The Object-Oriented (OO) Paradigm

Pillar: Computer Programming

As discussed in lessons early in the curriculum, three paradigms of programming languages have emerged over the years: the imperative paradigm, the functional paradigm, and the logical paradigm. A language is classified as belonging to a particular paradigm based on the programming features it supports. In addition, during the past decade or so these paradigms have been extended to include object-oriented features. Some computer scientists view object-oriented programming as a fourth paradigm. Others prefer to view it as an extension to the imperative, functional, and logical paradigms, in that object-oriented constructs and behaviors are often viewed as higher-level organizational attributes that can be incorporated into each of the three basic paradigms, rather than as a separate programming paradigm unto itself.

Object-oriented concepts have revolutionized programming languages. The vast majority of widely used programming languages are now object-oriented. In fact, they are, by far, the most popular type of programming languages. Python, Java, and C++ are object-oriented, imperative languages. Specifically, Python is an imperative, interpreted language that can *optionally* be object-oriented. That is, non-object-oriented programs can be written in Python if desired. In a sense, this makes Python quite powerful (like a Swiss army knife of programming).

The **object-oriented paradigm** is an elegant and clean way to conceptually think about computer programming. When used properly, it can produce programs that are more robust, less likely to have errors, and are easy for others to understand and modify. Specifically, the object-oriented approach adds the concepts of objects and messages to the paradigms listed above. We don't think of programs as procedures or lists of instructions to be executed in order from beginning to end; rather, we think of them as modeling a collection of objects interacting with each other.

Essentially, programs and the data on which they act are viewed as objects. In order to perform a task, an object must receive a message indicating that work needs to be done. Object-oriented languages are extremely useful for writing complex programs; for example, programs that support mouse-based, graphical user interfaces. Object-oriented programming helps in the construction of software systems by enabling large, complex systems to be subdivided into isolated functional units with well-defined external interfaces to other system components. There are many other distinguishing characteristics of object-oriented programs, including inheritance, polymorphism, and data encapsulation. Some of these will be discussed in this lesson, while others will be covered in later lessons.

State and behavior...the basic idea

We live in a world in which objects exist all around us. In fact, we interact with objects all the time. For example, the author of this document interacted with a keyboard and mouse while writing this sentence! The author's brain (an object) somehow sent a message to hands and fingers (all objects) to make contact with keyboard keys (again, all objects). The keyboard (an object) somehow sent a message to the computer (an object made up of many other objects) to interpret key presses as characters to be placed in this document (an object) and displayed on the screen (yet another object).

Fundamentally, an object is a thing. In object-oriented programming, the objects model things in our problem domain. Objects have properties (or attributes) that, in a sense, define them. They capture the properties or characteristics of an object. For example, a *person* object has many attributes, some of which include sex, name, age, height, hair color, eye color, and so on. These are the things that a person can *be*.

Definition: *The collection of attributes that make up an object are called its* ***state****.*

Objects can also *do* things. That is, they can have behaviors. In object-oriented programming, these behaviors are implemented by program modules (e.g., methods, procedures, functions, etc) that contain the instructions required to model the behavior in computer software. For example, a *person* object can eat, sleep, talk, walk, and so on.

Definition: *The collection of actions that an object can do are called its* ***behavior****.*

Collectively, **state** and **behavior** define an object. When you begin to adopt this way of thinking, you can begin to see many things in our world as objects with attributes and behaviors interacting with other objects.

Objects, classes, and instances

Definition: *An* ***object*** *represents a specific thing from the real world with defined attributes.*

For example, "the white truck down there parked by the road" is an object. It is a truck that could ostensibly be observed on the road. In fact, it could be a white 2016 4x4 Dodge Ram 1500 with 450 miles on it.

Clearly, there exist other trucks in the world. In fact, there may even be other trucks parked by the road next to the one just described. One could say, then, that the generic term *truck* could represent all kinds of truck objects. Trucks are all basically different versions of the same thing. That is, they all behave the same and have the same set of attributes; however, the values of those attributes is what sets them apart. For example, one truck could be red and another white; one truck could be a Dodge and another a Toyota.

Definition: *A* ***class*** *represents a blueprint or template for those real world things with defined attributes (i.e., for all of the objects that can be derived).*

For example, a truck class could be used to create many truck objects. Another way of saying this is that a class is a collection of objects that share the same attributes and behaviors. The differences between individual objects are abstracted away and ignored. So the class can be thought of as the perfect idea of something. This occurs in the real world in, for example, the way a child learns to abstract away the differences between Aunt Jamie's schnauzer, a best friend's bulldog, and dad's boxer – and learns to classify them all as dogs.

This is not a new idea. Plato, quoting Socrates in *The Republic*, discusses the *Theory of Forms or Ideas*. For example, no one has ever seen a perfect circle; however, we have an idea of what a perfect circle should be. We have drawn many circles, but none of them were absolutely perfect. The perfect idea of a circle would be considered a class, and each of the circles we draw would be considered objects.

Formally, a class defines the state and behavior of a *class* of objects. The fact that a truck has a color, year, make, model, mileage, and so on, is defined in the class. The fact that a truck can haul, drive, turn, honk, and so on, is also defined in the class. In fact, how a truck hauls, drives, turns, and honks is specified in the truck class as well. From the truck class, many truck **instances** can be created, each with potentially different attribute values making up each truck's unique state. We say that, from this class, we can **instantiate** many objects.

Definition: ***Instantiation** is the process of creating a new instance (a.k.a. an object) of a class.*

Usually, we use the term object and instance interchangeably. That is, a truck object, for example, is just an instance of the truck class.

Activity 1: The zoo

In this activity, you will play a game using an animal class. This class, formally called **Animal**, will define what animals can be and do. Some of the students in the class will become objects of the class **Animal**. The animal class defines several attributes that an animal has:

- `type`: a string that represents the animal's type (e.g., dog)
- `appetite`: an integer that represents how much daily food units the animal requires to live
- `stomach`: an integer that represents how much food units are currently in the animal's stomach
- `alive`: a Boolean that represents whether or not the animal is alive
- `sound`: a sound that represents the sound the animal makes

The class also defines several behaviors that an animal can do (and that students will perform when called upon):

- `talk()`: make the sound the animal makes
- `burn()`: use the animal's daily food units by subtracting `appetite` from `stomach`
- `eat(amount)`: increase the animal's `stomach` food units by the provided `amount`
- `getType()`: tell the requester what the animal's type is (i.e., the value of `type`)
- `isAlive()`: tell the requester if the animal is alive or not (i.e., the value of `alive`)

Note that if `stomach` is less than 0, then `alive` becomes false – and the animal dies...

Representing state and behavior

Objects store their state in **instance variables**.

Definition: *An **instance variable** is a variable defined within a class. Each object can have a different value for their copy of the instance variable.*

For example, a truck class could define the variable `year` to represent the year a truck was manufactured. A specific truck object (or instance) would set this variable to the year of its manufacture. In Python, this would be done with a simple assignment statement. If, for example, the truck object were manufactured in 2016, then the statement `year = 2016` would appropriately set the truck's year of manufacture. Another truck object could have a different year of manufacture. Ultimately, the class defines instance variables; however, each object stores its own unique set of values.

There are certain attributes that all instances of a class may want to share. Consider a class that defines a *person*. Although each instance/object of the person class can be different and thus have unique values stored in its instance variables (e.g., different sex, name, age, etc), all persons are Homo sapiens. All persons share this *scientific name*. In fact, if an expert in the field were to rename (or perhaps refine) the term Homo sapiens to something else, this would change for all persons, effectively at the same time. This kind of behavior can also be replicated in the object-oriented paradigm.

Definition: *A **class variable** defines a value that is shared among all the instances of a class.*

Unlike instance variables that, when changed, only affect a single object, a change in a class variable affects all instances of a class simultaneously. Essentially, a class variable is stored in memory that is shared among all the instances of a class.

The behavior of objects is defined in **methods** (or functions) that can be invoked. For example, the *turn* behavior of a truck could be defined in a function called `turn`. If necessary, this function could take parameters as input and return some sort of output.

Ultimately, a class has source code that specifies its state (through instance variables) and behavior (through methods).

Definition: *Collectively, state and behavior are referred to as the **members** of a class.*

Let's take a look at a simple example of a *dog* class in Python. For this example, a dog only has a name, and all dogs are canines.

```
1:   class Dog:
2:       kind = "canine"

3:       def __init__(self, dog_name):
4:           self.name = dog_name
```

Line 1 represents the class header, which includes the Python keyword `class` and the name of the class (in this case, *Dog*). Class headers are terminated with a colon, much like function headers. It is typical to capitalize the names of classes. Moreover, class names should always be singular nouns since they define the blueprint for a single thing.

Line 2 defines a class variable named `kind` that is initialized with the string “canine”. This value is shared among all dogs. The reason that we consider `kind` to be a class variable is that it is defined inside the class but outside any methods that are in the class. That is, class variables are defined at the class level.

Lines 3 and 4 represent a function called `__init__`. In Python, functions that begin and end with two underscores have special meaning. In fact, they are called **magic functions**[1].

[1] Some people refer to these are “dunder” methods due to the double underscores used to denote them

Definition: ***Magic functions*** *are functions in Python that have special meaning. They are normally called automatically by Python in the right situations (sometimes we call them explicitly). They have two underscores before and two underscores after the name of the function. There are several magic functions in Python that we can use and declare.*

The `__init__` function provides an initialization procedure for each instance of the class. That is, the source code contained within this method effectively defines what it means to initialize a new instance of the class. When we want new instances of the dog class, this function is automatically invoked (since it is a magic function). Formally, this type of function is called a **constructor.**

Definition: *A* ***constructor*** *contains the source code required to construct a new instance of a class. A constructor is automatically invoked each time we create an instance of a class. Its purpose is to initialize the object, which typically means to set default values for one or more of the instance variables (e.g. we don't have to initialize them all). These initial values may come from parameters of the constructor or we can give the instance variable a value ourselves (such as initializing a variable to 0 or 1).*

In the dog class above, the constructor takes two parameters: `self` and `dog_name`. The first parameter represents the instance that is about to be instantiated. **This parameter is always required!** The second parameter represents the name of this new dog (e.g., Bosco). The function header indicates that, to create a new instance of the dog class, a dog name must be provided. Line 4 then sets the instance variable `name` for the object to be created. Note that `self.name` (on the left side of the assignment statement) represents the instance variable (called `name`) for the dog class, and specifically targets this dog instance's name (via `self.name`). The *dot* in between `self` and `name` is called the dot operator and will be covered shortly. The variable `dog_name` (on the right side of the assignment statement) is passed in to the function when a new instance of a dog is desired.

Instances of the dog class can be created as we need them. This typically occurs outside of the dog class (for example, in a program that requires dog objects to interact with each other). Objects that are instances of the dog class can be easily instantiated as follows:

```
5:    d1 = Dog("Maya")
6:    d2 = Dog("Biff")
```

Line 5 declares a variable, `d1`, that represents an instance of the dog class. Specifically, `d1` is a dog whose name is "Maya". Line 6 defines a variable, `d2`, that represents another instance of the dog class. Specifically, `d2` is a dog whose name is "Biff".

When line 5 is executed, the variable `d1` is mapped to the variable `self` in the `__init__` function (constructor) of the dog class. The variable `self` is a formal parameter. The variable `d1` is the actual parameter that is mapped to the formal parameter. Similarly, the string "Maya" (actual parameter) is mapped to the variable `dog_name` (formal parameter). The statement `self.name = dog_name` ultimately sets the instance variable `name` for this instance of the dog class to whatever was passed in as the variable `dog_name` (i.e., Maya in this case).

Formally, the variable `d1` is called an **object reference**. That is, it refers to an object (or instance) of the dog class. The variable `d2` is also an object reference of the dog class. We can access the members of a class by using the **dot operator**. For example, we could change the name of `d1` to Bosco as follows:

```
	d1.name = "Bosco"
```

The above example shows how to modify an instance variable. Note that it only changes the name of `d1` and not `d2` because the specified object reference is `d1`. Simply accessing (without changing) a member of a class is just as easy; for example:

```
	print(d2.name)
```

This statement would produce the output Biff (because `d2`'s name is Biff). In fact, let's list the state of each instance of the class dog, `d1` and `d2` (note that this is not Python source code; rather, it is an enumeration of the instance variables and their associated values for the objects `d1` and `d2`):

`d1.name` →	Bosco
`d2.name` →	Biff
`d1.kind` →	canine
`d2.kind` →	canine

You have actually seen (and used) the dot operator before. Consider the following statements:

```
	name = "Joe"
	welcome_string = "Hello, {}"
	print(welcome_string.format(name))
```

From these statements, we can infer that strings (specifically the string `welcome_string` in the example above) are objects! In addition, the function `format` must be a member of the string class since it can be accessed using the dot operator! In fact, the function `format` is part of the behavior of the string class. This function takes one or more parameters that replace the empty braces in the string.

Did you know?

In Python, instance variables don't need to be formally declared in the class. That is, they can be defined as needed, dynamically. For example, although the dog class doesn't yet specify an instance variable that defines a dog's breed, the following statement in the main part of the program effectively adds the instance variable `breed`. It only adds it to the specific instance, though, as other instances will not have breed added to them. Specifically, it sets `d1`'s breed to German Shepherd:

```
	d1.breed = "German Shepherd"
```

It is standard practice, however, to formally define all instance variables in the class. This will be further discussed later.

Instance variables vs. class variables

Suppose that an expert in the field decided to change the scientific name for dogs from canine to something like "caten". You know, to account for inflation[2]. Changing class variables separately for each instance of the dog class doesn't make sense. The purpose of a class variable is that its value is

[2] See Victor Borge's Inflationary Language (Google it!) for the meaning behind this.

shared simultaneously among all of the instances of the class. The proper method of changing a class variable so that it appropriately affects all of the instances is to use the class name as follows:

```
     Dog.kind = "caten"
```

This statement simultaneously changes the class variable `kind` (to caten) for all instances of the dog class. To illustrate this, here is some source code for a dog class that illustrates the behavior and differences of class and instance variables:

```
     class Dog:
         kind = "canine"
         def __init__(self, dog_name):
             self.name = dog_name

     d1 = Dog("Maya")
     d2 = Dog("Biff")

     print("I have a dog named {} that is a {} and another named {}\
            that is also a {}.".format(d1.name, d1.kind, d2.name,\
            d2.kind))

     d1.name = "Bosco"
     print("I have a dog named {} that is a {} and another named {}\
            that is also a {}.".format(d1.name, d1.kind, d2.name,\
            d2.kind))

     Dog.kind = "caten"
     print("I have a dog named {} that is a {} and another named {}\
            that is also a {}.".format(d1.name, d1.kind, d2.name,\
            d2.kind))

     d1.breed = "German Shepherd"
     d2.breed = "mutt"
     print("I have a dog named {} that is a {} and another named {}\
            that is a {}.".format(d1.name, d1.breed, d2.name,\
            d2.breed))
```

Did you know?

You may have noticed above that some statements seem to be spread across multiple lines. Each of the lines that make up these statements end with a backslash (\), except for the last line of the statement. Python allows the use of a backslash to note that the remainder of a statement is provided on the next line. For example, take a look at the following statement:

```
     a = 1 - 1 + 1 - 1 + 1 - 1 + 1 - 1 + 1 - 1 + 1 - 1 + 1 - 1 + 1 -
1 + 1 - 1 + 1 - 1 + 1 - 1 + 1 - 1 + 1 - 1 + 1 - 1 + 1
```

This statement can be spread across multiple lines as follows:

```
     a = 1 - 1 + 1 - 1 + 1 - 1 + 1 - 1 + 1 - 1 + 1 - 1 + 1 - 1 + 1\
           - 1 + 1 - 1 + 1 - 1 + 1 - 1 + 1 - 1 + 1 - 1 + 1 - 1 + 1
```

Here is the output of the program:

```
I have a dog named Maya that is a canine and another named Biff
 that is also a canine.
I have a dog named Bosco that is a canine and another named Biff
 that is also a canine.
I have a dog named Bosco that is a caten and another named Biff
 that is also a caten.
I have a dog named Bosco that is a German Shepherd and another
 named Biff that is a mutt.
```

Note a few things: (1) the class variable `kind` is applied to both instances, `d1` and `d2`; and (2) the instance variable `breed` is dynamically created and applied (separately) to each instance.

Did you know?

For readability, Python source code is presented as it is formatted in IDLE throughout this lesson. the main reason for this is that presenting source code this way provides syntax highlighting. **Syntax highlighting** is the feature of highlighting (or coloring) certain portions of source code so that it helps to categorize constructs, keywords, variables, and so on. It essentially helps to make the source code more readable. For example, Python keywords are colored orange and strings are colored green.

It is important to understand the difference between instance variables and class variables. Although they seem similar, they are actually quite different. Perhaps this is best illustrated with an example. Consider the following modified dog class:

```
class Dog:
    kind = "canine"
    tricks = []

    def __init__(self, dog_name):
        self.name = dog_name

    def add_trick(self, trick):
        self.tricks.append(trick)

d1 = Dog("Maya")
d2 = Dog("Biff")
d1.add_trick("roll over")
d2.add_trick("play dead")

print("I have a dog named {} that can {}.".format(d1.name,\
       d1.tricks))
print("I have a dog named {} that can {}.".format(d2.name,\
       d2.tricks))
```

The only difference in the class is the addition of the list `tricks` and the function `add_trick`. After all, a dog can do tricks! Adding a trick to an instance of the dog class can be done by accessing the add_trick function (using the dot operator) on an object reference of a dog instance and providing the trick to add (as a string). As shown before, the dog instance is automatically passed in and mapped to the formal parameter self in the function. The string that represents the trick to add is passed in and mapped to the formal parameter trick. The function appends a new trick to the end of the list.

The expected behavior of the source code above may be that each instance of the dog class (i.e., d1 and d2) can define their own set (or list) of tricks. In fact, we expect that Maya can "roll over" and that Biff can "play dead".
However, take a look at the output:

```
I have a dog named Maya that can ['roll over', 'play dead'].
I have a dog named Biff that can ['roll over', 'play dead'].
```

The fact that both dog objects can do the same tricks can be explained by noting that the list tricks is defined at the class level and is therefore considered a class variable. As such, all instances of the dog class share the list. A change to it (even through the function add_trick) affects all instances of the dog class! To fix this and make the list of tricks an instance variable, we can define it in the __init__ method as follows:

```
class Dog:
    kind = "canine"

    def __init__(self, dog_name):
        self.name = dog_name
        self.tricks = []

    def add_trick(self, trick):
        self.tricks.append(trick)

d1 = Dog("Maya")
d2 = Dog("Biff")
d1.add_trick("roll over")
d2.add_trick("play dead")

print("I have a dog named {} that can {}.".format(d1.name,\
       d1.tricks))
print("I have a dog named {} that can {}.".format(d2.name,\
       d2.tricks))
```

Since it is no longer at the class level, it is considered an instance variable and thus allows unique values to be stored for each instance of the dog class. Here is the output of the above modified Python code:

```
I have a dog named Maya that can ['roll over'].
I have a dog named Biff that can ['play dead'].
```

Now, take a look at this more complete dog class:

```
class Dog:
    kind = "canine"

    def __init__(self, name, breed):
        self.name = name
        self.breed = breed
        self.tricks = []
        self.friends = []

    def add_trick(self, trick):
        self.tricks.append(trick)

d1 = Dog("Maya", "mutt")
d2 = Dog("Biff", "Black Lab")
d1.add_trick("roll over")
d2.add_trick("play dead")

print("I have a dog named {} that can {}.".format(d1.name,\
       d1.tricks))
print("I have a dog named {} that can {}.".format(d2.name,\
       d2.tricks))

d1.friends = [ "Finca", "Shane" ]
d2.friends = [ "Sadie", "Bosco" ]
print("{}'s friends are {}.".format(d1.name, d1.friends))
print("{}'s friends are {}.".format(d2.name, d2.friends))
```

Note the addition of several new instance variables: `breed` and `friends`. This class defines all dogs to have a name, a breed, a list of tricks, and a list of friends.

Here is the program's output:

```
I have a dog named Maya that can ['roll over'].
I have a dog named Biff that can ['play dead'].
Maya's friends are ['Finca', 'Shane'].
Biff's friends are ['Sadie', 'Bosco'].
```

At this point, it may be worthwhile to summarize the difference between class variables, instance variables, and function parameters. Class variables are relevant to an entire class. The values of class variables are shared among all of the instances of a class. Think of a class variable as being stored in a single memory location that all the instances of a class can refer to. Instance variables are also relevant to an entire class. However, the values of instance variables are unique for each instance of a class. That is, an instance variable is stored in a different memory location for each instance of a class. Function parameters are relevant to a function and are only accessible inside the function. They are short-lived and last until the function has finished its execution.

Accessors and mutators
Consider the following simple dog class:

```
class Dog:
    kind = "canine"

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

d1 = Dog("Maya")
print("I have a dog named {} that is {} year(s)\
      old.".format(d1.name, d1.age))
d1.age = -5
print("I have a dog named {} that is {} year(s)\
      old.".format(d1.name, d1.age))
```

Now take a look at the output:

```
I have a dog named Maya that is 0 year(s) old.
I have a dog named Maya that is -5 year(s) old.
```

Everything seems to work fine; however, note that no dog can actually be -5 years old. This value is not possible for a dog's age (at least not in the world that we live in). This illustrates an important point: sometimes, we may want to check that the values supplied to function parameters are sensible. For numeric types, we typically call this **range checking**. That is, we may need to ensure that a supplied value falls within a valid range. For example, a valid range for a dog's age could be 0 to 29[3].

Definition: *Range checking is a subset of a more general concept called* ***input validation****, which attempts to validate input (whether it be from a user during program execution, from actual parameters passed in to a function's formal parameters, etc).*

To ensure proper execution of a program that processes inputs, the inputs must first be validated. In the example above, the input to a dog's age must first be validated before the instance variable is assigned the value of the input.

To accomplish this, we can define a mutator (also known as a *setter*) that provides write access to an instance variable defined in a class.

Definition: *A* ***mutator*** *is a method that wraps an instance variable for the purpose of input validation (and often access control in some object-oriented programming languages).*

The instance variable still exists; however, to change it, the mutator must be called instead. Once the supplied input is validated, the instance variable is then changed with the provided value.
Here is a modified dog class with a mutator for the instance variable `age`:

```
class Dog:
    kind = "canine"
    def __init__(self, name):
        self.name = name
        self.age = 0
```

[3] The oldest dog that ever lived was an Australian cattle dog named Bluey. He reached almost 29.5 years of age!

```
        def setAge(self, age):
            if (age >= 0 and age <= 29):
                self.age = age

    d1 = Dog("Maya")
    print("I have a dog named {} that is {} year(s)\
           old.".format(d1.name, d1.age))
    d1.setAge(-5)
    print("I have a dog named {} that is {} year(s)\
           old.".format(d1.name, d1.age))
```

Note that the mutator is called `setAge`. Typically, we specifically set the name of a mutator to the word "set" followed by the name of the instance variable (initially capitalized). Since the mutator's purpose is to change the value of an instance variable, then that value must be passed in as a function parameter. The mutator then performs range checking. In the case of the dog class above, a value from 0 through 29 is valid (and would subsequently be assigned to the instance variable `age`). To change the value of a dog's age, the mutator must be called.

Here is the output now:

```
    I have a dog named Maya that is 0 year(s) old.
    I have a dog named Maya that is 0 year(s) old.
```

Note that the attempt to change `d1`'s age to -5 was not successful.

Using the function `setAge` as the mutator that enables modification of the instance variable `age` seems a bit tedious. In a perfect world, changing `d1`'s age (with input validation) would perhaps be done as follows:

```
    d1.age = 11
```

However, doing it this way would effectively bypass the mutator, `setAge`, and ignore input validation (as seen in the earlier example). Python does provide a neat way to accomplish this, however. We often call this kind of *neat* behavior syntactic sugar.

Definition: ***Syntactic sugar*** *just means that a programming language provides a sensible (and often shorthand) way to accomplish a task that may, under the hood, be a bit more convoluted.*

Python provides direct support for wrapping instance variables with mutators that perform input validation through a concept called a decorator. For now, a **decorator** is just a wrapper. It is something that wraps something else. In this case, it is a mutator in the form of a function that wraps an instance variable.

However, to properly explain how Python supports this, we must first discuss the concept of an accessor.

Definition: *An* ***accessor*** *(also known as a getter) is a method that wraps an instance variable for the purpose of providing read access (i.e., to allow us to read the value of an instance variable).*

In Python, the meaning behind this is lost because all of a class' instance variables are directly accessible. However, in other object-oriented programming languages (such as Java, for example), we can enforce the privacy of instance variables. That is, we can restrict them such that they can only be accessed through accessors and mutators. Nevertheless, the only way that Python supports decorators as mutators is to additionally provide decorators as accessors. It may be best to first show the source code that demonstrates this:

```
class Dog:
    kind = "canine"

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

    # accessor
    @property
    def age(self):
        return self._age

    # mutator
    @age.setter
    def age(self, age):
        if (age >= 0 and age <= 29):
            self._age = age

d1 = Dog("Maya")
print("I have a dog named {} that is {} year(s)\
       old.".format(d1.name, d1.age))
d1.age = -5
print("I have a dog named {} that is {} year(s)\
       old.".format(d1.name, d1.age))
```

Note that there are seemingly erroneous statements beginning with that "@" symbol (e.g., `@property` and `@age.setter`). In Python, these *tags* formally define decorators. The tag `@property` defines a decorator (or wrapper) that serves as an accessor, and the tag `@age.setter` defines a decorator (or wrapper) that serves as a mutator for a member called `age`.

Both the accessor and mutator are functions with the same name. In the case above, both are functions called `age`. Semantically, they refer to a dog's age. Since the identifier `age` is now used to refer to the accessor and mutator, the instance variable that these methods wrap must be renamed. In Python, it is typical to begin instance variables with an underscore. For example, the instance variable that stores a dog's age would be called `_age`.

Let's explain the accessor and mutator, one at a time. First, the accessor:

```
@property
def age(self):
    return self._age
```

Here, the tag @property defines a decorator that will serve as an accessor for the instance variable that represents a dog's age. The next statement defines the accessor itself. The function is called age (and only takes a single parameter, the object). Since the sole purpose of an accessor is to provide read access to an instance variable, then all that is required is to return its value (via the return keyword). Since the identifier age is used as the function's name, then the instance variable has been renamed to _age as noted earlier.

Now, the mutator:

```
    @age.setter
    def age(self, age):
        if (age >= 0 and age <= 29):
            self._age = age
```

Here, the tag @age.setter defines a decorator that will serve as a mutator for the instance variable that represents a dog's age. The next statement defines the mutator itself. The function is also called age (and takes two parameters: the object and the value to change the instance variable to). Since the purpose of a mutator is to provide write access to an instance variable with input validation, it appropriately ensures that the provided value is within an acceptable range. If so, the instance variable _age is changed to reflect the provided input value.

You may have noticed that the decorator for the mutator, @age.setter, contains the name of the function, age. This must be adhered to when defining a decorator as a mutator. If, for instance, we wished to provide a mutator for a dog's name, we could use the decorator tag @name.setter, call the mutator function name, and use the instance variable _name.

Note the following statement in the constructor:

```
    self.age = 0
```

Be careful! Here, self.age does not refer to an instance variable. It actually refers to the mutator. This assignment statement effectively calls the mutator, passing in the value on the right-hand side (0) as the second parameter of the mutator (age). That is, the value 0 is passed in to the mutator, which is then validated in the mutator. Since it is within the acceptable range (0 through 29), then the instance variable _age is set to 0.

It is important to note that the accessor must be defined **before** the mutator. Using the @property tag defines the property by name (e.g., age) so that it can be used to subsequently define the mutator (@age.setter).

To illustrate accessors and mutators a bit more, consider the following class that defines a 2D point (with an x- and y-coordinate):

```
    # points must fall within the range (-10,-10) and (10,10)
    class Point:
        def __init__(self, x=0, y=0):
            self.x = x
            self.y = y
        # getter for x
        @property
        def x(self):
            return self._x

        # setter for x
        @x.setter
        def x(self, value):
            if (value < -10):
                self._x = -10
            elif (value > 10):
                self._x = 10
            else:
                self._x = value

        # getter for y
        @property
        def y(self):
            return self._y

        # setter for y
        @y.setter
        def y(self, value):
            if (value < -10):
                self._y = -10
            elif (value > 10):
                self._y = 10
            else:
                self._y = value

    p1 = Point()
    p2 = Point(5, 5)
    p3 = Point(-50, 50)
    print("p1=({},{})".format(p1.x, p1.y))
    print("p2=({},{})".format(p2.x, p2.y))
    print("p3=({},{})".format(p3.x, p3.y))
```

Although the class for a 2D point is a bit more involved, it only contains two instance variables. The first, `_x`, represents the position of the point in the x-direction. The second, `_y`, represents the position of the point in the y-direction. Accessors and mutators for each are provided (via the methods called `x` for the instance variable `_x`, and the methods called `y` for the instance variable `_y`). In addition, range checking is performed for both the x- and y-components. Each component may not be less than -10 or greater than 10. Here is the output of the program:

```
p1=(0,0)
p2=(5,5)
p3=(-10,10)
```

Notice that the input validation works (i.e., declaring the point `p3` at -50,50 results in a point initialized at -10,10).

Did you notice something odd in the constructor? Here it is for reference:

```
def __init__(self, x=0, y=0):
    self.x = x
    self.y = y
```

Take a look at the constructor's parameters: `(self, x=0, y=0)`. You probably expected something more like this: `(self, x, y)`. In Python, we can provide **default** values for function parameters. This works for any function (not just the constructor). This means that, should parameter values be unspecified when the function is called, the default values will be used. In this example, a point's default x- and y-values are 0 and 0 respectively. Therefore, a point at the origin could be instantiated as follows:

```
p = Point()
```

Of course, such a point could also be instantiated as follows:

```
p = Point(0, 0)
```

The default values are only used if parameter values are not specified when the function is called. It is important to note that a function can have a mix of both standard and default parameters. In fact, the constructor is just like this (`self` is a standard parameter without a default value, while `x` and `y` have default values). In Python, all parameters with default values must be specified after standard parameters. This way, it is clear if values for default parameters are specified in a function call. For example:

```
def foo(a, b, c, d=5, e=7, f=8):
    pass

...

foo(1, 2, 3, 4)
```

In this case, the actual parameters 1, 2, 3, and 4 are mapped to the formal parameters `a`, `b`, `c`, and `d`. The default values for the formal parameters `e` and `f` are used.

To wrap up this section, let's add line numbers to the point class above and trace the program's execution:

```
1:  # points must fall within the range (-10,-10) and (10,10)
2:  class Point:
3:      def __init__(self, x=0, y=0):
4:          self.x = x
5:          self.y = y

6:      # getter for x
7:      @property
8:      def x(self):
 9:           return self._x

10:       # setter for x
11:       @x.setter
12:       def x(self, value):
13:           if (value < -10):
14:               self._x = -10
15:           elif (value > 10):
16:               self._x = 10
17:           else:
18:               self._x = value

19:       # getter for y
20:       @property
21:       def y(self):
22:           return self._y

23:       # setter for y
24:       @y.setter
25:       def y(self, value):
26:           if (value < -10):
27:               self._y = -10
28:           elif (value > 10):
29:               self._y = 10
30:           else:
31:               self._y = value

32:   p1 = Point()
33:   p2 = Point(5, 5)
34:   p3 = Point(-50, 50)

35:   print("p1=({},{})".format(p1.x, p1.y))
36:   print("p2=({},{})".format(p2.x, p2.y))
37:   print("p3=({},{})".format(p3.x, p3.y))
```

In the space below, trace the execution path of the program by listing the lines numbers:

Did you know?

Although some object-oriented languages actually prevent accessing instance variables that are protected (or wrapped) with accessors and mutators, Python does not enforce this. For example, it is possible to change the x-component or access the y-component of a point via statements such as:

```
p1._x = -22
print(p1._y)
```

Many Python programmers prefer to change the way they implement classes so that any value that requires protection is not stored in instance variables. There are other mechanisms that will, in fact, protect these values. However, this discussion is beyond the scope of this lesson.

Activity 2: Fractions

In this activity, we will create a class that represents a fraction. The first step is to determine what makes up a fraction (i.e., what it can *be* – its state). This task is pretty simple! Fractions have a numerator and a denominator. We can create instance variables for these and also provide accessors and mutators for each.

We may also want to provide the numeric representation of a fraction. For example, the fraction 1/2 has the numeric representation 0.5. In Python, simply dividing the numerator by the denominator (i.e., 1/2) won't produce the anticipated result because it will perform integer division. That is, the expression 1/2 will result in 0 (since 2 goes into 1 exactly 0 times). To produce a floating point result, we must convert one of the two operands to a floating point value. In Python, we can do this as follows:

```
float(1) / 2
```

The expression `float(1)` converts the integer value 1 into the floating point value 1.0. Generally, the expression `float(`*`x`*`)` converts the operand *x* into a floating point value. Formally, this conversion is called a **typecast**, in that the operand's *type* is *cast* to a different type.

The expression `float(1) / 2` produces the expected result (0.5). In fact, typecasting either operand works – as shown in the example below:

```
>>> 1 / 2
0
>>> float(1) / 2
0.5
>>> 1 / float(2)
0.5
```

There are other typecast operators that perform various type conversions. Here are a few of them:

`int(`*`x`*`)` – converts *x* to an integer
`long(`*`x`*`)` – converts *x* to a long integer
`complex(`*`x`*`, `*`y`*`)` – creates a complex number; *x* is the real part, *y* is the imaginary part
`str(`*`x`*`)` – converts *x* to a string

Lastly, we must not allow the denominator of a fraction to ever be 0 (since division by 0 is mathematically illegal). Therefore, we will need to provide range checking (via if-statements, for example), to ensure that such an assignment is prevented.

Here's the beginning of the fraction class, along with a brief main program to test the class:

```
# defines a fraction
class Fraction:
    # by default, a fraction is 0/1
    def __init__(self, num=0, den=1):
        self.num = num
        # make sure not to set the denominator to 0 if specified
        if (den == 0):
            den = 1
        self.den = den

    # getter for the numerator
    @property
    def num(self):
        return self._num

    # setter for the numerator
    @num.setter
    def num(self, value):
        self._num = value

    # getter for the denominator
    @property
    def den(self):
        return self._den

    # setter for the denominator
    @den.setter
    def den(self, value):
        # ignore if the specified denominator is 0
        if (value != 0):
            self._den = value

    # returns a fraction's numeric representation
    def getReal(self):
        return float(self.num) / self.den

# main program
f1 = Fraction()
f2 = Fraction(1, 2)
f3 = Fraction(0, 0)

print("{}/{} ({})".format(f1.num, f1.den, f1.getReal()))
```

```
    print("{}/{} ({})".format(f2.num, f2.den, f2.getReal()))
    print("{}/{} ({})".format(f3.num, f3.den, f3.getReal()))
```

And here's the output:

```
    0/1 (0.0)
    1/2 (0.5)
    0/1 (0.0)
```

Note how the class prevents the third fraction from being initialized as 0/0 and instead changes it to 0/1.

Did you know?

There is a better way of displaying a fraction than what is shown in the example above. Note how we earlier structured a print statement that built the string representation of a fraction:

```
    print("{}/{} ({})".format(f1.num, f1.den, f1.getReal()))
```

In Python, we can define a built-in **magic function** that is automatically called when we wish to display an object. In fact, this built-in function is user-definable and is named using a similar format as the constructor (i.e., the function begins and ends with two underscores). The function is called `__str__` and must return a string representation of the class. So for a fraction, such a function could be implemented as follows:

```
    def __str__(self):
        return "{}/{} ({})".format(self.num, self.den, self.getReal())
```

Displaying a fraction would then be possible via the following much simpler statement (via syntactic sugar):

```
    print(f1)
```

Adding this function to the fraction class is simple. Here's a snippet of the addition:

```
    # defines a fraction
    class Fraction:

        ...

        # returns a fraction's string representation
        def __str__(self):
            return "{}/{} ({})".format(self.num, self.den,\
                   self.getReal())

    ...

    # main program
    f1 = Fraction(1, 2)
    f2 = Fraction(1, 4)
    f3 = f1.add(f2)
    print(f1)
```

```
     print(f2)
     print(f3)
```

Of course, the output is the same as before!

Activity 3: Reducing fractions

You may have noticed that instantiating the fraction 6/8 would work just fine. The problem is that this fraction is not expressed in lowest terms. That is, it can be reduced (to 3/4). Our fraction class would greatly benefit from a function that can reduce a fraction. Such a function could be called in the constructor after setting the numerator and denominator in case it is not in lowest terms.

Although there are many ways to reduce a fraction, here's a simple algorithm that calculates the greatest common divisor (GCD) among the numerator and denominator. First, initially assume that the GCD is 1. From there, iterate, starting with 2 through the smaller of the numerator or denominator. Each time, the objective is to try to find a value that evenly divides both the numerator and denominator. As such a value is found, the GCD is updated. The final step is to divide the numerator and denominator by the GCD (which reduces the fraction). As a cleanup operation, if the numerator is 0 (i.e., the fraction's numeric value is 0.0), the denominator is set to 1 (i.e., 0/1).

This is shown in the snippet of code below (which can be placed anywhere in the fraction class):

```
     # reduces a fraction
     def reduce(self):
         # we initially assume that the GCD is 1
         # from there, we iterate starting at 2 through the smaller
         #  of the numerator or denominator
         # since the numerator and denominator could be negative,
         #  we use their absolute values
         # each time, we try to find a value that evenly divides
         #  both the numerator and denominator
         # as we find such a value, we update the GCD
         # the final step is two divide the numerator and
         #  denominator by the GCD to reduce the fraction
         # as cleanup, if the numerator is 0 (i.e., the fraction is
         #  0) then set the denominator to 1
         gcd = 1
         minimum = min(abs(self.num), abs(self.den))

         # find common divisors
         for i in range(2, int(minimum + 1)):
             # when we find one, update the GCD
             if (self.num % i == 0 and self.den % i == 0):
                 gcd = i

         # divide the numerator and denominator by the GCD
         self.num /= gcd
         self.den /= gcd
```

```
        # if the numerator is 0, set the denominator to 1
        if (self.num == 0):
            self.den = 1
```

The Python *math* library has many useful functions. The **min** function returns the minimum value of a number of values passed in as parameters. This makes it quite easy to determine which of the numerator or denominator is smaller. Since a function's numerator or denominator could be negative, we use their absolute value to determine which is smaller. The **abs** function returns the absolute value of a specified value.

So where (and when) do we call the `reduce` function? For the fraction class shown earlier, we could do so in the constructor as follows:

```
    # defines a fraction
    class Fraction:
        # by default, a fraction is 0/1
        def __init__(self, num=0, den=1):
            self.num = num
            # make sure not to set the denominator to 0 if specified
            if (den == 0):
                den = 1
            self.den = den

            self.reduce()
```

This works; however, the mutators for the numerator and denominator may cause a fraction to no longer be reduced. We could, therefore, call the `reduce` function at the end of the mutators. There is a problem with this, however. Since the constructor uses the mutator for the numerator (in `self.num = num`) before the denominator is even set, a call to the function `reduce` in the mutator for the numerator would attempt to access the denominator (which doesn't exist). This would result in an error.

We could try to place a call to the `reduce` function in the mutator for the denominator. But this is also problematic, because the `reduce` function uses the mutator for the denominator to change the denominator (i.e., when dividing it by the *gcd*). Placing it here would cause the `reduce` function to be recursively called infinitely!

Perhaps the best place to put the call to the `reduce` function is in the `__str__` function (i.e., when displaying a fraction).

Activity 4: Adding fractions...and more

Let's now implement the functionality to add two fractions and produce the sum of these as a new fraction. We must first discuss how two fractions can be added. Typically, the least common denominator is found. A simpler version, however, is to multiply each fraction by the other's denominator to obtain a common denominator (that is not necessarily the least common denominator).

Here's an illustration:

$$\frac{a}{b}+\frac{c}{d} = \frac{a*d}{b*d}+\frac{b*c}{b*d}$$

As an example, take the following:

$$\frac{1}{2}+\frac{1}{4} = \frac{1*4}{2*4}+\frac{2*1}{2*4} = \frac{4}{8}+\frac{2}{8} = \frac{6}{8} = \frac{3}{4}$$

So now, how do we implement a method in the fraction class that does this? One way is as follows:

```
    # calculates and returns the sum of two fractions
    def add(self, other):
        num = (self.num * other.den) + (other.num * self.den)
        den = self.den * other.den
        sum = Fraction(num, den)
        sum.reduce()

        return sum
```

This function is called as follows (specifically in the third statement below):

```
    f1 = Fraction(1, 2)
    f2 = Fraction(1, 4)
    f3 = f1.add(f2)
```

Note that both fractions are effectively passed in to the `add` function. The first, `f1`, represents the current instance and is mapped to the first parameter, `self`. The second, `f2`, is mapped to the second parameter, `other`.

The function implements the common denominator method shown above and generates a fraction representing the sum of `self` and `other`. The new fraction is then reduced and returned. Note that calling the `reduce` function here is not necessary if it is called in the `__str__` function.

The function could be optimized (perhaps at the expense of not being quite as readable) by returning a new fraction directly instead of creating one and returning it. For example:

```
    return Fraction(num, den)
```

In fact, the function could be optimized even more as follows:

```
    return Fraction(self.num * other.den + other.num * self.den,\
self.den * other.den)
```

Of course, we could implement functions to subtract, multiply, and divide fractions! In fact, we could implement a subtract method by using the already defined add method. How? Recall that subtracting is just adding the negative. Since this will be assigned as a program later, it is left as an individual exercise for now.

Operator overloading

In the last example above, we defined a function in the fraction class that adds two fractions and returns the sum. We used this function similar to the following snippet of Python code:

```
f1 = Fraction(1, 2)
f2 = Fraction(3, 4)
print(f1.add(f2))
```

As expected, the output of these statements is: `5/4 (1.25)`

The way in which we call the addition function seems a bit tedious. Why can't we just use the addition operator? For example, why can't we just add two fractions, `f1` and `f2`, by merely using the expression `f1 + f2`? This would mean that the following modification of the example above would work as expected:

```
f1 = Fraction(1, 2)
f2 = Fraction(3, 4)
print(f1 + f2)
```

It turns out that such a thing is possible through a concept called operator overloading.

Definition: ***Operator overloading*** *is the act of redefining the behavior of operators (such as addition and subtraction) using their known symbols (+ for addition, – for subtraction, and so on) in order to support these operations on user-defined data types.*

For example, redefining the addition operator for the fraction class could mean implementing the common denominator method described earlier.

Python has various internal magic functions that support the redefinition of common operators. The main idea is to encapsulate the new, redefined behavior in a function that is automatically called (using syntactic sugar) when two objects of the class are used as operands with the specified operator. For the purpose of the fraction class, we will only consider the four arithmetic operators.

The addition operator (+) can be redefined in a function called `__add__` as follows:

```
def __add__(self, other):
    num = (self.num * other.den) + (other.num * self.den)
    den = self.den * other.den
    sum = Fraction(num, den)
    sum.reduce()
    return sum
```

Note that the source code in the new overloaded `__add__` function is exactly the same as it was in the original `add` function shown earlier.

The subtraction operator (–) can be redefined in a function called `__sub__` as follows:

```
def __sub__(self, other):
    ...
```

Note that for this and the remaining operators, the source code is not provided. Instead, appropriate code is replaced with an ellipsis (…).

The multiplication operator (*) can be redefined in a function called `__mul__` as follows:

```
def __mul__(self, other):
    ...
```

Lastly, The division operator (/) can be redefined in a function called `__truediv__` as follows:

```
def __truediv__(self, other):
    ...
```

The fraction class has now grown! Take a look:

```
# defines a fraction
class Fraction:
    # by default, a fraction is 0/1
    def __init__(self, num=0, den=1):
        self.num = num
        # make sure not to set the denominator to 0 if specified
        if (den == 0):
            den = 1
        self.den = den

        self.reduce()

    ...

    # calculates and returns the sum of two fractions
    def __add__(self, other):
        num = (self.num * other.den) + (other.num * self.den)
        den = self.den * other.den
        sum = Fraction(num, den)
        sum.reduce()

        return sum

    # calculates and returns the difference of two fractions
    def __sub__(self, other):
        # replace this with the function's actual implementation
        return None
    # calculates and returns the product of two fractions
    def __mul__(self, other):
        # replace this with the function's actual implementation
        return None

    # calculates and returns the division of two fractions
    def __truediv__(self, other):
        # replace this with the function's actual implementation
        return None

    ...
```

In fact, we can now perform all of the implemented arithmetic operations on fractions. Here's a snippet of Python code that tests the fraction class and assumes that the operator overload functions have been fully implemented:

```
# main program
f1 = Fraction(1, 2)
f2 = Fraction(1, 4)

print(f1)
print(f2)
print(f1 + f2)
print(f1 - f2)
print(f1 * f2)
print(f1 / f2)
```

And here is the output:

```
1/2 (0.5)
1/4 (0.25)
3/4 (0.75)
1/4 (0.25)
1/8 (0.125)
2/1 (2.0)
```

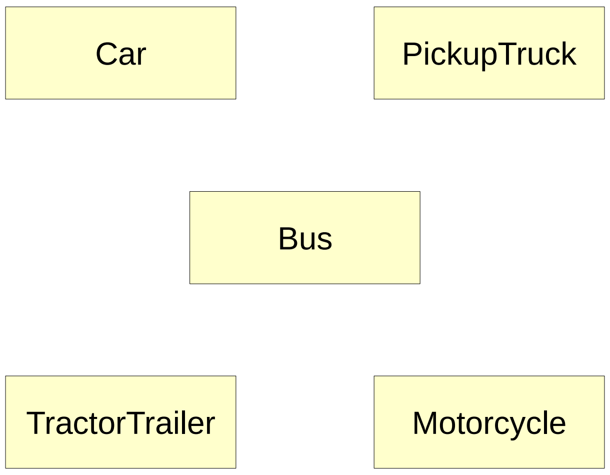
Class diagrams

In computer science courses, you are often asked to design simple programs. The ability to understand and hold everything in one's head when solving a simple problem is relatively straightforward and, well, usually pretty simple. However, when solving complicated problems and developing solutions to these problems as large and tedious applications, it becomes quite difficult to manage all of the parts and pieces. Often, we require the use of tools and techniques that incorporate visual aids and diagrams to assist us in managing the structure and components of these applications.

Definition: *A **class diagram** is a type of diagram that describes the structure of a program by visualizing its classes, their state and behavior, and their relationships.*

The most simple class diagram only shows the classes of a program, which are represented as rectangles.

To illustrate how a class diagram could be used to model an application's structure, let's consider one that models vehicle traffic in a large city for the purpose of analyzing how it manages traffic during rush hour. This kind of application would be useful in learning about traffic patterns, congestion, and so on. In fact, it could help to redesign roads, entrances to and exits from highways and interstates, the placement and timing of traffic signals, etc. Such an application may include classes for cars, pickup trucks, buses, tractor trailers, motorcycles, and so on, since all of these things contribute to the traffic in a city. In fact, the application could be modeled with a class diagram as follows:



The classes of an application are always singular nouns. Since a class is a blueprint for objects, then a class is essentially like a rubber stamp. For example, we can define a class that describes the blueprint for a car. This class would be considered the car class and be formally called **Car**. As mentioned earlier, the names of classes are typically capitalized. Since they are identifiers, they also must not contain spaces and abide by all of the rules for naming identifiers in the programming language. To summarize the rules of naming a class:

1. Always use singular nouns
2. Names of classes start with an initial capital letter (and usually follow CamelCasingLikeThis)
3. Names must not contain spaces or other invalid characters as per the naming identifier rules

In Python, the car class could be defined as follows:

```
class Car:
    ...
```

Instances of the car class would collectively be called cars (and there could be many of them). Similarly, the class for a pickup truck could be called **PickupTruck**, and would be defined in Python as follows:

```
class PickupTruck:
    ...
```

The beauty of a class diagram is that it allows us to very easily see the components of a system or application. In the class diagram above, there is no indication of the state and behavior of classes, nor is there any indication of any relationships between classes. We will get to this later.

Inheritance
As you know, the object-oriented paradigm attempts to mimic the real world, particularly in how it is made up of objects that interact with each other. In the real world, objects also have relationships, and this is useful! For example, a person inherits traits from parents. Specifically, a person inherits physical traits (e.g., height, hair color, etc) and behavioral traits (e.g., manner of speaking). This behavior is represented in the object-oriented paradigm as well.

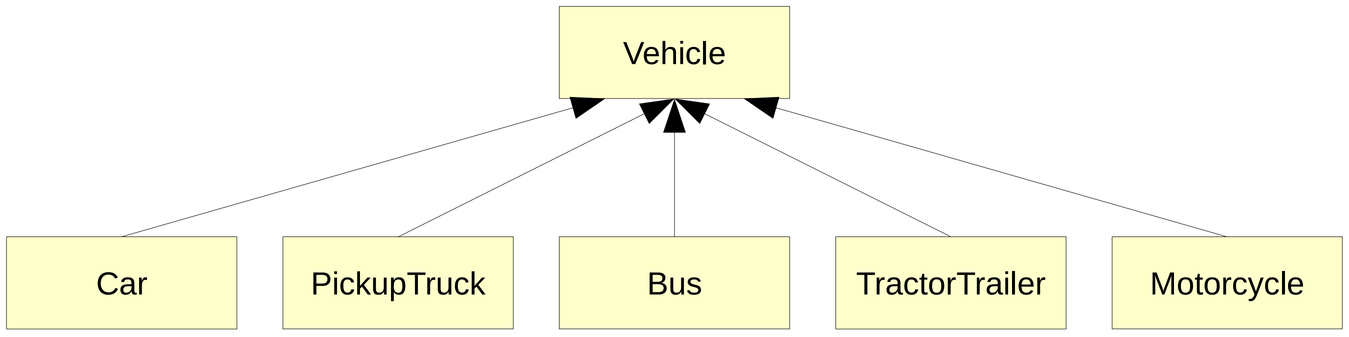
To illustrate how this is done, let's consider the application that models vehicle traffic described earlier. Let's begin with the car class that serves as the blueprint for a car in the traffic simulation. What might its state and behavior look like? That is, what are cars made up of? What can they be, and what can they do? Very quickly, we can think of attributes such as year, make, model, mileage, and so on. This represents the state of a car. We can also think of behaviors such as start, move, turn, park, and so on. In fact, we could quickly design a car class now that we know how to do so in Python!

Now let's consider a pickup truck class that serves as the blueprint for a pickup truck in the traffic simulation. Its state would most likely be very similar to that of the car class. And so would its behavior. In fact, not much differentiates a car from a pickup truck. They both generally have the same attributes and do the same thing. Imagine designing the classes for a car and a pickup truck. You may think that the classes would share many similarities in both state and behavior (and you would be right).

Now imagine maintaining such an application. Suppose that the implementation of some behavior that is similar across cars and pickup trucks needs to be modified. This would require changing both the car and pickup truck classes because code is duplicated across the two classes. Dealing with this type of thing increases the likelihood of bugs. The beauty of the object-oriented paradigm is that it allows the inheritance of state and behavior from class-to-class, just like we inherit traits from our ancestors!

The state and behavior that is shared among the car and pickup truck classes in the traffic simulation application could be captured in a more general class. Such a class could, for example, be called a **Vehicle**. All of the state and behavior that is shared among any type of vehicle would be defined in this class. Specific kinds of vehicles (like cars and pickup trucks) would then inherit these *traits*. Any modifications to the state and behavior of vehicles of all types could be made in the vehicle class and be automatically applied to all types of vehicles!

In fact, let's amend the class diagram shown earlier by including a vehicle class that defines the overall state and behavior that all types of vehicles (cars, pickup trucks, buses, tractor trailers, and motorcycles) share:



Note how all of the classes that inherit state and behavior from the vehicle class now have solid arrows pointing *toward* the vehicle class. In a class diagram, this indicates an inheritance relationship. Specifically, the car, pickup truck, and other classes shown at the bottom of the class diagram inherit state and behavior from the vehicle class.

Definition: *A class that defines state and behavior that is inherited by other classes is called a* ***superclass****. The classes that inherit from it are called* ***subclasses****.*

In the class diagram above, the class **Vehicle** is a superclass of the class **Car**, and the class **Car** is a subclass of the class **Vehicle**.

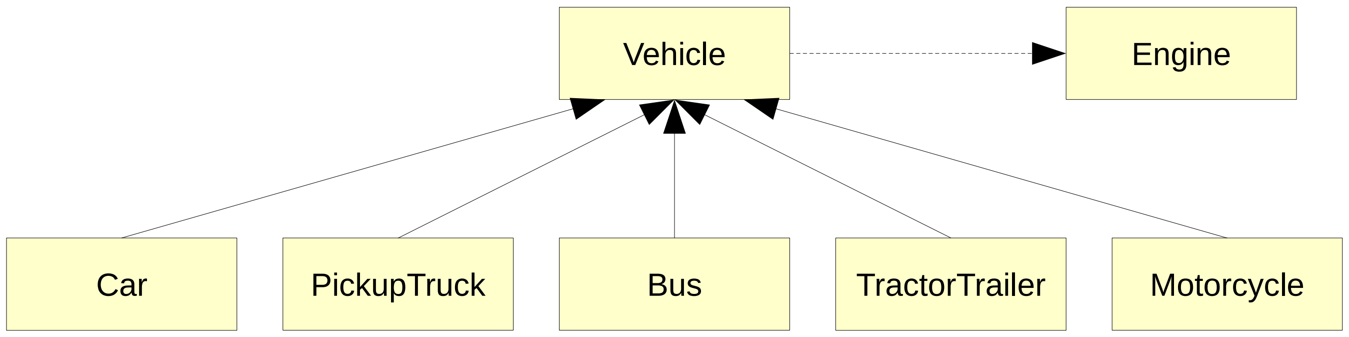
The inheritance relationship is often called the **is-a** relationship. This is actually quite clear from the class diagram: a Car *is a* Vehicle, a Bus *is a* Vehicle, and so on. There is also the **has-a** relationship. This represents a composition relationship and refers to the state of an object. Specifically, we often note the has-a relationship in class diagrams for classes that contain other classes.

In terms of how this is accomplished in Python source code, we merely need to specify the superclass in a subclass' class definition. For example, consider the class **Car** (which is a subclass of the superclass **Vehicle**). To note this relationship in Python, we merely need to define the **Car** class as follows:

```
class Car(Vehicle):
    ...
```

This establishes the relationship that the class **Car** is a subclass of the class **Vehicle**, and that the class **Vehicle** is a superclass of the class **Car**.

Next, consider an engine class that defines everything that an engine can be and do. Clearly, a car has an engine. So does a pickup truck, a bus, a motorcycle, and so on. In general, all of these vehicles have an engine. Since all vehicles have an engine, in the design of the application we may include the engine class as part of the state of the vehicle class. Specifically, we would include an instance of the engine class in the vehicle class. All subclasses of the vehicle class would then inherit this attribute. We note the has-a relationship in a class diagram with a dashed arrow that point toward the composed class. Here is an amended class diagram that now includes the engine class:

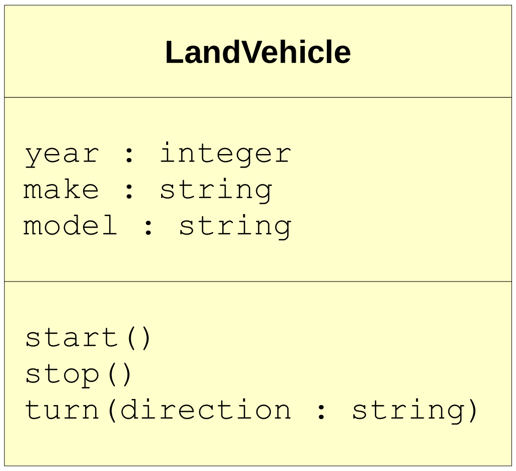


This important relationship illustrates that objects can, in fact, create other objects! In the example above, a vehicle can create an instance of an engine. Although the state and behavior of all vehicles is defined in the vehicle class, nothing stops any of its subclasses from redefining or specializing these attributes or behaviors. That is, although a car and a motorcycle both have an engine, they are quite different. Simply because the engine class is included in the vehicle class does not prevent a car or a motorcycle from specializing the engine and uniquely setting its state.

Let's further illustrate the concept of inheritance by expanding the world of vehicles. In this expanded world, there are two types of vehicles: land vehicles (that move on land) and air vehicles (that fly in the air). The types of land vehicles that exist include all of the vehicles described earlier (e.g., cars, pickup trucks, etc), and the types of air vehicles that exist include airplanes, helicopters, and ultralights. While we're at it, let's define multiple types of engines for land vehicles (e.g., V-6, V-8, and I-6), and multiple types of engines for air vehicles (e.g., turbo prop and jet engine).

Try to represent this expanded world with a class diagram in the space below:

Often, we include the state and behavior of classes in the class diagram. Suppose that the class **LandVehicle** has the instance variables `year`, `make`, and `model`, and the functions `start`, `stop`,



Typically, we include the types of instance variables and adhere to the following format:

```
variable_name : variable_type
```

For functions, we include the names and types of any parameters and adhere to the following format:

```
function_name(parameter1_name : parameter1_type, ...)
```

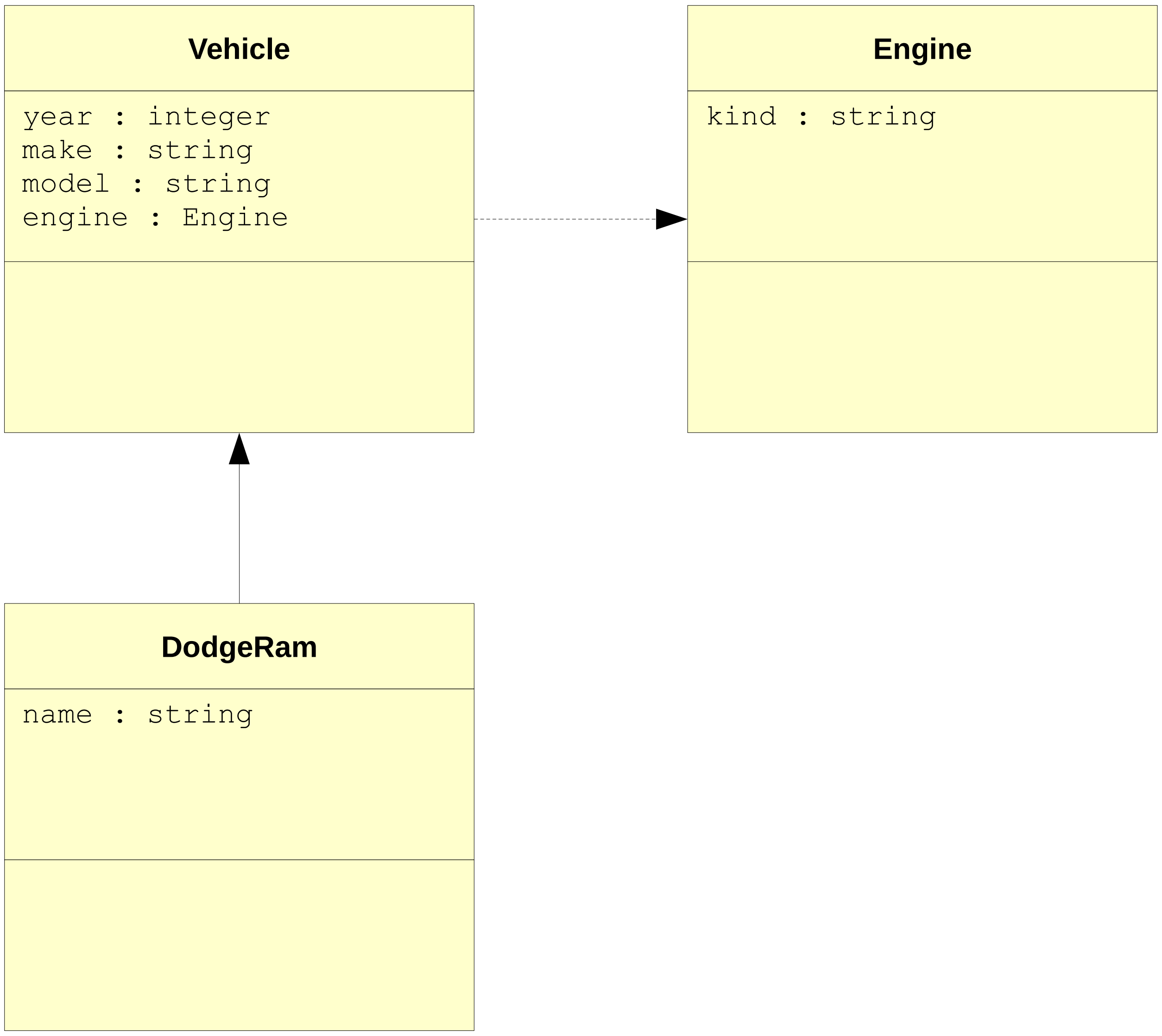
You have probably noticed that this extension of class diagrams makes it quite easy to implement the source code for the class!

The `object` class

The inheritance relationship is easily implemented in Python classes in the class header. In python 2.x, in order to define accessors and mutators, the class definition must **explicitly** include what you now know to be an inheritance relationship with a class called **object**. The inheritance from class **object** is **implicit** in Python 3.x. The class header for the dog class could have been written:

```
class Dog(object):
    ...
```

Formally, the class **object** is defined to be the ultimate superclass for all built-in (i.e., user-defined) types. As shown, it is possible to have multiple levels of inheritance (e.g., the class **Car** is a subclass of the class **LandVehicle** which is a subclass of the class **Vehicle**). At the top of this inheritance hierarchy lies the object class.



From this class diagram, we can quickly begin laying out the source code for the classes. In fact, the class headers can be directly inferred from the class diagram:

```
class Vehicle: #implicitly inherits from class object
    ...

class DodgeRam(Vehicle):
    ...

class Engine: #implicitly inherits from class object
    ...
```

Since the class diagram includes the state of each class, declaring instance variables in the constructors of each class and providing appropriate accessors and mutators is also relatively straightforward. In fact, here's the entire **Vehicle** class:

```
    class Vehicle: #implicitly inherits from class object
        def __init__(self, year, make, model):
            self.year = year
            self.make = make
            self.model = model
            self.engine = None

        @property
        def year(self):
            return self._year

        @year.setter
        def year(self, value):
            self._year = None
            if (value > 1800 and value < 2018):
                self._year = value

        @property
        def make(self):
            return self._make

        @make.setter
        def make(self, value):
            self._make = value

        @property
        def model(self):
            return self._model

        @model.setter
        def model(self, value):
            self._model = value

        @property
        def engine(self):
            return self._engine

        @engine.setter
        def engine(self, value):
            self._engine = value

        def __str__(self):
            return "Year: {}\nMake: {}\nModel: {}\nEngine:\
                   {}".format(self.year, self.make, self.model,\
                   self.engine)
```

The constructor includes parameters for a vehicle's year, make, and model. By default, a vehicle's engine is undefined (i.e., `None`). The class contains getters and setters for each of the instance variables. Finally, the `__str__` function defines how to represent a vehicle as a string, which is in the following format (with an example for clarity):

```
Year: 2016
Make: Dodge
Model: Ram
Engine: V6
```

Here is the entire **Engine** class:

```python
class Engine: #implicitly inherits from class object
    def __init__(self, kind=None):
        self.kind = kind

    @property
    def kind(self):
        return self._kind

    @kind.setter
    def kind(self, value):
        self._kind = value

    def __str__(self):
        return str(self.kind)
```

The **Engine** class has a single instance variable (its kind). Its constructor includes one parameter for the kind of engine (`None` by default). A getter and setter is provided for the instance variable. The string representation of an engine is just its kind.

Lastly, here is the **DodgeRam** class:

```python
class DodgeRam(Vehicle):
    make = "Dodge"
    model = "Ram"

    def __init__(self, name=None, year=None):
        super().__init__(year, DodgeRam.make, DodgeRam.model)
        self.name = name

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, value):
        self._name = value
    def __str__(self):
        return "Name: {}\n{}".format(self.name,\
               super().__str__())
```

The **DodgeRam** class has a single instance variable (its name). Its constructor takes two parameters (for a DodgeRam's name and year). Note that the class contains two class variables that are shared among all instances of the class: make and model. This makes sense, because all instances of the class **DodgeRam** are Dodge Rams! That is, their make is Dodge, and their model is Ram.

There are two more interesting (and new) things in the class **DodgeRam**. Take a look at the first statement in the constructor:

```
super().__init__(year, DodgeRam.make, DodgeRam.model)
```

When implementing inheritance relationships, it often becomes useful and sometimes necessary to invoke or call functions in a subclass' superclass. Formally, state and behavior that are defined in the superclass are inherited in a subclass. They can be redefined in the subclass; however, they don't necessarily need to be. In fact, a subclass may inherit a function and need to implement the inherited behavior first. This is accomplished by calling the function in the superclass. Since the current object (self) is not an instance of the superclass, then invoking a function in the superclass is done by using the super() function. So the first part of the statement, `super().__init__`, means to call the constructor in the **Vehicle** class (the superclass of the **DodgeRam** class).

In this case, the year, make, and model are passed as parameters to the constructor in the **Vehicle** class. This effectively sets up the appropriate instance variables in the **Vehicle** class (which are inherited in the **DodgeRam** class). Subsequently, the constructor in the **DodgeRam** class then initializes its only instance variable, `name`.

Another new thing in the class is the statement in the `__str__` function:

```
return "Name: {}\n{}".format(self.name, super().__str__())
```

The string representation of a **DodgeRam** is its name, followed by the string representation of a **Vehicle** (which was illustrated earlier). The latter part of the statement calls the `__str__` function in the Vehicle class:

```
super().__str__()
```

Again, this illustrates a call to a function in the superclass. This call returns the string representation of a vehicle – which is displayed below the name of the **DodgeRam**; for example:

```
Name: Boss Hog
Year: 2016
Make: Dodge
Model: Ram
Engine: V6
```

So the string representation of a **DodgeRam** is simply its name, followed by the inherited string representation of a **Vehicle**.

Why inheritance?
There are clear benefits of using inheritance in our programs. In a sense, it makes the reasoning of an application more possible since it attempts to mimic the world that we live in. But it also reduces code duplication, because similarities between objects can be encapsulated in superclasses. This has the downstream effect of promoting the reuse of code, and intrinsically makes code maintenance much easier. In fact, we often say that if software is not maintained, it dies! So we maintain software often. It behooves us to make this process easier.

Lastly, inheritance makes applications easier to extend. Think of adding a different type of vehicle (say, a **HondaCivic**). Without inheritance, we would have to include instance variables for year, make, model, and so on. However, these are already defined in the **Vehicle** class! We simply need to define the class **HondaCivic** as a subclass of the class **Vehicle** in order to inherit its state and behavior:

```
class HondaCivic(Vehicle):
    ...
```

Single inheritance vs. multiple inheritance
The inheritance relationship that has been discussed thus far is known as **single inheritance**. That is, a subclass inherits state and behavior from a single superclass. Most object-oriented programming languages support single inheritance. Often, however, there is a need to support **multiple inheritance**, where a subclass can inherit from more than one superclass.

To illustrate this, consider a grocery store's items. A banana, for example, is a fruit. Therefore, it may inherit traits from a fruit superclass such as type, country of origin, etc. However, in the context of a grocery store, a banana is also an item for sale. Such a sale item has a price, an inventory, a shelf location, etc. Inheriting from both a **Fruit** superclass and a **SaleItem** superclass, for example, would be useful in implementing the point-of-sale system for a grocery store.

Most object-oriented programming languages do not support multiple inheritance. Some do, but only in a limited form. Java, for example, supports it in a limited form by utilizing something known as an interface. The technical details of this are beyond the scope of this lesson. Python, however, directly supports multiple inheritance. One must merely list all of a subclass' superclasses in the class header; for example:

```
class Banana(Fruit, SaleItem):
    ...
```

OO quick reference
We have covered a lot of new terms in this lesson. This final section merely aggregates them all, along with their definition, so that you can easily and quickly refer to the terms should you need to.

Term	**Definition**
accessor	A special method in a class that wraps an instance variable for the purpose of providing read access.
behavior	All of the things that an object can *do*; implemented using functions in the class.
class	A blueprint for a thing; the definition of state and behavior for an entire class

	of things.
class diagram	A diagram that models the classes of a system or application, their relationships, and their members.
class variable	A variable that is defined at the class level; its value is shared among all instances of the class.
constructor	A special method in a class that is automatically invoked when a new instance of the class is instantiated; usually performs initialization tasks (e.g., assigning default or specified values to the instance variables).
decorator	A wrapper; in Python, accessors and mutators are wrapped using a decorator.
dot operator	When used on an object reference, accessed the specified member of the class.
has-a	A relationship among classes that implies one class making use of another; also means the ability of an object to create other objects.
inheritance	A relationship among classes that permits a class to inherit the state and behavior of another class; see **is-a**.
input validation	The process of validating a provided input to ensure that it conforms to some expected range or type.
instance	An object that represents the instantiation of a class.
instance variable	A variable defined in a method of a class (usually the constructor) that allows individual instances of the class to uniquely set values to.
instantiate	The process of constructing a new instance of a class.
is-a	A relationship among classes that permits a class to inherit the state and behavior of another class; see **inheritance**.
magic function	A special function in Python whose name begins and ends with two underscores (e.g., `__init__`, `__str__`, `__add__`).
member	How we collectively reference the state and behavior of a class.
method	How behavior is implemented in a class; they are functions that describe what an object can *do*.
multiple inheritance	The ability of a class to inherit the state and behavior of multiple classes simultaneously.
mutator	A special method in a class that wraps an instance variable for the purpose of providing write access; usually implements input validation.
object	An instance of a class, with specific values assigned to instance variables.
object class	The base class for all user-defined objects; the top-most superclass.
object reference	A variable name that refers to an object.
operator overloading	The redefining of an operator (e.g., the addition operator) on user-defined objects.
single inheritance	The ability of a class to inherit the state and behavior of a single class.

state	All of the things that an object can *be*; implemented using instance variables in the class.
subclass	A class that inherits state and behavior from another class.
superclass	A class that another class inherits state and behavior from.
typecast	The process of converting a value from one type to another (e.g., converting an integer to a floating point number).