Week 15 - Friday

COMP 1800

Last time

- What did we talk about last time?
- Review up to Exam 2

Questions?

Assignment 10

Review

Final Exam

- Format:
  - Multiple choice questions (~20%)
  - Short answer questions (~20%)
  - Programming problems (~60%)
- Written in class
  - No notes
  - Closed book
  - No calculator

Final exam

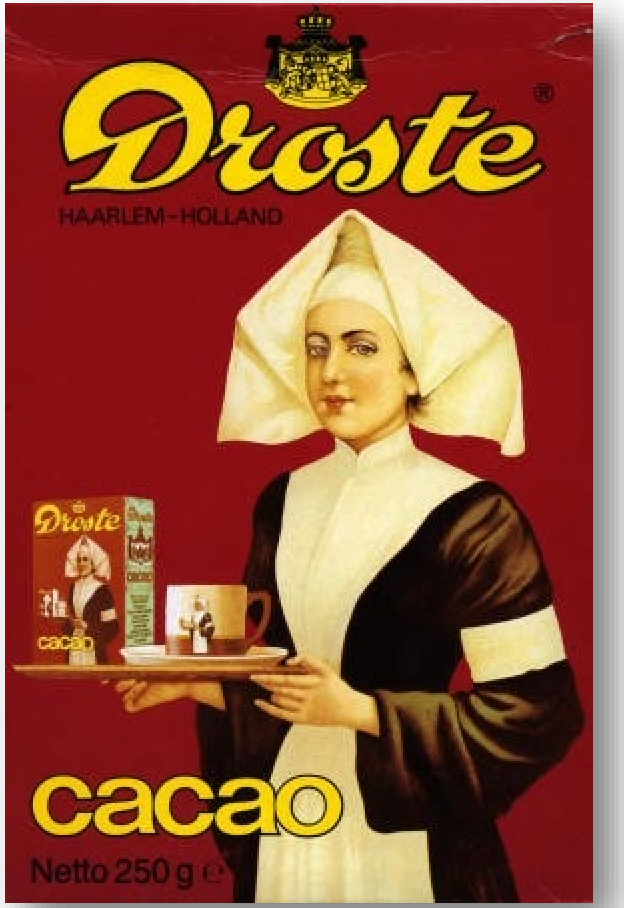
- Designed to be 50% longer than previous exams
- But you'll have 100% more time
- **Time:** Friday, 12/08/2023, 2:45 - 4:45 p.m.
- **Place:** Point 113

Recursion

To understand recursion, you must first understand recursion.

What is Recursion?

- Defining something in terms of itself
- To be useful, the definition must be based on progressively simpler definitions of the thing being defined



Useful Recursion

Two parts:

- Base case(s)
  - Tells recursion when to stop
  - For factorial, $\boldsymbol{n} = 1$ or $\boldsymbol{n} = 0$ are examples of base cases
- Recursive case(s)
  - Allows recursion to progress
  - "Leap of faith"
  - For factorial, $\boldsymbol{n} > 1$ is the recursive case

Approach for Problems

- Top down approach
- Don't try to solve the whole problem
- Deal with the next step in the problem
- Then make the "leap of faith"
- Assume that you can solve any smaller part of the problem

Implementing Factorial

- Base case ($n \leq 1$):
  - $1! = 0! = 1$
- Recursive case ($n > 1$):
  - $n! = n(n-1)!$

Code for Factorial

```
def factorial(n):
  if n <= 1:
    return 1
  else:
    return n*factorial(n - 1)
```

Base Case

Recursive Case

Recursion and loops are the same

- Any program that uses loops can be done with recursion
- Any program that uses recursion can be done with loops
- Sometimes it's easier to use loops
- Sometimes it's easier to use recursion
- A base case is necessary in recursion to tell the process when to stop
  - This is like a condition for while loop or the amount of iteration for a for loop
- A recursive case is necessary so that recursion can continue
  - This is similar to how a loop jumps back up to the top when it gets to the bottom

Adding up the numbers in a list

- Base case (Empty list):
  - 0

- Recursive case (At least one thing left in the list):
  - The value of the first thing plus the sum of the rest of the list

Code for Sum

```
def recursiveSum(list):
  if len(list) == 0:
     return 0
  else:
     return list[0] + recursiveSum(list[1:])
```

Base Case

Recursive Case

Tips for recursion

- Use it only in special circumstances, since it's usually slower than loops
- Recursive solutions are often impressive for how short the code is
- Some people love it, but it can be hard to think about
- Instead of trying to solve the entire problem, we think about unwrapping one layer of the problem
  - Don't think too much about what's going on in the other recursive calls since you can't access those variables
- You usually don't want to change the values of variables with **=** since that can make the recursion harder to think about

Drawing Recursively

Complex shapes

- Many natural things have recursive shapes:
  - Trees
  - Spiral shells
  - Blood vessels
  - Mountains
  - Snowflakes
- Using recursion, we can draw some complex, organic-looking shapes with only a little code

Drawing squares

- Let's start with a simple (non-recursive) function that draws a square with a turtle called **`yertle`** and a side length called **`side`**

```
def drawSquare(yertle, side):
    for i in range(4):
        yertle.forward(side)
        yertle.right(90)
```

- It works by going clockwise around the square
- It (importantly) returns **`yertle`** to the starting point

Nested squares

- We can use the **`drawSquare()`** function repeatedly to draw a series of nested squares with progressively smaller sides
- Base case (Side length < 1):
  - Do nothing (Seems odd but is not an unusual base case)
- Base case (Side length ≥ 1):
  - Draw a square with the given side length
  - Continue drawing nested squares with a side length that's 5 units smaller

Nested squares function

- Here is that function implemented in Python:

```
def nestedSquares(yertle, side):
    if side >= 1: # hidden base case
        drawSquare(yertle, side)
        nestedSquares(yertle, side - 5)
```

- This function is called like any normal function:

```
nestedSquares(someTurtle, 200)
```

Trees

- Squares are fine, but they're not very exciting (or very organic looking)
- We can extend the idea into drawing a tree shape
- A tree looks kind of like a capital Y
- But then, instead of straight lines, we can replace the two branches of the Y with smaller Y's
  - And so on …
    - And so on …

Recursion for tree drawing

- Base case (Trunk length < 5):
  - Do nothing
- Recursive case (Trunk length ≥ 5):
  - Move forward trunk length
  - Turn right 30°
  - Draw a tree (recursively) with a trunk length 15 units shorter
  - Turn left 60° (which turns back to the original heading plus another 30°)
  - Draw a tree (recursively) with a trunk length 15 units shorter
  - Turn right 30° (which turns back to the original heading)
  - Move backward the trunk length (returning to the starting point)

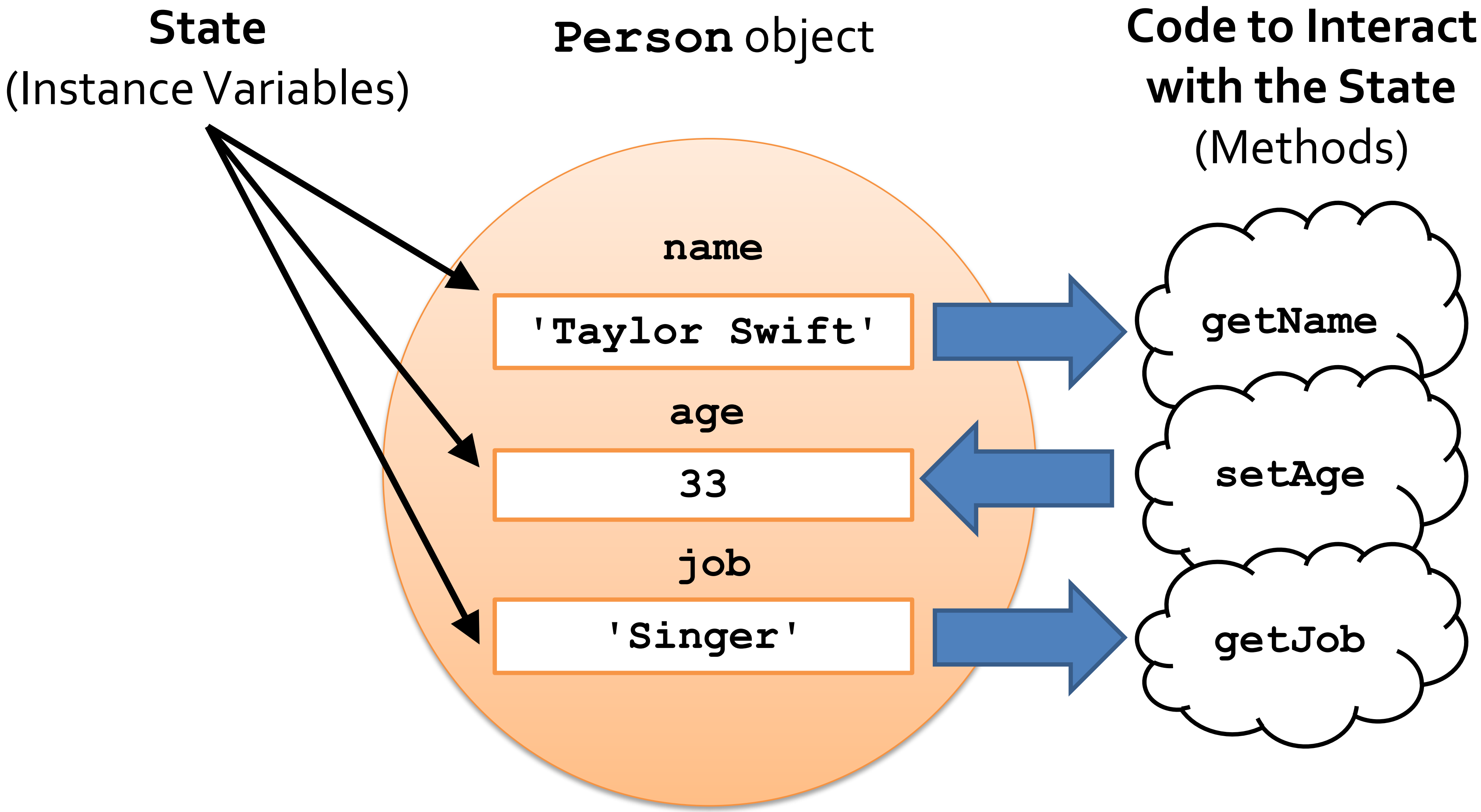
Tree function

- Here is that function implemented in Python:

```
def tree(yertle, trunkLength):
    if trunkLength >= 5: # hidden base case
        yertle.forward(trunkLength)
        yertle.right(30)
        tree(yertle, trunkLength - 15)
        yertle.left(60)
        tree(yertle, trunkLength - 15)
        yertle.right(30)
        yertle.backward(trunkLength)
```

Objects in Python

What's an object?



Objects

- The idea of an object is to group together data and code
- You have used objects a bit already
  - Strings are objects
  - Even lists are a special kind of object

Why are objects a good idea?

- Encapsulation: hiding data to keep it safe
- Methods provide useful ways to interact with the data
- It's convenient to keep related data grouped together
  - You could have a list of **`Person`** objects instead of three separate lists of names, ages, and jobs

Calling methods

- When you have an object, you can call methods on it
- A method is like a function, except that it has access to the details of the object
- To call a method, you type the name of the object, a dot, and the name of the method
- A method will always have parentheses after it
- Sometimes the parentheses will have arguments that the method uses

Method call examples

- You've called methods with strings:

```
phrase = 'BOOM goes the dynamite!'
other1 = phrase.lower() # gets lowercase version
other2 = phrase.upper() # gets uppercase version
words = phrase.split() # turns to list
```

- You've called methods on a list:

```
words.sort() # sorts the list
```

Instance variables

- Instance variables are the data **inside** of an object
- Like methods, you can access an instance variable with the name of the object, a dot, and then the name of the member
- Unlike methods, instance variables never have parentheses
- They are values, not functions that do things

Adding members

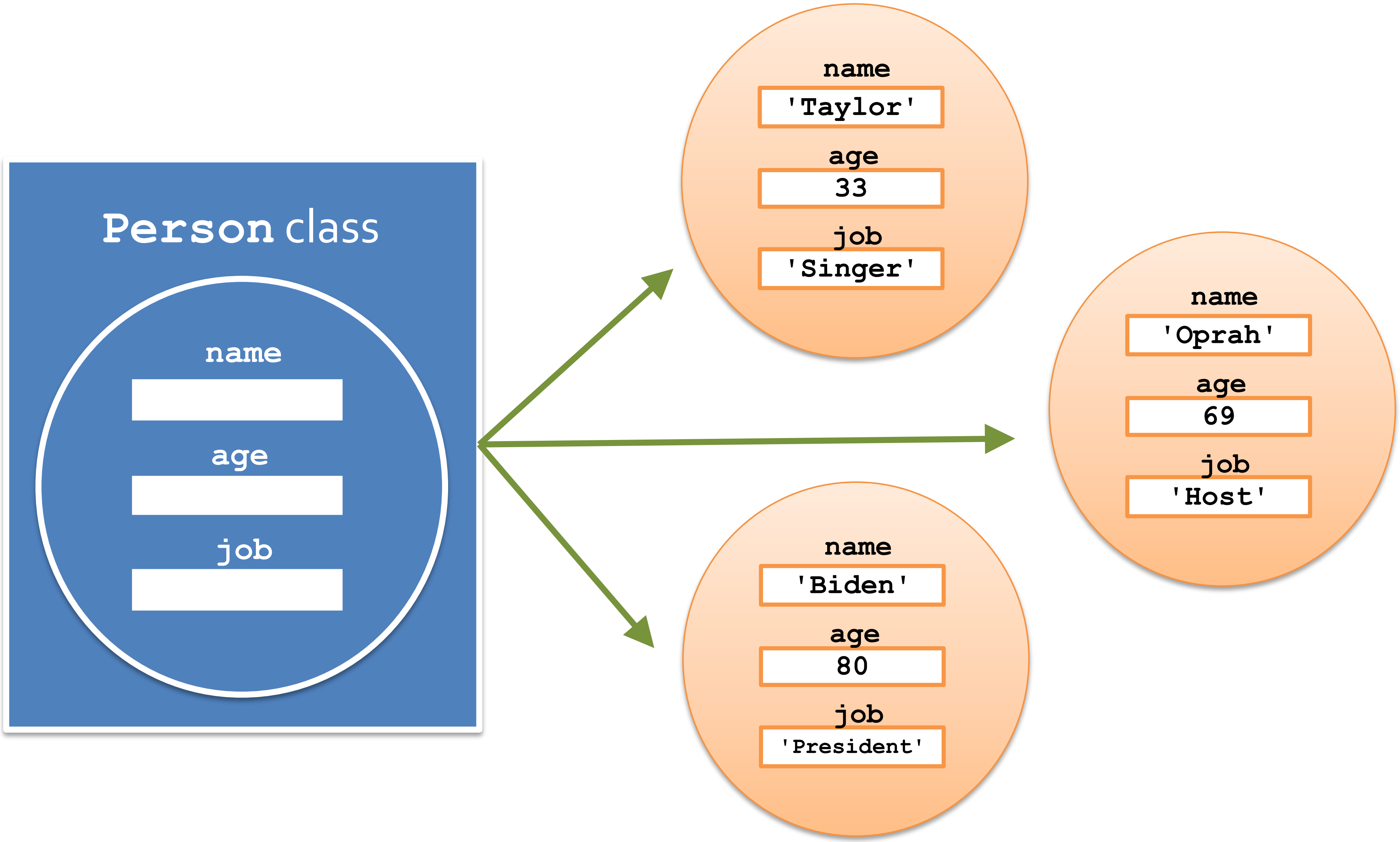
- Python allows us to add instance variables anytime we want
- Doing so lets us keep extra information in each object
- For example, we could give a **`Person`** object a **`nickname`** variable after creating it

```
taylor = Person('Taylor Swift', 33, 'Singer')
taylor.nickname = 'Tay Tay'
```

Creating entirely new classes

- Adding instance variables is fine, but what if you want to create an object from scratch?
- A **class** is a template for an object
- You can define a class that will allow you to create your own custom objects

Classes are like blueprints



Planet class

- Let's look at an example class that holds information about a planet

```
class Planet:
    def __init__(self, name, radius, mass, distance):
        self.name = name
        self.radius = radius
        self.mass = mass
        self.distance = distance

    def getName(self):
        return self.name

    def setName(self, name):
        self.name = name
```

What is `self`?

- **`self`** is a reference to the object that you're currently inside of
- If you forget to use **`self`**, you aren't talking about the current object, you're talking about an outside variable
- The Java or C++ equivalent of **`self`** is **`this`**
- When calling a method (or the constructor), you always ignore the **`self`** parameter
- The object itself is automatically supplied

Constructor

- A **constructor** is a special kind of method that initializes the values inside of an object
- It's how a new object is created
- In Python, its name is always `__init__`
- It takes in the initial values for the object

```
class Planet:
    def __init__(self, name, radius, mass, distance):
        self.name = name
        self.radius = radius
        self.mass = mass
        self.distance = distance
```

Creating a new object

- To create a new object, you call its constructor
- This means typing the name of the class with parentheses after it, including the initial values for the object
- When you call the constructor, you **don't** pass in **`self`**!
  - That happens automatically

```
planet1 = Planet('Jupiter', 69911, 1.9E27, 7.78E8)
planet2 = Planet('Mars', 3390, 6.4e23, 2.27E8)
```

Accessors

- An **accessor** is a kind of method that **gets** a value out of an object
- It can read an existing value or compute a new one
- An accessor doesn't change the data inside the object

```
def getName(self):
    return self.name
```

- Calling an accessor is like calling any other method on an object
  - Object name, dot, then method name
  - Leave off the **`self`**!

```
name = planet1.getName()
print(name)
```

Mutators

- A **mutator** is a kind of method that **sets** a value in an object
- Its purpose is to change the data inside the object

```
def setName(self, name):
    self.name = name
```

- It could do some checking to make sure that a good value is supplied

```
planet1.setName('Jove')   # new name
print(planet1.getName()) # prints Jove
```

Hiding data in Python

- Python doesn't have a **`private`** keyword
- Instead, it uses a naming convention to hide variables
- All member variables that you want to be hidden should have names that start with double underscore (__)
- Such variables cannot be accessed directly
- I didn't talks about data hiding before because:
  - Hiding variables in Python this way is not as universal as in languages like Java
  - It makes stuff ugly to read
  - It adds another layer of confusion
- If you're serious about writing object-oriented Python, you should still do it

Hiding example

- Here's part of the **`Planet`** class from before, with appropriate hiding

```
class Planet:
    def __init__(self, name, radius, mass, distance):
        self.__name = name
        self.__radius = radius
        self.__mass = mass
        self.__distance = distance

    def getName(self):
        return self.__name

    def setName(self, name):
        self.__name = name
```

Simulation

Continuous simulations

- The example we did of the solar system was a simulation
  - Using (totally unrealistic) physics
- Those kinds of simulations can be useful for scientists trying to model behavior
- Real simulations are much more complex
  - Important example: weather forecasting
- These kinds of simulations are **continuous simulations** because they show the system evolving continuously as time goes on

Discrete event simulations

- Discrete event simulations are another kind of simulation
- In these, events happen at particular times
- Then, the system progresses onward after each time step, based on what happened
- The elements of the system that can act are sometimes called **agents**
- Discrete event simulations are good for modeling situations like agents shopping, standing in line, visiting the BMV, etc.
- Another possibility is modeling an ecosystem

Ecosystem

- Our ecosystem simulation will contain fish and bears
- They will exist on a grid
- Only one creature can exist at any location on the grid
- Each turn, one creature is randomly selected to come alive and do actions
- Fish can breed, move, and die
- Bears can breed, move, eat, and die
- To model this simulation, we will create objects for the world, for fish, and for bears

UML

- The **Unified Modeling Language** (UML) is an international standard for making diagrams of software systems
- One of the most commonly used diagrams is called a **class diagram**
- One standard for class diagrams has three sections:
  - Name
  - Instance variables
  - Methods
- To the right is an example of what that looks like

Class Name
Instance variables
Methods

Class diagram for `World`

- Here is a UML class diagram for the **`World`** class

World

`maxX`
`maxY`
`thingList`
`grid`
`turtle`
`screen`

`draw`
`getMaxX`
`getMaxY`
`addThing`
`deleteThing`
`moveThing`
`live`
`emptyLocation`
`lookAtLocation`

Class diagram for `Bear`

- Here is a UML class diagram for the **`Bear`** class

Bear

`x`
`y`
`world`
`breedTick`
`starveTick`
`turtle`

`getX`
`getY`
`setX`
`setY`
`setWorld`
`appear`
`hide`
`move`
`live`
`tryToBreed`
`tryToMove`
`tryToEat`

Class diagram for `Fish`

- Here is a UML class diagram for the **`Fish`** class

`isinstance()`

- If you want to test to see if a variable has a certain type, you can also use the **`isinstance()`** function
- It's useful for **`if`** statements
- It will also help us find out if an object is a **`Fish`** or a **`Bear`**

```
x = 5
if isinstance(x, int):
    print("It's an int!")
else:
    print("What's going on?")
```

Inheritance

Inheritance

- The idea of inheritance is to take one class and generate a child class
- This child class has everything that the parent class has (members and methods)
- But, you can also add more functionality to the child
- The child can be considered to be a **specialized** version of the parent

Code reuse

- The key idea behind inheritance is safe code reuse
- You can use old code that was designed to, say, sort lists of `Vehicle`s, and apply that code to lists of `Car`s
- All that you have to do is make sure that `Car` is a subclass (or child class) of `Vehicle`

Creating a subclass

- All this is well and good, but how do you actually create a subclass?
- Let's start by writing the **Vehicle** class

```
class Vehicle:
  def travel(self, destination):
    print('Traveling to', destination)
```

Extending a superclass

- We use put the superclass name in parentheses when making a subclass

```
class Car(Vehicle):
  def __init__(self, model):
     self.model = model

  def getModel(self):
     return self.model

  def startEngine(self):
     print('Vroooom!')
```

- A **`Car`** can do everything that a **`Vehicle`** can, plus more

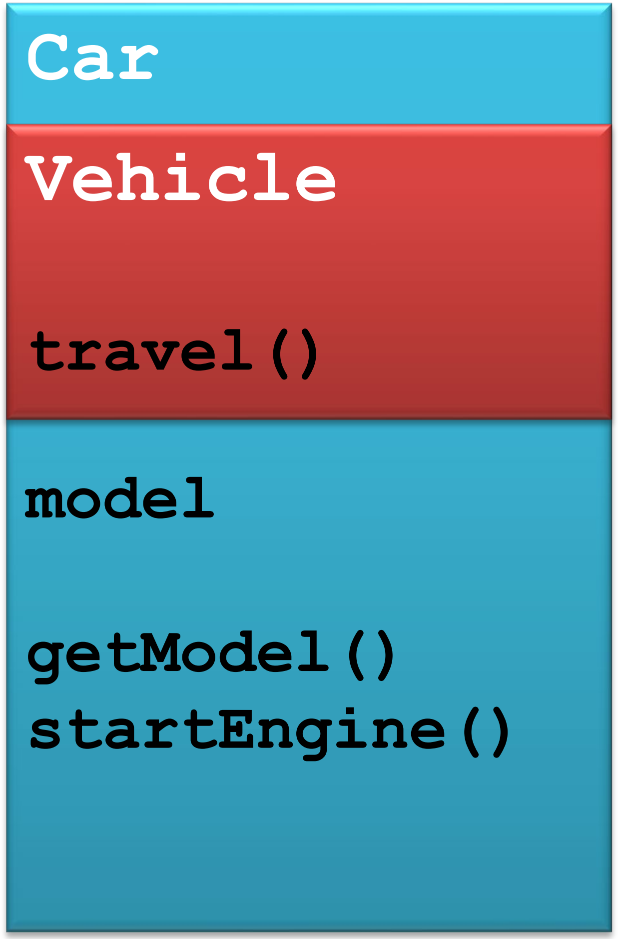
Power of inheritance

- There is a part of the **`Car`** class that knows all the **`Vehicle`** members and methods

```
car = Car('Camry')

#prints 'Camry'
print(car.getModel())

#prints 'Vroooooom!'
car.startEngine()

#prints 'Traveling to New York City'
car.travel('New York City')
```

A look at a Car

- Each **`Car`** object actually has a **`Vehicle`** object buried inside of it
- If code tries to call a method that isn't found in the **`Car`** class, it will look deeper and see if it is in the **`Vehicle`** class
- The outermost method will always be called



Calling the parent constructor

- If a class's parent has a constructor (the `__init__()` method), that constructor needs to get called too
  - That way, your parent gets set up correctly
- The best way to do that is to access the parent with the `super()` function
- Inside a class's constructor, it should call `super().__init__()`
  - Inserting arguments if appropriate

Parent example

- The **`Car`** class has a constructor that takes a model
- So, if we make a child class, it needs to call the parent constructor with a model

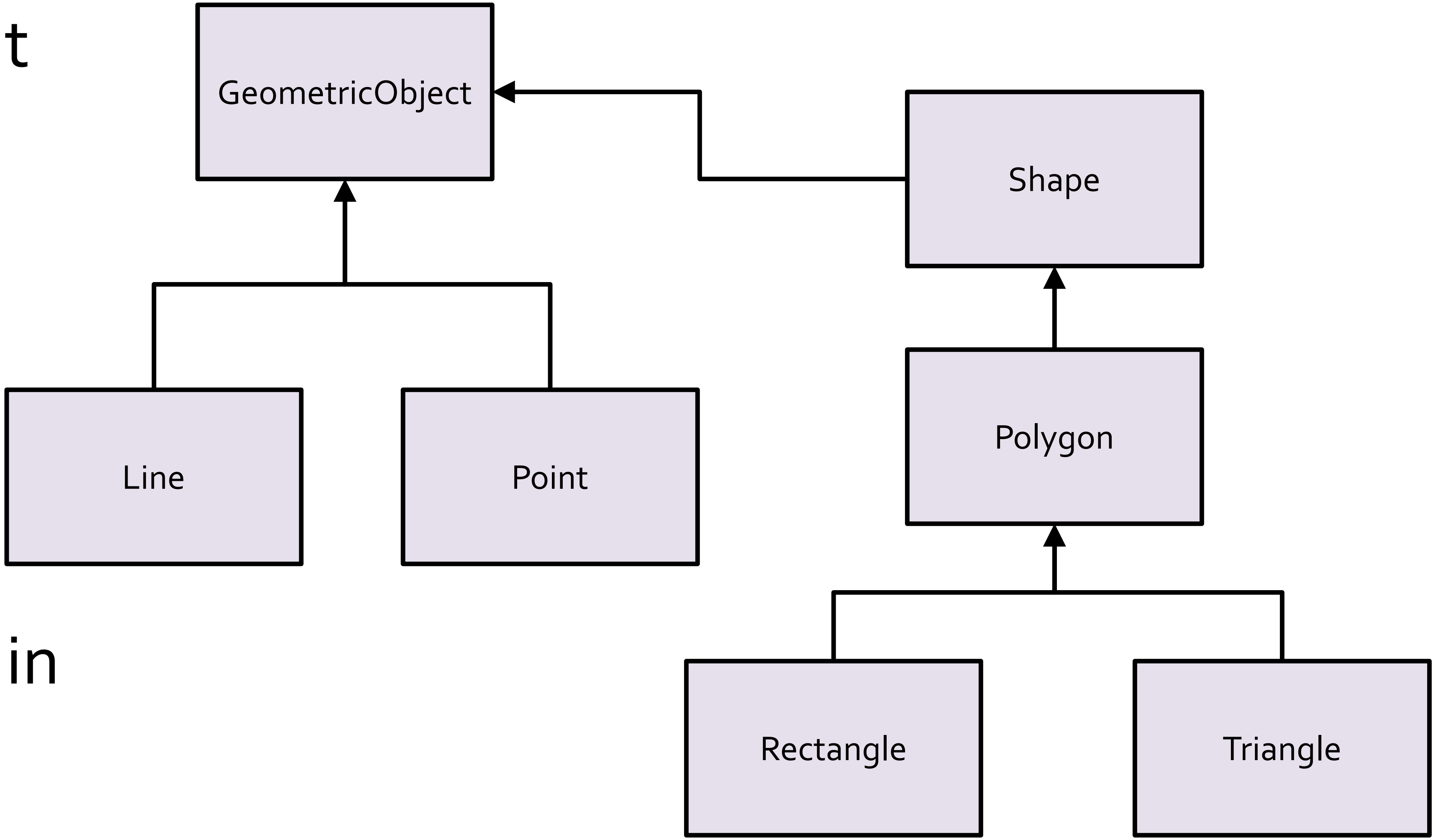
```
class RocketCar(Car):
  def __init__(self):
    super().__init__('Rocket Car')

  def fireRockets(self):
    print('Rockets firing!')
```

Inheritance hierarchies

- In large, object-oriented systems, it's common for there to be many classes with many children (and grandchildren, and great-grandchildren…)
- This kind of arrangement is called an **inheritance hierarchy**
- Using UML, we can draw inheritance relationships between classes with arrows
- Although it is counterintuitive, the UML standard is for the arrow to point from the child to the parent

Shapes

- Drawing different kinds of shapes can be a useful task for inheritance
- Consider the following inheritance hierarchy shown in UML



Drawing shapes

- The classes shown in the previous slide have an inheritance relationship with **`GeometricShape`**
  - The *is-a* relationship, since each of those shapes is a **`GeometricShape`**
- We also need a place to draw those shapes
- We can create a **`Canvas`** class to draw them
- A **`Canvas`** is *not* a **`GeometricShape`**
- Instead, it provides a turtle that **`GeometricShape`** objects can use to draw themselves

One final bit of Python syntax

- You can't have a function (or an **`if`** statement or a loop) with nothing in it
- For these rare circumstances, there's a special keyword that means do nothing
  - The **`pass`** keyword

```
def doNothing():
  pass # would have errors otherwise
```

Adding to existing classes is nice…

- Sometimes you want to do more than add
- You want to change a method to do something different
- You can write a method in a child class that has the same name as a method in a parent class
- The child version of the method will always get called
- This is called **overriding** a method

Mammal example

- We can define the **`Mammal`** class as follows:

```
class Mammal:
  def makeNoise(self):
    print('Grunt!')
```

Mammal subclasses

- From there, we can define the **`Dog`**, **`Cat`**, and **`Human`** subclasses, overriding the **`makeNoise()`** method appropriately

```
class Dog(Mammal):
  def makeNoise(self):
    print('Woof')
```

```
class Cat(Mammal):
  def makeNoise(self):
    print('Meow')
```

```
class Human(Mammal):
  def makeNoise(self):
    print('Hello')
```

Studying Advice

Studying advice

- Focus on quizzes
- Focus on assignments
- Memorizing things about Python is okay
- Practicing programming is better

Upcoming

Next time…

- There is no next time!
- Consider visiting CodingBat.com for Python practice

Reminders

- **Fill out course evaluations!**
- Finish Assignment 10
  - Due tonight by midnight!
- Study for Final Exam
  - Friday, 12/08/2023, 2:45 - 4:45 p.m.