Software objects in Python

Summarizing Python so far…

- Variables and data
  - Single data values (like integers and floats)
  - Collections (like lists and dictionaries)

- Control structures
  - Conditionals (like if-else)
  - Loops

- Functions
  - With different kinds of input

Summarizing Python so far…

- File manipulation
  - Inputting data from different file types
  - Outputting data into a file

- Exceptions
  - How to prevent and anticipate them

So far…

- We’ve been using variables types defined by Python (or in added modules)
  - Like strings, lists, dictionaries, etc.

- Like the grade book example, they’re not always a perfect fit

- What if we could create our own type of variable?
  - Like Animal, Car, Student

Object oriented programming

- Sometimes abbreviated as OOP

- So far, we’ve been programming procedurally
  - Data and functions are separate
  - Very few metaphors or representations of things in code

- OOP is used in many commercial products – even Python itself

- Basic building block is the software object
  - We’ll just call it an object

Objects we’ve already used

- So far we’ve *used* containers – variables that contain other variables
    - For example, a list can contain several integers
- These containers have attributes
    - A dictionary has a size
- These containers have functionality
    - A string can be made lower case
- These are all really just **objects**

Seeing things as objects

- Consider this string

```
name = "Bob Jarvis"
```

- What are the attributes (or parts) of the string?
  - The characters that make up the string

```
"Bob Jarvis"
```

- What is its functionality (or things it can do)?
  - Uppercase it and display it

```
print(name.upper())
```

Representing things in code

- OOP lets you represent real-life things as software objects
  - Concrete items: teacher, furniture, book
  - Abstract items: checking account, course, word

- Real life objects have features and functionality
  - The car is blue
  - The car can accelerate

- Software objects can have features and functionality too
  - Sometimes called attributes and abilities
  - Sometimes called variables and functions

Consider a vehicle

- What attributes does a vehicle have?
  - Think “facts about a vehicle”
  - This vehicle is __________
  - This vehicle has _________

- What behaviors can a vehicle perform?
  - Think “actions a vehicle can perform”
  - This vehicle can _________

About a vehicle

- Attributes
  - Number of wheels
  - Color
  - Make
  - Model
  - Year
  - License plate number
  - Number of passengers

- Behaviors
  - Turn right
  - Turn left
  - Accelerate
  - Decelerate
  - Honk horn
  - Turn on AC
  - Turn off AC

An example vehicle

- Attributes
  - Number of wheels = 4
  - Color = brown
  - Year = 2006
  - Make = Disney
  - Model = Mater

- Abilities
  - Accelerate (forward)
  - Accelerate (backwards)
  - Tow

Another example vehicle

- Attributes
  - Number of wheels = 4
  - Color = black
  - Year = 2005
  - Make = Wayne Enterprises
  - Model = Tumbler

- Abilities
  - Accelerate (forward)
  - Fire weapons
  - Eject Batman

About objects

- Objects are created (instantiated) from a definition called a class

- An object is not a class – it is the realization of a class

- A programmer can create many objects from the same class

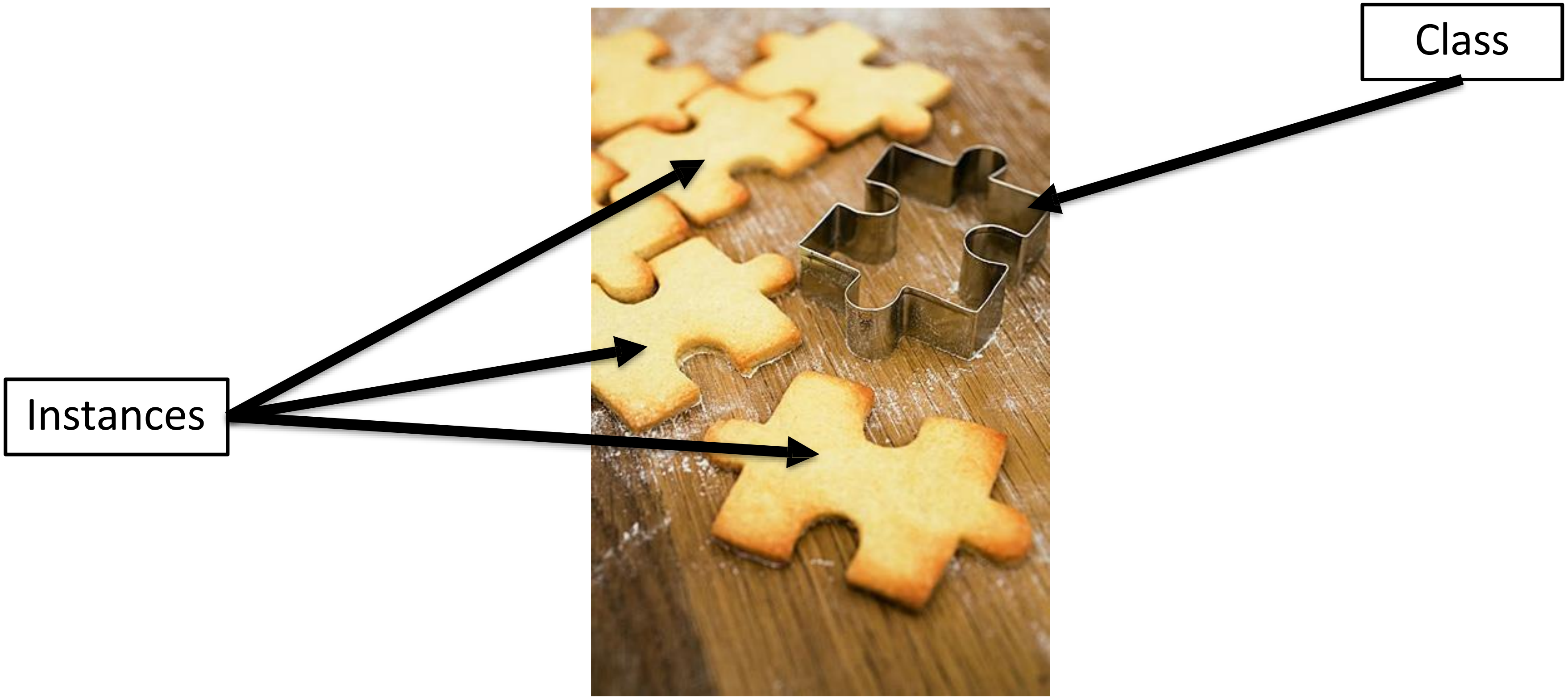
- Each object (or instance) created from the same class will have similar structure

About classes

- Classes are like blueprints
- A class is not an object – it is the design for an object
- Classes are code that define attributes and functions

Classes and instances

- Think of a class as a cookie cutter



Class

Instances

- Objects (or instances) are the cookies

A very loose analogy

- Think about working with functions
  - First, we had to define the function
  - Only then can we call the function

- Similarly, when working with classes
  - First, we need to define a class
  - Then we can instantiate an object of that class

Designing a class

- When designing an algorithm ask, “Does it make sense to use objects here?”

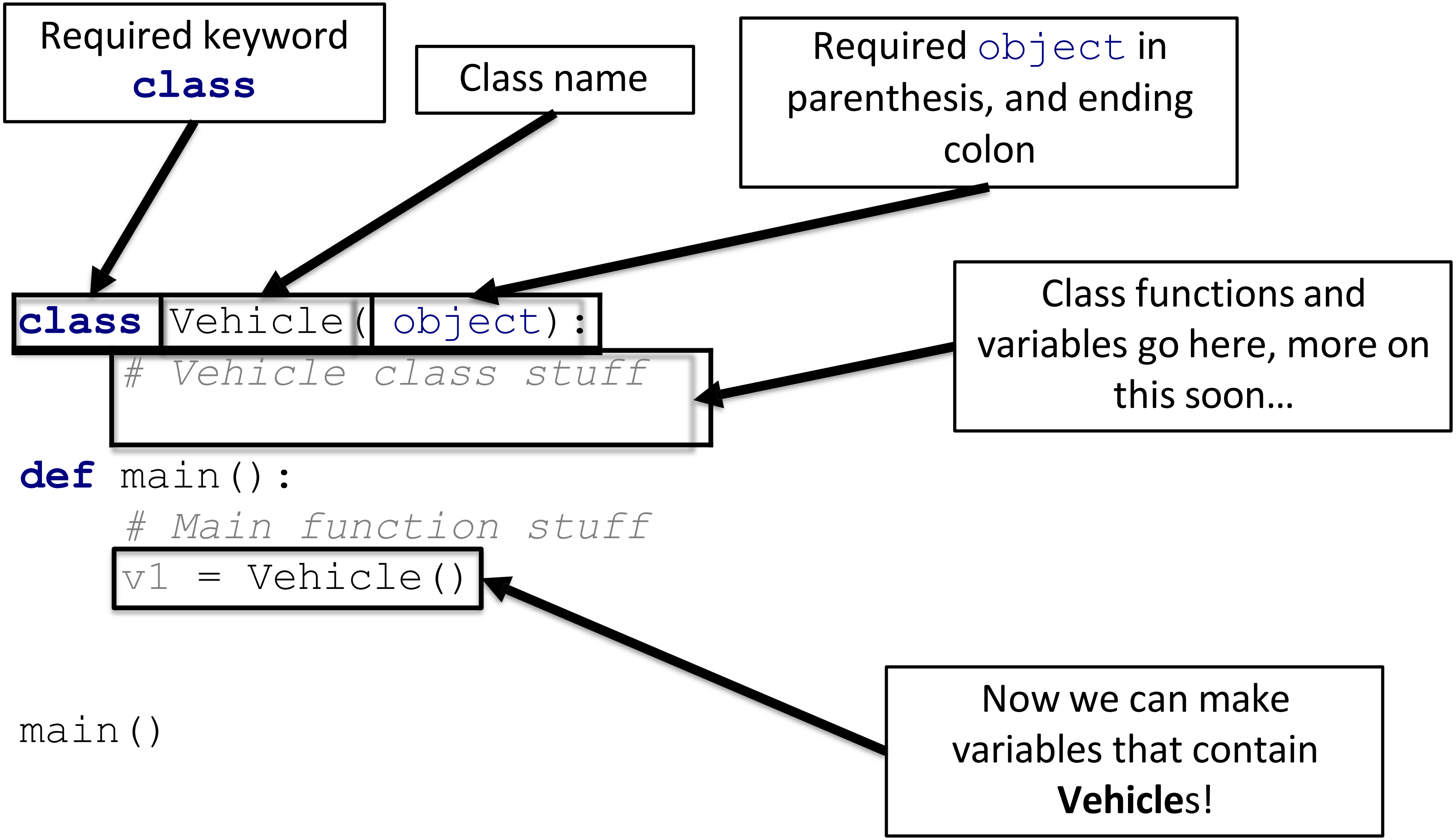
- If so, consider which attributes and functions a class will need
  - Attributes are variables ***every*** object will store
  - Methods are functions ***every*** object can perform

Defining a class

- By convention name all classes using **<u>U</u>**pper**<u>C</u>**amel**<u>C</u>**ase

- Classes are defined “globally”
  - They are aligned to the far left
  - They exist outside of and separate from main or other functions

Defining a class in code



Required keyword **class**

Class name

Required object in parenthesis, and ending colon

```
class Vehicle(object):
    # Vehicle class stuff

def main():
    # Main function stuff
    v1 = Vehicle()


main()
```

Class functions and variables go here, more on this soon…

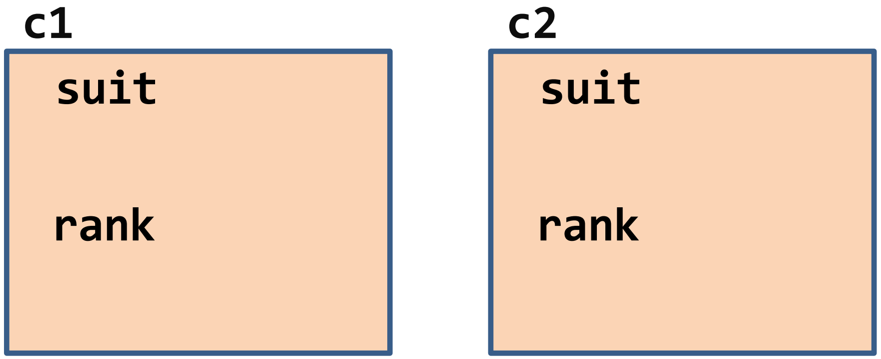
Now we can make variables that contain **Vehicle**s!

Attributes

- Just like we can store other variables inside a list or dictionary variable, we can also store other variables inside an object

- Variables contained inside an object are called **attributes**
  - Some other languages call these *instance variables* or *member variables*

- For example, given a PlayingCard class that describes each card with a `suit` and a `rank` we can have 2 variables, `c1` and `c2`



Attributes and the constructor

- We use a **constructor** to define what attributes will exist inside a object

- A constructor is **method** that is used to create an instance of an object
  - A **method** is basically a **function** that is in a class (more later)
  - Just be warned: I may use function and method interchangeably

- Constructors have **no return value**

- Constructors are **called automatically** when you create an object

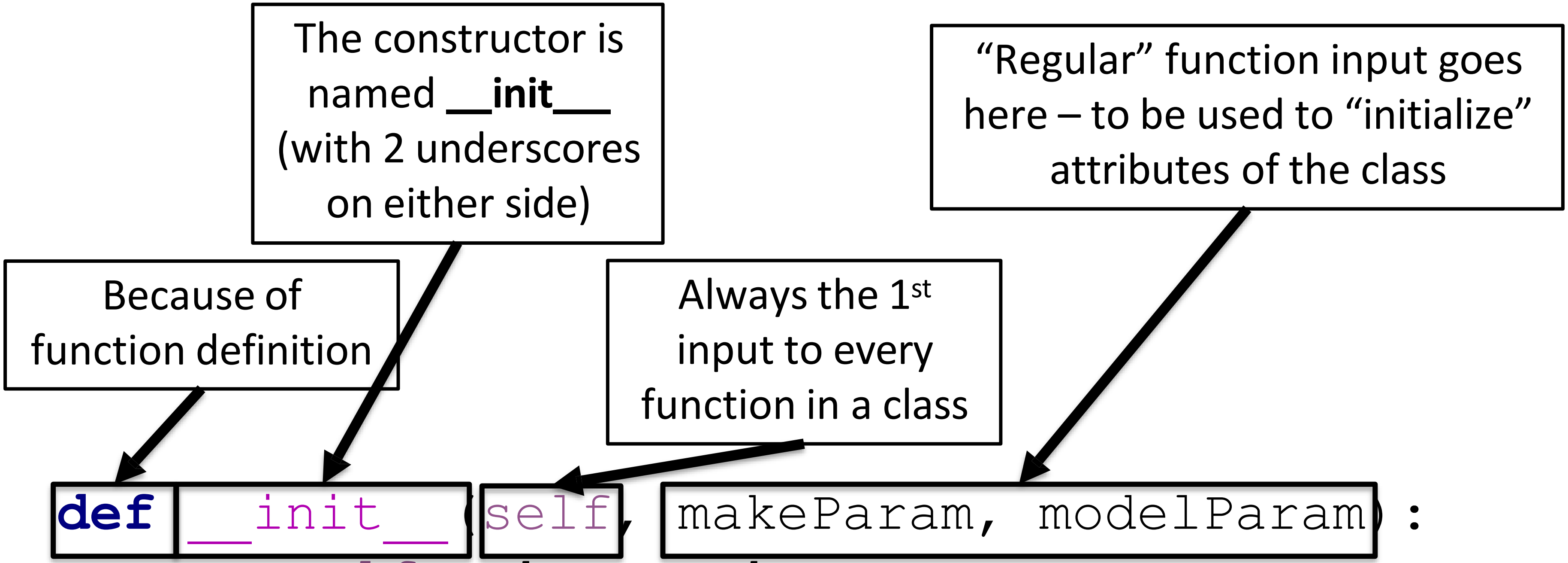
A class with a constructor

```
class Vehicle(object):
    def __init__(self, makeParam, modelParam):
        self.make = makeParam
        self.model = modelParam
        self.year = 2017


def main():
    v1 = Vehicle("Disney", "Mater")
    v2 = Vehicle("Wayne Enterprises", "Tumbler")


main()
```

Constructor syntax



The constructor is named **__init__** (with 2 underscores on either side)

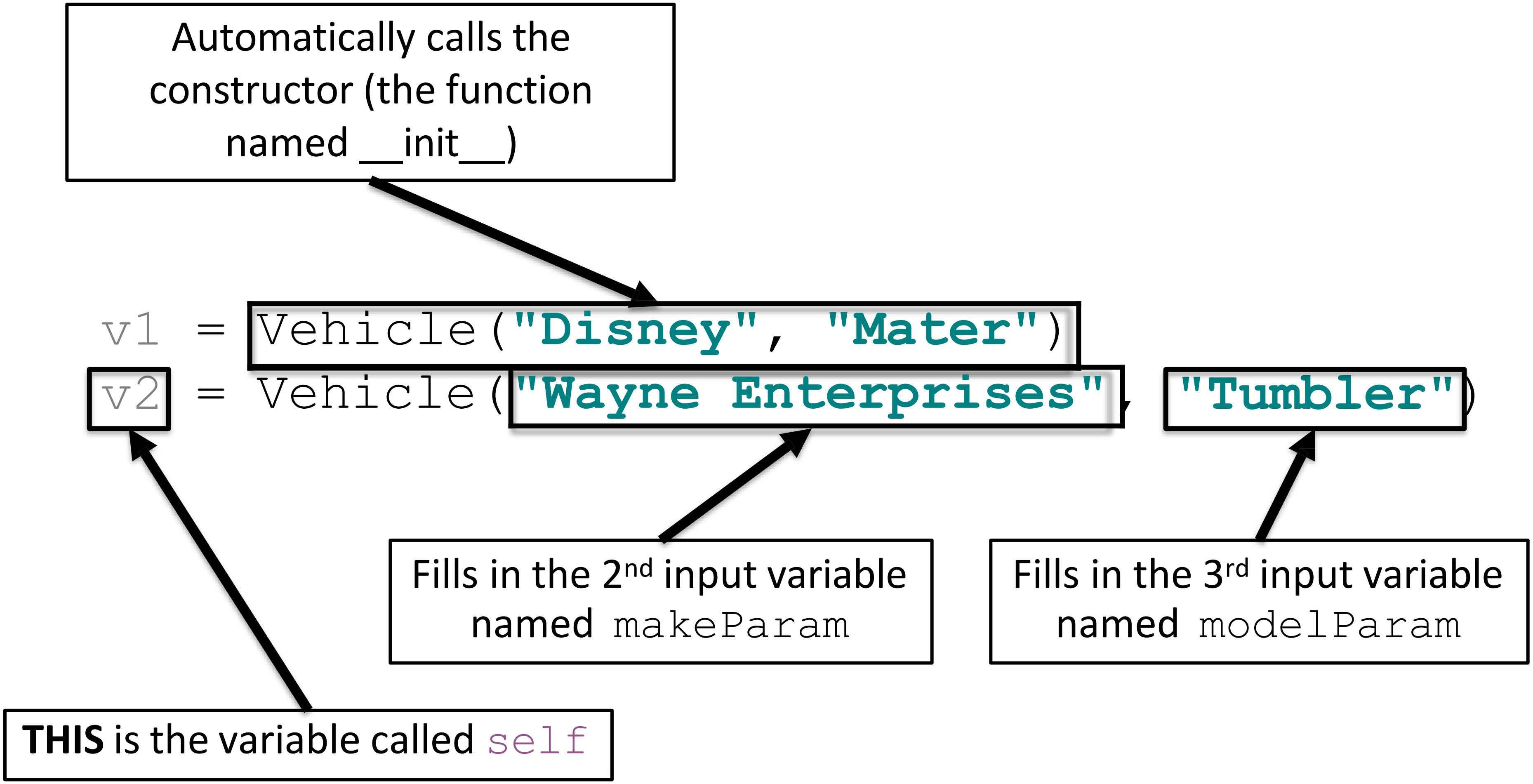
"Regular" function input goes here – to be used to "initialize" attributes of the class

Because of function definition

Always the 1st input to every function in a class

```
def __init__(self, makeParam, modelParam):
    self.make = makeParam
    self.model = modelParam
    self.year = 2017
```

Using the constructor



Automatically calls the constructor (the function named __init__)

```
v1 = Vehicle("Disney", "Mater")
v2 = Vehicle("Wayne Enterprises", "Tumbler")
```

Fills in the 2nd input variable named `makeParam`

Fills in the 3rd input variable named `modelParam`

THIS is the variable called `self`

Imagine a conversation

- Mater:
  - "**I** knew it! **I** knowed **I** made a good choice!"
- Bruce Wayne:
  - "**I**'m Batman!"
- To whom does **I** refer?
  - It depends on who is speaking
  - The word "**I**" is a way for people to refer to themselves

About self

- **self** is a way for an object to refer to itself

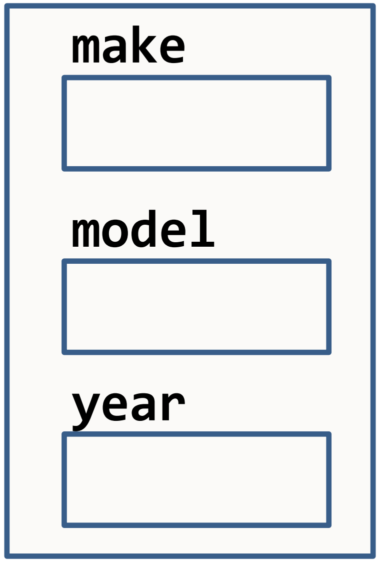
- **self** always refers to a unique object that called a method (or was created by a constructor)

- Attributes are part of the object so it must be preceded by **self**
  - Attributes are stored inside of the object, they exist when the constructor ends
  - Unlike regular local variables in a function they persist after the function is over

Classes and objects visualized

- When we define a class, we describe the blueprint for what objects will look like

`Vehicle` class



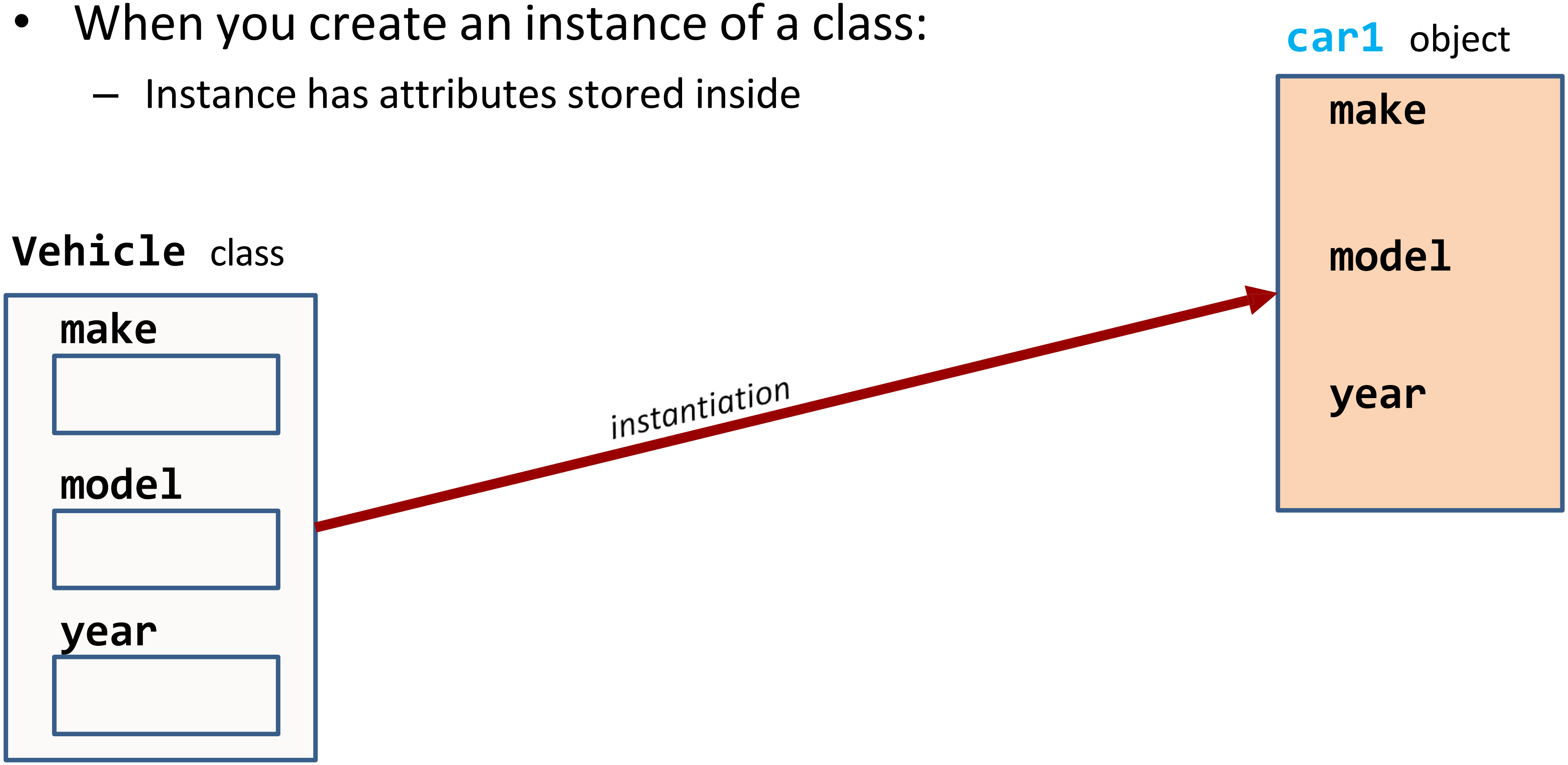
`make`

`model`

`year`

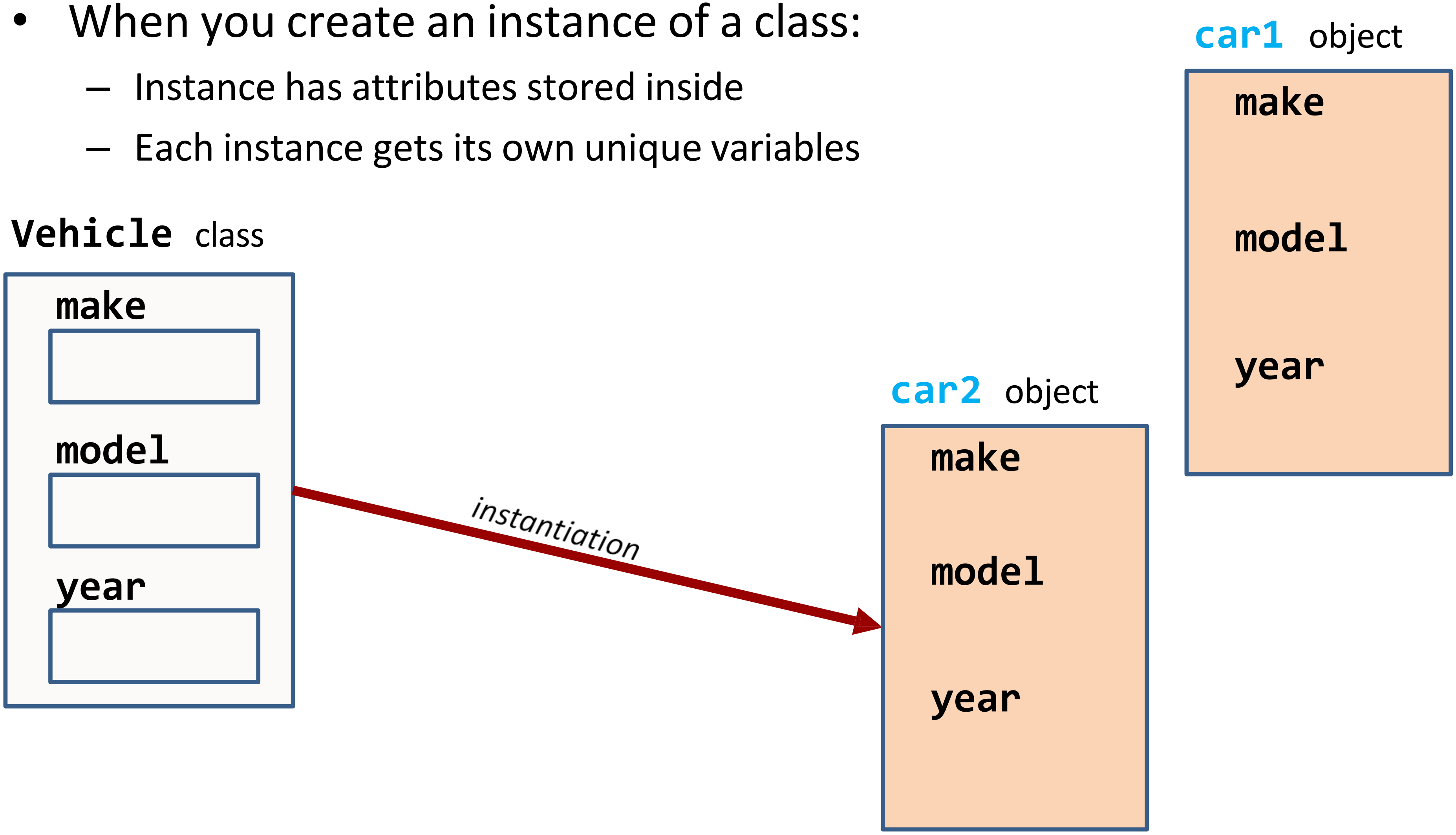
Classes and objects visualized

- When you create an instance of a class:
  - Instance has attributes stored inside



Classes and objects visualized

- When you create an instance of a class:
  - Instance has attributes stored inside
  - Each instance gets its own unique variables



Putting it all together

```
class Vehicle(object):
    def __init__(self, makeParam, modelParam):
        self.make = makeParam
        self.model = modelParam
        self.year = 2017


def main():
    v1 = Vehicle("Disney", "Mater")
    v2 = Vehicle("Chevy", "Nova")
    print(v1.model)
    print(v2.model)


main()
```

```
Mater
Nova
```

`__init__(self):`

- A constructor is a **method** that is used to create an instance of an object

- A constructor defines what attributes will exist inside a object

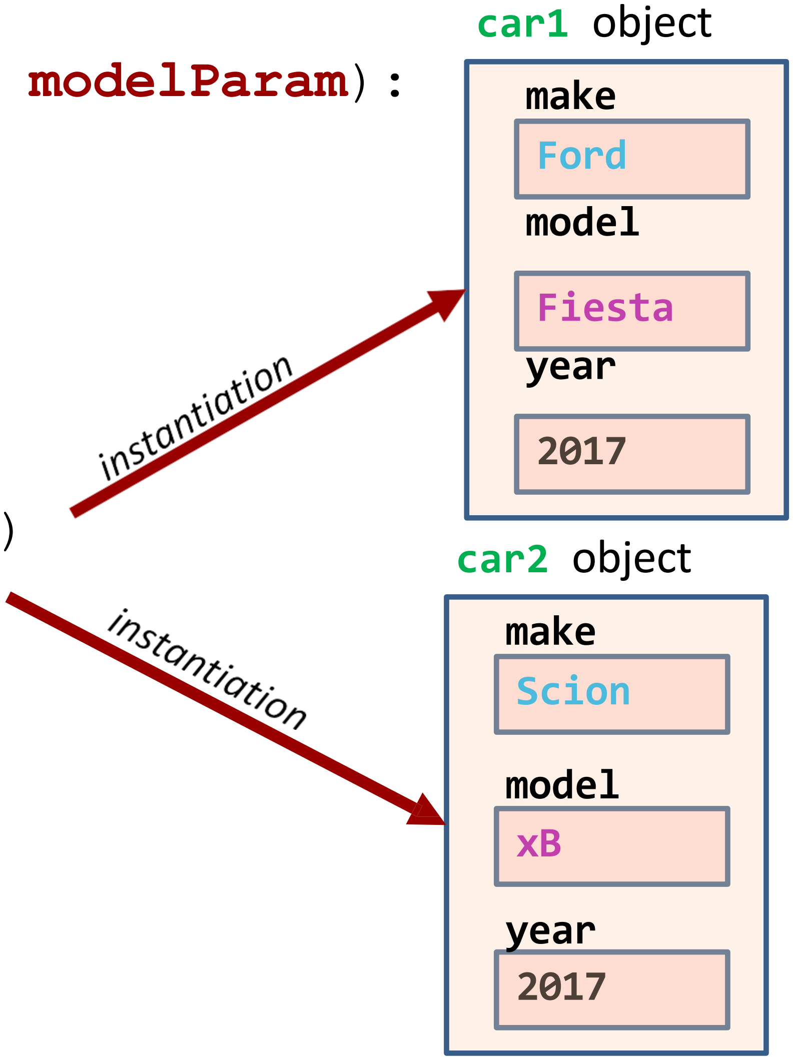
- Constructors are called **automatically** when you create an object

Attributes and constructors

```
class Vehicle(object):
    def __init__(self, makeParam, modelParam):
        self.make = makeParam
        self.model = modelParam
        self.year = 2017


def main():
    v1 = Vehicle("Ford", "Fiesta")
    v2 = Vehicle("Scion", "xB")

main()
```



Methods

- Classes can have methods (or behaviors)

- You can think of methods as functions associated with an object

- Methods are defined inside of the class, this is what makes them different from regular functions

Functions and Methods

- **Functions** are free-standing blocks of code that we can use

```
print("Hello world")
drink = input("What coffee do you want?")
```

- **Methods** are basically functions that are part of an `object`

```
word = word.upper()
numList = line.split(",")
```

Methods are like Other Functions

- Output
  - Can return a value
- Input
  - Can take input parameters
- Contents of the method are in a block (indented)

Methods

- Methods are part of the object just like attributes

- Methods can access the attributes defined in the constructor using `self`

Defining a method

- Syntax:

  ```
  def methodName(self):
  ```

  - `methodName` is the name of the method

- Every method special first parameter is `self`
  - Provides a way for a method to refer to the object itself

Calling a Method

- Syntax

  ```
  varName.methodName()
  ```

  - `varName` is the name of the variable object *(not the name of the class)*
  - `methodName` is the name of the method

Calling a Method

- Recall:
  - Every list object has a **sort** method

```
myList = ["cat", "yeti"]
myList.sort()
```

- Once we have created / instantiated an object, we can call methods

Calling a method

- Example:

```
class Vehicle(object):
    def startEngine(self):
        #...




def main():
    v1 = Vehicle()
    v1.startEngine()
```

Calling a method

- Example:

```
class Vehicle(object):
    def startEngine(self):
        #...



def main():
    v1 = Vehicle()
    v1.startEngine()
```

self refers to the object that called the method

Method input and output

- As with functions, methods can receive input parameters and return output values

- Syntax

```
def methodName(self, param1, param2, ...):
  ...
  return someVariable
```

Example: method input and output

```
class Vehicle(object):
    # Constructor goes here!

    def calcTripCost(self, miles):
        totalCost = 0
        # perform some calculations
        return totalCost

def main():
    v1 = Vehicle()
    cost = v1.calcTripCost(100)
```

Example: method input and output

```
class Vehicle(object):
    # Constructor goes here!

    def calcTripCost(self, miles):
        totalCost = 0
        # perform some calculations
        return totalCost

def main():
    v1 = Vehicle()
    cost = v1.calcTripCost(100)
```

Example: method input and output

```
class Vehicle(object):
    # Constructor goes here!

    def calcTripCost(self, miles):
        totalCost = 0
        # perform some calculations
        return totalCost

def main():
    v1 = Vehicle()
    cost = v1.calcTripCost(100)
```

`self` refers to the object that called the method

Example: method input and output

```
class Vehicle(object):
    # Constructor goes here!

    def calcTripCost(self, miles):
        totalCost = 0
        # perform some calculations
        return totalCost

def main():
    v1 = Vehicle()
    cost = v1.calcTripCost(100)
```

Rest of the parameters go in order

Example: method input and output

```
class Vehicle(object):
    # Constructor goes here!

    def calcTripCost(self, miles):
        totalCost = 0
        # perform some calculations
        return totalCost

def main():
    v1 = Vehicle()
    cost = v1.calcTripCost(100)
```

Return values just like we did with functions

__str__()

- Special method you can create that can be used to display the attributes of an object

- This called automatically whenever you attempt to “print” an instance
  - Pass instance as argument to print function

__str__()

- Syntax

```
def __str__(self):
    return "This will have stuff to display"
```

- **Important:** This method ***returns*** a string; it does not print directly

__str__()

- Example

```
class Vehicle(object):
    # Other stuff appears here...
    def __str__(self):
        msg = "Make: " + self.make
        msg += "\nModel: " + self.model
        return msg
```

__str__()

- Now we can use it in main!

```
def main():
    v = Vehicle("Ariel", "Atom")
    print(v)
```

```
Make: Ariel
Model: Atom
```

Consider a car

- We use brake pedal, accelerator pedal, steering wheel – we know **what** they do

- We do <u>not</u> see mechanical details of **how** they do their jobs

- The complexity of how a car works has been abstracted away
  - **What** a car does (drive) is separate from **how** it works (engine, etc).

The BFD

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

- OOP means organizing our code differently to solve these issues

Before OOP programmers considered 2 roles

- User
  - Interacts with the program (through keyboard, mouse, etc.)
  - Doesn't need to know anything about the code

- Programmer, class user (you)
  - Writes overall program logic, main()

With OOP programmers consider 3 roles

- User
  - Interacts with the program (through keyboard, mouse, etc.)
  - Doesn't need to know anything about the code

- Programmer, class user (you)
  - Writes overall program logic, main()
  - **Uses classes**

- Programmer, class designer (also you, or another programmer)
  - **Creates class definition** to be used by other programmers
  - Structures classes to be updated with little impact on everyone else

Encapsulation

- Encapsulation means knowing **what** a class does without needing to know **how** it does it

- Ex: How does a dictionary actually work?
  - To us, it isn't important
  - We just need to know what a dictionary can do

Information Hiding

- Class design defines a method / class so it can be used without knowing details

- Programmer using a class / method need not know details of implementation
  - Only needs to know *what* the method does

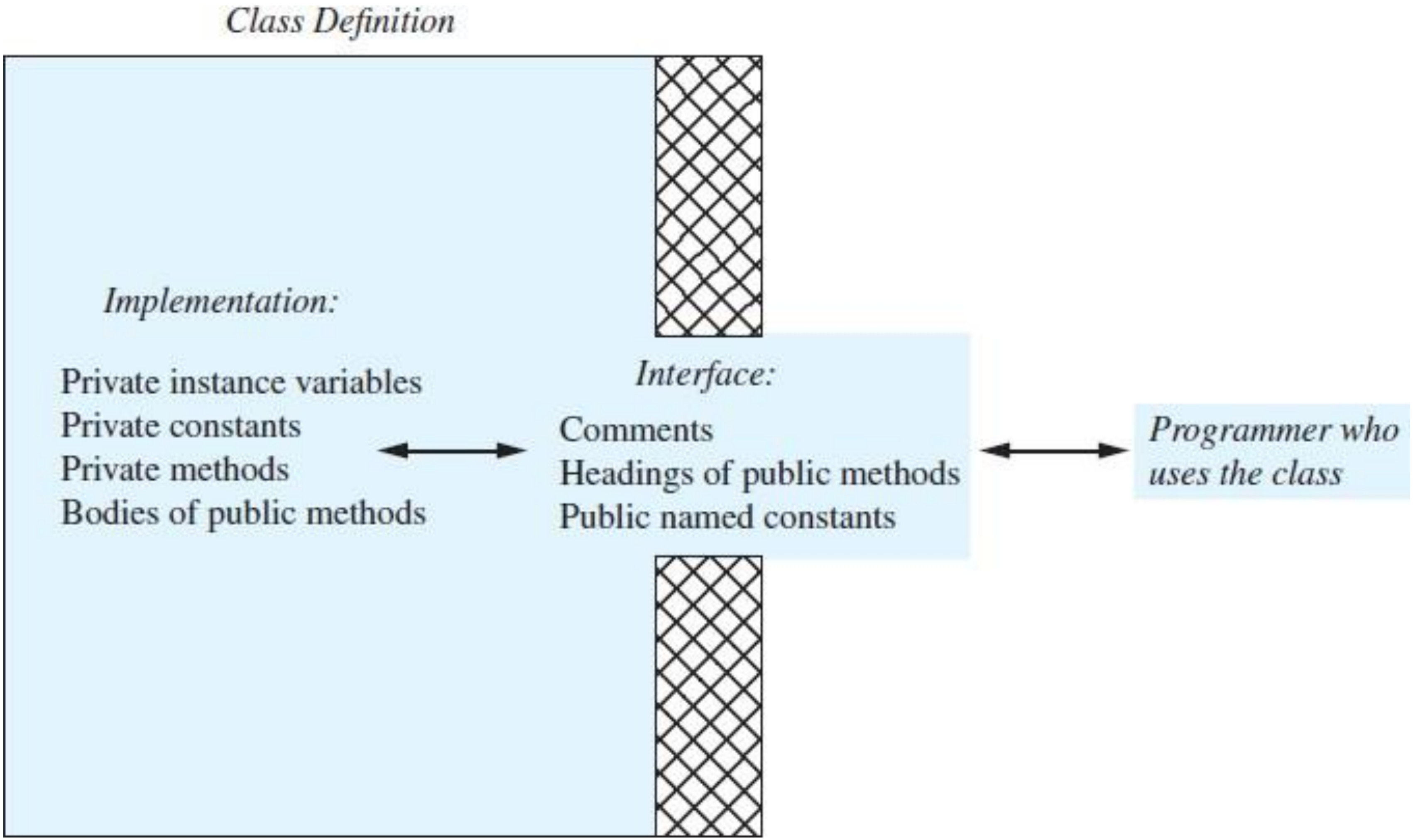
- Method design should separate ***what*** from ***how***

Encapsulation Separates Classes into 2 Parts

- A class interface
  - Tells **what** the class does (not how)
  - Gives **headings** from public methods (the ones we can use) and comments about them

- A class implementation
  - Contains private attributes (the ones we can’t see)
  - Includes **definitions** (details) of public and private methods

Encapsulation in pictures



Class Definition

Implementation:

Private instance variables
Private constants
Private methods
Bodies of public methods

Interface:

Comments
Headings of public methods
Public named constants

Programmer who uses the class

Advantages of Encapsulation

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

Public attributes

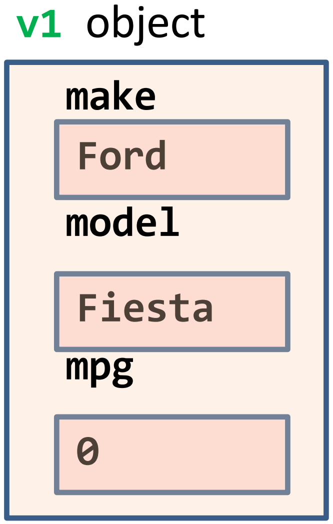
- By default, all of an object's attributes and methods are **public**

- They can be directly accessed or invoked by a class user (e.g. in **main()**)

Example

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

def main():
    v1 = Vehicle("Ford", "Fiesta")
    print("The MPG is" + v1.mpg)
```


v1 object

make

Ford

model

Fiesta

mpg

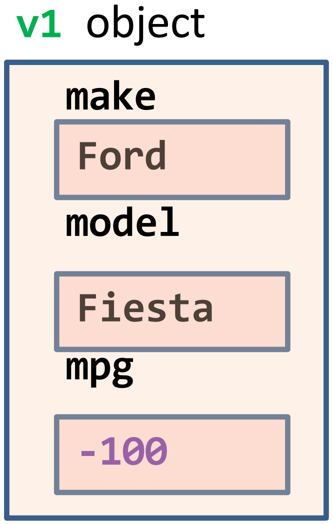
0

```
The MPG is 0
```

Example

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

def main():
    v1 = Vehicle("Ford", "Fiesta")
    print("The MPG is" + v1.mpg)
    v1.mpg = -100
```



v1 object

make

Ford

model

Fiesta

mpg

-100

Should this be allowed?

Private attributes

- To create a private attribute, begin the attribute name with **two underscores**

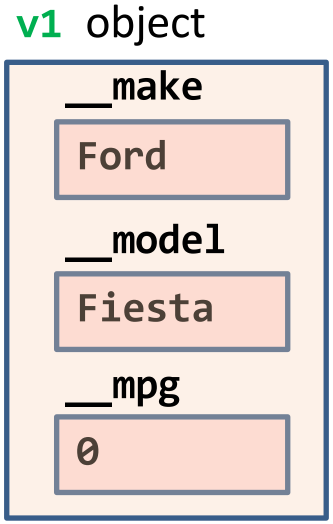
```
class Vehicle(object):
    def __init__(self, make, model):
        self.__make = make
        self.__model = model
        self.__mpg = 0
```

- **Private attributes** can only be directly accessed by the objects

Example

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


def main():
    v1 = Vehicle("Ford", "Fiesta")
```



v1 object

__make

Ford

__model

Fiesta

__mpg

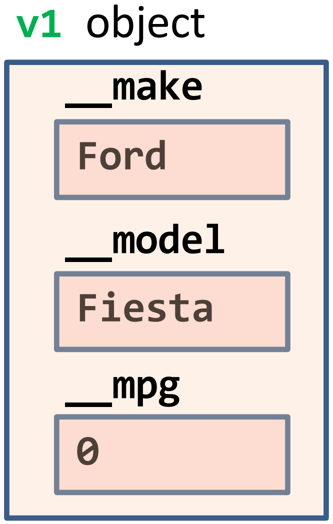
0

Same as before

Example

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


def main():
    v1 = Vehicle("Ford", "Fiesta")
    print(v1.__mpg)
```



v1 object

__make

Ford

__model

Fiesta

__mpg

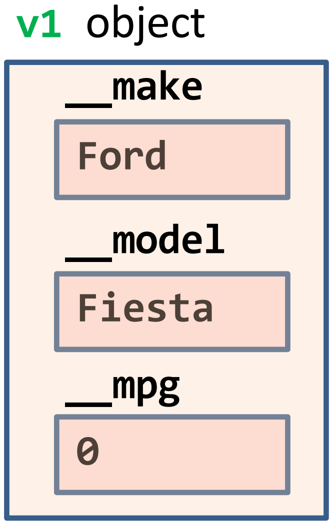
0

Error! main() can't directly access **mpg**

Example

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


def main():
    v1 = Vehicle("Ford", "Fiesta")
    print(v1.__mpg)
    v1.__mpg = -100
```



v1 object

__make

Ford

__model

Fiesta

__mpg

0

Error! main() can't directly change **mpg**

Private attributes

- Data is now **`private`**…
  - But we can't access it or change it at all

- We would like a way to **control** access and modification

- We can allow indirect access to attributes and often impose some sort of restrictions on that access (like error checking)

Using getters

- One type of access method is a **get method**
  - Provides read access to a private attribute
  - Referred to as an **accessor** or **getter** method

- Syntax

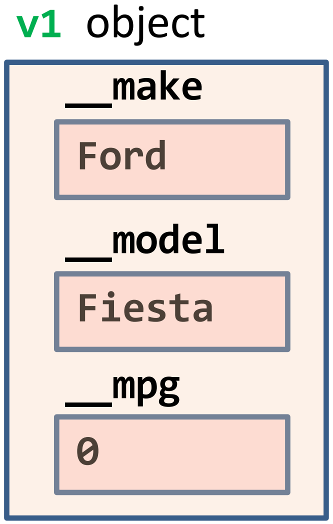
```
getXXXX(self)
```

  - Always **returns** the value of the attribute

Using getters

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

    def getMPG(self):
        return self.__mpg
```



v1 object

__make

Ford

__model

Fiesta

__mpg

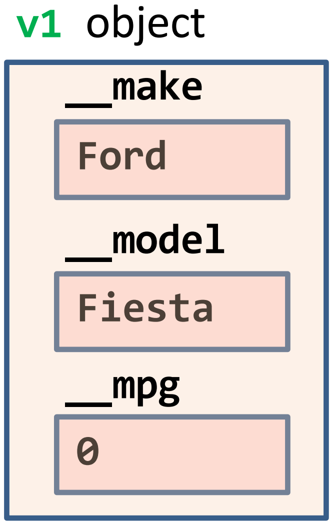
0

Using getters

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

    def getMPG(self):
        return self.__mpg

def main():
    v1 = Vehicle("Ford", "Fiesta")
    print(v1.getMPG())
```



```
0
```

Using setters

- To allow controlled changes to an attribute, use a **set method**
  - Modifies the value of a private attribute
  - Referred to as a **mutator** or **setter** method

- Syntax

  ```
  setXXXX(self, newXXXX)
  ```

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

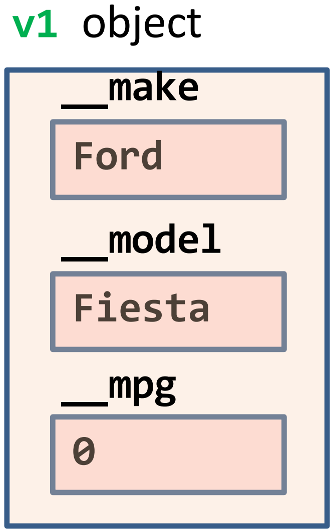
Using setters

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

    def getMPG(self):
        return self.__mpg

    def setMPG(self, newMPG):
        if newMPG > 0:
            self.__mpg = newMPG

def main():
    v1 = Vehicle("Ford", "Fiesta")
    print(v1.getMPG())
```


v1 object

__make

Ford

__model

Fiesta

__mpg

0

```
0
```

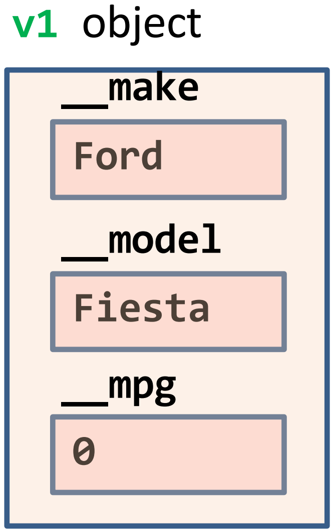
Using setters

```python
class Vehicle(object):
    def __init__(self, make, model):
        self.__make = make
        self.__model = model
        self.__mpg = 0

    def getMPG(self):
        return self.__mpg

    def setMPG(self, newMPG):
        if newMPG > 0:
            self.__mpg = newMPG

def main():
    v1 = Vehicle("Ford", "Fiesta")
    v1.setMPG(-18)
    print(v1.getMPG())
```



v1 object

__make

Ford

__model

Fiesta

__mpg

0

```
0
```

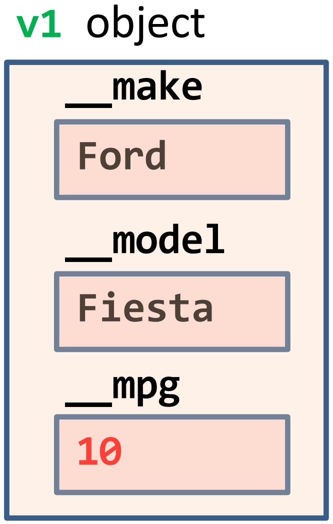
Using setters

```python
class Vehicle(object):
    def __init__(self, make, model):
        self.__make = make
        self.__model = model
        self.__mpg = 0

    def getMPG(self):
        return self.__mpg

    def setMPG(self, newMPG):
        if newMPG > 0:
            self.__mpg = newMPG

def main():
    v1 = Vehicle("Ford", "Fiesta")
    v1.setMPG(10)
    print(v1.getMPG())
```



v1 object

__make

Ford

__model

Fiesta

__mpg

10

```
10
```

Guidelines for Implementing Privacy in a Class

- What should be public?
  - **get** and **set** methods for each instance variable
  - methods the user needs to use your class

- What should be private?
  - attributes / instance variables
  - any methods that the user shouldn't access *(all methods in our course will be public*)