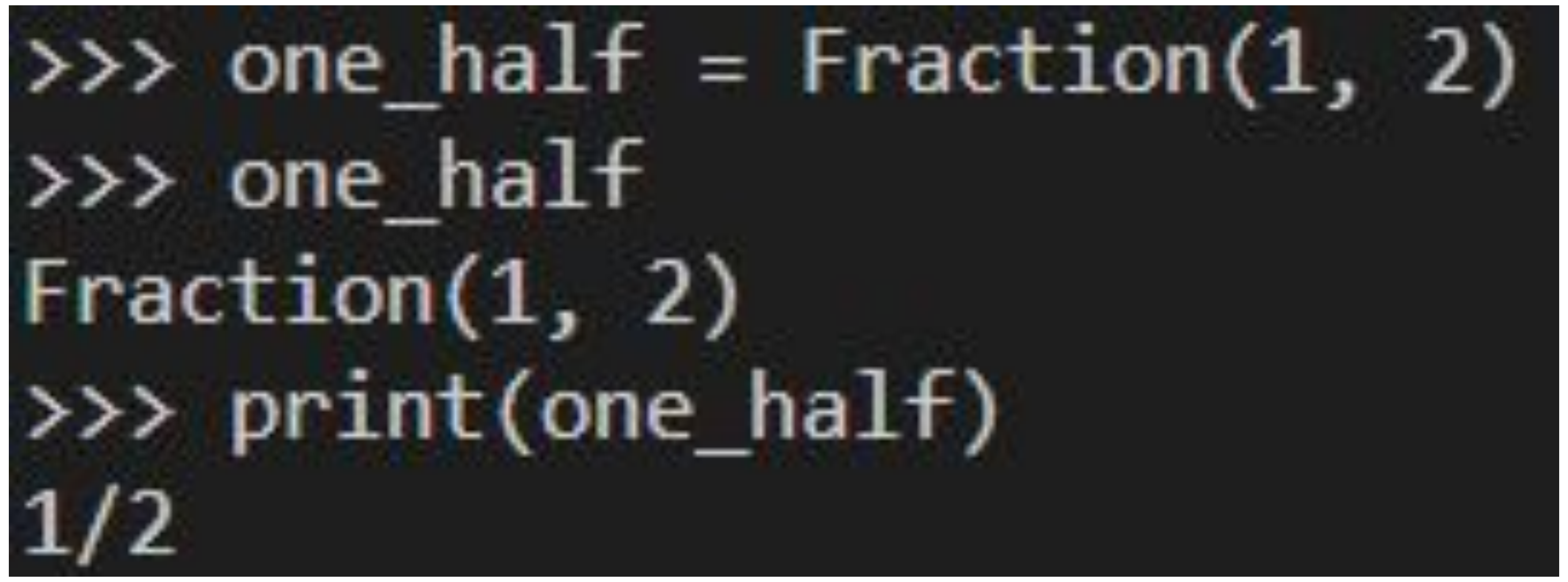
- How will Python know how to represent our own objects as strings?
    - default representation is almost useless
- Defining the methods **__str__** and **__repr__** in a class tells Python what to return when calling str and repr on objects of that class
    - two underscores before, two underscores after, just like __init__
- If __repr__ is defined but __str__ is not, then str will use __repr__
    - this is not true the other way around
- Demo

__str__ and __repr__

By defining __str__ and __repr__, we can control how Fraction objects are represented as strings.



```
"""
Represents x divided by y
"""
class Fraction:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return str(self.x) + "/" + str(self.y)

    def __repr__(self):
        return "Fraction(" + str(self.x) + ", " + str(self.y) + ')'
```



```
>>> one_half = Fraction(1, 2)
>>> one_half
Fraction(1, 2)
>>> print(one_half)
1/2
```

F-strings

F-strings

- In the Fraction example, we had to manually call str and use string addition to form the strings we wanted to return
- F-strings give us a way to neatly include expressions in strings
    - e.g. `f"2 + 2 is {2 + 2}"` is equivalent to `"2 + 2 is 4"`
    - anything in curly braces gets evaluated in the current environment
    - the results get str called on them and that gets substituted in
- Demo