Chapter 8

Software Objects: The Critter Caretaker Program

Understanding Object-Oriented Basics

- *Object-oriented programming* (OOP) is a different way of thinking about programming. It’s a modern methodology that’s been embraced by the software industry and is used in the creation of the majority of new, commercial software.

- The basic building block in OOP is the *software object*—often just called an *object*.

- OOP can represent real-life objects as software objects. Like real-life objects, software objects have characteristics (*attributes*) and behaviors (*methods*).

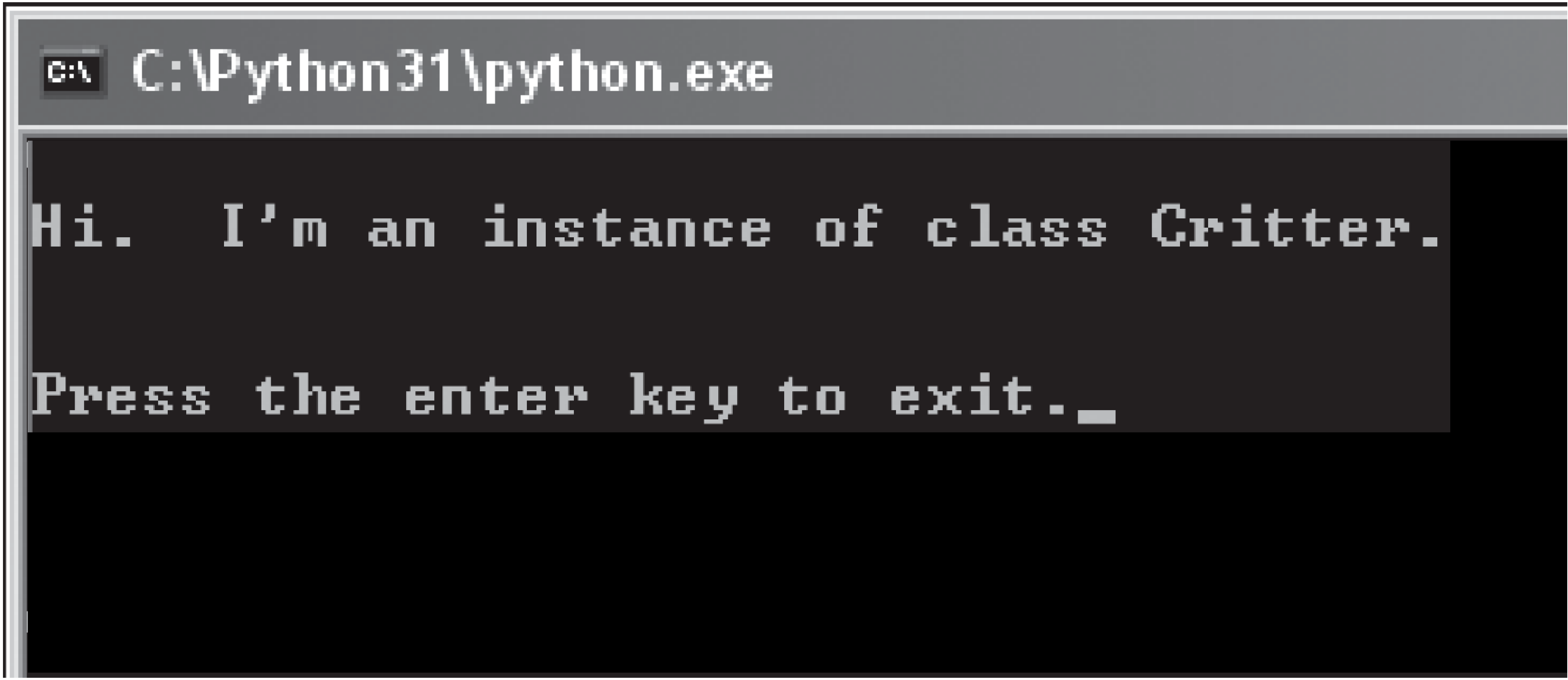
- Objects are created (*instantiated*) from a definition called a *class*—code that can define attributes and methods.

- Classes are like blueprints. A class isn’t an object, it’s a design for an object. One can create many objects from the same class.

- As a result, each object (*instance*) instantiated from the same class will have a similar structure.

- you can have 2 objects of the same class and give each its own unique set of attribute values.

Introducing the Simple Critter Program



```
Hi.  I'm an instance of class Critter.

Press the enter key to exit._
```

simple_critter.py

```python
# Simple Critter
# Demonstrates a basic class and object

class Critter(object):
    """A virtual pet"""
    def talk(self):
        print("Hi.  I'm an instance of class Critter.")

# main
crit = Critter()
crit.talk()

input("\n\nPress the enter key to exit.")
```

Defining a Class

```
class Critter(object):
```

- The keyword **class** is followed by the class name.
- **object** is a fundamental, built-in type.

Defining a Method

```
def talk(self):
    print("Hi.  I'm an instance of class Critter.")
```

- Every *instance method*—a method that every object of a class has—must have a special 1st parameter, called **self** by convention.

- This parameter provides a way for a method to refer to the object itself.

- If you create an instance method without any parameters, you’ll generate an error when you invoke it.

Instantiating an Object

- After we wrote the class, instantiating a new object is:

```
crit = Critter()
```

- This line creates an object of the Critter class and assigns it to the variable crit.

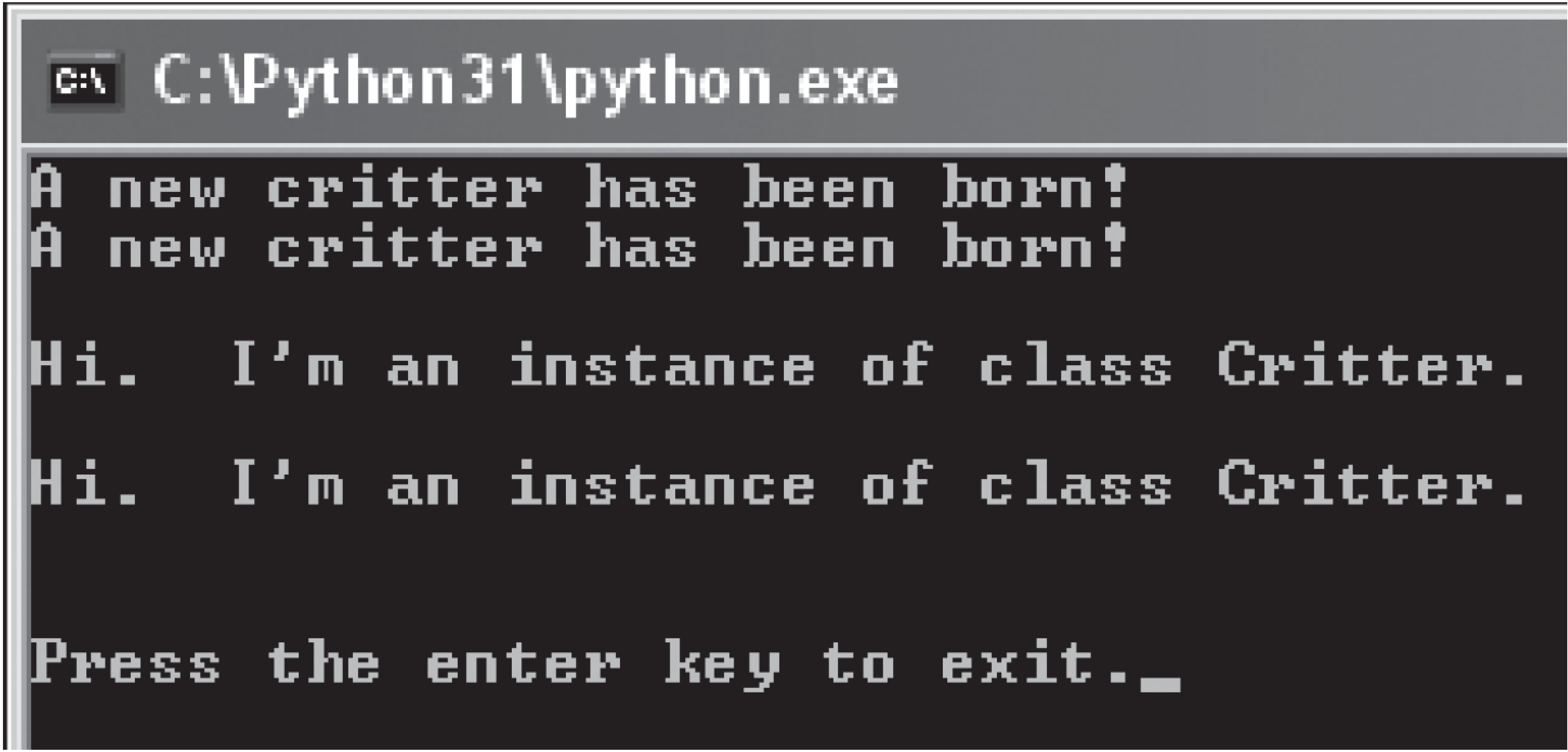
- Notice the parentheses after the class name Critter in the assignment statement. It’s critical to use them if you want to create a new object.

Invoking a Method

- Invoke this method talk() just like any other, using dot:

```
crit.talk()
```

the Constructor Critter Program



```
A new critter has been born!
A new critter has been born!

Hi.  I'm an instance of class Critter.

Hi.  I'm an instance of class Critter.


Press the enter key to exit._
```

constructor_critter.py

```
# Constructor Critter
# Demonstrates constructors

class Critter(object):
    """A virtual pet"""
    def __init__(self):
        print("A new critter has been born!")

    def talk(self):
        print("\nHi.  I'm an instance of class Critter.")

# main
crit1 = Critter()
crit2 = Critter()

crit1.talk()
crit2.talk()
input("\n\nPress the enter key to exit.")
```

Creating a Constructor

- The constructor method (the *initialization* method) in the class definition is:

```
def __init__(self):
    print("A new critter has been born!")
```

- By naming the method **__init__**, we told Python that this is the constructor method.

- As a constructor method, **__init__()** is automatically called by any newly created Critter object right after the object springs to life.

- Python has a collection of built-in “special methods” whose names begin and end with 2 underscores, like **__init__** , the constructor method.

Creating Multiple Objects

```
crit1 = Critter()
crit2 = Critter()
```

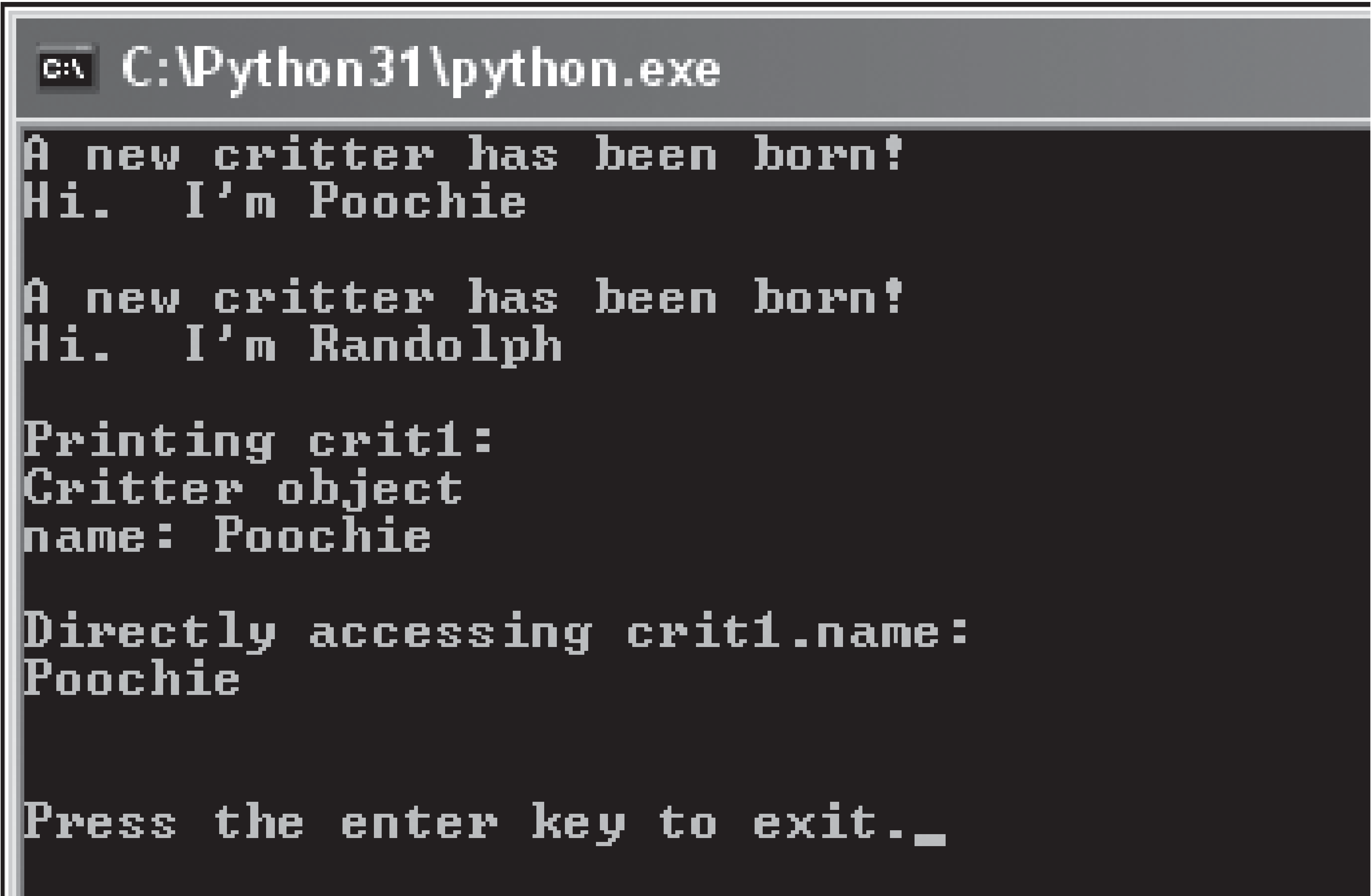
- As a result, 2 objects are created. After each is instantiated it prints "A new critter has been born!" via its constructor.

- Each object is its very own full-fledged critter. We invoke their talk() methods:

```
crit1.talk()
crit2.talk()
```

- Even though these 2 lines of code print the exact same string, "\nHi. I’m an instance of class Critter." , each is the result of a different object.

the Attribute Critter Program



```
A new critter has been born!
Hi.  I'm Poochie

A new critter has been born!
Hi.  I'm Randolph

Printing crit1:
Critter object
name: Poochie

Directly accessing crit1.name:
Poochie



Press the enter key to exit._
```

handle_it.py

```
# Attribute Critter
# Demonstrates creating/accessing object attributes

class Critter(object):
    """A virtual pet"""
    def __init__(self, name):
        print("A new critter has been born!")
        self.name = name

    def __str__(self):
        rep = "Critter object\n"
        rep += "name: " + self.name + "\n"
        return rep

    def talk(self):
        print("Hi.  I'm", self.name, "\n")
```

```
# main
crit1 = Critter("Poochie")
crit1.talk()

crit2 = Critter("Randolph")
crit2.talk()

print("Printing crit1:")
print(crit1)

print("Directly accessing crit1.name:")
print(crit1.name)

input("\n\nPress the enter key to exit.")
```

Initializing Attributes

- In the constructor, it creates the attribute **name** for the new object and sets it to the value of the parameter name:

```
def __init__(self, name):
    print("A new critter has been born!")
    self.name = name
```

- So, in the main part: **crit1 = Critter("Poochie")** results in the creation of a new Critter object with an attribute **name** set to "Poochie". And the object is assigned to crit1.

- As the 1st parameter in every method, self automatically receives a reference to the object invoking the method. This means that, through self, a method can get at the object invoking it and access the object’s attributes and methods.

- You shouldn’t name the 1st parameter in a method header something other than self.

```
        self.name = name
```

creates the attribute **name** for the object and sets it to the value of name, which is "Poochie".

```
crit2 = Critter("Randolph")
```

kicks off the same basic chain of events. But this time, a new Critter object is created with its own attribute name set to "Randolph". And the object is assigned to crit2.

Accessing Attributes

- Get the 1st critter to say hi by invoking its talk() method:

```
crit1.talk()
```

- talk() receives the automatically sent reference to the object into its self parameter:

```
    def talk(self):
        print("Hi.  I'm", self.name, "\n")
```

- By default, we can access and modify an object’s attributes outside of its class. In the main part, we directly accessed the name attribute of crit1:

```
print(crit1.name)
```

- Usually, you want to avoid directly accessing an object’s attributes outside of its class definition.

Printing an Object

- If we were to print an object with print(crit1), Python would come back with something like the cryptic:

```
<__main__.Critter object at 0x00A0BA90>
```

- This tells us that we’ve printed a Critter object, but doesn’t give me any useful information about the object.

- By including the special method **__str__()** in a class definition, we can create a string for the objects that will be displayed whenever one is printed:

```
def __str__(self):
    rep = "Critter object\n"
    rep += "name: " + self.name + "\n"
    return rep
```

- **__str__()** returns a string that includes the value of the object’s name attribute. So

```
print(crit1)
```

gives

```
Critter object
name: Poochie
```

- Even if you never plan to print an object in your program, creating a **__str__()** is still not a bad idea. You may find that being able to see the values of an object’s attributes helps you understand how a program is working (or not working).

Using Class Attributes & Static Methods

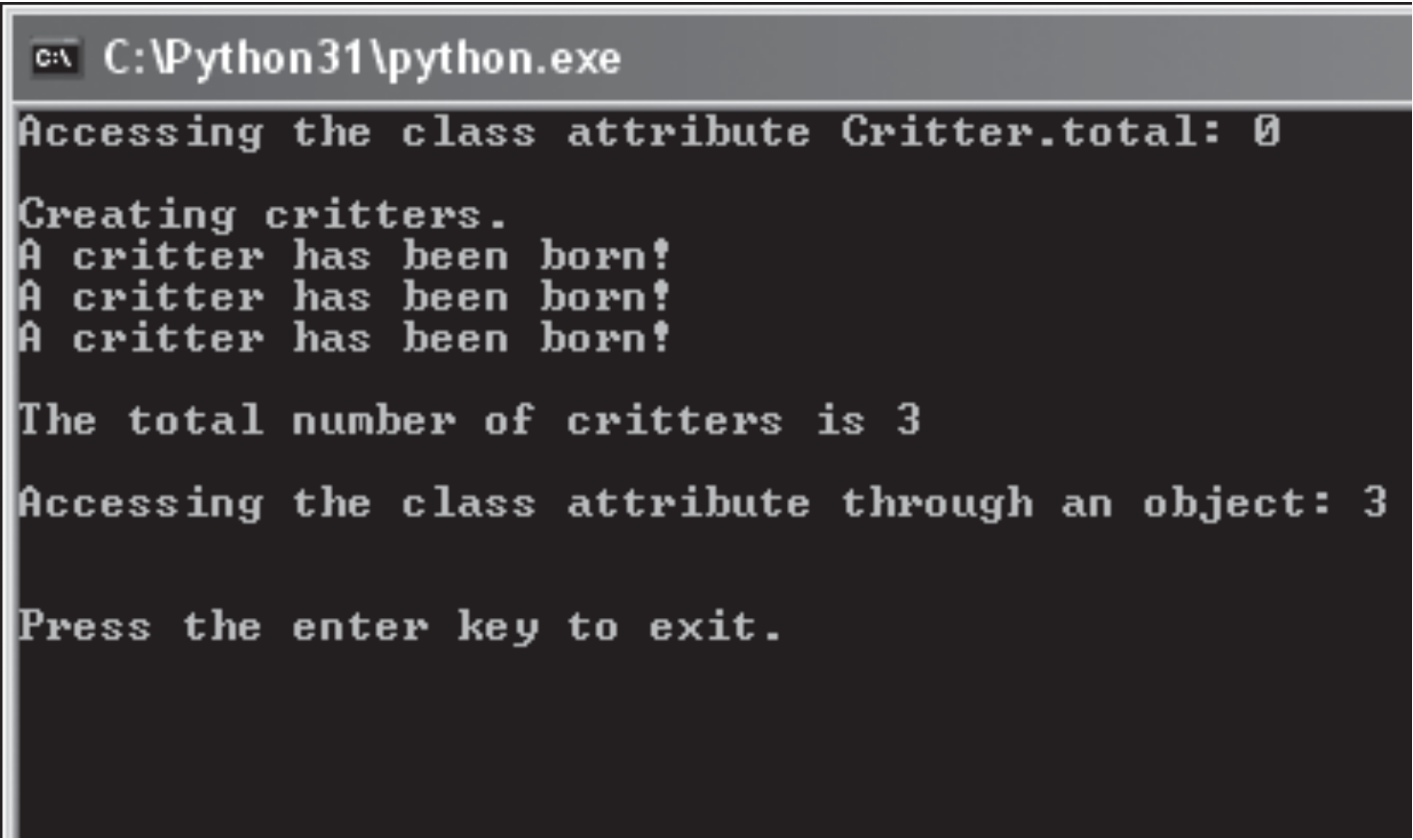
- Through attributes, different objects of the same class can each have their own, unique values.

- But you may have some information that relates not to individual objects, but the entire class.

- Python offers a way to create a single value associated with a class itself, called a *class attribute*.

- A method that’s associated with the class is called *static method*. They’re often used to work with class attributes.

Introducing the Classy Critter Program



```
Accessing the class attribute Critter.total: 0

Creating critters.
A critter has been born!
A critter has been born!
A critter has been born!

The total number of critters is 3

Accessing the class attribute through an object: 3


Press the enter key to exit.
```

classy_critter.py

```python
# Classy Critter
# Demonstrates class attributes and static methods

class Critter(object):
    """A virtual pet"""
    total = 0

    @staticmethod
    def status():
        print("\nThe total No. of critters is", Critter.total)

    def __init__(self, name):
        print("A critter has been born!")
        self.name = name
        Critter.total += 1
```

```
#main
print("Accessing the attribute Critter.total:", end=" ")
print(Critter.total)

print("\nCreating critters.")
crit1 = Critter("critter 1")
crit2 = Critter("critter 2")
crit3 = Critter("critter 3")

Critter.status()

print("\nAccessing the class attribute through an \
object:", end= " ")
print(crit1.total)

input("\n\nPress the enter key to exit.")
```

Creating a Class Attribute

```
total = 0
```

creates a class attribute total and assigns 0 to it.

- Any assignment statement like this—a variable assigned a value outside of a method—creates a class attribute.

- The assignment statement is executed only once, when Python first sees the class definition.

- Thus the class attribute exists even before a single object is created.

- So you can use a class attribute without any objects of the class in existence.

Accessing a Class Attribute

- In the main part:

```
print(Critter.total)
```

- In the static method status():

```
        print("\nThe total No. of critters is", Critter.total)
```

- In the constructor method:

```
        Critter.total += 1
```

- Every time a new object is instantiated, the value of the attribute is incremented by 1.

- In general, to access a class attribute, type the class name, followed by a dot, followed by the attribute name.

- Finally, you can access a class attribute through an object of that class:

```
print(crit1.total)
```

- This line prints the value of the class attribute total and not an attribute of the object itself.

- You can read the value of a class attribute through any object that belongs to that class.

- Although you can use an object of a class to access a class attribute, you can’t assign a new value to a class attribute through an object.

- If you want to change the value of a class attribute, access it through its class name.

Creating a Static Method

```
    def status():
        print("\nThe total No. of critters is", Critter.total)
```

the definition is part of creating a static method. Notice that the definition doesn’t have self in its parameter list.

- That’s because it’s designed to be invoked through a class, not an object. So, the method won’t be passed a reference to an object and therefore won’t need a parameter, like self, to receive such a reference.

- A *decorator* is put before the definition. This decorator creates a static method with the same name:

```
    @staticmethod
```

- The class now has a static method, status(), showing the Critter objects’ number by printing the class attribute total.

- To create your static method, begin with **@staticmethod** decorator, followed by the class method definition. And since the method is for the entire class, you won’t include the self parameter, necessary only for object methods.

Invoking a Static Method

- Invoke the static method with:

Critter.status()

- notice that we are able to invoke the method without a single object in existence.

- Since static methods are invoked through a class, no objects of the class need to exist before you can invoke them.

- After creating 3 objects. We invoke status() again, which prints a message stating that 3 critters exist.

- This works because, during the constructor method for each object, the class attribute total is increased by 1.

Understanding Object Encapsulation

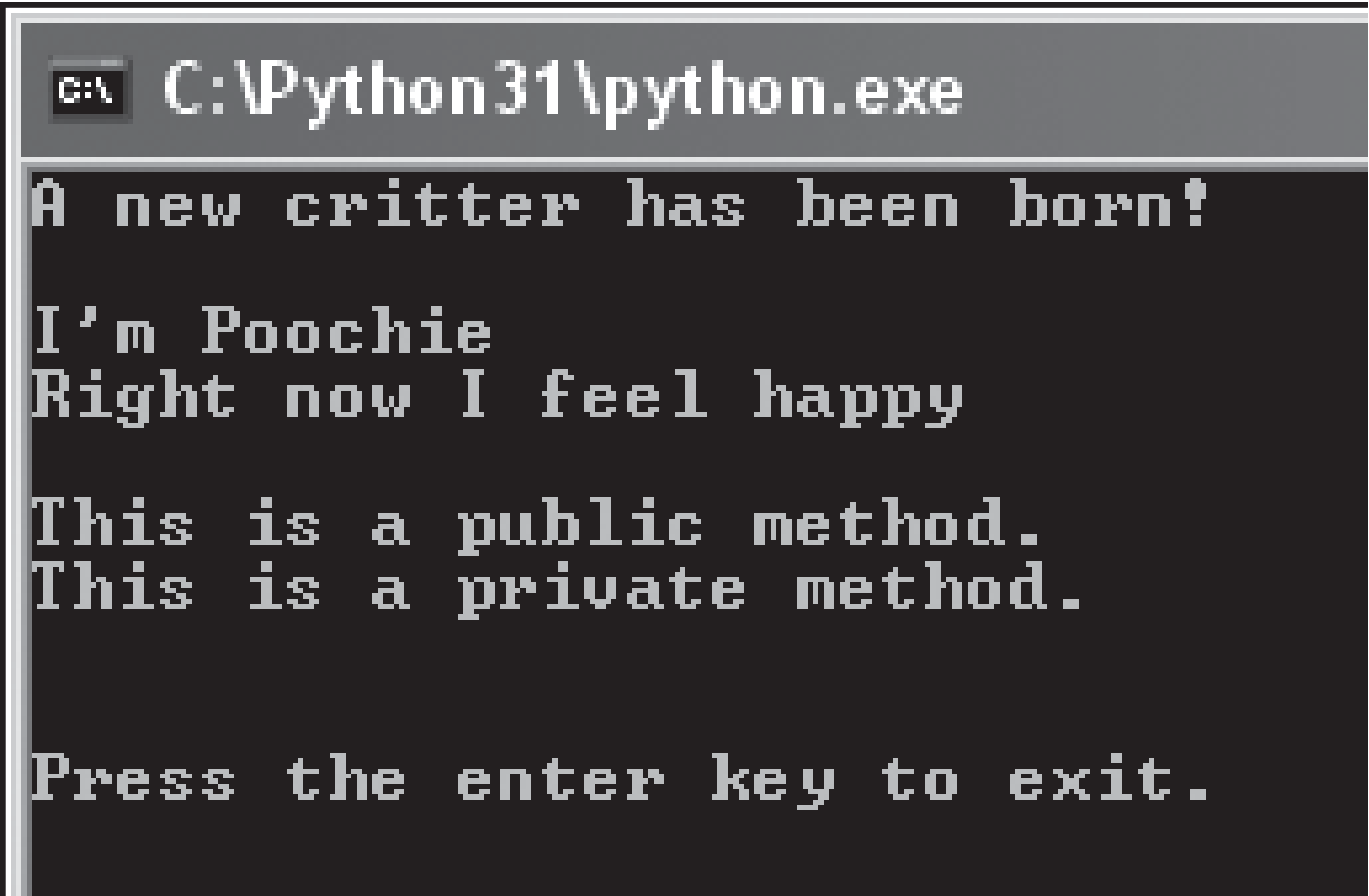
- For the function encapsulation, functions are encapsulated and hide the details of their inner workings from the part of your program that calls it (called the *client* of the function).

- The client of a well-defined function communicates with the function only through its parameters and return values.

- Objects should be treated the same way. Client code should avoid directly altering the value of an object’s attribute.

- Altering directly an object’s attribute by a careless client could cause a catastrophic consequence. Employing a safe method offered by the class instead can avoid the situation from happening.

Introducing the Private Critter Program



```
A new critter has been born!

I'm Poochie
Right now I feel happy

This is a public method.
This is a private method.


Press the enter key to exit.
```

private_critter.py

```
# Private Critter
# Demonstrates private variables and methods

class Critter(object):
    """A virtual pet"""
    def __init__(self, name, mood):
        print("A new critter has been born!")
        self.name = name                # public attribute
        self.__mood = mood              # private attribute

    def talk(self):
        print("\nI'm", self.name)
        print("Right now I feel", self.__mood, "\n")

    def __private_method(self):
        print("This is a private method.")
    def public_method(self):
        print("This is a public method.")
        self.__private_method()

# main
crit = Critter(name = "Poochie", mood = "happy")
crit.talk()
crit.public_method()

input("\n\nPress the enter key to exit.")
```

Creating Private Attributes

- By default, all of an object's attributes/methods are *public*, ie, they can be directly accessed or invoked by a client.

- For encapsulation, attributes/methods can be defined as *private*, ie, only other methods of the object itself can access and invoke them.

- In the constructor method, we create 2 attributes, one public and one private:

```
self.name = name            # public attribute
self.__mood = mood          # private attribute
```

- The 2 underscore characters that begin the 2nd attribute name tell Python that this is a private attribute.

- To create a private attribute, including class attributes, just begin the attribute name with 2 underscores.

Accessing Private Attributes

- It’s perfectly fine to access an object’s private attribute inside the class definition of the object:

```
    def talk(self):
        print("\nI'm", self.name)
        print("Right now I feel", self.__mood, "\n")
```

- If we tried to access this attribute outside of the Critter class definition, we’d have trouble:

```
>>> crit = Critter(name = "Poochie", mood = "happy")
A new critter has been born!
>>> print(crit.mood)
Traceback (most recent call last):
   File "<pyshell#1>", line 1, in <module>
      print(crit.mood)
AttributeError: 'Critter' object has no attribute 'mood'
```

- Another trial:

```
>>> print(crit.__mood)
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    print(crit.__mood)
AttributeError: 'Critter' object has no attribute '__mood'
```

- A private attribute is still accessible outside. Python hides the attribute through a special naming convention:

```
>>> print(crit._Critter__mood)
happy
```

- Privacy is an indicator that the attribute or method is only for an object’s internal use. In addition it helps prevent inadvertent access to such an attribute or method. So don’t try to directly access the private attributes or methods of an object from outside of its class definition.

Creating Private Methods

```
def __private_method(self):
    print("This is a private method.")
```

- This is a private method but it can easily be accessed by any other method in the class.

- Like private attributes, private methods are meant only to be accessed by an object’s own methods.

Accessing Private Methods

- Just as with private attributes, accessing an object’s private methods within its class definition is simple:

```
    def public_method(self):
        print("This is a public method.")
        self.__private_method()
```

- Like private attributes, private methods aren’t meant to be directly accessed by clients:

```
>>> crit.private_method
Traceback (most recent call last):
    File "<pyshell#0>", line 1, in <module>
        crit.private_method
AttributeError: 'Critter' object has no attribute
                'private_method'
```

- Another trial:

```
>>> crit.__private_method()
Traceback (most recent call last):
    File "<pyshell#1>", line 1, in <module>
        crit.__private_method()
AttributeError: 'Critter' object has no attribute
                         '__private_method'
```

- Just as with private attributes, it is technically possible to access private methods from anywhere in a program:

```
>>> crit._Critter__private_method()
This is a private method.
```

- A client should never attempt to directly access an object’s private methods.

- You can create a private static method by beginning the method’s name with 2 underscores.

Respecting an Object’s Privacy

In the main part, we create an object and invoke its 2 public methods, without prodding into the object’s private attributes or methods:

```
# main
crit = Critter(name = "Poochie", mood = "happy")
crit.talk()
crit.public_method()

input("\n\nPress the enter key to exit.")
```

Understanding When to Implement Privacy

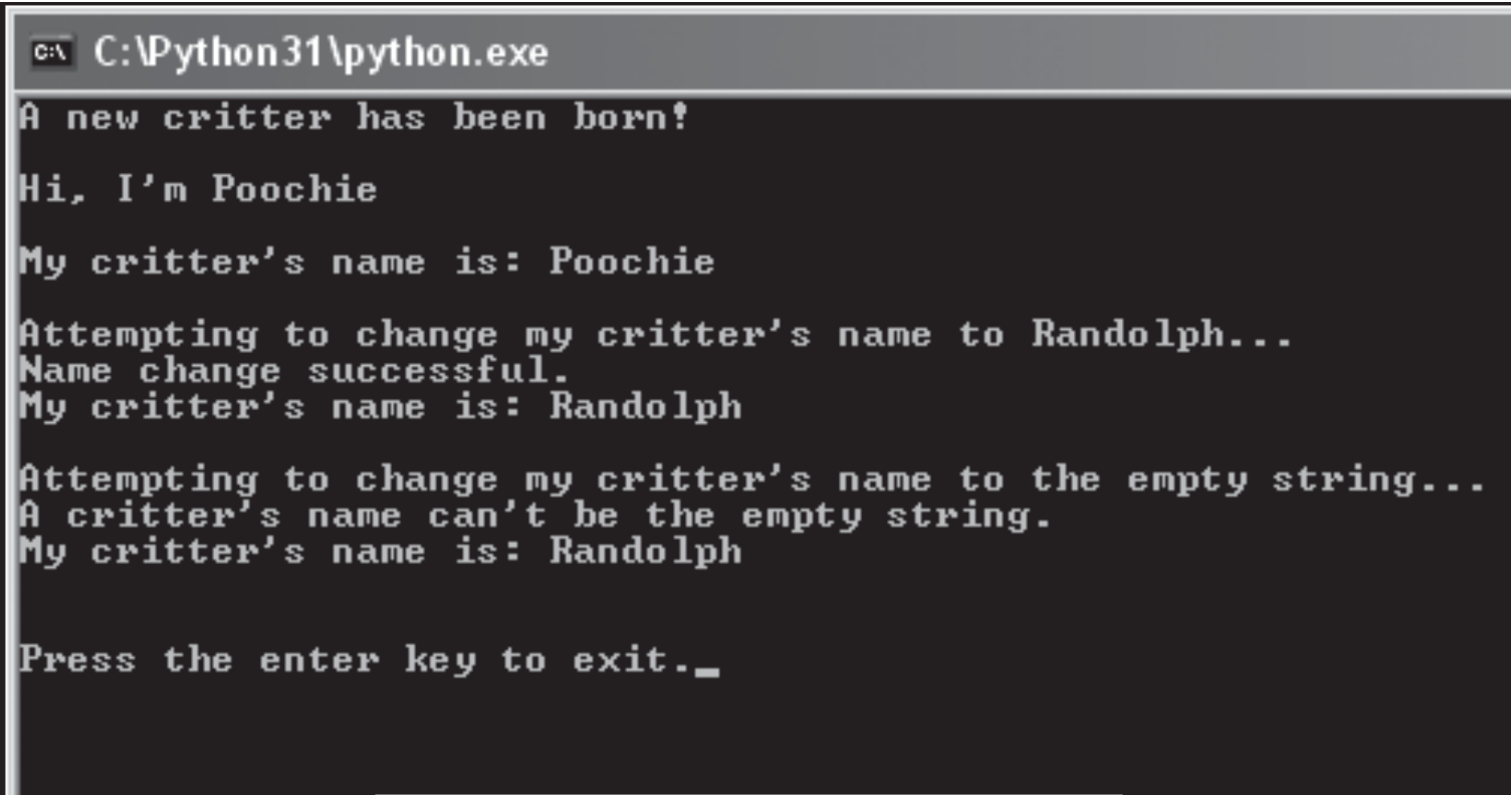
- Make private any method you don’t want a client to invoke. If it’s critical that an attribute never be directly accessed by a client, you can make it private.

- The philosophy among many Python programmers is to trust that clients will use an object’s methods and not directly alter its attributes.

- When you write a class:
  * Create methods to reduce the need for clients to directly access an object’s attributes.
  * Use privacy for those attributes and methods that are completely internal to the operation of objects.

- When you use an object:
  * Minimize the direct reading of an object’s attributes.
  * Avoid directly altering an object’s attributes.
  * Never attempt to directly access an object’s private attributes or methods.

Introducing the Property Critter Program



```
A new critter has been born!

Hi, I'm Poochie

My critter's name is: Poochie

Attempting to change my critter's name to Randolph...
Name change successful.
My critter's name is: Randolph

Attempting to change my critter's name to the empty string...
A critter's name can't be the empty string.
My critter's name is: Randolph


Press the enter key to exit._
```

property_critter.py

```
# Property Critter
# Demonstrates properties

class Critter(object):
    """A virtual pet"""
    def __init__(self, name):
        print("A new critter has been born!")
        self.__name = name

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

    @name.setter
    def name(self, new_name):
        if new_name == "":
            print("A critter's name can't be empty.")
        else:
            self.__name = new_name
            print("Name change successful.")

    def talk(self):
        print("\nHi, I'm", self.name)

# main
crit = Critter("Poochie")
crit.talk()

print("\nMy critter's name is:", end= " ")
print(crit.name)

print("\nTrying my critter's name to Randolph...")
crit.name = "Randolph"
print("My critter's name is:", end= " ")
print(crit.name)
print("\nTrying to change the name to be empty...")
crit.name = ""
print("My critter's name is:", end= " ")
print(crit.name)

input("\n\nPress the enter key to exit.")
```

Creating Properties

- One way to control access to a private attribute is to create a *property*—an object with methods that allow indirect access to attributes and often impose restriction on that access:

```
@property
def name(self):
    return self.__name
```

- We create the property by writing a method that returns the value I want to provide indirect access to and precede the method definition with the **@property** decorator.

- The property has the same name as the method—in this case, name. Now we can use the name property of any Critter object to get the value of the object's private __name attribute, inside or outside the class definition using the familiar dot notation.

- To create a property, write a method that returns the value you want to provide indirect access to and precede the method definition with the **@property** decorator. The property will have the same name as the method.

- By creating a property, you can provide read access to a private attribute. A property can also provide write access—and even impose some limits on that access:

```
@name.setter
def name(self, new_name):
    if new_name == "":
        print("A critter's name can't be empty.")
    else:
        self.__name = new_name
        print("Name change successful.")
```

- **@name.setter** accesses the **setter** attribute of the name property.

- It means that the following method definition will provide a way to set the value of the property name.

- You can create your own decorator for setting a property value: start with the @ symbol, followed by the name of the property, followed by a dot (.), followed by **setter**.

- This method is called name just like the property; it has to be. When establishing a setter method in this way, the method must have the same name as the property.

- When you create a method for setting the value of a property, the method definition must have a parameter to receive the new value.

Accessing Properties

- By creating the name property, I can get the name of a critter through dot notation:

```
    def talk(self):
        print("\nHi, I'm", self.name)

# main
crit = Critter("Poochie")
crit.talk()
```

- The code self.name accesses the name property and indirectly calls the method that returns __name.

- Not only can we use the name property of an object inside its class definition, but we can also use it outside the definition:

```
print("\nMy critter's name is:", end= " ")
print(crit.name)
```

- Although this code is outside the Critter class, the code crit.name accesses the name property of the Critter object and indirectly calls the method that returns __name.

```
crit.name = "Randolph"
```

accesses the name property of the object and indirectly calls the method that attempts to set __name.

- Display the critter’s name using the name property:

```
print("My critter's name is:", end= " ")
print(crit.name)
```

gives My critter's name is Randolph.

- Attempt to change the critter’s name to the empty string:

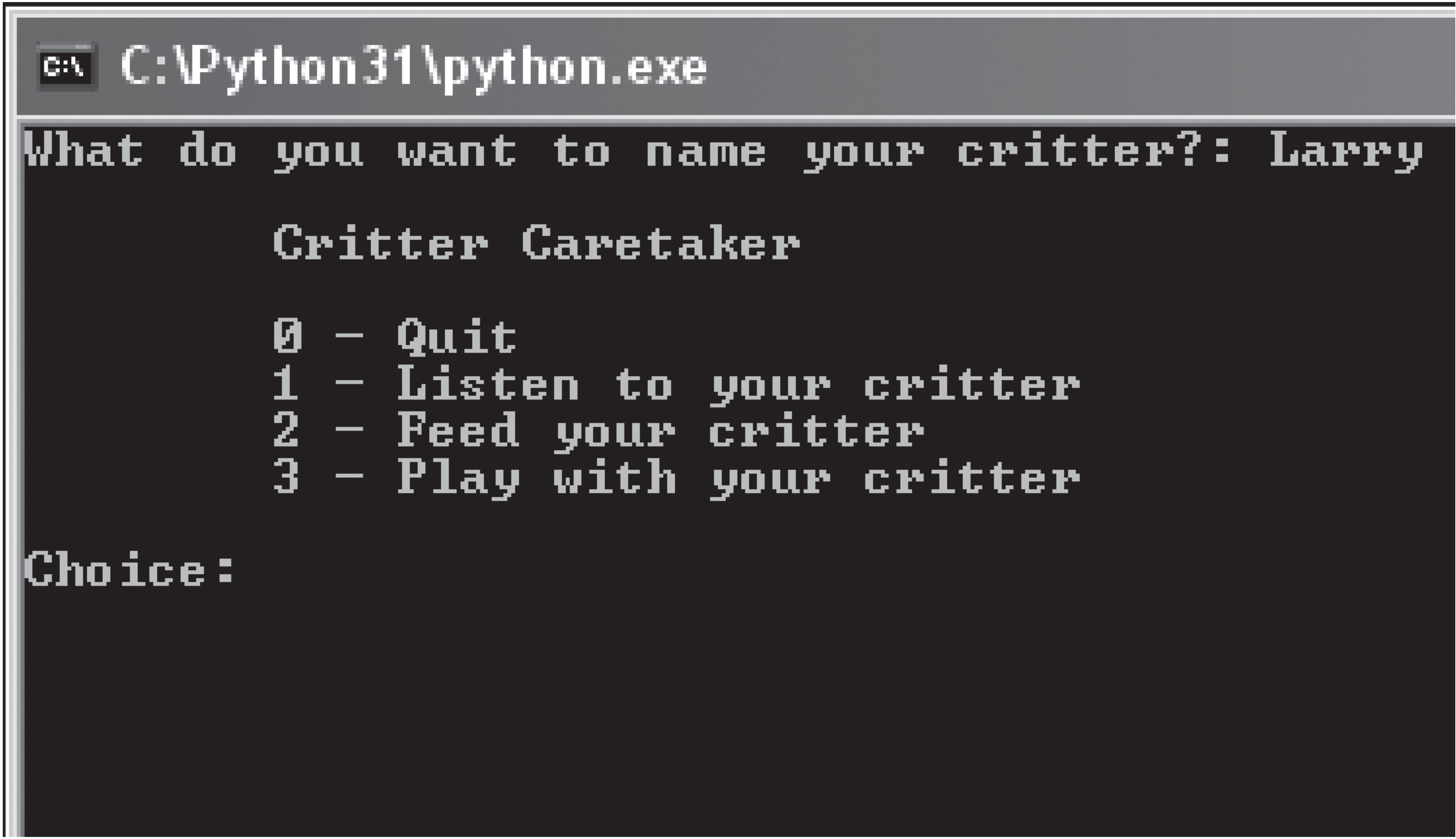
```
print("\nTrying to change the name to be empty...")
crit.name = ""
```

gives A critter's name can't be the empty string and the object’s __name attribute remains unchanged.

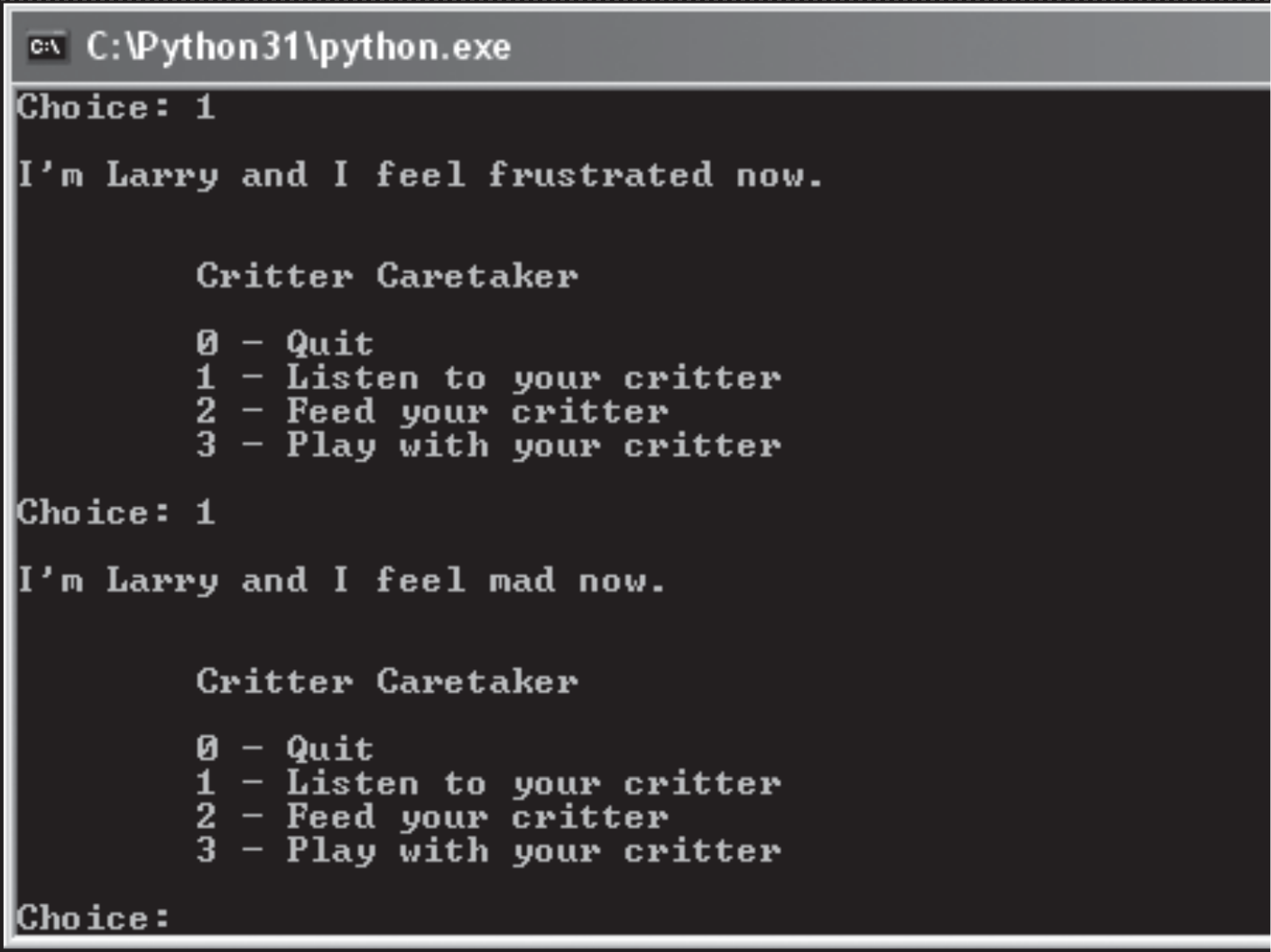
- Finally, check if the critter’s name hasn’t been changed to the empty string:

```
print("My critter's name is:", end= " ")
print(crit.name)
```

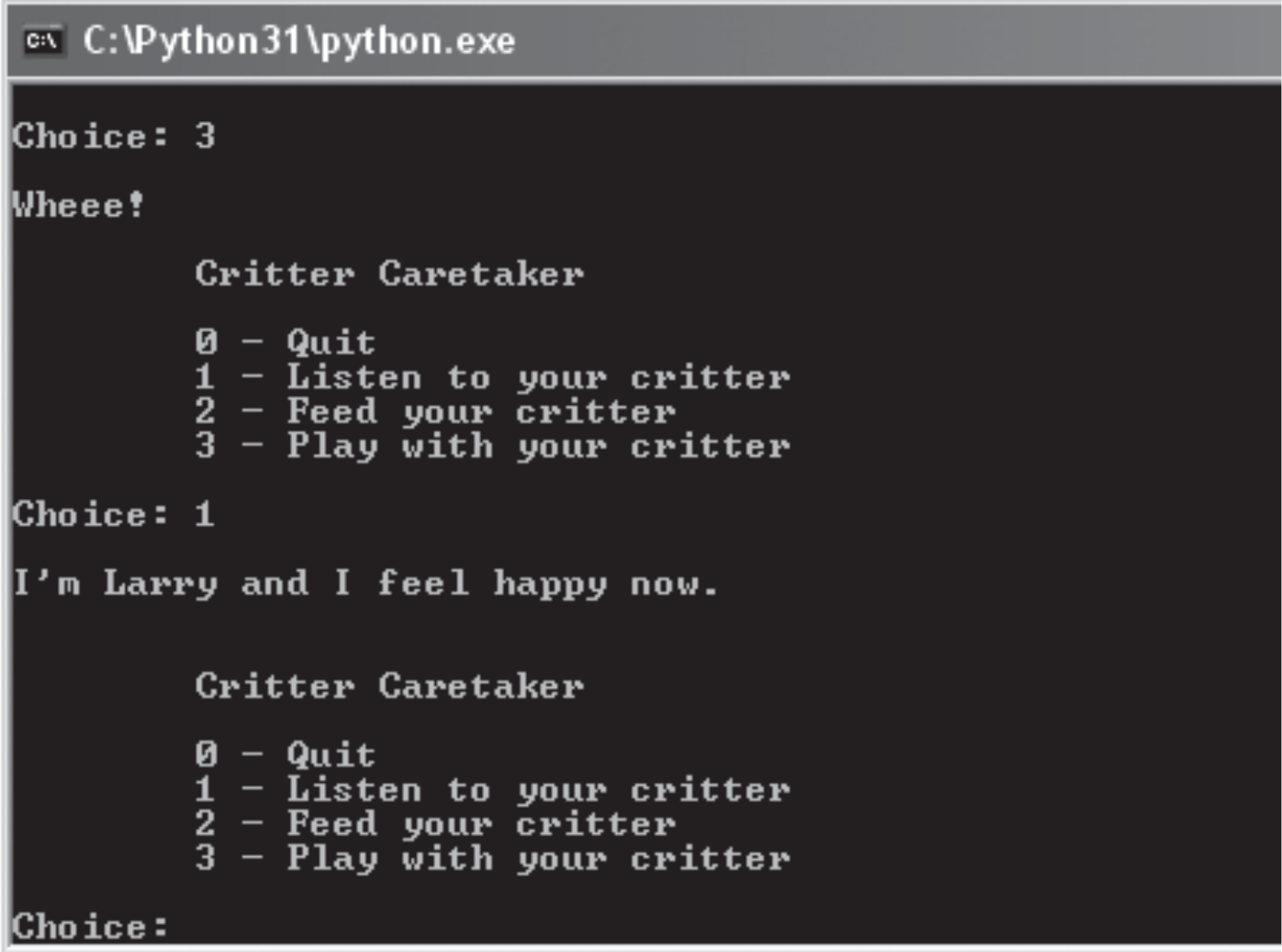
Introducing the Critter Caretaker Program



```
What do you want to name your critter?: Larry

        Critter Caretaker

        0 - Quit
        1 - Listen to your critter
        2 - Feed your critter
        3 - Play with your critter

Choice:
```



```
Choice: 1

I'm Larry and I feel frustrated now.


         Critter Caretaker

         0 - Quit
         1 - Listen to your critter
         2 - Feed your critter
         3 - Play with your critter

Choice: 1

I'm Larry and I feel mad now.


         Critter Caretaker

         0 - Quit
         1 - Listen to your critter
         2 - Feed your critter
         3 - Play with your critter

Choice:
```



```
Choice: 3

Wheee!

        Critter Caretaker

        0 - Quit
        1 - Listen to your critter
        2 - Feed your critter
        3 - Play with your critter

Choice: 1

I'm Larry and I feel happy now.


        Critter Caretaker

        0 - Quit
        1 - Listen to your critter
        2 - Feed your critter
        3 - Play with your critter

Choice:
```

critter_caretaker.py

```
# Critter Caretaker
# A virtual pet to care for

class Critter(object):
    """A virtual pet"""
    def __init__(self, name, hunger = 0, boredom = 0):
        self.name = name
        self.hunger = hunger
        self.boredom = boredom

    def __pass_time(self):
        self.hunger += 1
        self.boredom += 1
```

```
@property
def mood(self):
    unhappiness = self.hunger + self.boredom
    if unhappiness < 5:
        m = "happy"
    elif 5 <= unhappiness <= 10:
        m = "okay"
    elif 11 <= unhappiness <= 15:
        m = "frustrated"
    else:
        m = "mad"
    return m

def talk(self):
    print("I'm", self.name, "and I feel", self.mood,
             "now.\n")
    self.__pass_time()
    def eat(self, food = 4):
        print("Brruppp.  Thank you.")
        self.hunger -= food
        if self.hunger < 0:
            self.hunger = 0
        self.__pass_time()

    def play(self, fun = 4):
        print("Wheee!")
        self.boredom -= fun
        if self.boredom < 0:
            self.boredom = 0
        self.__pass_time()


def main():
    crit_name = input("What’s your critter’s name?:")
    crit = Critter(crit_name)
    choice = None
    while choice != "0":
        print \
        ("""
        Critter Caretaker

        0 - Quit
        1 - Listen to your critter
        2 - Feed your critter
        3 - Play with your critter
        """)
        choice = input("Choice: ")
        print()

        # exit
        if choice == "0":
            print("Good-bye.")

        # listen to your critter
        elif choice == "1":
            crit.talk()
        # feed your critter
        elif choice == "2":
            crit.eat()

        # play with your critter
        elif choice == "3":
            crit.play()

        # some unknown choice
        else:
            print("\nSorry,", choice, "isn't a valid choice.")

main()
("\n\nPress the enter key to exit.")
```

The Constructor Method

The constructor method of the class initializes the 3 public attributes of a Critter object: name, hunger, boredom. Notice that hunger and boredom have default values of 0, allowing a critter to start off in a very good mood:

```
class Critter(object):
    """A virtual pet"""
    def __init__(self, name, hunger = 0, boredom = 0):
        self.name = name
        self.hunger = hunger
        self.boredom = boredom
```

The __pass_time() Method

- __pass_time() is a private method that increases a critter’s hunger and boredom levels.

- It’s invoked at the end of each method where the critter does something (eats, plays, or talks) to simulate the passage of time:

```
def __pass_time(self):
    self.hunger += 1
    self.boredom += 1
```

The mood Property

- mood represents a critter’s mood. It adds the values of a Critter object’s hunger and boredom attributes and, based on the total, returns "happy", "okay", "frustrated", or "mad".

- mood doesn’t provide access to a private attribute because the string representing a critter’s mood is not a part of the Critter object, but is calculated on the fly:

```
@property
def mood(self):
    unhappiness = self.hunger + self.boredom
    if unhappiness < 5:
        m = "happy"
    elif 5 <= unhappiness <= 10:
        m = "okay"
    elif 11 <= unhappiness <= 15:
        m = "frustrated"
    else:
        m = "mad"
    return m
```

The talk() Method

- talk() announces a critter’s mood by accessing the Critter object’s mood property. Then it invokes __pass_time():

```
def talk(self):
    print("I'm", self.name, "and I feel", self.mood,
          "now.\n")
    self.__pass_time()
```

The eat() Method

- eat() reduces a critter’s hunger level by an amount passed to food. food’s default value is 4. The critter’s hunger level is kept in check and not allowed to go below 0. Finally, the method invokes __pass_time():

```
def eat(self, food = 4):
    print("Brruppp.  Thank you.")
    self.hunger -= food
    if self.hunger < 0:
        self.hunger = 0
    self.__pass_time()
```

The play() Method

- play() reduces the critter’s boredom level by an amount passed to fun. fun’s default value is 4. The critter’s boredom level is kept in check and not allowed to go below 0. Finally, the method invokes __pass_time():

```
def play(self, fun = 4):
    print("Wheee!")
    self.boredom -= fun
    if self.boredom < 0:
        self.boredom = 0
    self.__pass_time()
```

Creating the Critter

- We put the main part of the program into main().

- At the start of the program, we get the name of the critter from the user. Next, we instantiate a new Critter object.

- Since we don’t supply values for hunger or boredom, the attributes start out at 0:

```
def main():
    crit_name = input("What is your critter’s name?: ")
    crit = Critter(crit_name)
```

Creating a Menu System

- We then created a menu system. If 0 is entered, the code ends. If 1 is entered, the object’s talk() is invoked. If 2 is entered, the object’s eat() is invoked. If 3 is entered, the object’s play() is invoked. If anything else is entered, the code shows the choice is invalid:

```
choice = None
while choice != "0":
    print \
    ("""
    Critter Caretaker

    0 - Quit
    1 - Listen to your critter
    2 - Feed your critter
    3 - Play with your critter
    """)
```

```
choice = input("Choice: ")
print()

# exit
if choice == "0":
    print("Good-bye.")

# listen to your critter
elif choice == "1":
    crit.talk()
# feed your critter
elif choice == "2":
    crit.eat()

# play with your critter
elif choice == "3":
    crit.play()

# some unknown choice
else:
    print("\nSorry,", choice, "isn't a valid choice.")
```

Starting the Program

- The next line of code calls main() and begins the program:

```
main()
```