BIFX 502 Foundations in Computer Science

Chapter 10: Classes and Object-Oriented Programming

Dr. Jim

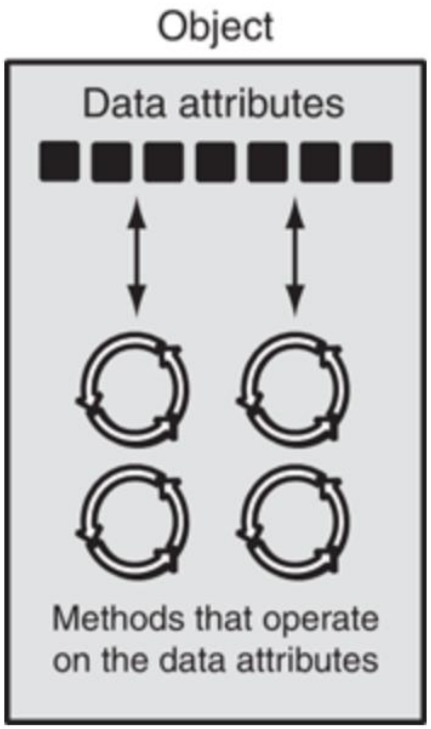
Hood College

Procedural Programming

- **Procedural programming: writing programs made of functions that perform specific tasks**
  - Procedures typically operate on data items that are separate from the procedures
  - Data items commonly passed from one procedure to another
  - Focus: to create procedures that operate on the program's data

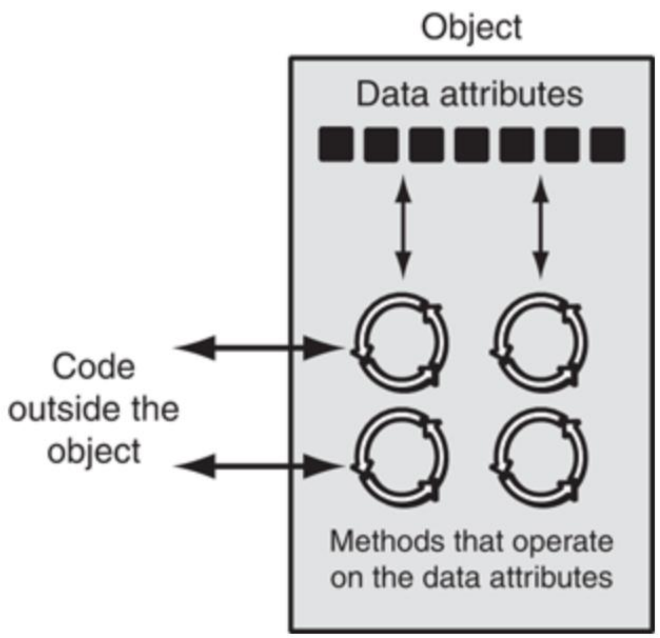
Object-Oriented Programming

- **<u>Object-oriented programming</u>: focused on creating objects**
- **<u>Object</u>: entity that contains data and procedures**
  - Data is known as data attributes and procedures are known as methods
    - Methods perform operations on the data attributes



Object-Oriented Programming (cont'd.)

- **Encapsulation: combining data and code into a single object**
- **Data hiding: object's data attributes are hidden from code outside the object**
  - Access restricted to the object's methods
- **Object reusability: the same object can be used in different programs**



An Everyday Example of an Object

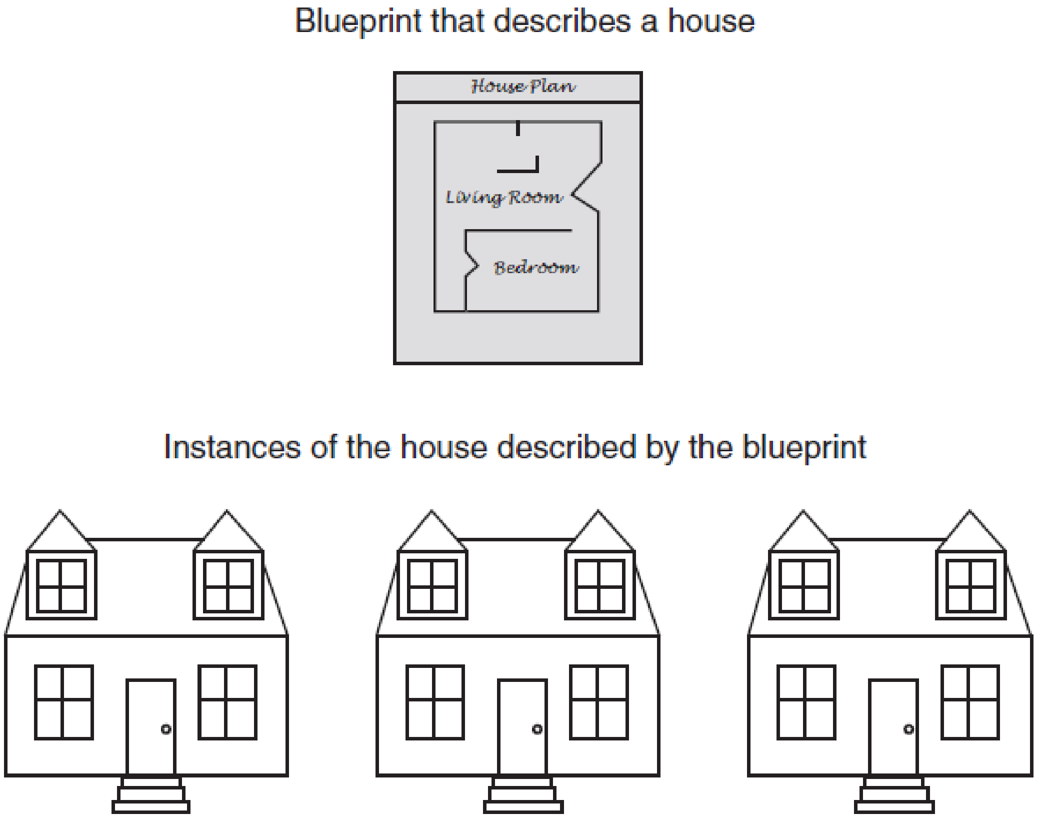
- **<u>Data attributes</u>: define the state of an object**
  - Example: clock object would have `second`, `minute`, and `hour` data attributes
- **<u>Public methods</u>: allow external code to manipulate the object**
  - Example: `set_time, set_alarm_time`
- **<u>Private methods</u>: used for object’s inner workings**
  - Example: `increment_current_second`, `increment_current_minute`, `increment_current_hour`

Classes

- **Class: code that specifies the data attributes and methods of a particular type of object**
  - Similar to a blueprint of a house or a cookie cutter
- **Instance: an object created from a class**
  - Similar to a specific house built according to the blueprint or a specific cookie
  - There can be many instances of one class

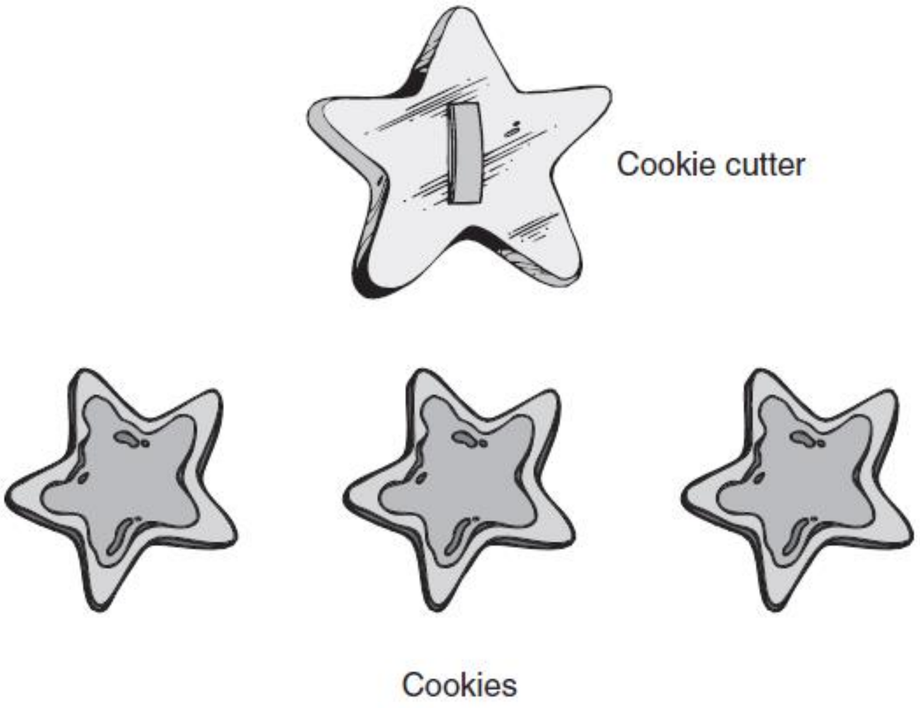
Classes (cont'd.)

Figure 10-3 A blueprint and houses built from the blueprint


Blueprint that describes a house
House Plan
Living Room
Bedroom
Instances of the house described by the blueprint

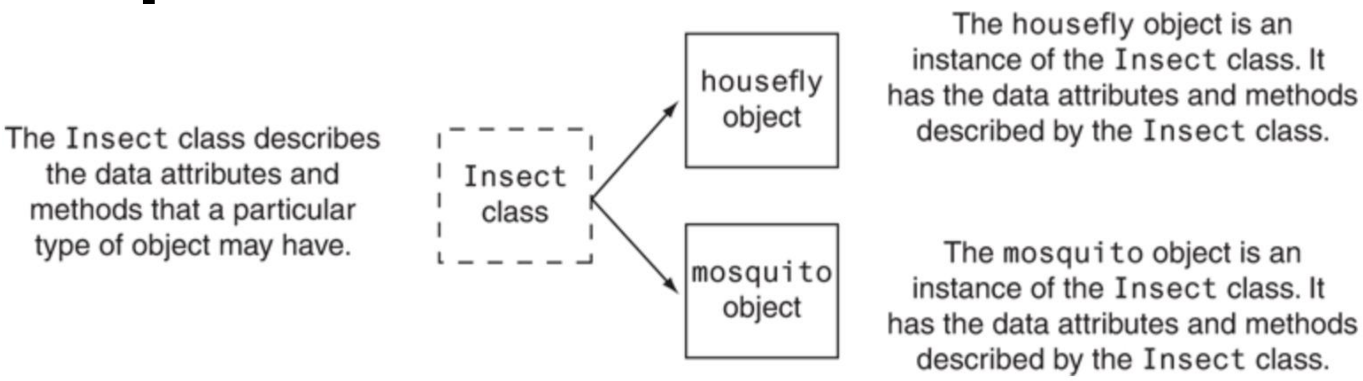
Classes (cont'd.)

Figure 10-4 The cookie cutter metaphor



Classes (cont'd.)

- **Example 1:**

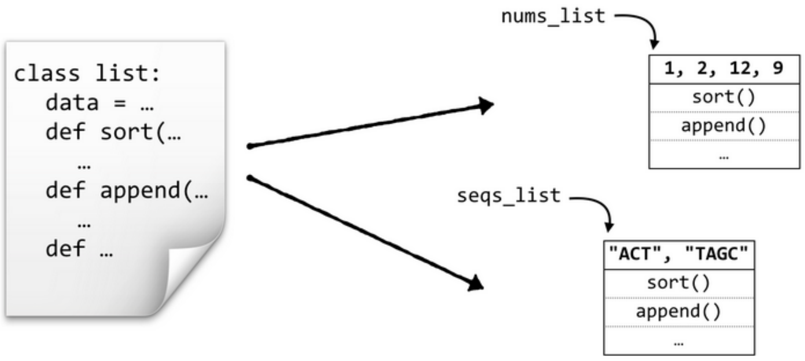


The Insect class describes the data attributes and methods that a particular type of object may have.

The housefly object is an instance of the Insect class. It has the data attributes and methods described by the Insect class.

The mosquito object is an instance of the Insect class. It has the data attributes and methods described by the Insect class.

- **Example 2:**



```
class list:
   data = …
   def sort(…
     …
   def append(…
     …
   def …
```

Class definition: code written
that defines the structure of objects

Objects: data and functions
in RAM, referenced by variables

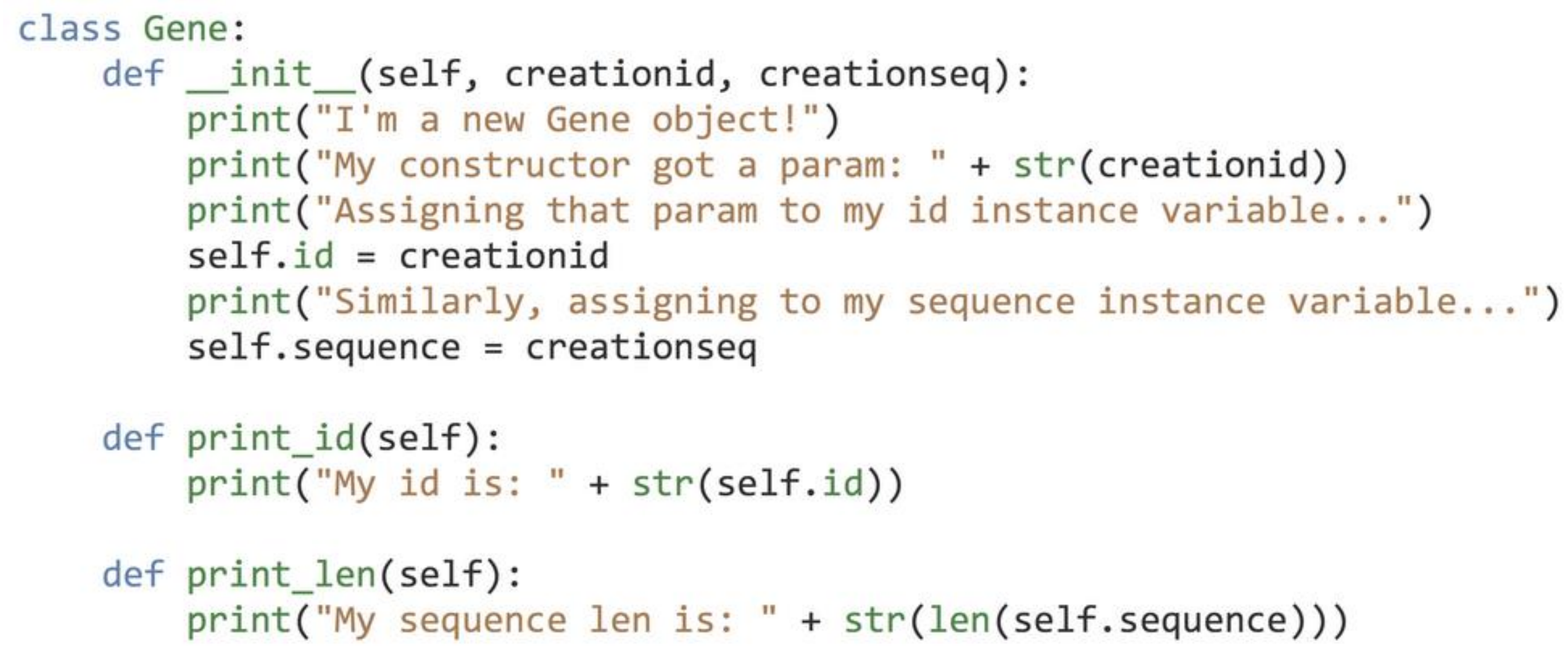
Class Definitions

- **Class definition: set of statements that define a class's methods and data attributes**
  - Format: begin with `class` *`Class_name`*`:`
    - Class names often start with uppercase letter
  - Method definition like any other Python function definition
    - `self` parameter: required in every method in the class – references the specific object that the method is working on

Class Definitions (cont'd.)

- **Initializer method: automatically executed when an instance of the class is created**
  - Initializes object's data attributes and assigns `self` parameter to the object that was just created
  - Format: `def __init__ (self):`
  - Usually the first method in a class definition

Class Definitions (cont’d.)



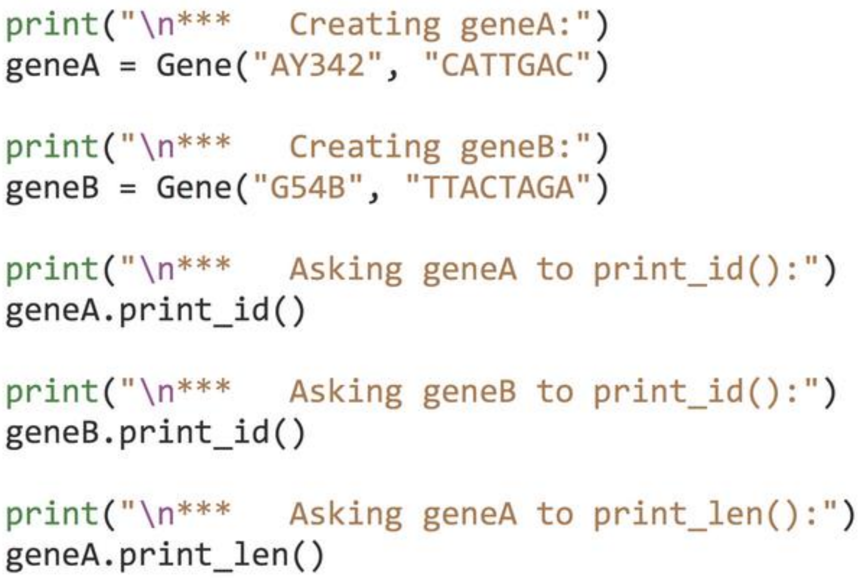
```
class Gene:
    def __init__(self, creationid, creationseq):
        print("I'm a new Gene object!")
        print("My constructor got a param: " + str(creationid))
        print("Assigning that param to my id instance variable...")
        self.id = creationid
        print("Similarly, assigning to my sequence instance variable...")
        self.sequence = creationseq

    def print_id(self):
        print("My id is: " + str(self.id))

    def print_len(self):
        print("My sequence len is: " + str(len(self.sequence)))
```

Class Definitions (cont'd.)

- **To create a new instance of a class call the initializer method**
  - Format: `My_instance = Class_Name()`
- **To call any of the class methods using the created instance, use dot notation**
  - Format: `My_instance.method()`
  - Because the `self` parameter references the specific instance of the object, the method will affect this instance
    - Reference to `self` is passed automatically

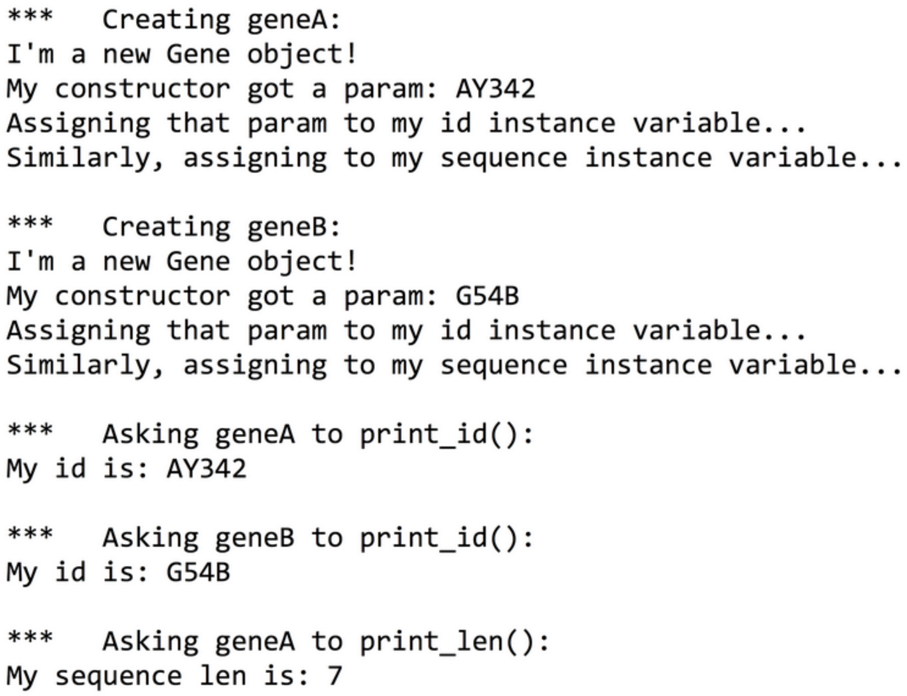
Class Definitions (cont’d.)

Create and interact with Gene objects:



```
print("\n***   Creating geneA:")
geneA = Gene("AY342", "CATTGAC")

print("\n***   Creating geneB:")
geneB = Gene("G54B", "TTACTAGA")

print("\n***   Asking geneA to print_id():")
geneA.print_id()

print("\n***   Asking geneB to print_id():")
geneB.print_id()

print("\n***   Asking geneA to print_len():")
geneA.print_len()
```

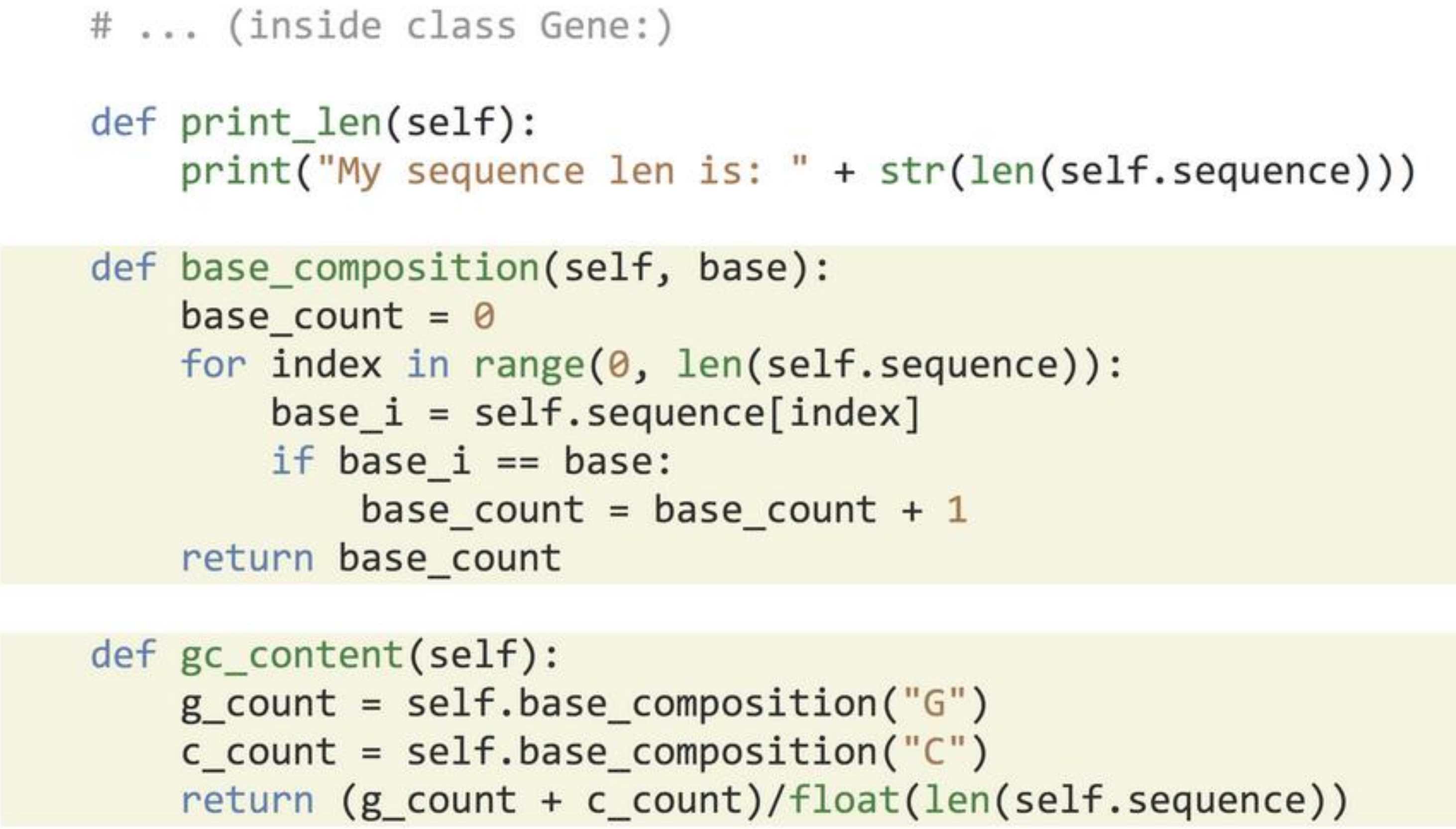
Output:



```
***   Creating geneA:
I'm a new Gene object!
My constructor got a param: AY342
Assigning that param to my id instance variable...
Similarly, assigning to my sequence instance variable...

***   Creating geneB:
I'm a new Gene object!
My constructor got a param: G54B
Assigning that param to my id instance variable...
Similarly, assigning to my sequence instance variable...

***   Asking geneA to print_id():
My id is: AY342

***   Asking geneB to print_id():
My id is: G54B

***   Asking geneA to print_len():
My sequence len is: 7
```

Class Definitions (cont’d.)

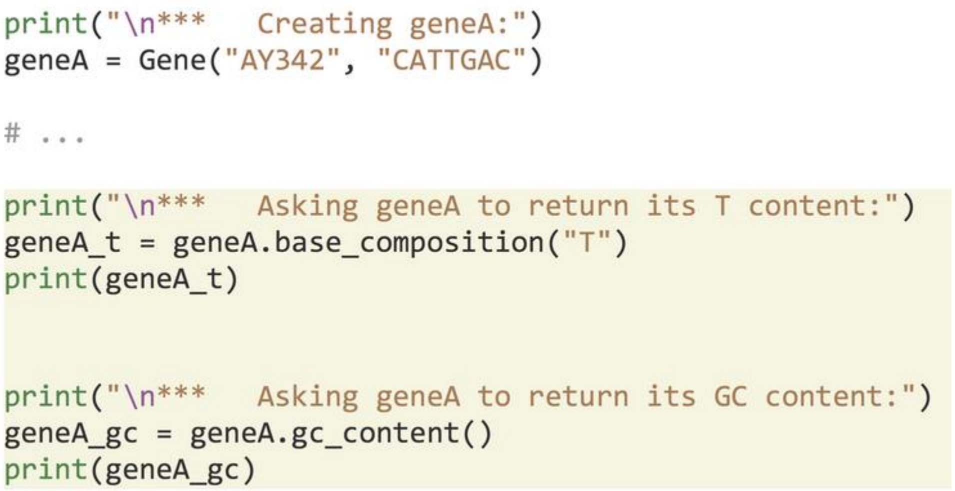
- **Expanding the Gene class**



```
    # ... (inside class Gene:)

    def print_len(self):
        print("My sequence len is: " + str(len(self.sequence)))

    def base_composition(self, base):
        base_count = 0
        for index in range(0, len(self.sequence)):
            base_i = self.sequence[index]
            if base_i == base:
                base_count = base_count + 1
        return base_count

    def gc_content(self):
        g_count = self.base_composition("G")
        c_count = self.base_composition("C")
        return (g_count + c_count)/float(len(self.sequence))
```

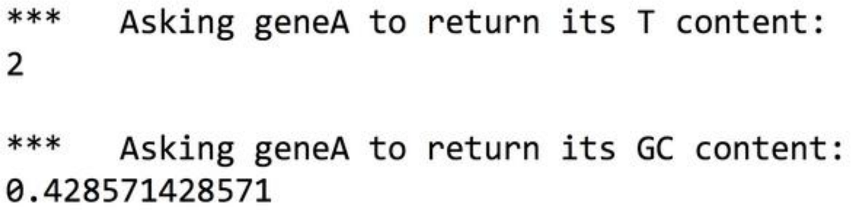
Class Definitions (cont'd.)

Create and interact with Gene objects:



```
print("\n***   Creating geneA:")
geneA = Gene("AY342", "CATTGAC")

# ...

print("\n***   Asking geneA to return its T content:")
geneA_t = geneA.base_composition("T")
print(geneA_t)


print("\n***   Asking geneA to return its GC content:")
geneA_gc = geneA.gc_content()
print(geneA_gc)
```

Output:



```
***   Asking geneA to return its T content:
2

***   Asking geneA to return its GC content:
0.428571428571
```

Summary of Steps for Writing a Class Definition

- **1. Decide what concept or entity the objects of that class will represent, as well as what data (instance variables) and methods (functions) they will have**
- **2. Create a constructor method and have it initialize all of the instance variables**
- **3. Write methods that set or get the instance variables, compute calculations, call other methods or functions, and so on. Don’t forget the `self` parameter!**

Hiding Attributes and Storing Classes in Modules

- **An object’s data attributes should be private**
  - To make sure of this, place two underscores (__) in front of attribute name
    - Example: `__current_minute`
- **Classes can be stored in modules**
  - Filename for module must end in .py
  - Module can be imported to programs that use the class

The `__str__` method

- **<u>Object's state</u>: the values of the object's attribute at a given moment**
- **`__str__ method`: displays the object's state**
  - Automatically called when the object is passed as an argument to the `print` function
  - Automatically called when the object is passed as an argument to the `str` function

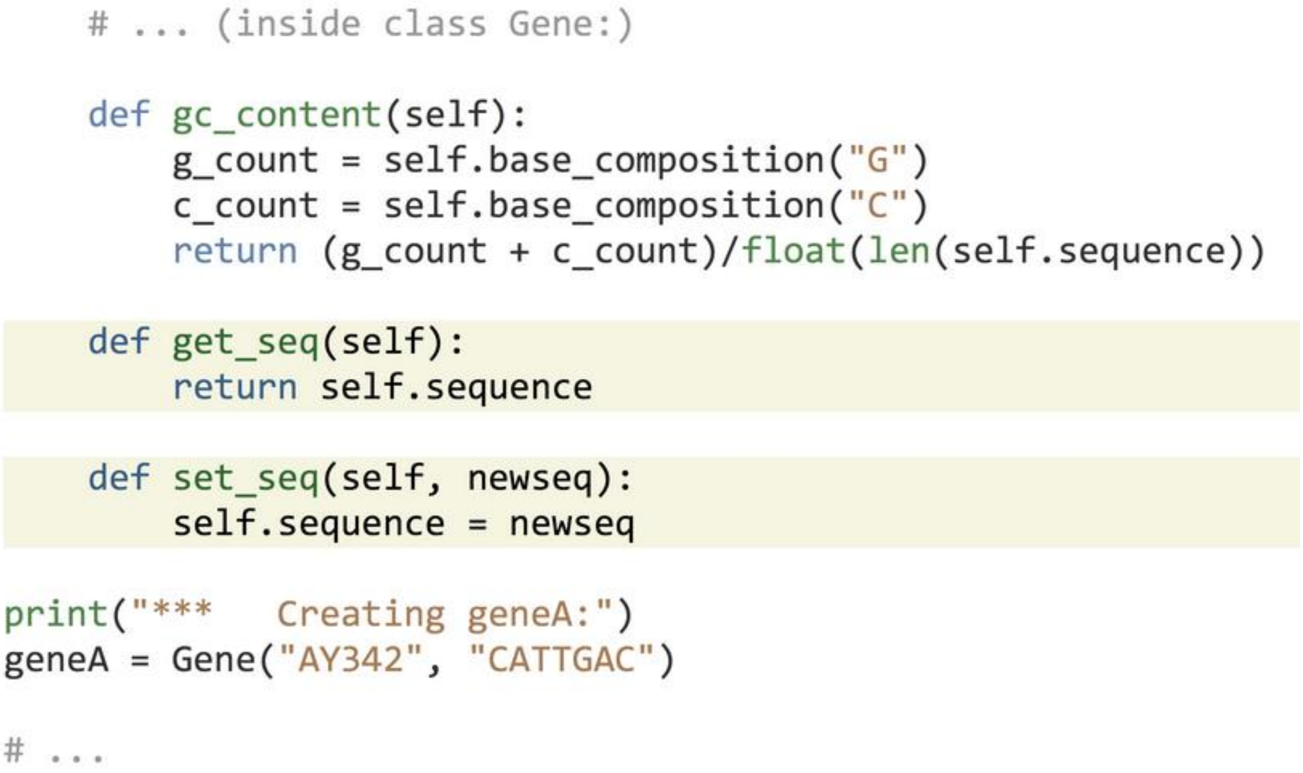
Working With Instances

- **Instance attribute: belongs to a specific instance of a class**
  - Created when a method uses the `self` parameter to create an attribute
- **If many instances of a class are created, each would have its own set of attributes**

Accessor and Mutator Methods

- **Typically, all of a class's data attributes are private and provide methods to access and change them**
- **Accessor methods: return a value from a class's attribute without changing it**
  - Safe way for code outside the class to retrieve the value of attributes
- **Mutator methods: store or change the value of a data attribute**

Accessor and Mutator Methods (cont’d.)



```
    # ... (inside class Gene:)

    def gc_content(self):
        g_count = self.base_composition("G")
        c_count = self.base_composition("C")
        return (g_count + c_count)/float(len(self.sequence))

    def get_seq(self):
        return self.sequence

    def set_seq(self, newseq):
        self.sequence = newseq

print("***   Creating geneA:")
geneA = Gene("AY342", "CATTGAC")

# ...
```

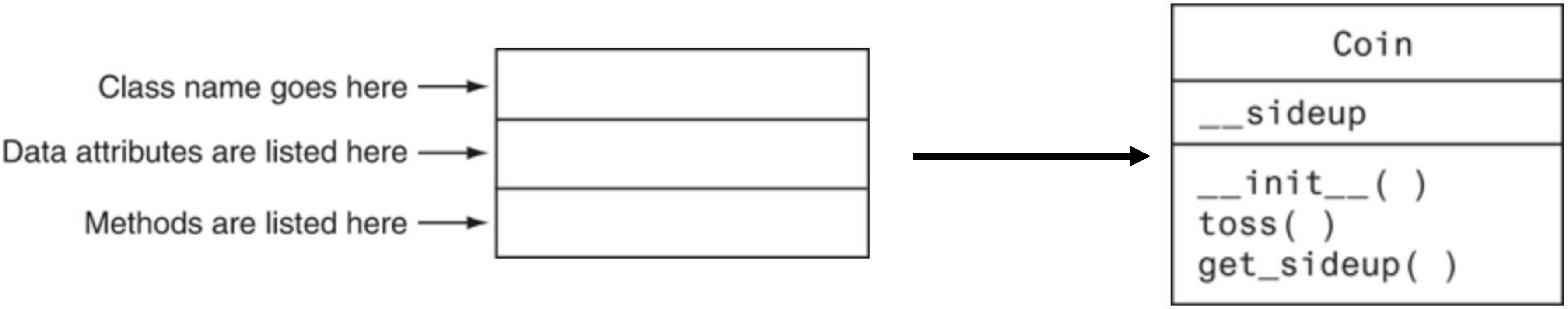
```
print("gene A's sequence is " + geneA.get_seq())
geneA.set_seq("ACTAGGGG")
```

Passing Objects as Arguments

- **Methods and functions often need to accept objects as arguments**
- **When you pass an object as an argument, you are actually passing a reference to the object**
    - The receiving method or function has access to the actual object
        - Methods of the object can be called within the receiving function or method, and data attributes may be changed using mutator methods

Techniques for Designing Classes

- **<u>UML diagram</u>: standard diagrams for graphically depicting object-oriented systems**
  - Stands for Unified Modeling Language
- **General layout: box divided into three sections**



Class name goes here

Data attributes are listed here

Methods are listed here

Coin

__sideup

__init__( )
toss( )
get_sideup( )

Summary

- **This chapter covered:**
  - Procedural vs. object-oriented programming
  - Classes and instances
  - Class definitions, including:
    - The `self` parameter
    - Data attributes and methods
    - `__init__` and `__str__` functions
    - Hiding attributes from code outside a class
  - Storing classes in modules