ITP 115 – Programming in Python

Objects
part 2

Review

Object-Oriented Programming (OOP)

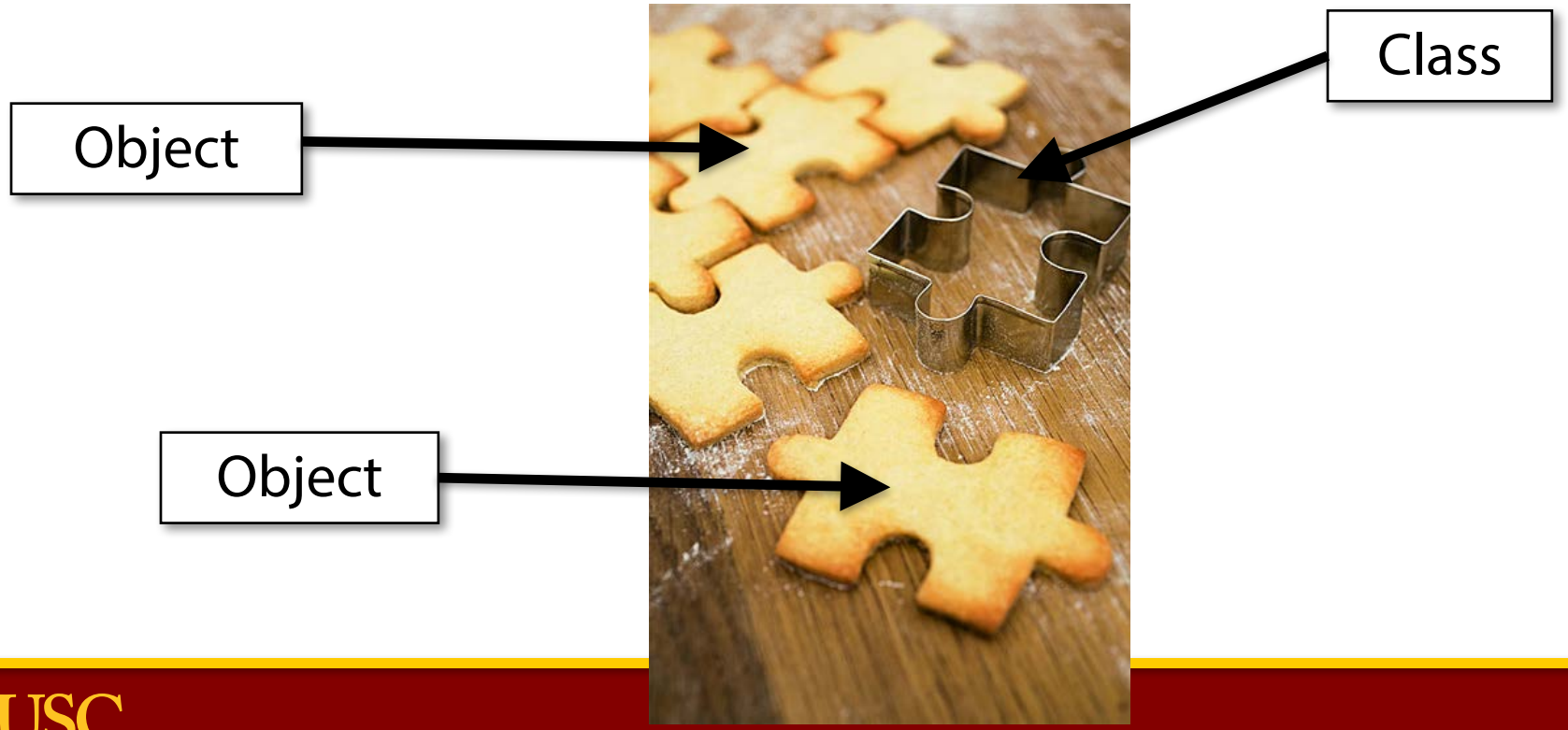
- A different way of thinking about programming
- A modern methodology used in the creation of the majority of new, commercial software
- The basic building block is the **software object**
  - just called an **object**

Classes and Objects

- Classes are like blueprints and defined by **`class`**
  - A class isn't an object, it's a design for one
- Objects are created (*instantiated*) from a class definition
- Classes contain
  - Attributes: set of object variables given to every object
  - Methods: functions that are part of each object

Classes and Objects

- Think of a **class** as a cookie cutter
- **Objects** (or **instances**) are the cookies



Creating an Instance of a Class

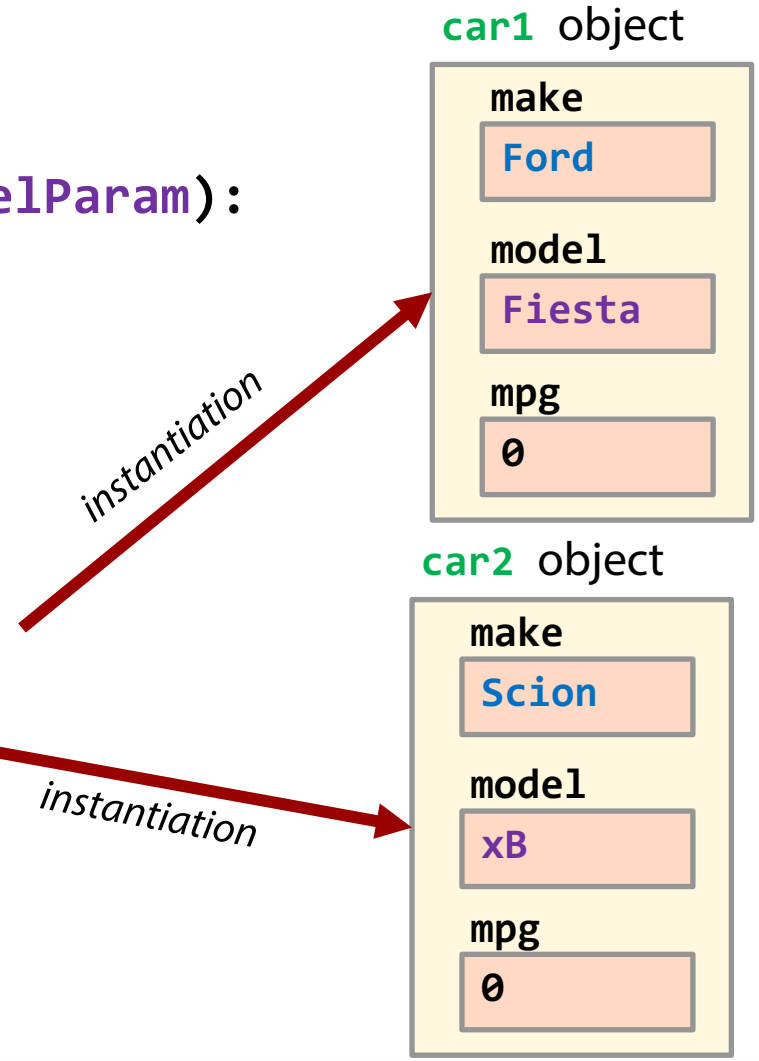
```
class Vehicle(object):
  ...

def main():
  car1 = Vehicle()

main()
```

`__init__(self):`

- A constructor is **method** that is used to create an instance of an object
- A constructor define what attributes will exist inside a object
- Constructors are called **automatically** when you create an object

Attributes and Constructors

```
class Vehicle(object):
  def __init__(self, makeParam, modelParam):
    self.make = makeParam
    self.model = modelParam
    self.mpg = 0

def main():
    car1 = Vehicle("Ford", "Fiesta")
    car2 = Vehicle("Scion", "xB")
```



Methods

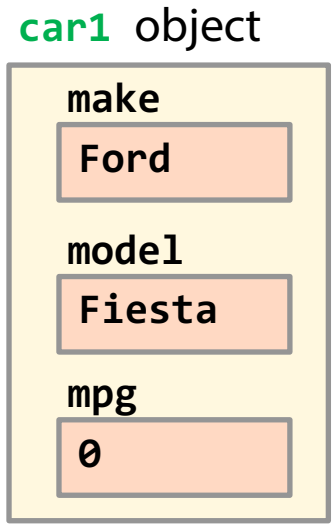
- Classes can have methods (or behaviors)
- Methods are part of the object just like attributes
  - Think: *functions associated with an object*
- Methods can access the attributes defined in the constructor using **`self`**

Method Input and Output

```
class Vehicle(object):
   def calcTripCost(self, miles):
      ... #perform some calculations
      return totalCost #new variable


def main():
   car1 = Vehicle()
   cost = car1.calcTripCost(100)
```

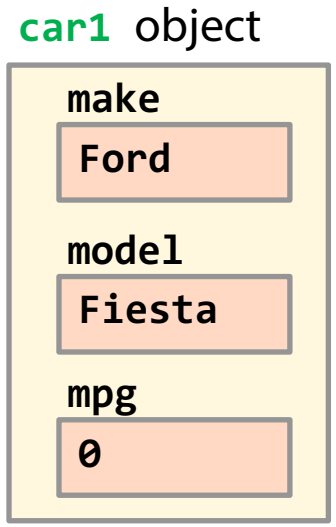
Changing Attributes

```
class Vehicle(object):
    def __init__(self, make, model):
      self.make = make
      self.model = model
      self.mpg = 0

def main():
   car1 = Vehicle("Ford", "Fiesta")
```



Changing Attributes

```
class Vehicle(object):
    def __init__(self, make, model):
      self.make = make
      self.model = model
      self.mpg = 0

def main():
   car1 = Vehicle("Ford", "Fiesta")
   print("The MPG is", car1.mpg)
```

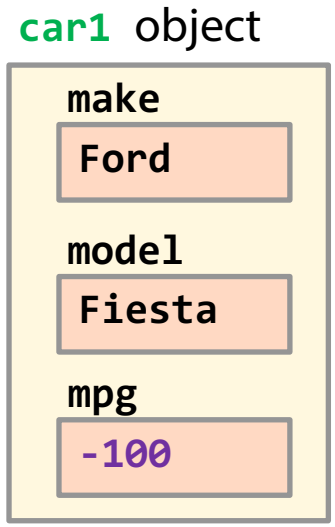


Output

```
The MPG is 0
```

Changing Attributes

```
class Vehicle(object):
    def __init__(self, make, model):
      self.make = make
      self.model = model
      self.mpg = 0

def main():
   car1 = Vehicle("Ford", "Fiesta")
   print("The MPG is", car1.mpg)
   car1.mpg = -100
```



Should this be allowed?

Protecting Attributes

- Instead of changing attributes directly, we can provide a method
- Method will allow changes to attributes
- Method will be able to do error checking to make sure the new value is valid
- We call this a **set method** (also called a **setter** or **mutator**)

Using Set Methods

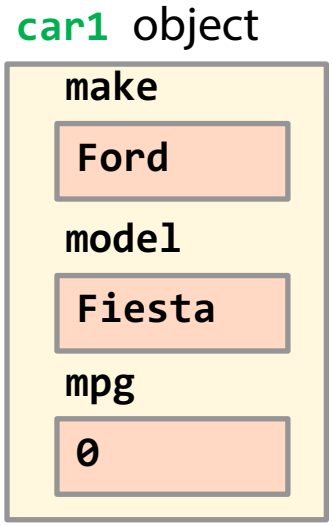
- Syntax

```
setAttribute(self, newAttribute)
```

  - Assigns the parameter value to the attribute
  - May perform error checking
  - Doesn't return anything

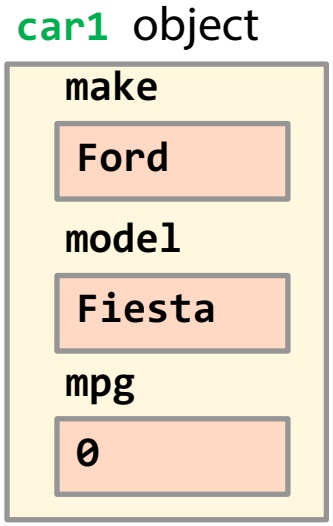
Using Set Methods

```
class Vehicle(object):
    def __init__(self, make, model):
        self.make = make
        self.model = model
        self.mpg = 0

    def setMPG(self, newMPG):
        if newMPG >= 0:
            self.mpg = newMPG
        else:
            print("Invalid MPG")

def main():
    car1 = Vehicle("Ford", "Fiesta")
```



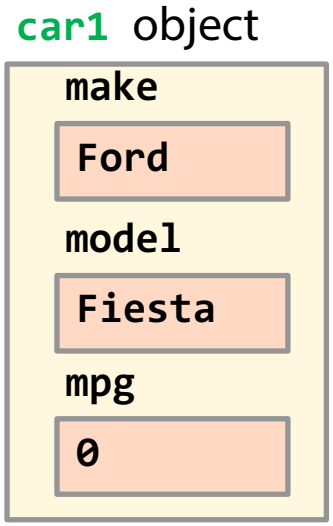
Using Set Methods

```
class Vehicle(object):
    def __init__(self, make, model):
        self.make = make
        self.model = model
        self.mpg = 0

    def setMPG(self, newMPG):
        if newMPG >= 0:
            self.mpg = newMPG
        else:
            print("Invalid MPG")

def main():
    car1 = Vehicle("Ford", "Fiesta")
```



Using Set Methods

```
class Vehicle(object):
    def __init__(self, make, model):
        self.make = make
        self.model = model
        self.mpg = 0

    def setMPG(self, newMPG):
        if newMPG >= 0:
            self.mpg = newMPG
        else:
            print("Invalid MPG")

def main():
    car1 = Vehicle("Ford", "Fiesta")
    car1.setMPG(-18)
```

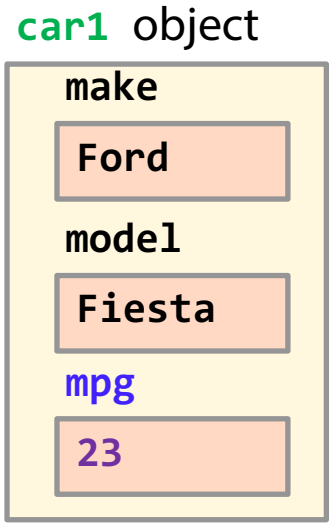


Output

```
Invalid MPG
```

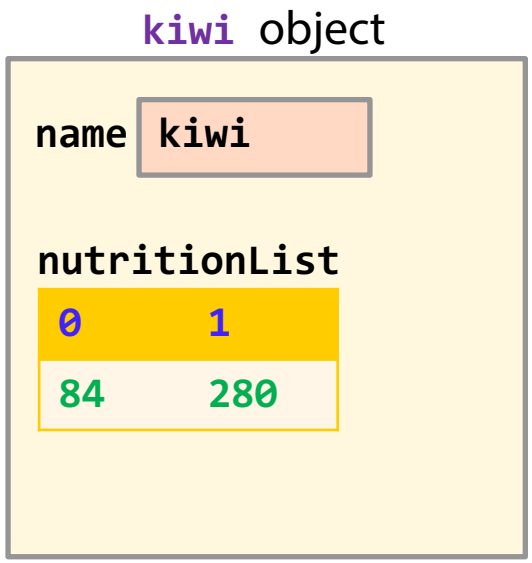
Using Set Methods

```
class Vehicle(object):
    def __init__(self, make, model):
        self.make = make
        self.model = model
        self.mpg = 0

    def setMPG(self, newMPG):
        if newMPG >= 0:
            self.mpg = newMPG
        else:
            print("Invalid MPG")

def main():
    car1 = Vehicle("Ford", "Fiesta")
    car1.setMPG(-18)
    car1.setMPG(23)
```



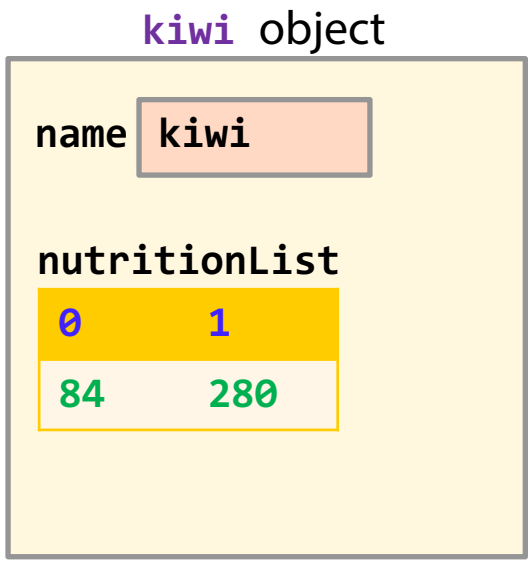
Accessing Attributes

```
class Fruit(object):
  def __init__(self, nameParam, vitC, vitA):
    self.name = nameParam
    self.nutritionList = [vitC, vitA]


def main():
  kiwi = Fruit("kiwi", 84, 280)
```



Accessing Attributes

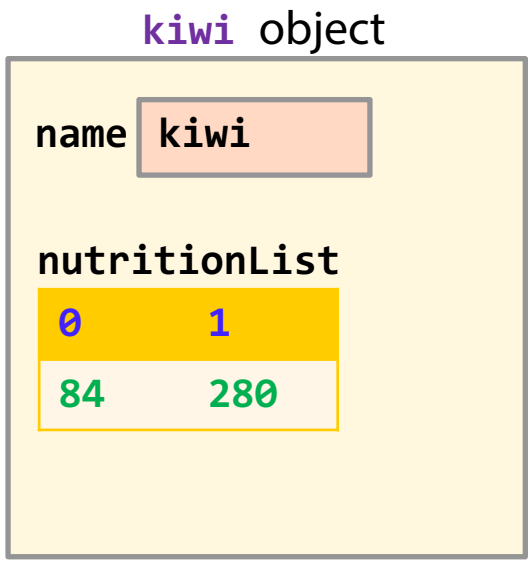
```
class Fruit(object):
  def __init__(self, nameParam, vitC, vitA):
    self.name = nameParam
    self.nutritionList = [vitC, vitA]


def main():
  kiwi = Fruit("kiwi", 84, 280)
```



How do we access nutrition values?

Accessing Attributes

```
class Fruit(object):
  def __init__(self, nameParam, vitC, vitA):
    self.name = nameParam
    self.nutritionList = [vitC, vitA]
```



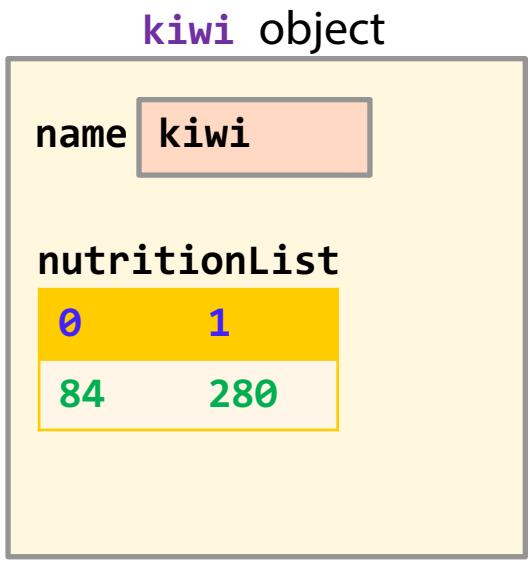

```
def main():
  kiwi = Fruit("kiwi", 84, 280)
  print(kiwi.name, "has", kiwi.nutritionList[1], "mg Vitamin A")
```

Output

```
kiwi has 280 mg vitamin A
```

Accessing Attributes

```
class Fruit(object):
  def __init__(self, nameParam, vitC, vitA):
    self.name = nameParam
    self.nutritionList = [vitC, vitA]
```




```
def main():
  kiwi = Fruit("kiwi", 84, 280)
  print(kiwi.name, "has", kiwi.nutritionList[2], "mg fiber")
```

What if the index doesn't exist?

Accessing Attributes

- Instead of accessing attributes directly, we can provide a method to access attributes
- Method will be able to do make sure values exists
- This also makes the code easier to read
- We call this a **get method** (also called a **getter** or **accessor**)

Using Get Methods

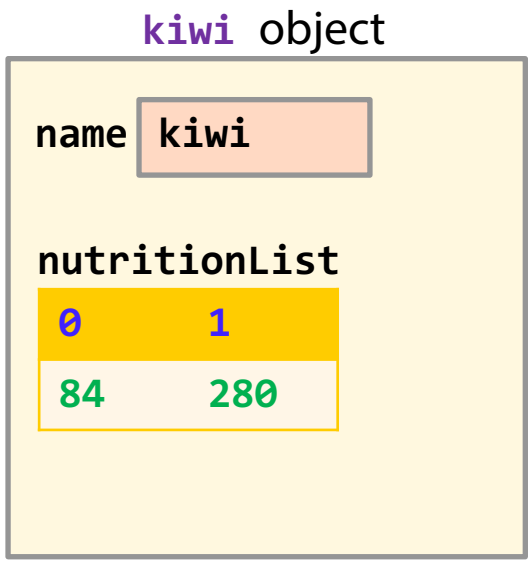
- Syntax

  ```
  getAttribute(self)
  ```

  – **Returns** the value of the attribute

Accessing Attributes

```
class Fruit(object):
    def getNutrition(self,param):
        if param.lower() == "vitamin c":
            return self.nutritionList[0]
        elif param.lower() == "vitamin a":
            return self.nutritionList[1]
        else:
            return 0


def main():
    kiwi = Fruit("kiwi", 84, 280)
    print(kiwi.name, "has", kiwi.getNutrition("vitamin a"), "mg Vitamin A")
```

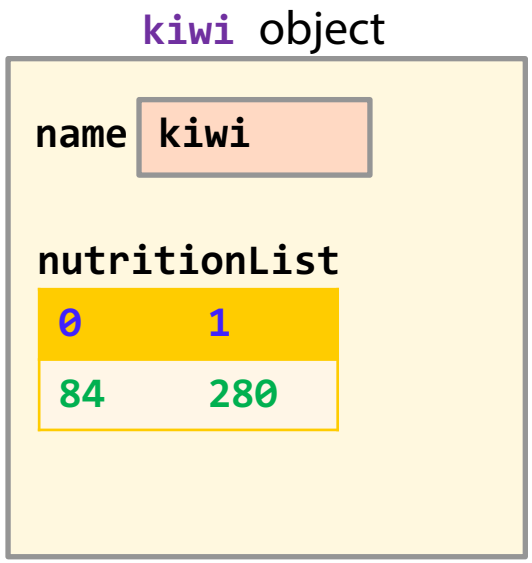


Output

```
kiwi has 280 mg vitamin A
```

Accessing Attributes

```
class Fruit(object):
    def getNutrition(self,param):
        if param.lower() == "vitamin c":
            return self.nutritionList[0]
        elif param.lower() == "vitamin a":
            return self.nutritionList[1]
        else:
            return 0

def main():
    kiwi = Fruit("kiwi", 84, 280)
    print(kiwi.name, "has", kiwi.getNutrition("fiber"), "mg fiber")
```

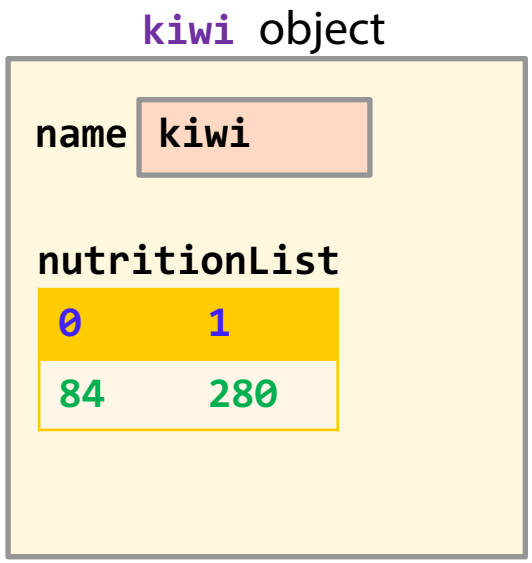


Output

```
kiwi has 0 mg fiber
```

Accessing Attributes

```
class Fruit(object):
...
   def getName(self):
      return self.name
```




```
def main():
  kiwi = Fruit("kiwi", 84, 280)
  print(kiwi.getName(), "has", kiwi.getNutrition("fiber"), "mg fiber")
```

Output

```
kiwi has 0 mg fiber
```

Side Note: Encapsulation

- Creating get / set methods is part of **encapsulation**
- Goal is to help other programmers to **use a class** without needing to know **how the class works**
- Good method design should separate ***what*** from ***how***

Examples

- How does a list actually work?
  - To us, it isn't important
  - We just need to know how to use list
- How does a car work?
  - We do not see mechanical details of **how** engine, wheels, etc. work
  - We see the brake pedal, accelerator, steering wheel and know **what** they do
- The complexity has been **abstracted** away
  - **What** a car does (drive) is separated from **how** it works

Is all this really necessary?

- OOP means organizing our code differently to solve these issues

- On a large software project, there might be dozens of programmers, hundreds of classes, and millions of lines of code

How Complex is Software?

- Assignment 9 ~ 250 lines of code
- Average iPhone app ~ 50 thousand lines of code
- Google Chrome ~ 6.7 million lines of code
- Android ~ 12-15 million lines of code
- Software in new car ~ 100 million lines of code
- All Google services ~ 2 billion lines of code

http://www.visualcapitalist.com/millions-lines-of-code/

Advantages of Encapsulation

- Reduces errors
  - Prevents other programmers from directly changing attributes of objects
- Makes it easier to collaborate / work on large projects
  - Simplifies uses classes through public interface
- Code is easier to maintain / read

- End lecture

SEPARATING CLASSES INTO MULTIPLE FILES

Using Separate Files

- Common practice with object programming
- Use separate files for each class
- Use one (or multiple) files to "drive" your program (this file has **main** method)

Using Separate Files

- Class file – **`Vehicle.py`**
  - Define class, methods, variables as before

```
class Vehicle(object):
    def __init__(self):
        ...
```

Using Separate Files

- "Driver" file – **`Program.py`**
  - This file contains the **`main()`** function
  - **`main()`** contains the logic that runs the entire program
  - In **`main()`** you will create **`Vehicle`** objects
  - To create **`Vehicle`** objects, you need to tell Python what / where **`Vehicle`** is defined

Using Separate Files

- "Driver" file – **`Program.py`**
  - General Syntax

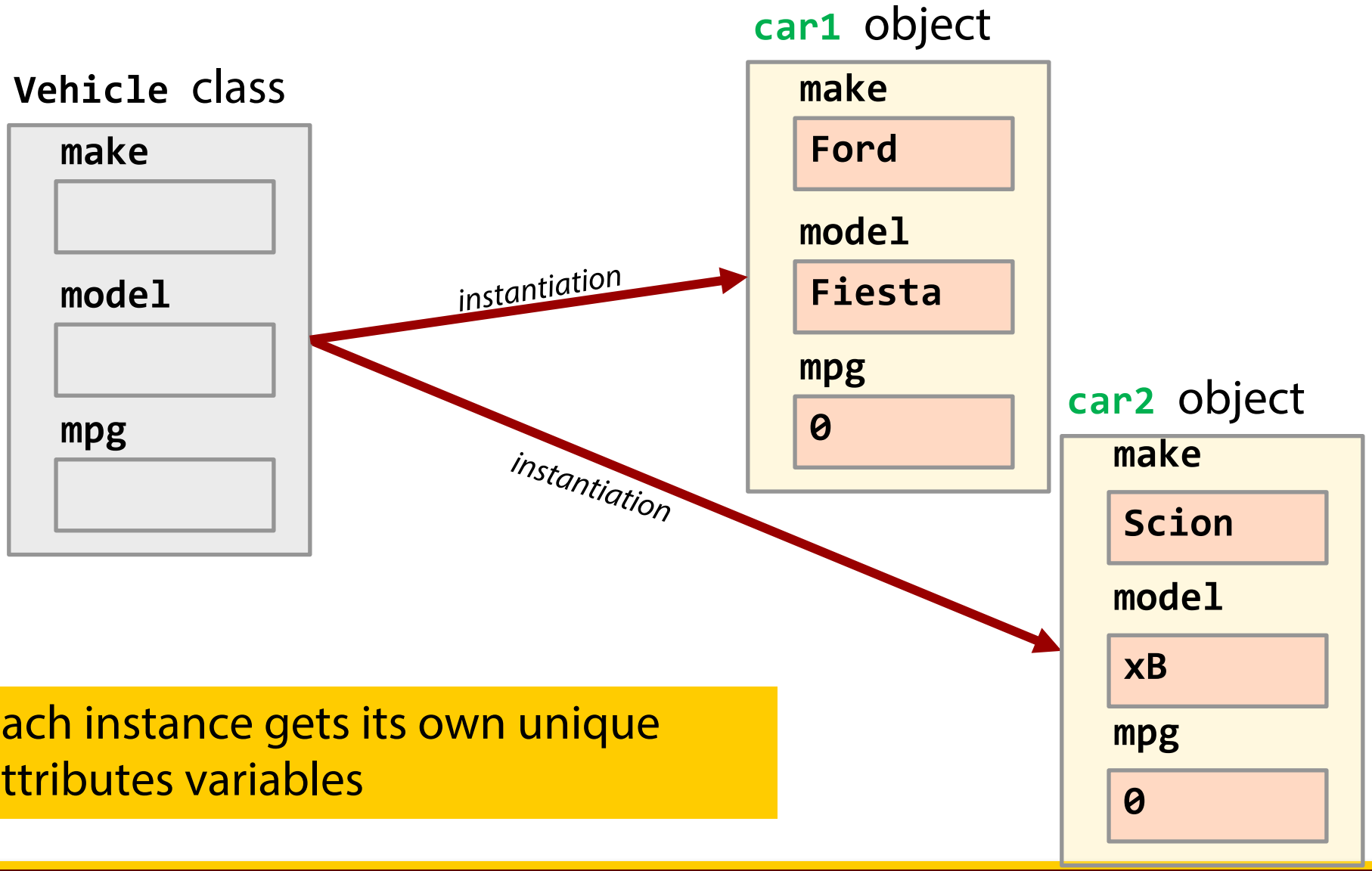
```
from fileName import className
```

```
from Vehicle import Vehicle
```

Instance Attributes

```
class Vehicle(object):
  def __init__(self, make, model):
    self.make = make
    self.model = model
    self.mpg = 0

def main():
    car1 = Vehicle("Ford", "Fiesta")
    car2 = Vehicle("Scion", "xB")
```

Instance Attributes



Each instance gets its own unique attributes variables

Shared Attributes

- What is we want similar object to be able to share some data?

- Example
  - Constants used by all objects of a class
  - Count of number of objects created

Class Variables

- Attributes are shared by all instances of a class
- Can be accessed by all objects of that class type
- Only 1 version of a class variable exists
  - Even if many objects exist
- These are sometimes called ***static variables***

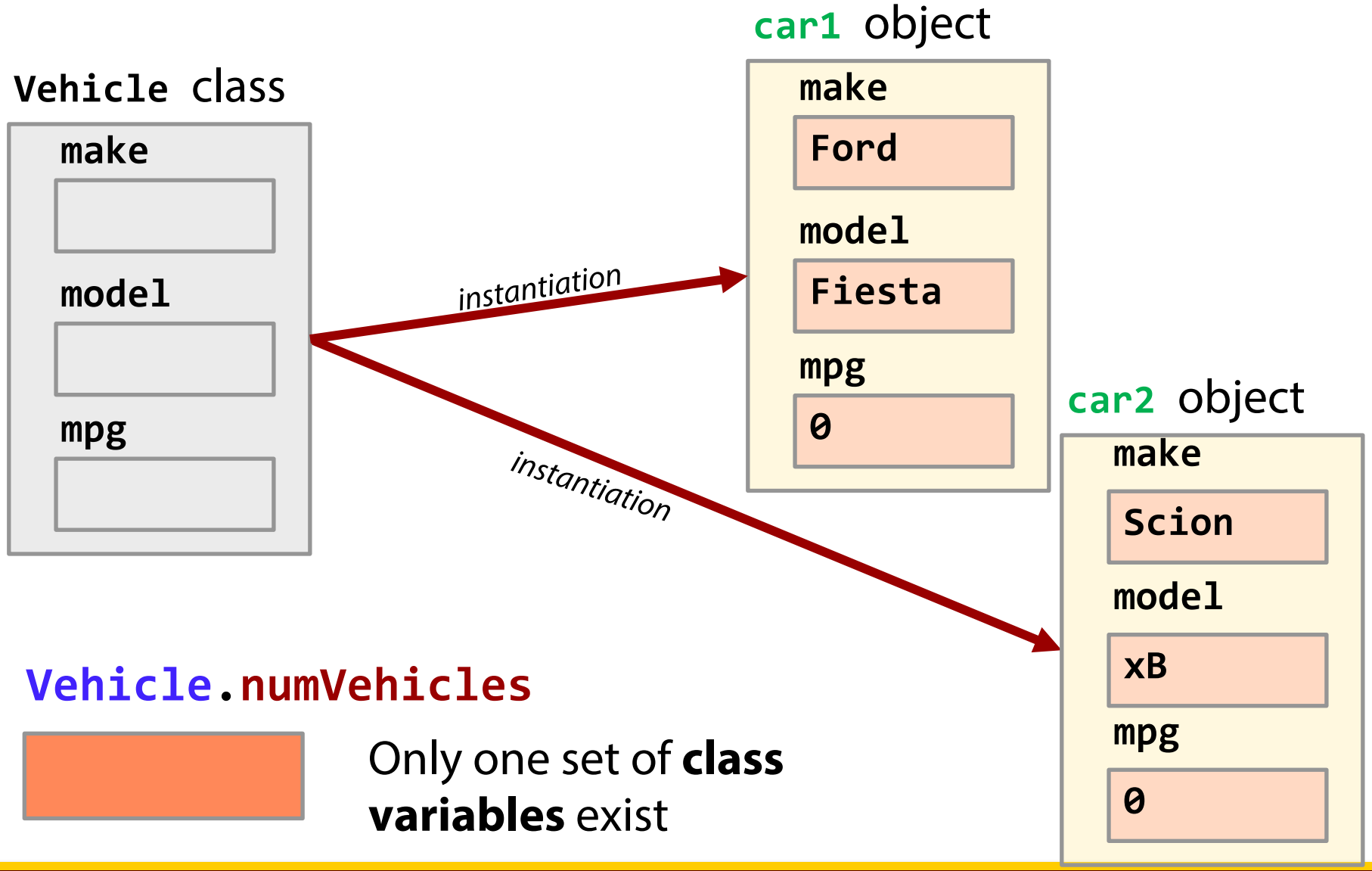
Class Variables

```
class Vehicle(object):
  numVehicles = 0
  def __init__(self, make, model):
    self.make = make
    self.model = model
    self.mpg = 0
    Vehicle.numVehicles += 1

def main():
  car1 = Vehicle("Ford", "Fiesta")
  car2 = Vehicle("Scion", "xB")
```

class variables are declared *outside* of **__init__**

class variables are accessed by **ClassName.variable**

Instance Attributes



Class Variables

```
def main():
  print("Total num is", Vehicle.numVehicles)
  car1 = Vehicle("Ford", "Fiesta")
  print("Total num is", Vehicle.numVehicles)
  car2 = Vehicle("Scion", "xB")
  print("Total num is", Vehicle.numVehicles)
```

class variables can be accessed before objects have been created

Output

```
Total vehicles is 0
Total vehicles is 1
Total vehicles is 2
```

Global Variables → Class Variables

- Global variables work within one file
- If you want a global variable for a class, make it a class variable
- Student Example:
  - The maximum number of courses is 6
  - **MAX_COURSES** is defined before the class and only available in in that file
  - You want it attached to the Student class and available in other files

Global Variable

```
MAX_COURSES = 6

class Student(object):
  def __init__(self, studentName, studentID):
    self.name = studentName
    self.idNumber = studentID
    self.courses = []

  def addCourse(self, course):
    if len(self.courses) < MAX_COURSES:
      self.courses.append(course)
```

Class Variable

```
class Student(object):
  MAX_COURSES = 6

  def __init__(self, studentName, studentID):
    self.name = studentName
    self.idNumber = studentID
    self.courses = []

  def addCourse(self, course):
    if len(self.courses) < Student.MAX_COURSES:
      self.courses.append(course)
```

Summary: 4 Types of Variables

- Local variables
- Global constants
- Instance variables
- Class variables

Local Variables

```
def main():
    msg = "hello world"
```

- Declared in a function (or method)
- These variable exist **only** during the function's execution
- Use them for temporary operations
- Remember **scope**

Global Variables

```
SPEED_OF_LIGHT = 30000000
def main():
```

- Declared outside of any function
- These variable exist **everywhere** in the file
- Use them for values that are constant and need to be accessed in multiple places

Instance (or Object) Variables

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

- Declared in a class
- Exist as long as the object exists
- Every object of the class has a **unique set** of variables

Class (or Static) variables

```
class Vehicle(object):
    numVehicles = 0
```

- Declared in a class
- Exist as long as the program is running
- Every object of the class **shares only one** copy of the variable

For reference only

APPENDIX

Static methods

- Static methods are declared in a class…
- But are invoked without using a specific object
- Instead use the class name

```
Vehicle.showCount()
```

For reference only

Static Methods

```
class Vehicle(object):
  numVehicles = 0

  def __init__(self, make, model):
    self.make = make
    self.model = model
    Vehicle.numVehicles += 1

  @staticmethod
  def status():
    print("Total number of Vehicles ", Vehicle.numVehicles)

def main():
  car1 = Vehicle("Ford", "Fiesta")
  Vehicle.status()
```

For reference only

Class Parts

- Attributes
  - Instance variables
    - Each instance of the class has its own values for the attributes
  - Class (or *static*) variables
    - If a class is like a blueprint, then a class attribute is like a Post-it note stuck to the blueprint
- Methods
  - Instance methods
    - Special ones – constructor (`__init__`) and print (`__str__`)
  - Static methods *(reference only)*
    - Use `@staticmethod` decorator