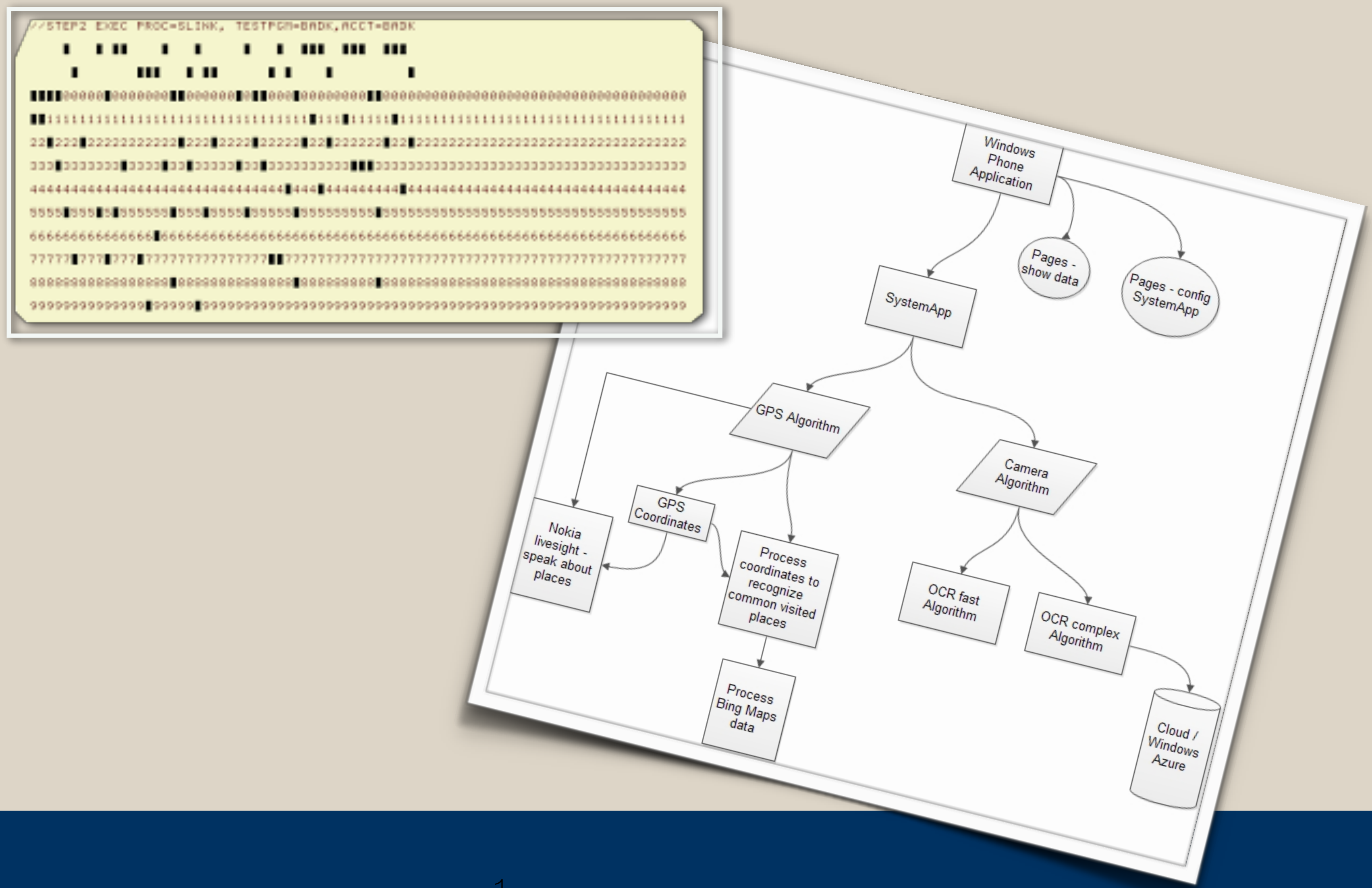
Week 8: Object-Oriented Design



Tonight’s agenda

Points from your homeworks/scrabble
Docstrings
PEP8, Dunder, Private vs. Public
Project 1 & Schedule

- Why classes?
  - Inheritance
  - Polymorphism
  - super() and pass()

Quack Typing

Magic Methods

Project 1 and rest of course

```
x = 4
repr(x) outputs ‘4’

str(x) outputs ‘4’

y = ‘hello’
repr(y) outputs “‘hello’”

str(y) outputs ‘hello’
```

Homeworks & Scrabble Implementations

- *Good Algorithm Design:*
  - Iterated through scrabble words list only once
  - For wildcards: did *not* make a list of all possible rack letters
- *Compared words to rack while keeping track of wildcards:*
  - If word length > rack length - immediately discard (& move to next word)
  - If letter is not in the rack & no wildcards - discard
    - if wildcards, word could be 'off" by that many letters (1 or 2)
    - if word was off by more than # of wildcards - discard
  - If word passes the above tests - add to valid word list & go to next word.

Homeworks

- *Tabs* v *Spaces - An eternal debate*
  - One reason 4 spaces are better: 4 spaces = 4 spaces whenever copying & pasting the code. 1 tab isn't necessarily 4 spaces [tab sizes are different; space size isn't]
  - *79 character limit:*
    - a little arbitrary today (with wide-screen monitors, etc)
    - Based in the old unix default window of 80 chars [itself based on the old keypunch cards]
    - Designed to increase the readability of code
    - Jupiter notebook doesn't wrap code so the code may continue off the right side of the screen

Docstrings:

- *Usually do this after defining a function to indicate the input arguments, type, and what the function does and returns. Example:*

```
def function_with_docstring(param1, param2):
    """This function takes a string, param2, and checks if a number, param1,
     is in that string.

      Args:
         param1 (int): The number to check for
            param2 (str): The string to check

         Returns:
            bool: True for success, False otherwise.
     """
```

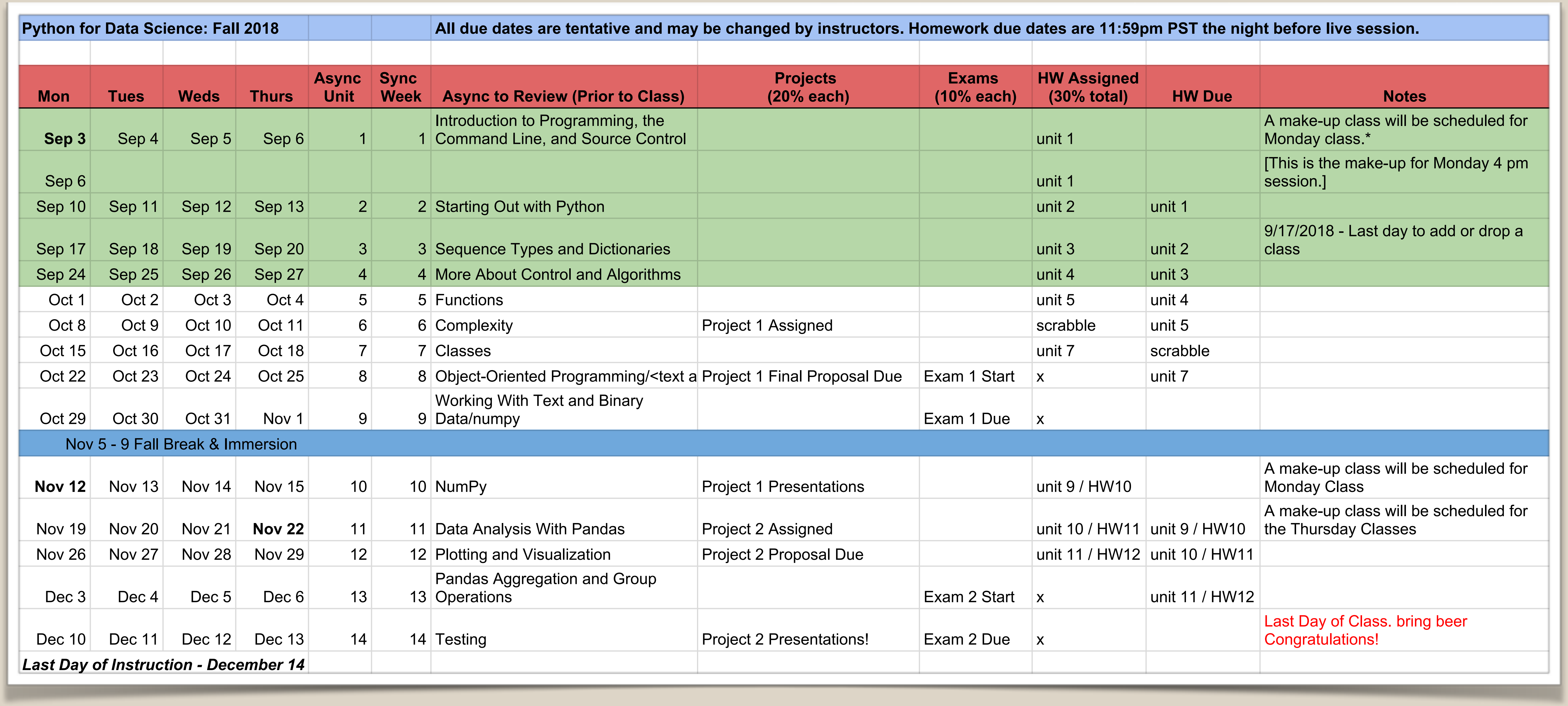
Other points …

- *Autoformat PEP8?:*
  - Yes, in PyCharm (and maybe in others)
- *"Dunder" is the __ (as in __init__):*
  - Invoked behind the scenes – that is, you don't specifically call that method.
  - Using __init__ as an example - it's a method that is called automatically when the object is substantiated
- *Public* vs. *private variables:*
  - *Private variables* are annotated by a dander __ or one __ (like __count). The means that variable is accessed only by that class, not from outside the class. Python doesn't truly have private vars - because they can still be modified. [Not the case in other OOP languages; private vars cannot be modified outside class.]

Project 1 reminder

- *The proposal was due, but treat is as* a hypothesis *as how you'll solve the problem. In the two weeks left, implement, refine and adjust.*
- *An iterative process while you learn.*
- *Next week there will be more time to discuss your progress and discuss outstanding issues in breakout groups.*

Schedule



Python for Data Science: Fall 2018

All due dates are tentative and may be changed by instructors. Homework due dates are 11:59pm PST the night before live session.

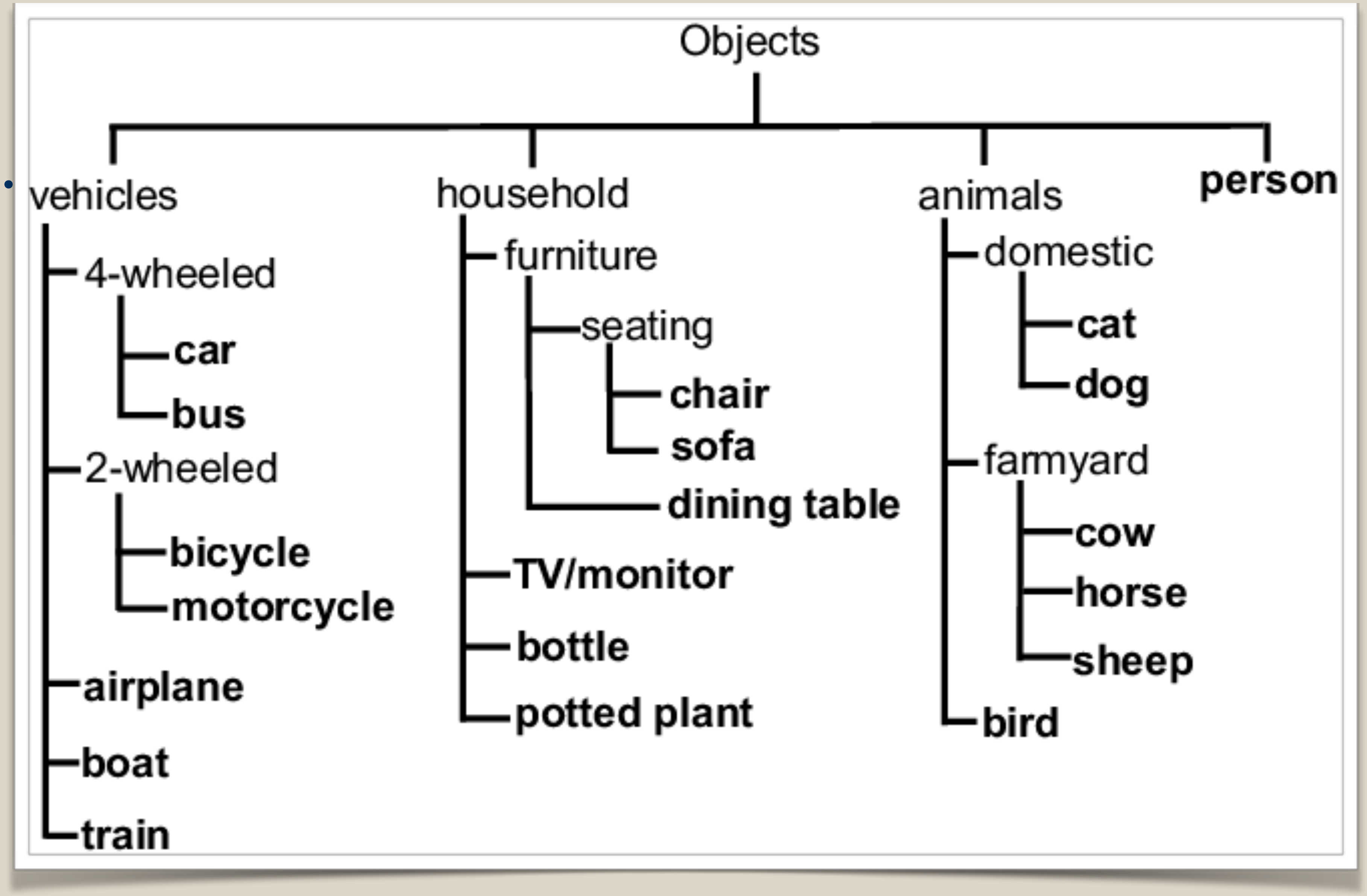
Mon	Tues	Weds	Thurs	Async Unit	Sync Week	Async to Review (Prior to Class)	Projects (20% each)	Exams (10% each)	HW Assigned (30% total)	HW Due	Notes
Sep 3	Sep 4	Sep 5	Sep 6	1	1	Introduction to Programming, the Command Line, and Source Control			unit 1		A make-up class will be scheduled for Monday class.*
Sep 6									unit 1		[This is the make-up for Monday 4 pm session.]
Sep 10	Sep 11	Sep 12	Sep 13	2	2	Starting Out with Python			unit 2	unit 1	
Sep 17	Sep 18	Sep 19	Sep 20	3	3	Sequence Types and Dictionaries			unit 3	unit 2	9/17/2018 - Last day to add or drop a class
Sep 24	Sep 25	Sep 26	Sep 27	4	4	More About Control and Algorithms			unit 4	unit 3	
Oct 1	Oct 2	Oct 3	Oct 4	5	5	Functions			unit 5	unit 4	
Oct 8	Oct 9	Oct 10	Oct 11	6	6	Complexity	Project 1 Assigned		scrabble	unit 5	
Oct 15	Oct 16	Oct 17	Oct 18	7	7	Classes			unit 7	scrabble	
Oct 22	Oct 23	Oct 24	Oct 25	8	8	Object-Oriented Programming/<text a	Project 1 Final Proposal Due	Exam 1 Start	x	unit 7	
Oct 29	Oct 30	Oct 31	Nov 1	9	9	Working With Text and Binary Data/numpy		Exam 1 Due	x		
Nov 5 - 9 Fall Break & Immersion											
Nov 12	Nov 13	Nov 14	Nov 15	10	10	NumPy	Project 1 Presentations		unit 9 / HW10		A make-up class will be scheduled for Monday Class
Nov 19	Nov 20	Nov 21	**Nov 22**	11	11	Data Analysis With Pandas	Project 2 Assigned		unit 10 / HW11	unit 9 / HW10	A make-up class will be scheduled for the Thursday Classes
Nov 26	Nov 27	Nov 28	Nov 29	12	12	Plotting and Visualization	Project 2 Proposal Due		unit 11 / HW12	unit 10 / HW11	
Dec 3	Dec 4	Dec 5	Dec 6	13	13	Pandas Aggregation and Group Operations		Exam 2 Start	x	unit 11 / HW12	
Dec 10	Dec 11	Dec 12	Dec 13	14	14	Testing	Project 2 Presentations!	Exam 2 Due	x		Last Day of Class. bring beer Congratulations!
Last Day of Instruction - December 14											

https://docs.google.com/spreadsheets/d/1sVV7-4OHZ-EDNqkMJ55OPUfz_QLJ4LZNuZl-cRaxgVo/edit#gid=0

Why classes?

- *Can be challenging…*

Encapsulation
Modularity
Inheritance
Polymorphism



Objects

vehicles
- 4-wheeled
  - car
  - bus
- 2-wheeled
  - bicycle
  - motorcycle
- airplane
- boat
- train

household
- furniture
  - seating
    - chair
    - sofa
  - dining table
- TV/monitor
- bottle
- potted plant

animals
- domestic
  - cat
  - dog
- farmyard
  - cow
  - horse
  - sheep
- bird

person

Why did we create a "card" class, even tho we didn't need its functions?

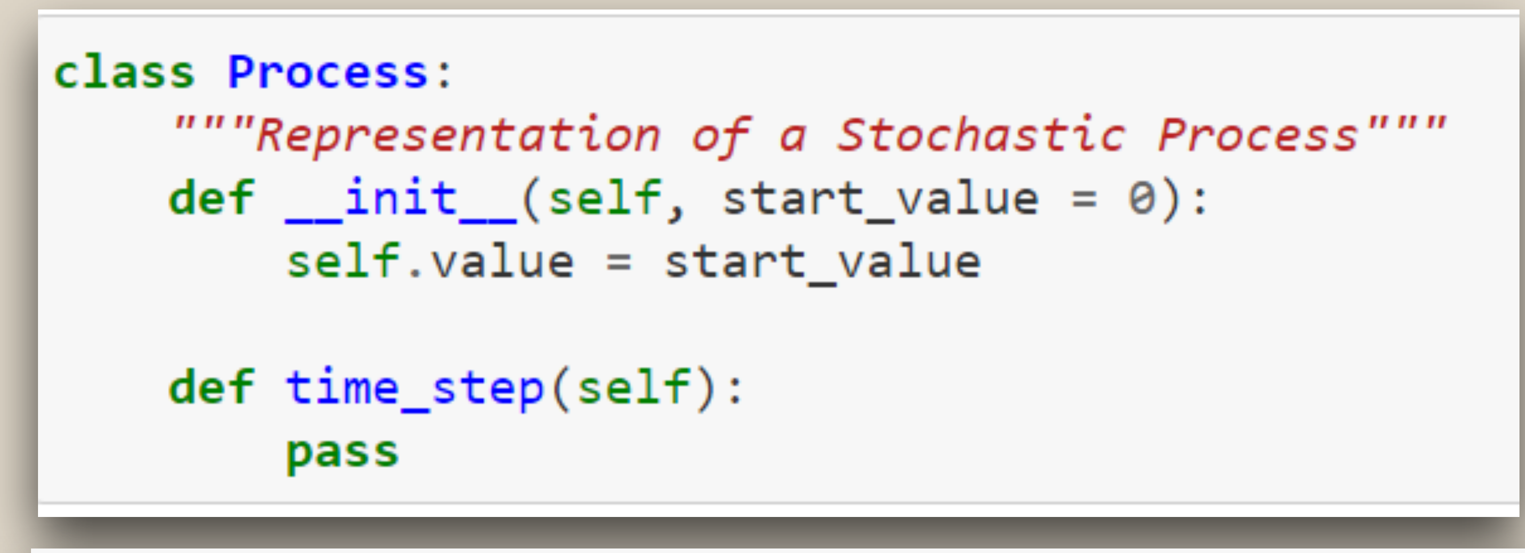
How can you decide when to use an "object" versus an "attribute" for a given part of your code?

Discussion points (from *asynch notebooks*)

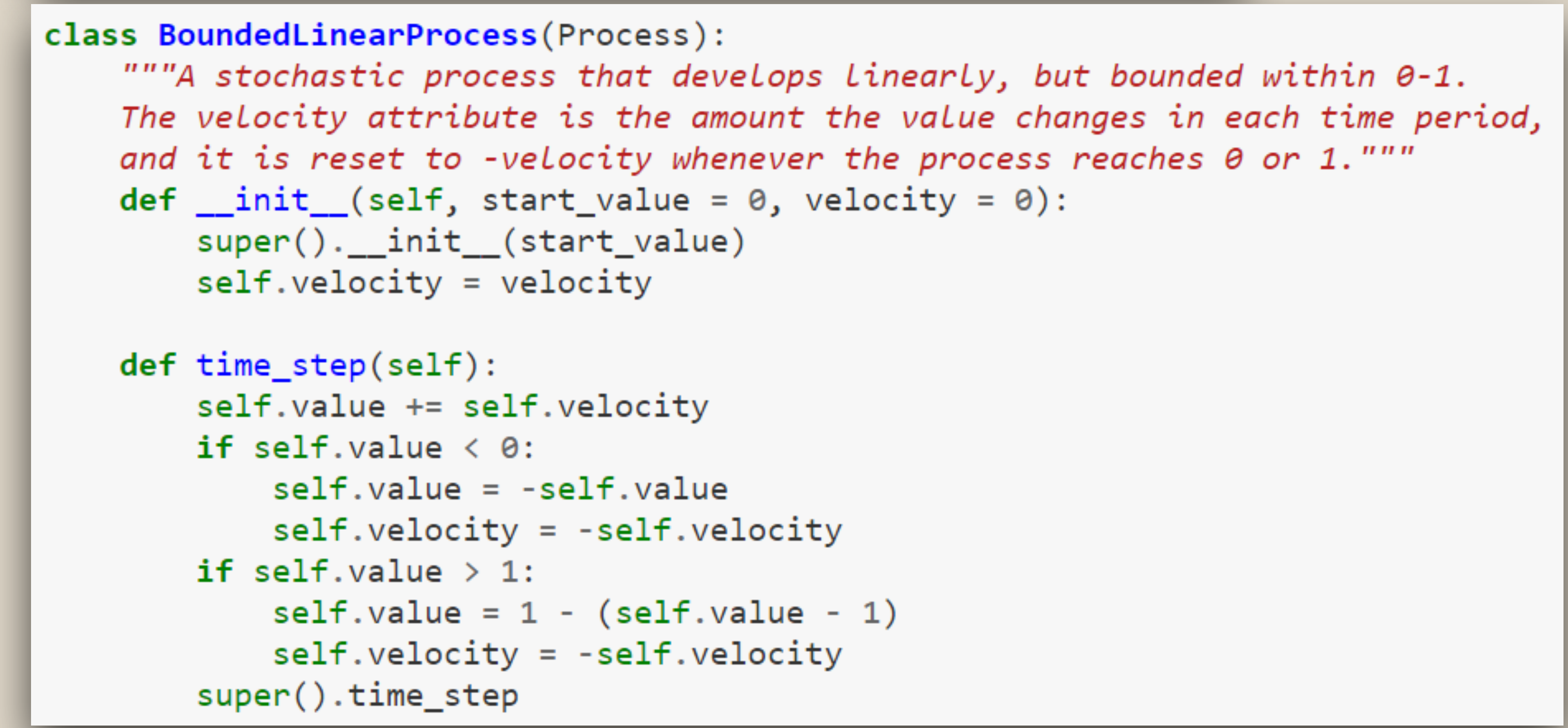
- *Inheritance (§8.4)*
- *Polymorphism (§8.7)*
- *Magic Methods (§8.9)*

Why inheritance?

- *Why would we want to use inheritance?*
- *What's super?*
- *What's pass?*

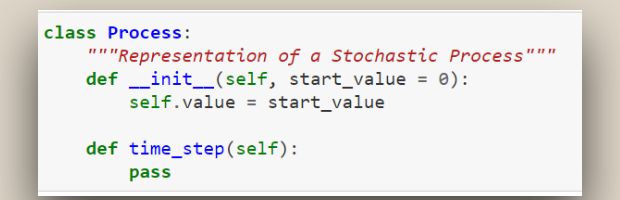


```
class Process:
    """Representation of a Stochastic Process"""
    def __init__(self, start_value = 0):
        self.value = start_value

    def time_step(self):
        pass
```

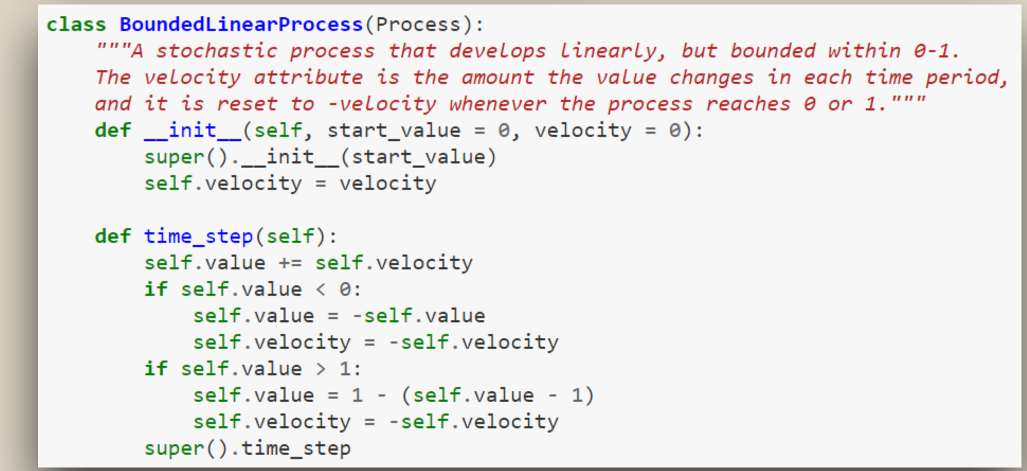


```
class BoundedLinearProcess(Process):
    """A stochastic process that develops linearly, but bounded within 0-1.
    The velocity attribute is the amount the value changes in each time period,
    and it is reset to -velocity whenever the process reaches 0 or 1."""
    def __init__(self, start_value = 0, velocity = 0):
        super().__init__(start_value)
        self.velocity = velocity

    def time_step(self):
        self.value += self.velocity
        if self.value < 0:
            self.value = -self.value
            self.velocity = -self.velocity
        if self.value > 1:
            self.value = 1 - (self.value - 1)
            self.velocity = -self.velocity
        super().time_step
```

Why inheritance?



```python
class Process:
    """Representation of a Stochastic Process"""
    def __init__(self, start_value = 0):
        self.value = start_value

    def time_step(self):
        pass
```



```python
class BoundedLinearProcess(Process):
    """A stochastic process that develops linearly, but bounded within 0-1.
    The velocity attribute is the amount the value changes in each time period,
    and it is reset to -velocity whenever the process reaches 0 or 1."""
    def __init__(self, start_value = 0, velocity = 0):
        super().__init__(start_value)
        self.velocity = velocity

    def time_step(self):
        self.value += self.velocity
        if self.value < 0:
            self.value = -self.value
            self.velocity = -self.velocity
        if self.value > 1:
            self.value = 1 - (self.value - 1)
            self.velocity = -self.velocity
        super().time_step
```

- *Why would we want to use inheritance?*

  A template for other classes

- *What's* super?

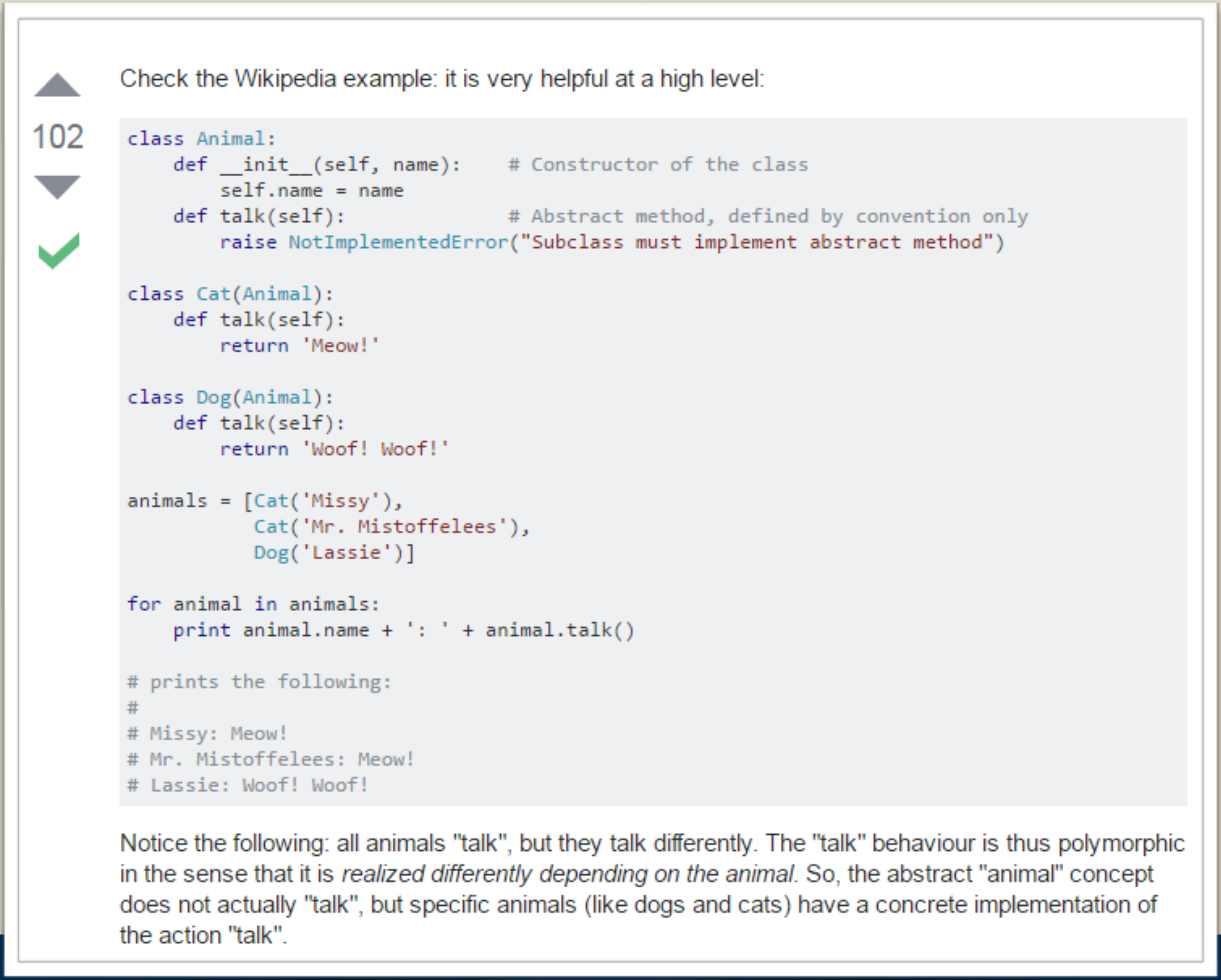
  Run the function as defined in the superclass (parent)

- *What's* pass?

  Define a function to be implemented in subclasses

What's *polymorphism?*

- *The provision of a single interface to entities of different types.*
- *Discuss: How might this relate to inheritance?*

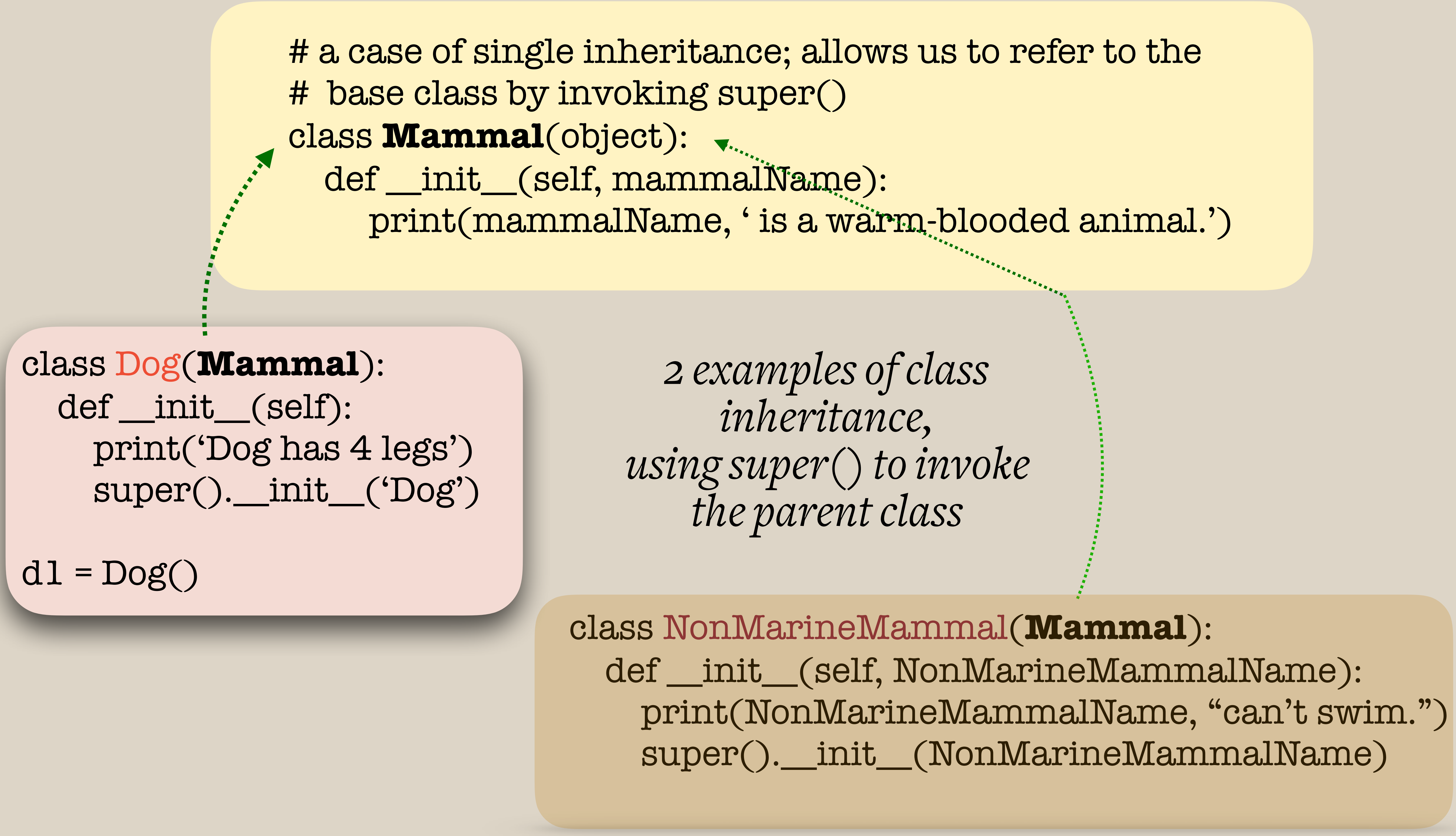


Check the Wikipedia example: it is very helpful at a high level:

```
class Animal:
    def __init__(self, name):    # Constructor of the class
        self.name = name
    def talk(self):              # Abstract method, defined by convention only
        raise NotImplementedError("Subclass must implement abstract method")

class Cat(Animal):
    def talk(self):
        return 'Meow!'

class Dog(Animal):
    def talk(self):
        return 'Woof! Woof!'

animals = [Cat('Missy'),
           Cat('Mr. Mistoffelees'),
           Dog('Lassie')]

for animal in animals:
    print animal.name + ': ' + animal.talk()

# prints the following:
#
# Missy: Meow!
# Mr. Mistoffelees: Meow!
# Lassie: Woof! Woof!
```

Notice the following: all animals "talk", but they talk differently. The "talk" behaviour is thus polymorphic in the sense that it is *realized differently depending on the animal*. So, the abstract "animal" concept does not actually "talk", but specific animals (like dogs and cats) have a concrete implementation of the action "talk".

Optional: super() and inheritance[s]



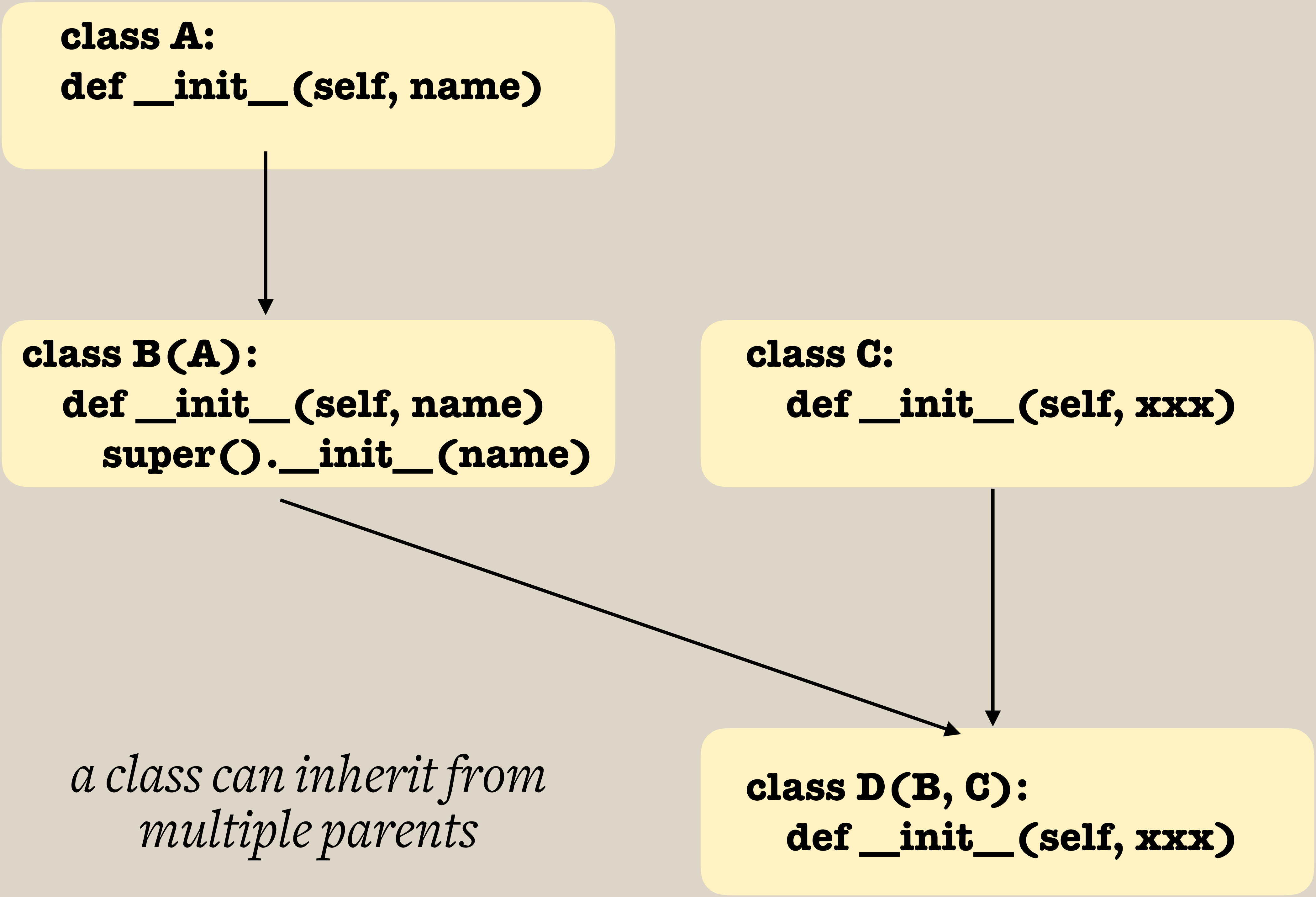
```
# a case of single inheritance; allows us to refer to the
#  base class by invoking super()
class **Mammal**(object):
    def __init__(self, mammalName):
        print(mammalName, ‘ is a warm-blooded animal.’)
```

```
class Dog(Mammal):
    def __init__(self):
        print(‘Dog has 4 legs’)
        super().__init__(‘Dog’)

d1 = Dog()
```

2 examples of class inheritance,
using super() to invoke the parent class

```
class NonMarineMammal(Mammal):
    def __init__(self, NonMarineMammalName):
        print(NonMarineMammalName, “can’t swim.”)
        super().__init__(NonMarineMammalName)
```

Optional: super() and multiple inheritance



```
class A:
def __init__(self, name)
```

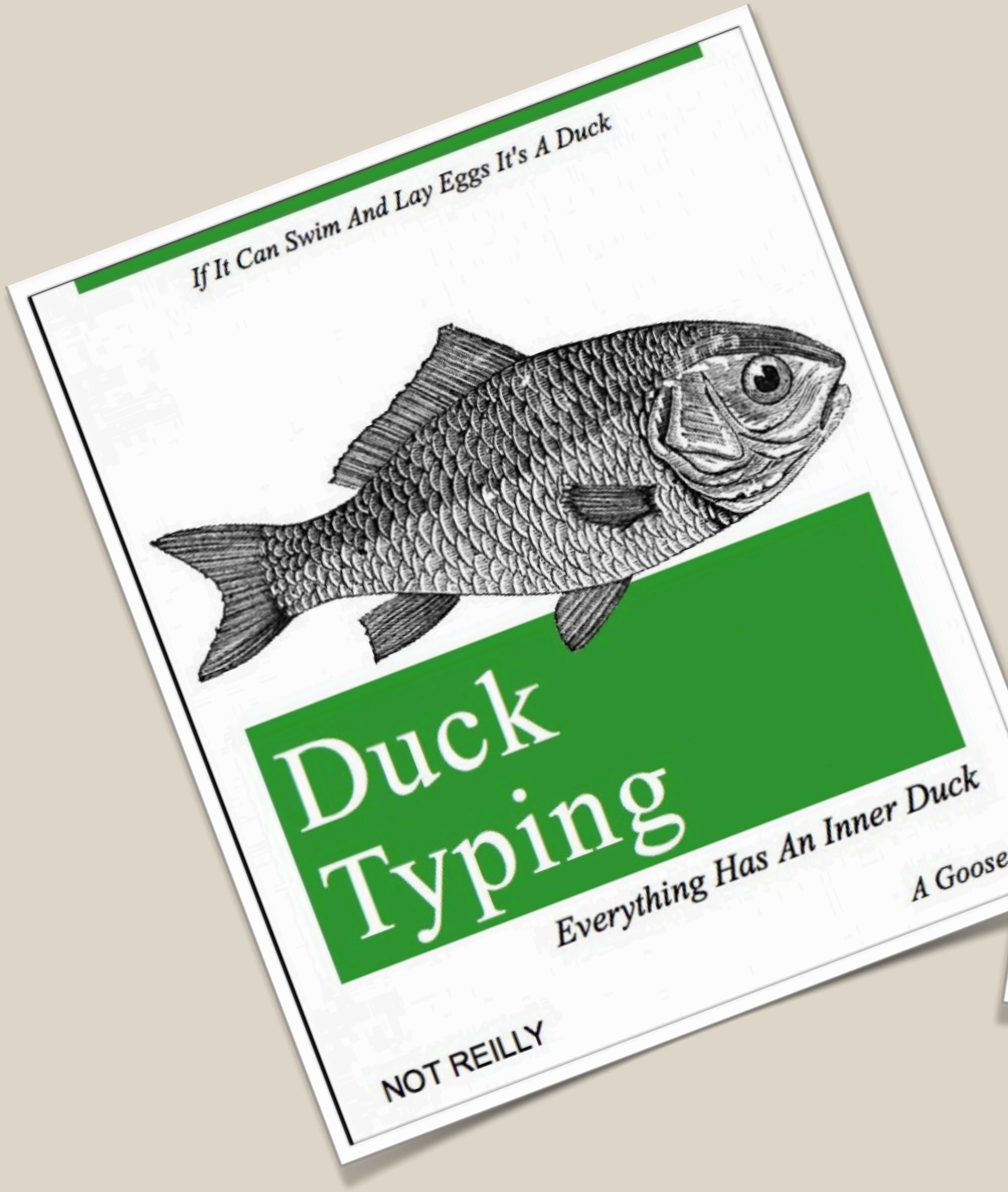
```
class B(A):
  def __init__(self, name)
    super().__init__(name)
```

```
class C:
  def __init__(self, xxx)
```

```
class D(B, C):
  def __init__(self, xxx)
```

a class can inherit from multiple parents

Quack. Repeat, quack. What is duck typing?


If It Can Swim And Lay Eggs It's A Duck
Duck Typing
Everything Has An Inner Duck
A Goose
NOT REILLY

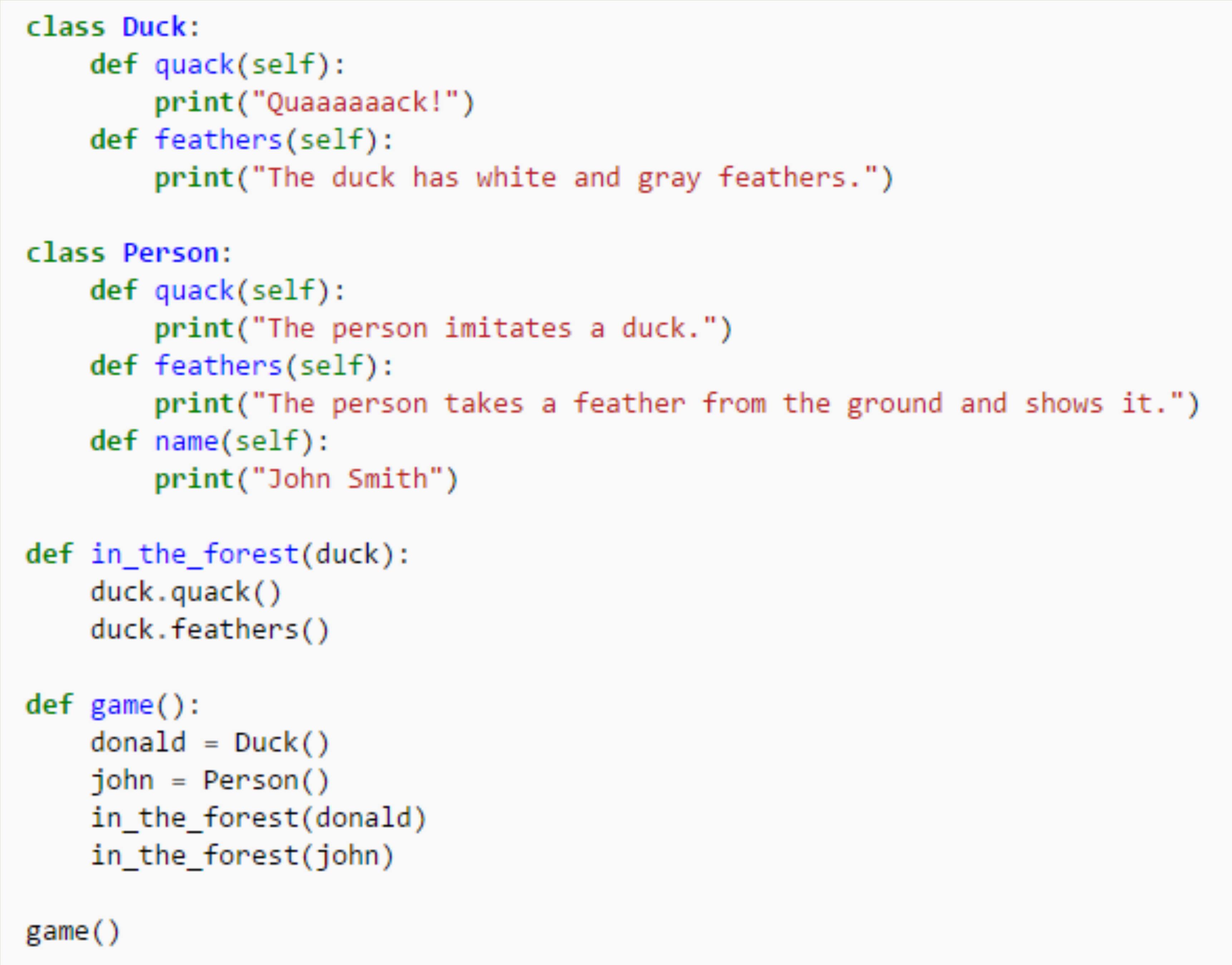
What is duck typing?

- *Python won't check variable* types *before running a function.*

if it looks like a duck

and quacks like a duck,

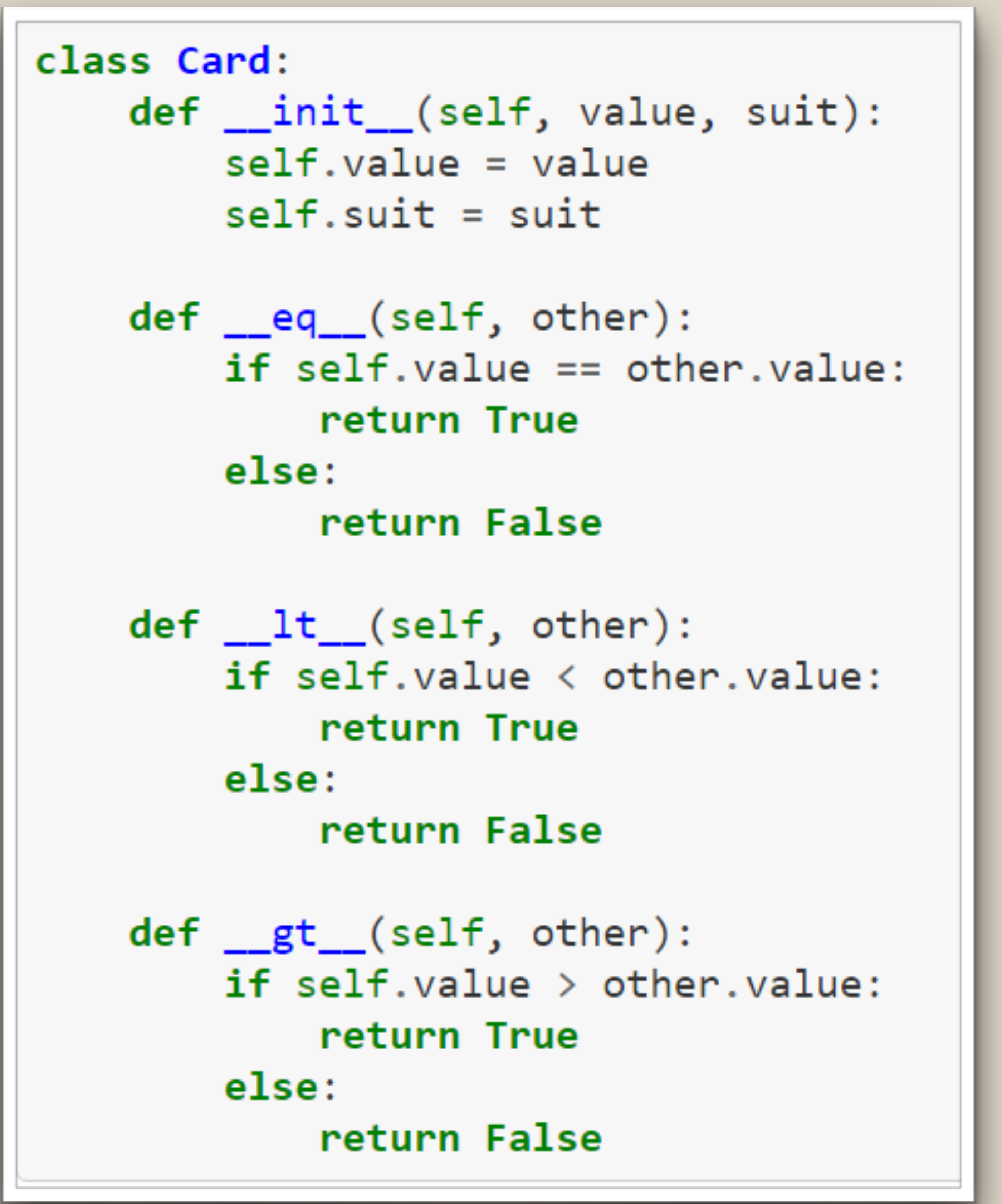
it's a duck.



```python
class Duck:
    def quack(self):
        print("Quaaaaaack!")
    def feathers(self):
        print("The duck has white and gray feathers.")

class Person:
    def quack(self):
        print("The person imitates a duck.")
    def feathers(self):
        print("The person takes a feather from the ground and shows it.")
    def name(self):
        print("John Smith")

def in_the_forest(duck):
    duck.quack()
    duck.feathers()

def game():
    donald = Duck()
    john = Person()
    in_the_forest(donald)
    in_the_forest(john)

game()
```

Magic Methods

- *Read and think about these methods. What is the purpose of 'em?*
- *How might you apply them in your own classes?*



```
class Card:
    def __init__(self, value, suit):
        self.value = value
        self.suit = suit

    def __eq__(self, other):
        if self.value == other.value:
            return True
        else:
            return False

    def __lt__(self, other):
        if self.value < other.value:
            return True
        else:
            return False

    def __gt__(self, other):
        if self.value > other.value:
            return True
        else:
            return False
```

Magic Methods (from p. 138ff)

__eq__(self, other)
__ne__(self, other)
__lt__(self, other)
__gt__(self, other)
__le__(self, other)
__ge__(self, other)

__add__(self, other)
__sub__(self, other)
__mul__(self, other)
__floordiv__(self, other)
__truediv__(self, other)
__mod__(self, other)
__pow__(self, other)

__str__(self)
__repr__(self)
__len__(self)
__name__(self)
__main__

Recap of the schedule:

Project 1 | build your own object oriented project

- *Code at home and collaborate in class*

Unit 9 | Working With Text and Binary Data

Unit 10 | NumPy

Unit 11 | Data Analysis With Pandas

Unit 12 | More Analysis With Pandas; Data Vis

Unit 13 | Testing