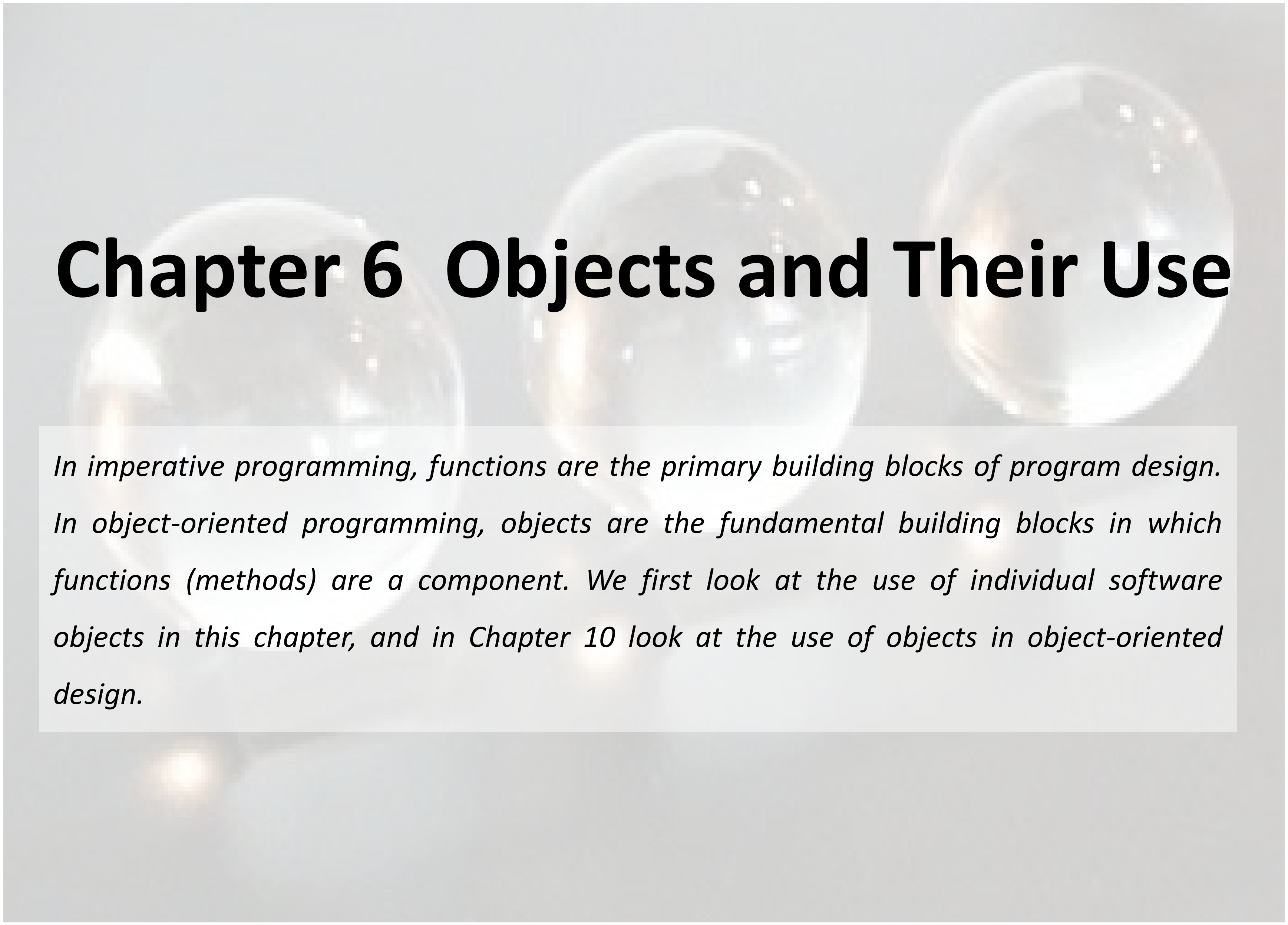


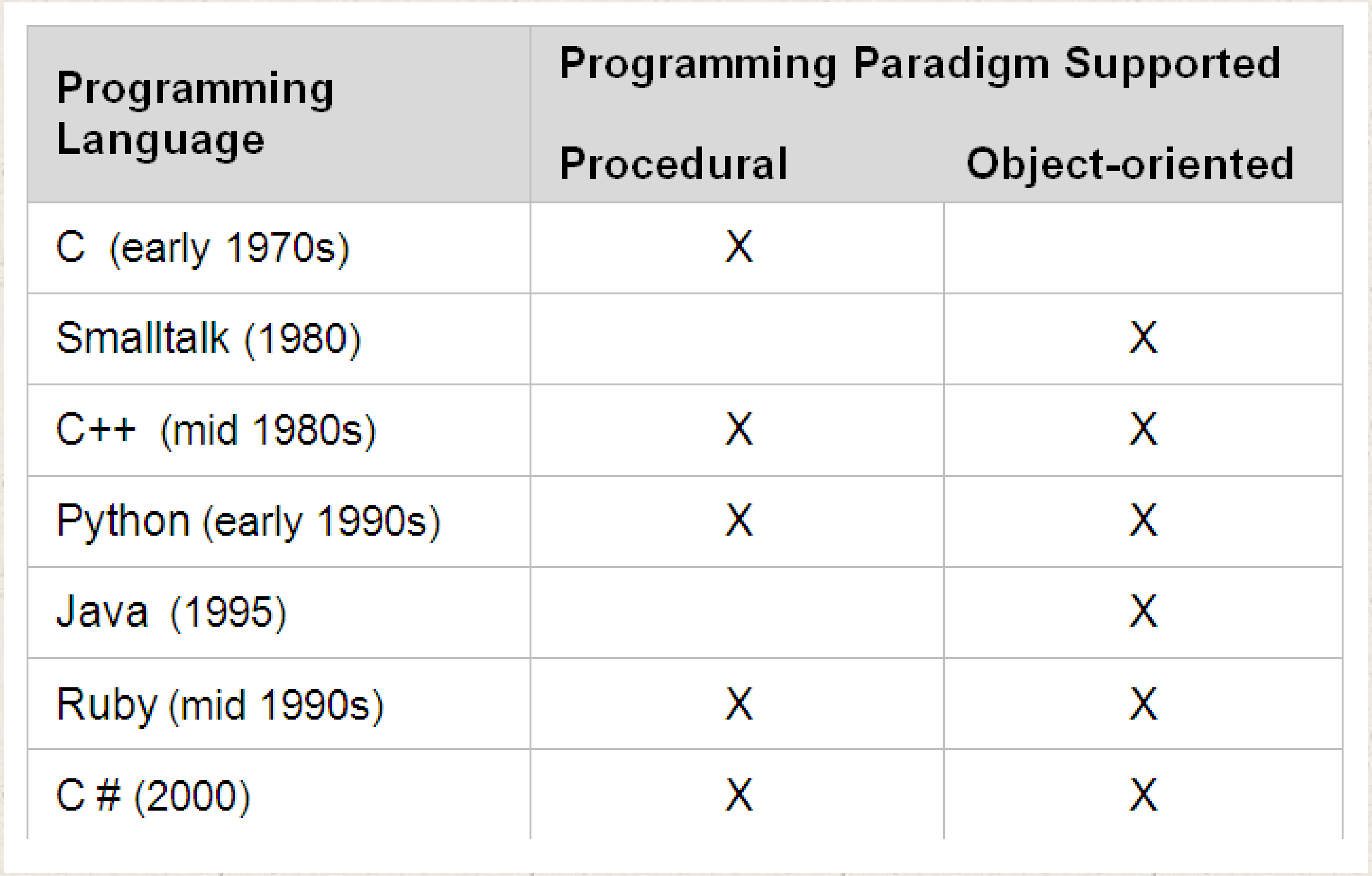
Chapter 6 Objects and Their Use

In imperative programming, functions are the primary building blocks of program design. In object-oriented programming, objects are the fundamental building blocks in which functions (methods) are a component. We first look at the use of individual software objects in this chapter, and in Chapter 10 look at the use of objects in object-oriented design.

Motivation

The idea of incorporating “objects” into a programming language came out of work in computer simulation. Given the prevalence of objects in the world, it was natural to provide the notion of an object within simulation programs that simulate some aspect of the world.

In the early 1970s, Alan Kay at Xerox PARC (Palo Alto Research Center) fully evolved the notion of object-oriented programming with the development of a programming language called Smalltalk. The language became the inspiration for the development of graphical user interfaces (GUIs) — the primary means of interacting with computer systems today. Before that, all interaction was through typed text. In fact, it was a visit to Xerox PARC by Steve Jobs of Apple Computers that led to the development of the first commercially successful GUI-based computer, the Apple Macintosh in 1984. In this chapter, we look at the creation and use of objects in Python.



Programming Language	Programming Paradigm Supported	
	Procedural	Object-oriented
C (early 1970s)	X	
Smalltalk (1980)		X
C++ (mid 1980s)	X	X
Python (early 1990s)	X	X
Java (1995)		X
Ruby (mid 1990s)	X	X
C # (2000)	X	X

Some Commonly-Used Programming Languages

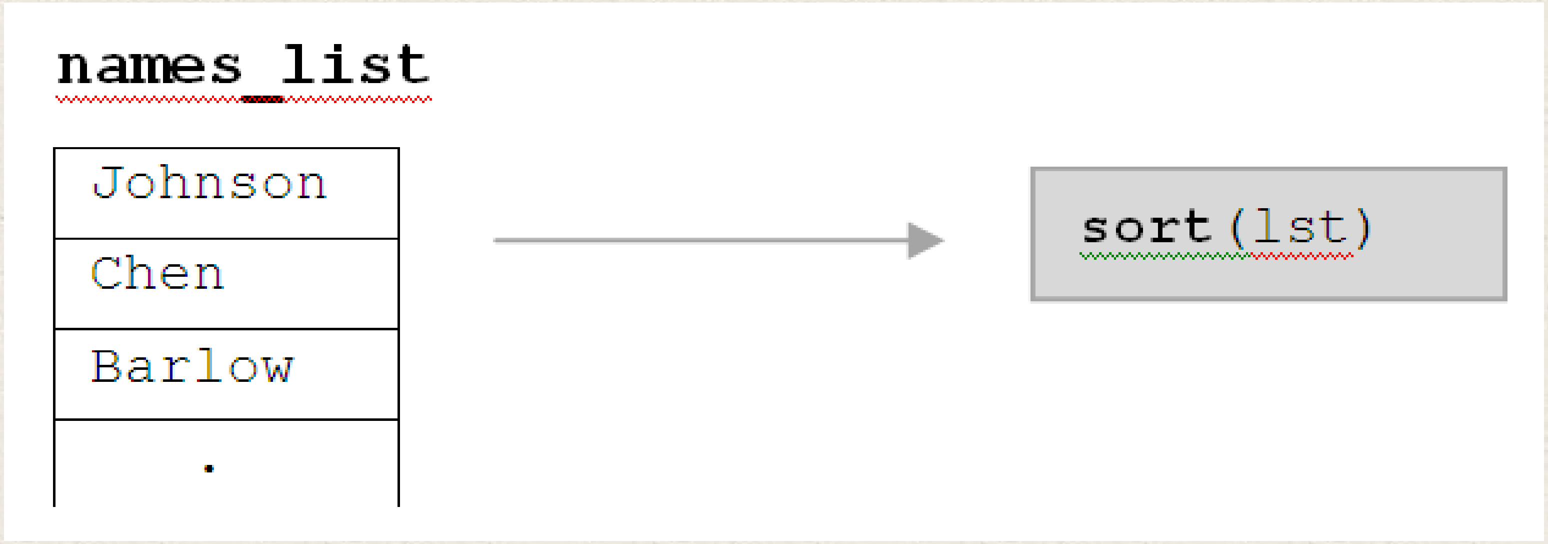
Software Objects

Objects are the fundamental component of object-oriented programming. Although we have not yet stated it, all values in Python are represented as objects. This includes, for example, lists, as well as numerical values.

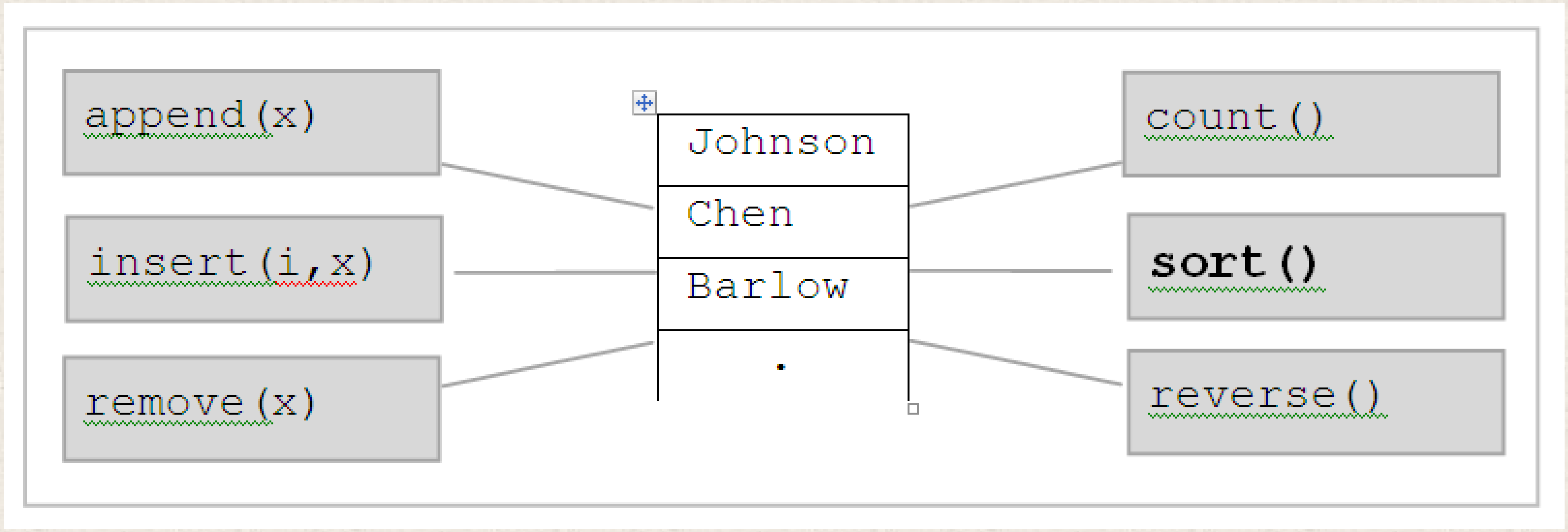
What is an Object?

An **object** contains a set of attributes, stored in a set of instance variables, and a set of functions called methods that provide its behavior.

When sorting a list in procedural programming, there are two distinct entities — a *sort function* and the *list* to pass it to.



In object-oriented programming, the sort routine would be part of the object containing the list.



All list objects contain the same set of methods. Thus, names_list is sorted by simply calling that object's sort method,

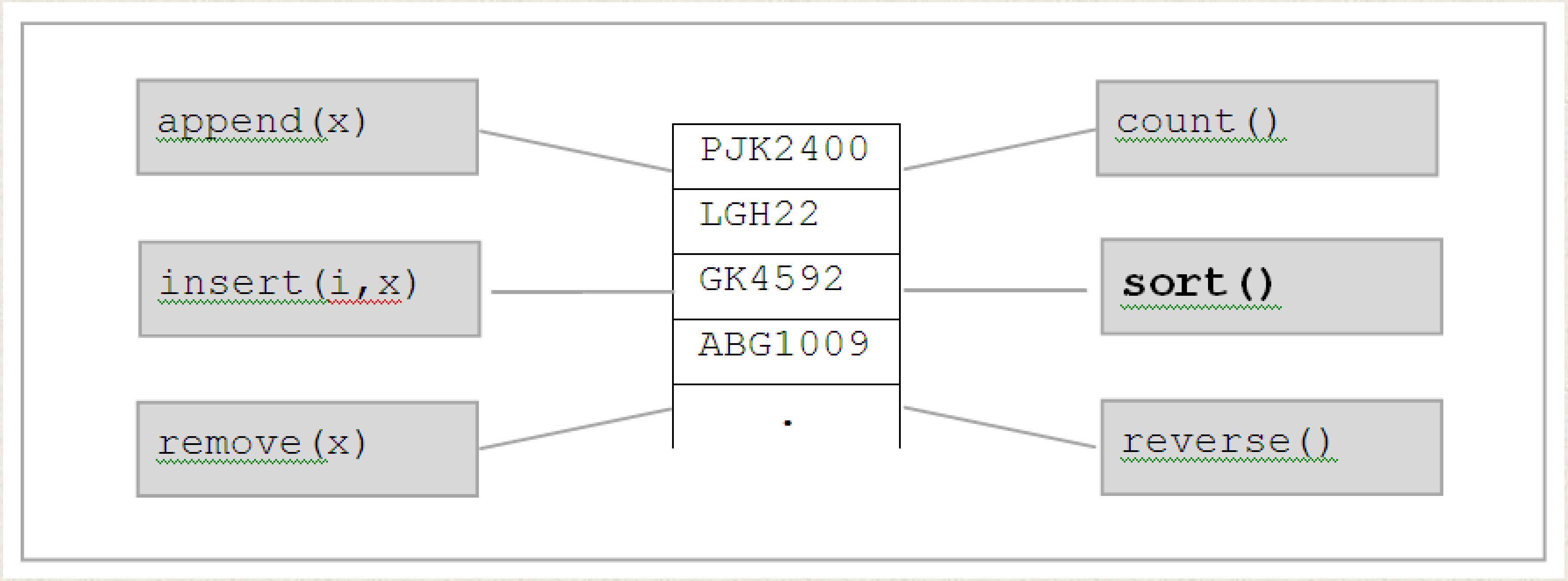
names_list.sort()

names_list.sort()

The period is referred to as the dot operator, used to select a member of an object. In this case, a method is selected (method sort).

Note that no arguments are passed to the method. This is because methods operate on the data of the object that they are part of. Thus, the sort method does not need to be told which list to sort. Calling the method is more of a message saying "sort yourself."

Suppose there was another list object called `part_numbers`, containing a list of automobile part numbers.



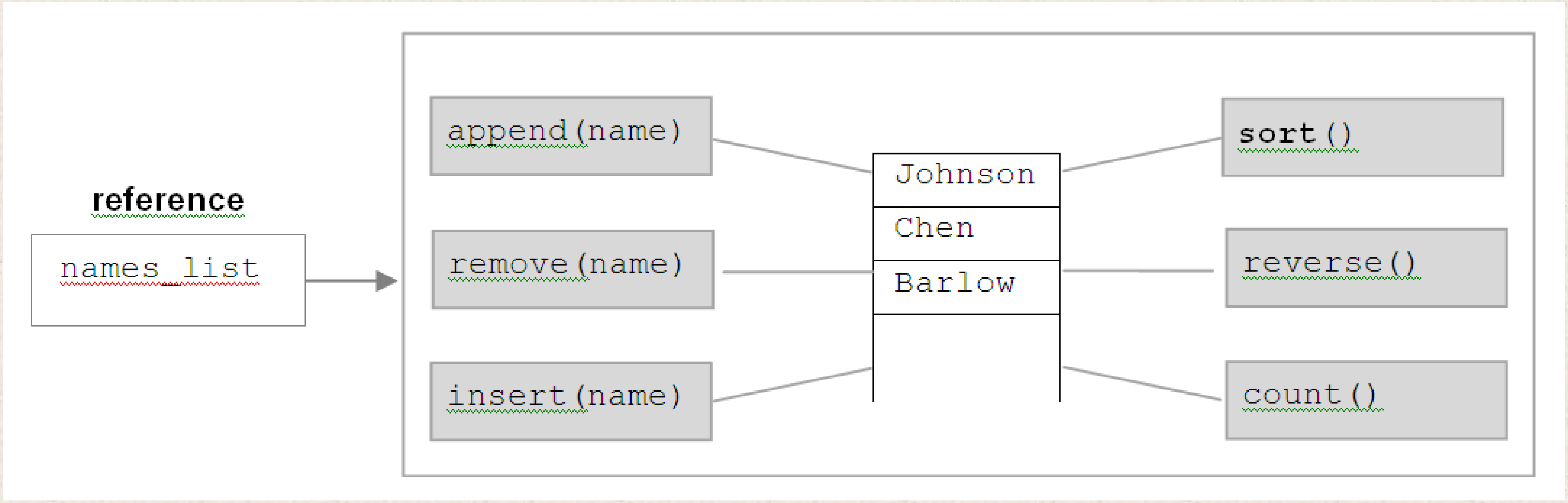
Since all list objects behave the same, `part_numbers` would contain the identical set of methods as `names_list`. Only the data that they operate on would be different.

Object References

We look at how objects (values) are represented and the effect it has on the operations of assignment and comparison, as well as parameter passing of objects.

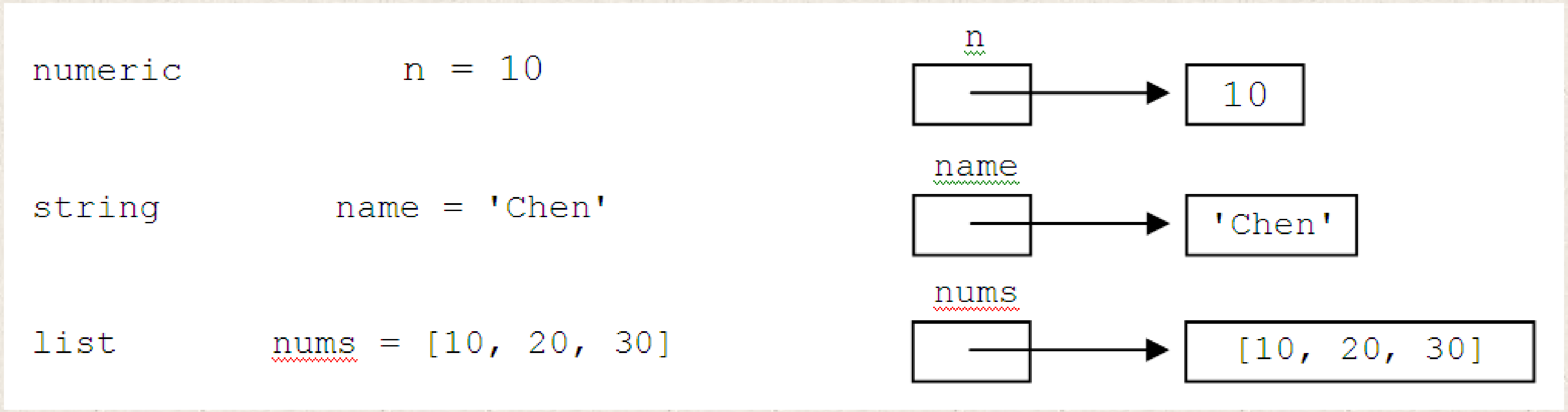
References in Python

In Python, **objects are represented as a *reference*** to an object in memory.



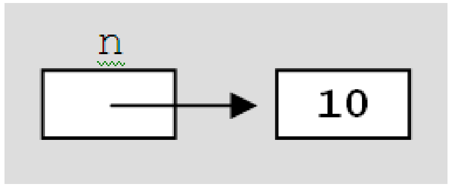
A **reference** is a value that references, or "points to," the location of another entity.

When a new object in Python is created, two values are stored — the object, and a variable holding a reference to the object. **All access to the object is through the reference value.**

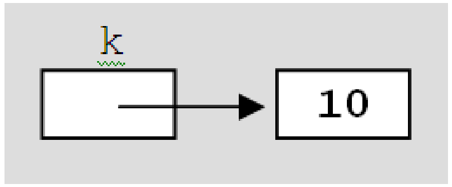


```
numeric          n = 10
string        name = 'Chen'
list       nums = [10, 20, 30]
```

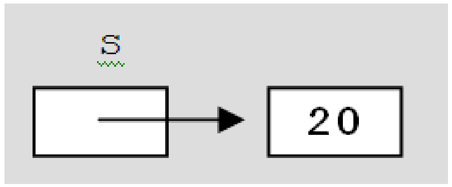
The value that a reference points to is called the dereferenced value. This is the value that the variable represents.



```
>>> n
10
```



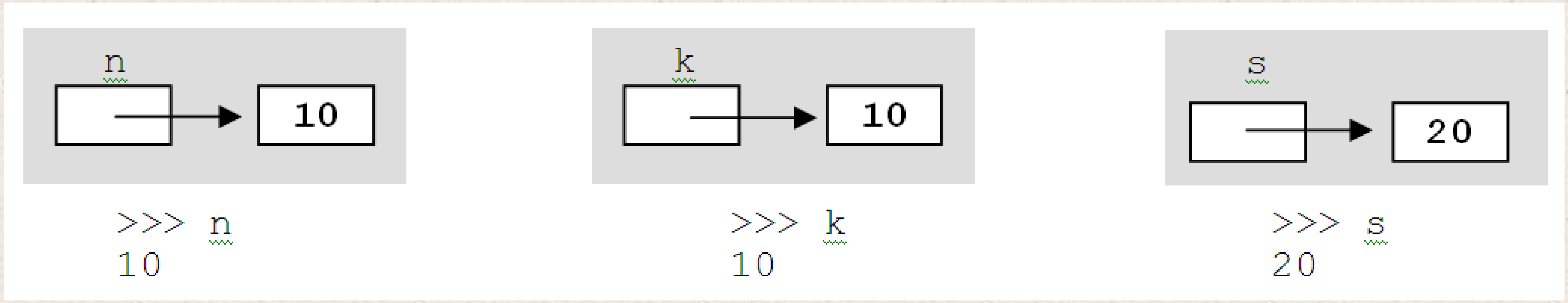
```
>>> k
10
```



```
>>> s
20
```

We can get the reference value of a variable (the location in which the dereferenced value is stored) by use of **built-in function id**.

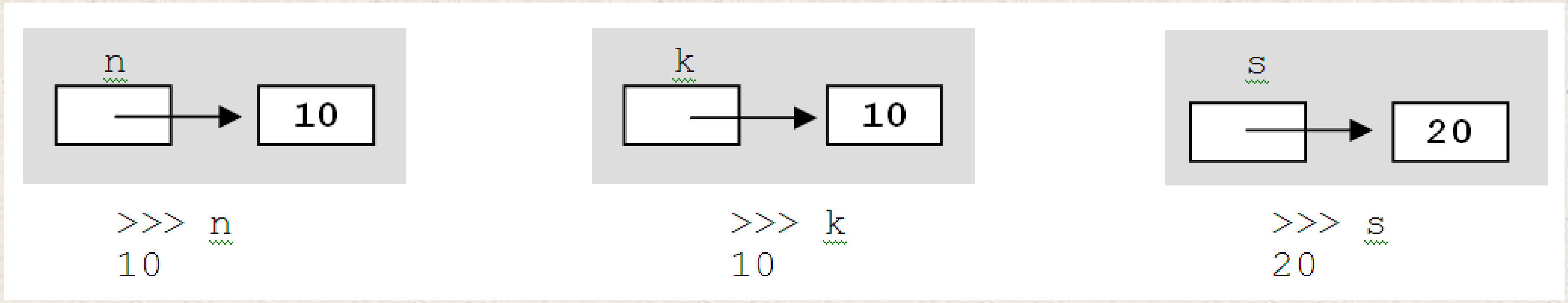
```
>>> id(n)          >>> id(k)          >>> id(s)
505498136          505498136          505498296
```



```
>>> n              >>> k              >>> s
10                 10                 20
```

We see that the dereferenced values of n and k, 10, is stored in the same memory location (505498136), whereas the dereferenced value of s (505498296), 20, is stored in a different location.

Even though n and k are separately assigned literal value 10, they reference the *same instance* of 10 in memory (505498136). We would expect that two instances of the value 10 be stored.



```
>>> n
10
```

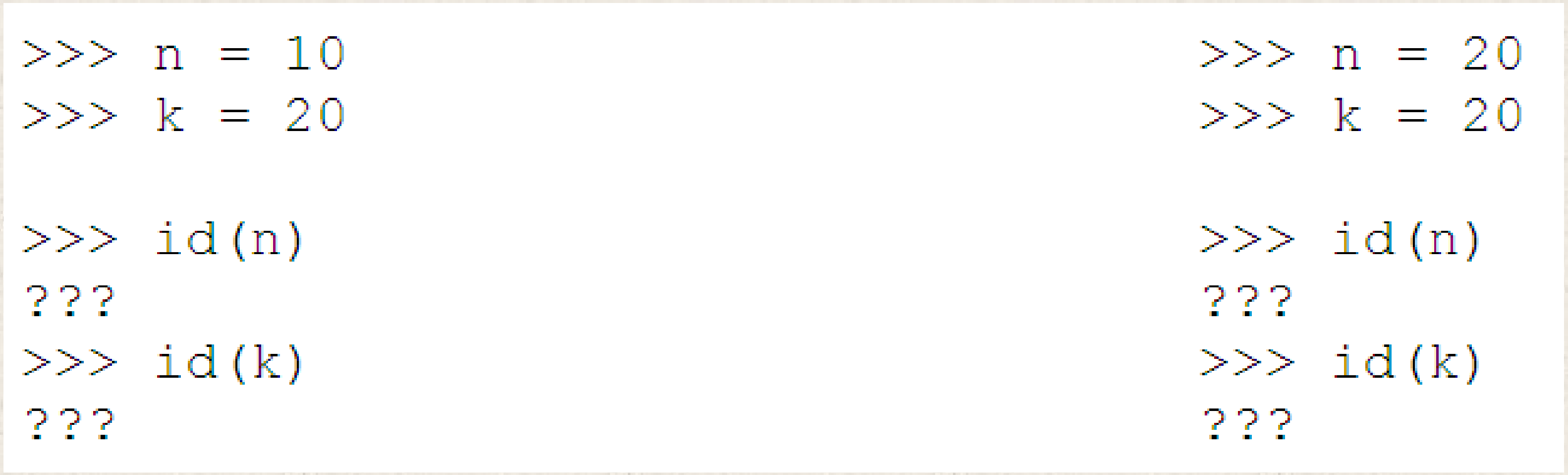
```
>>> k
10
```

```
>>> s
20
```

Python is being clever here. Since integer values are immutable, it assigned both n and k to the same instance. This saves memory and reduces the number of reference locations that Python must maintain. From the programmer's perspective however, they can be treated as if each holds its own instance.

Let’s Try It

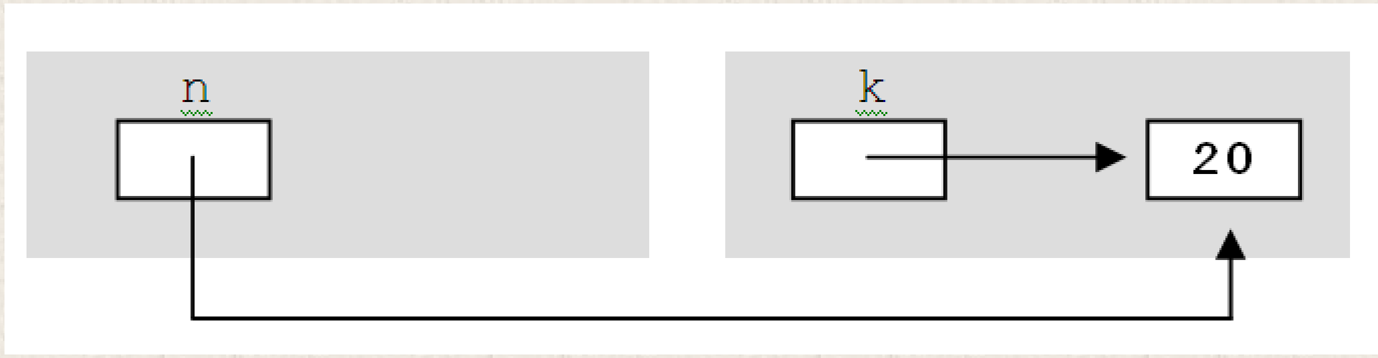
From the Python Shell, enter the following and observe the results.



```
>>> n = 10                                      >>> n = 20
>>> k = 20                                      >>> k = 20

>>> id(n)                                       >>> id(n)
???                                             ???
>>> id(k)                                       >>> id(k)
???                                             ???
```

The Assignment of References

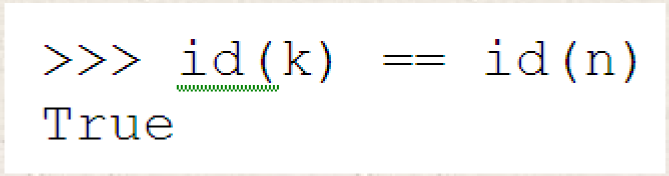
Consider what happens when one variable, n, is assigned to another, k, depicted below.



```
k = 20
n = k
n
>>> 20
```

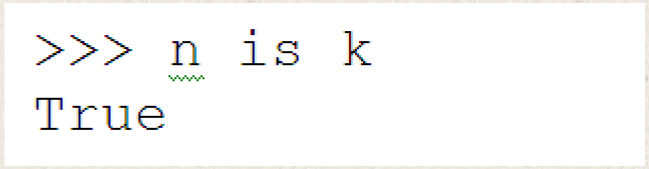
When variable n is assigned to k, it is the *reference value* of k that is assigned, not the dereferenced value 20.

To verify that two variables refer to the same object instance, we can compare their id values using the comparison operator.



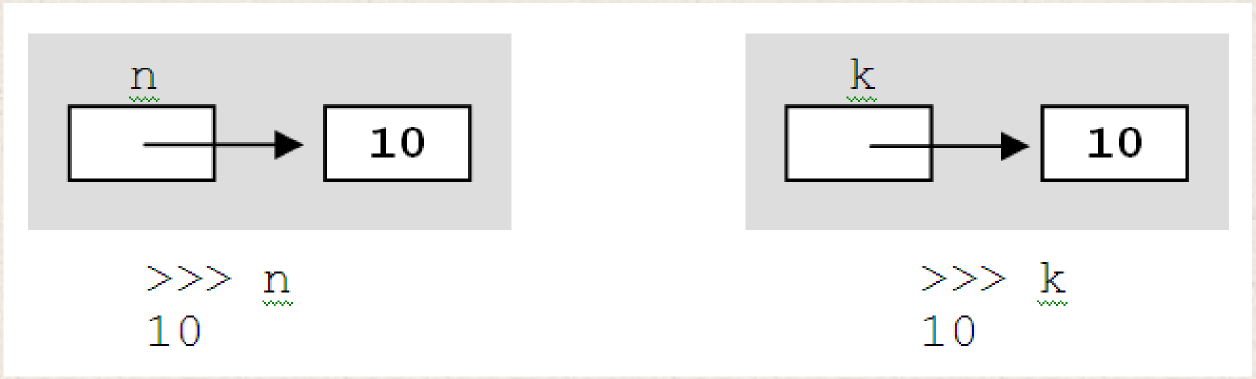
```
>>> id(k) == id(n)
True
```

We an also verify this by use of the **is operator**, which performs id(k) == id(n).



```
>>> n is k
True
```

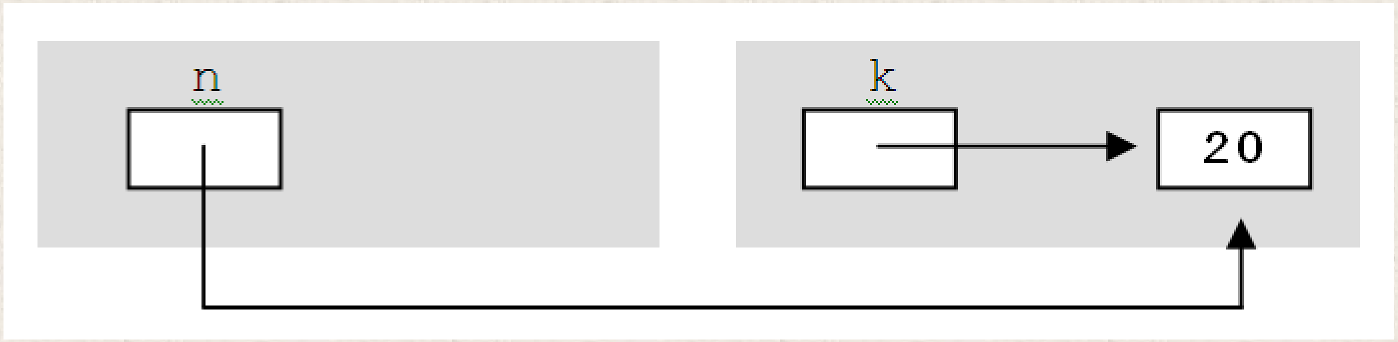
When variables n and k were separately assigned the value 10, each ended up referring to the same instance of 10 in memory because integer values are immutable, and Python decided to have both variables share the same instance.



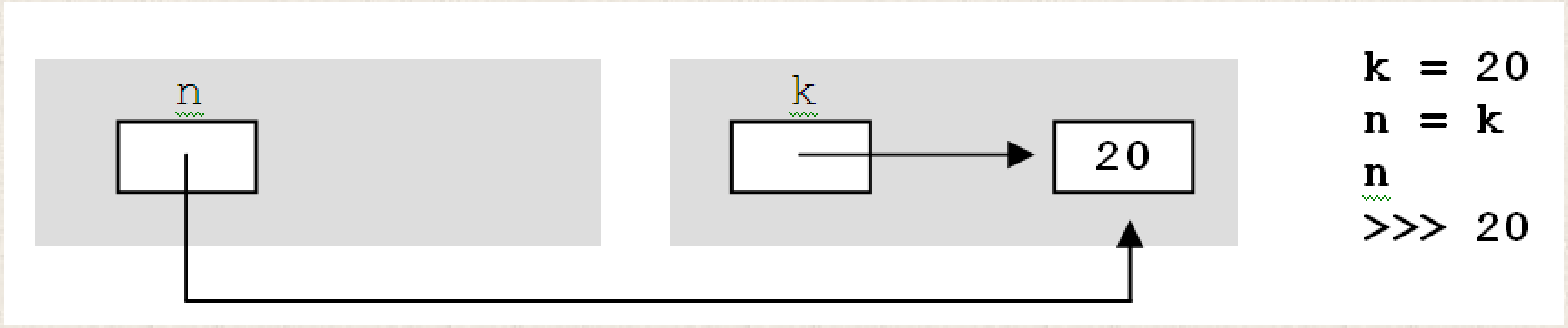
```
>>> n
10
```

```
>>> k
10
```

When variable n was directly assigned to k, both variables ended up referring to the same instance of 20 in memory because assignment in Python assigns reference values, and not the dereference values.

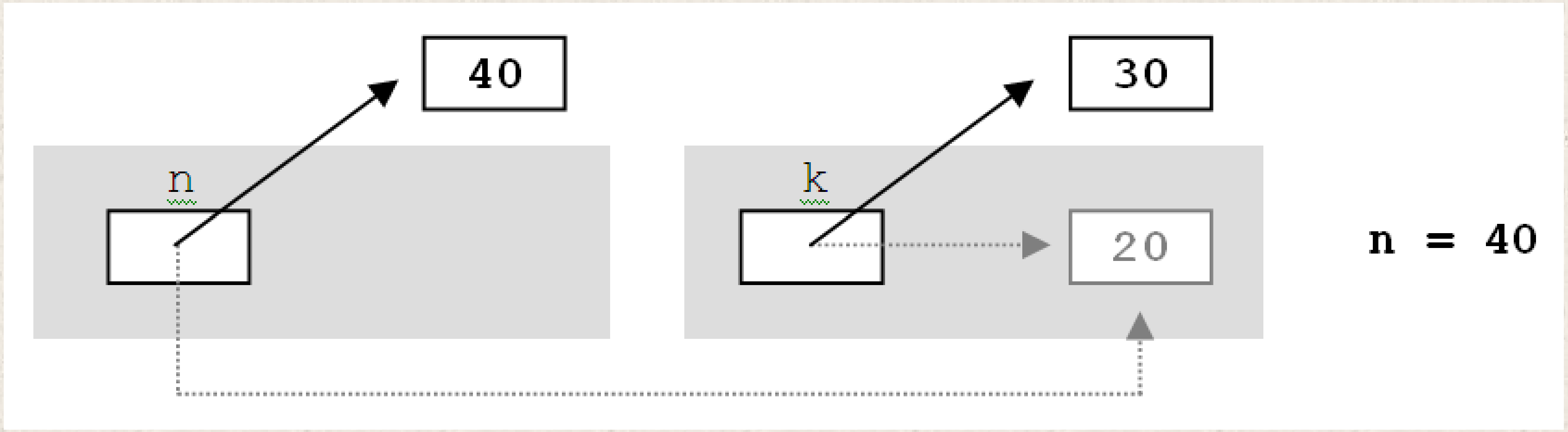


If one of the two variables is assigned a new value, then they would no longer reference the same memory location.

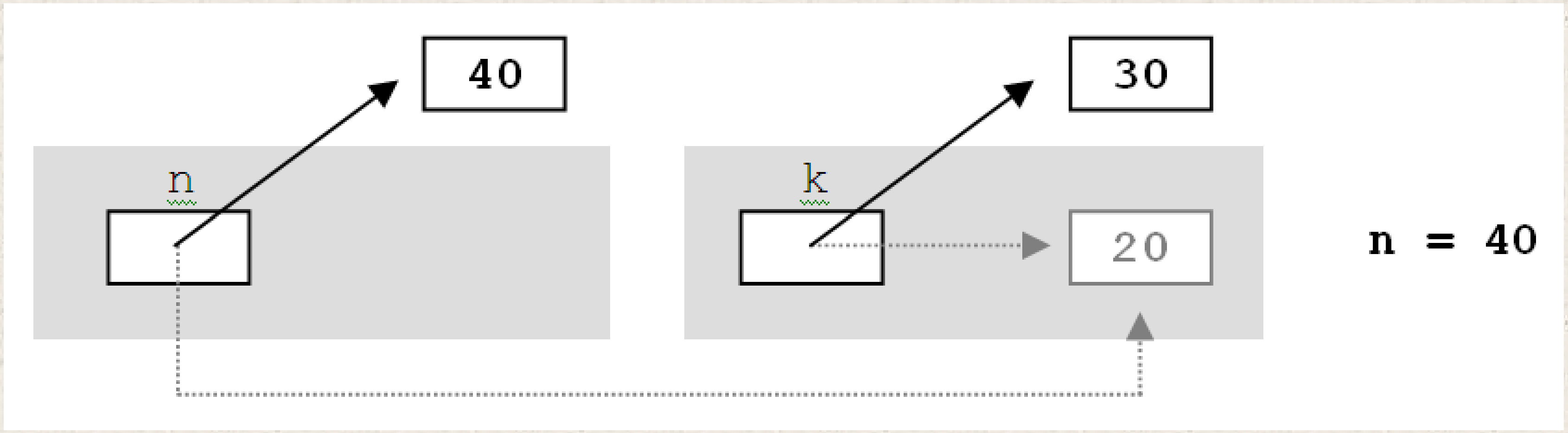


```
k = 20
n = k
n
>>> 20
```

Memory Deallocation and Garbage Collection



Consider what happens when a variable (n) is assigned a new value, (40) and no other variable references the original value (20).



After n is assigned to 40, the memory location storing integer value 20 is no longer referenced — thus, it can be *deallocated*.

To deallocate a memory location means to change its status from "currently in use" to "available for reuse."

In Python, memory deallocation is automatically performed by a process called *garbage collection.*

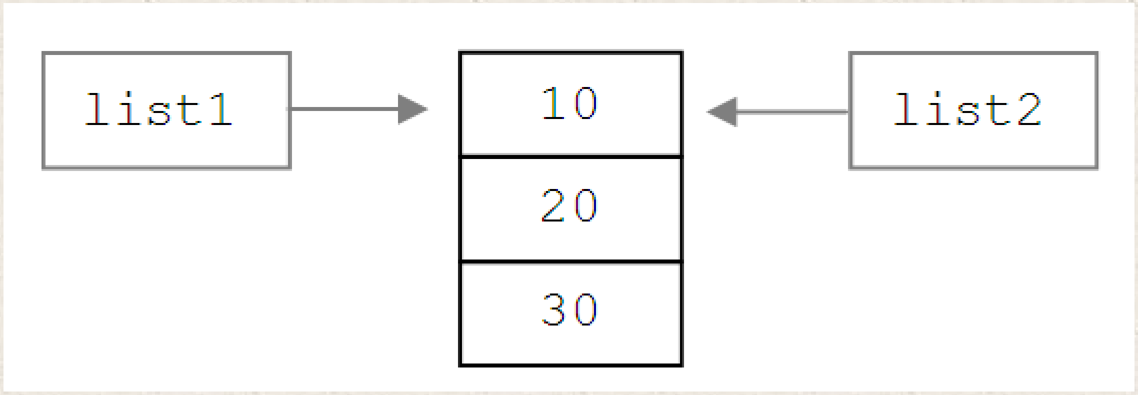
Garbage collection is a method of automatically determining which locations in memory are no longer in use, and deallocating them.

The garbage collection process is ongoing during the execution of a Python program.

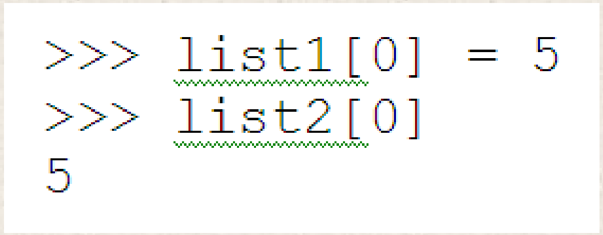
List Assignment and Copying

Now that we understand the use of references in Python, we can revisit the discussion on copying lists from Chapter 4.

When a variable is assigned to another variable referencing a list, each variable ends up referring to the same instance of the list in memory.



Thus, any change to the elements of list1 will result in changes to list2.

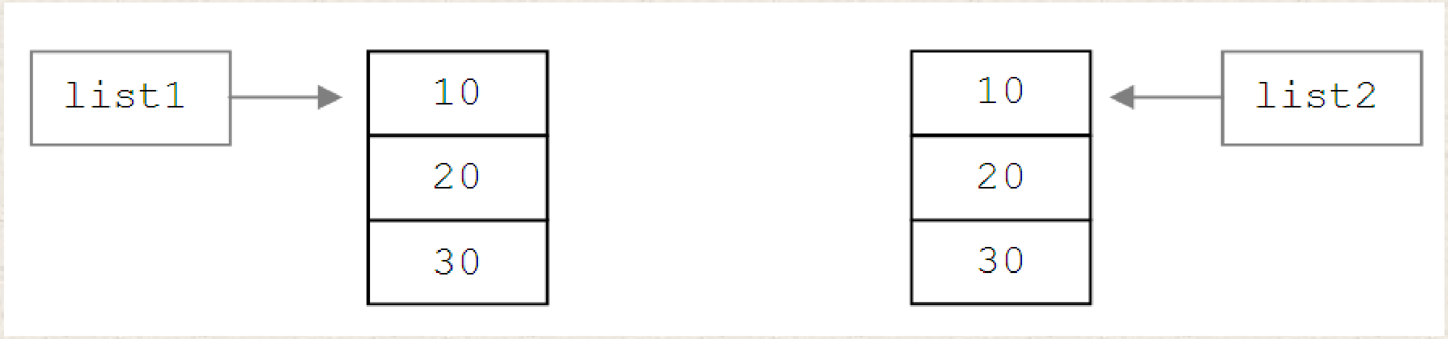


```
>>> list1[0] = 5
>>> list2[0]
5
```

We also learned that **a copy of a list can be made as follows**.

```
>>> list2 = list(list1)
```

`list()` is referred to as a **list constructor**. The result of the copying is depicted below.



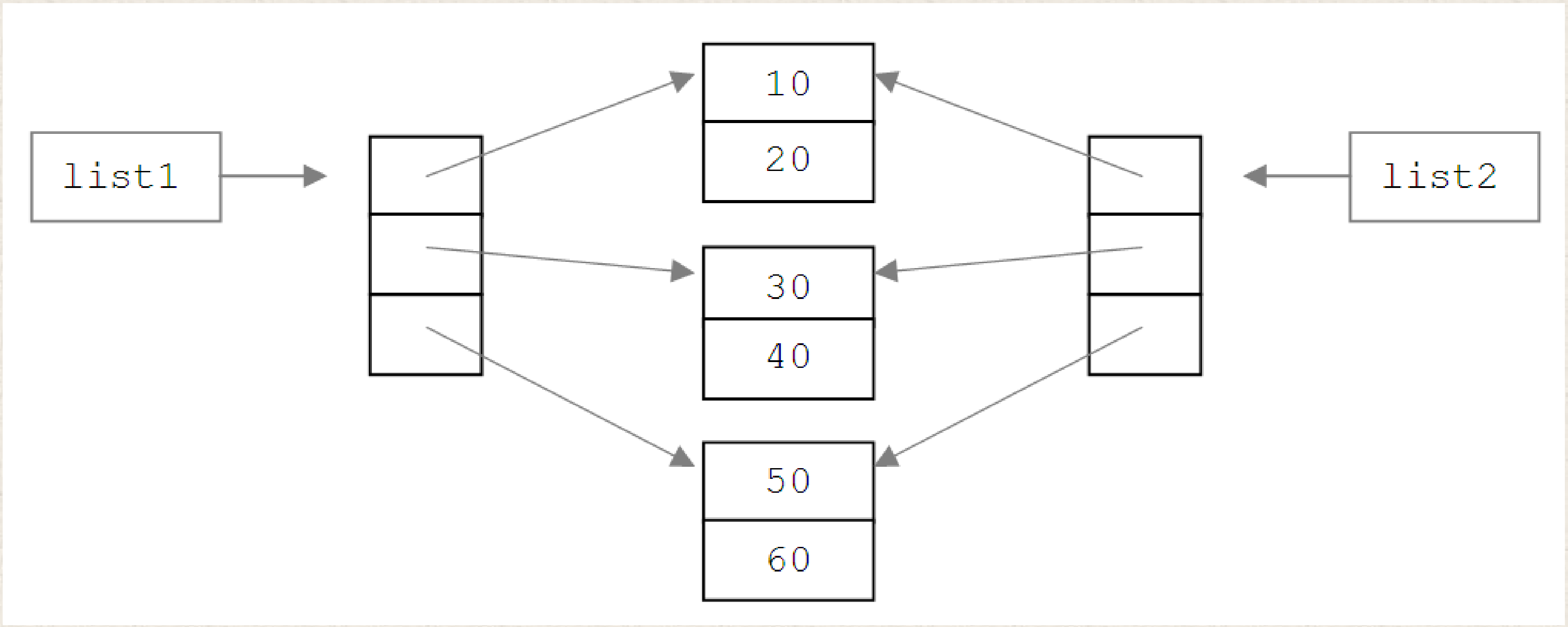
Since a copy of the list structure has been made, changes to the elements of one list will not result in changes to the other.

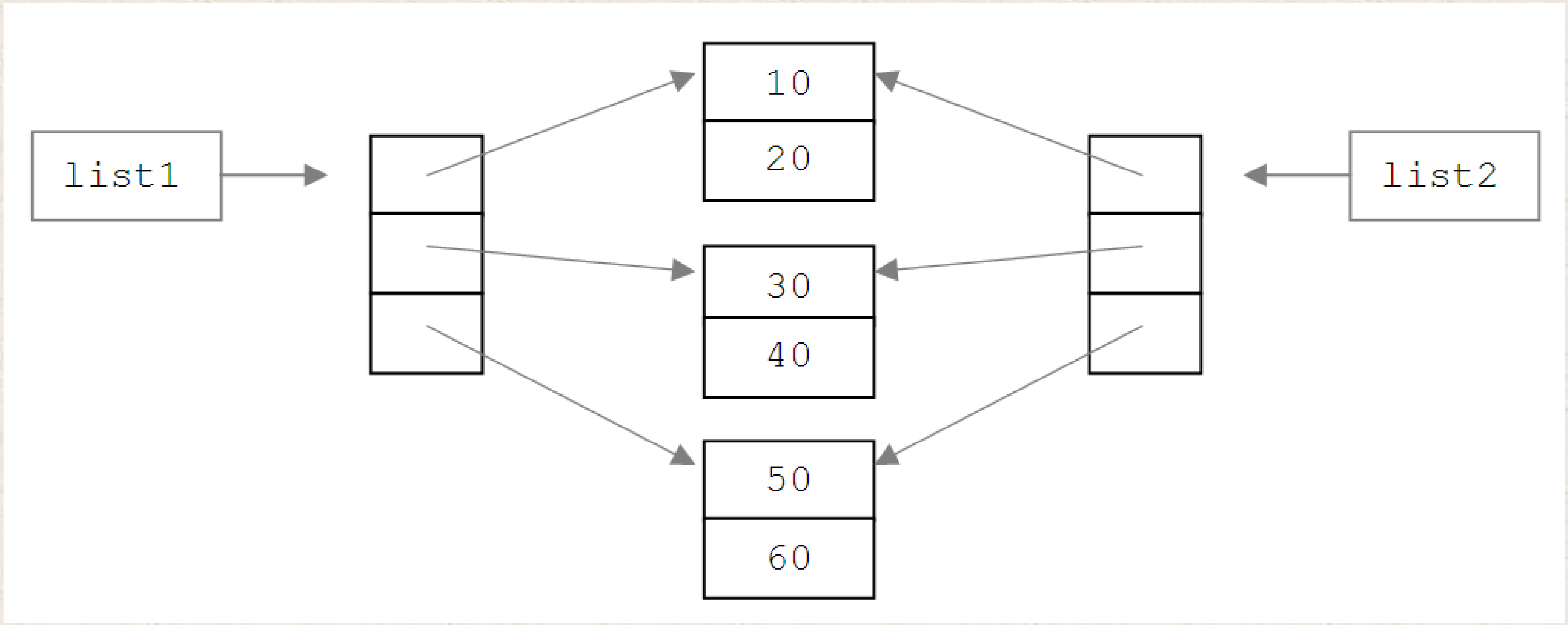
```
>>> list1[0] = 5
>>> list2[0]
10
```

The situation is different if the list contains sublists, however.

```
>>> list1 = [[10, 20], [30, 40], [50, 60]]
>>> list2 = list(list1)
```

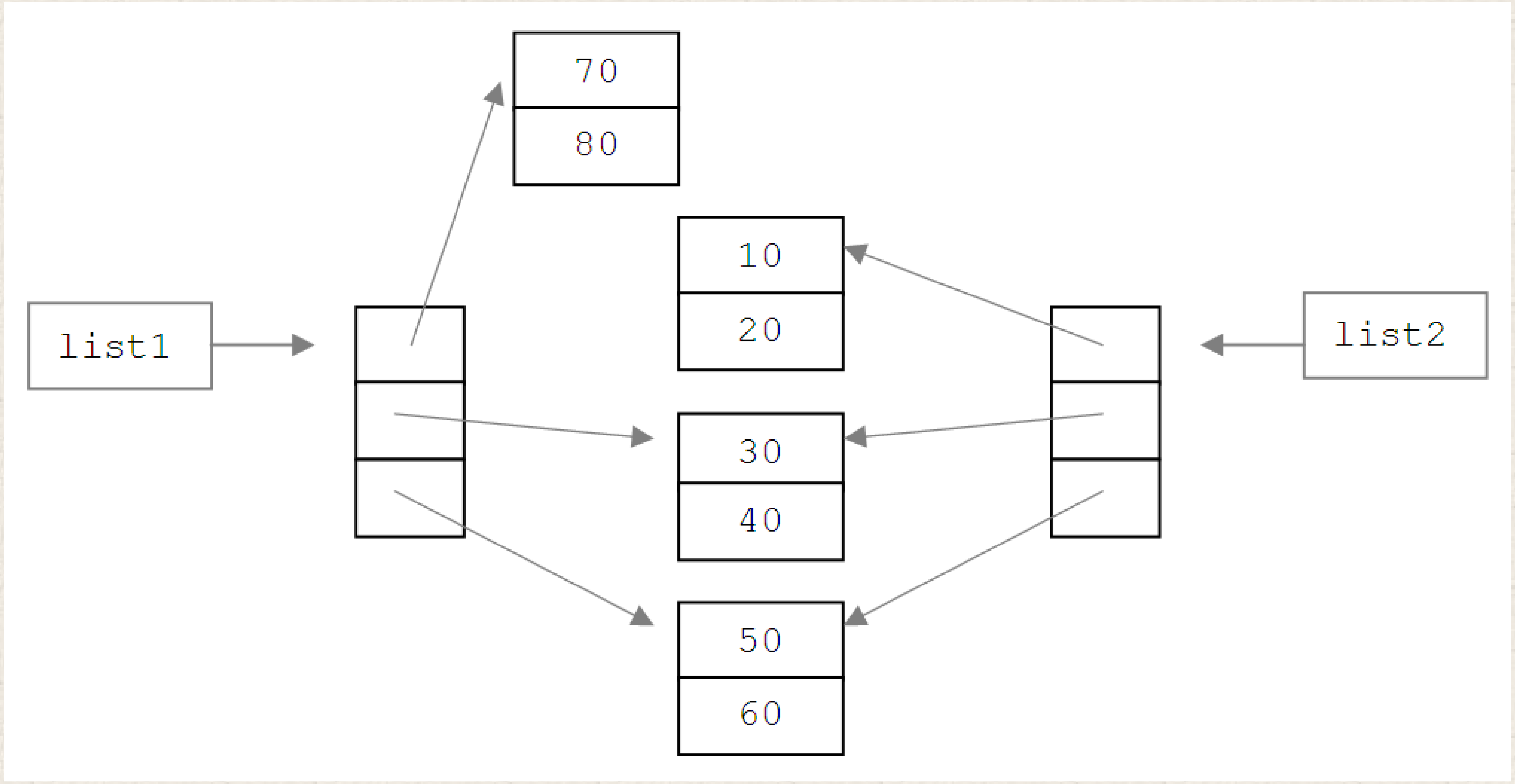
Below is the resulting list structure after this assignment.



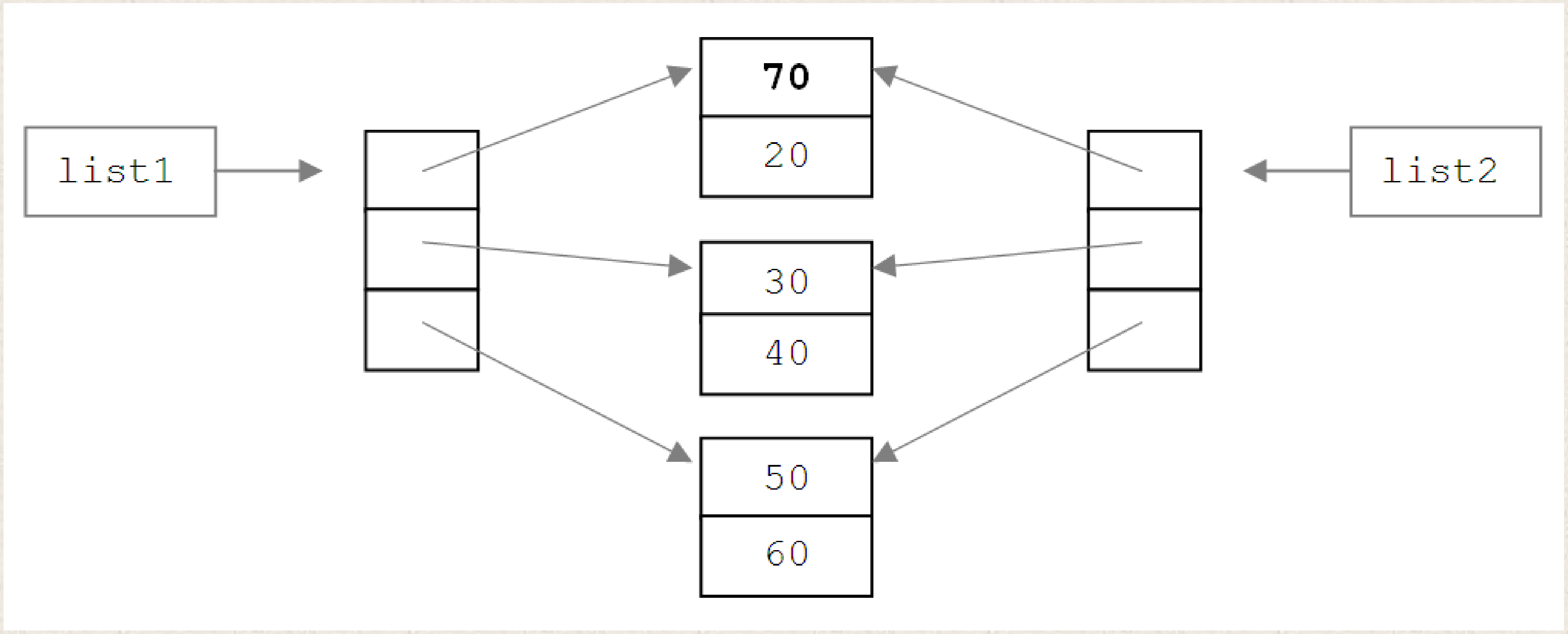


Although copies were made of the top-level list structures, the elements *within* each list were not copied. This is referred to as a **shallow copy**.

If a top-level element of one list is reassigned, for example list1[0] = [70, 80], the other list would remain unchanged.

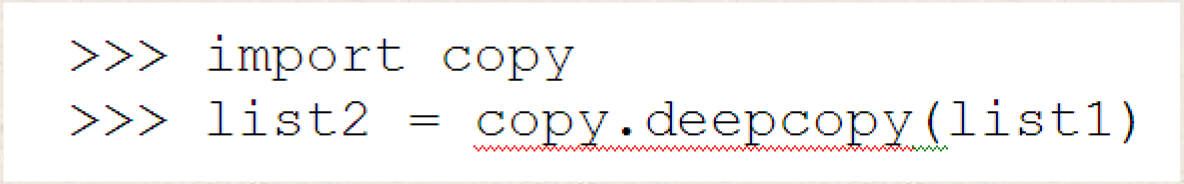


If, however, a change to one of the *sublists* is made, for example, list1[0][0] = 70, the corresponding change would be made to the other list as well. Thus, list2[0][0] would also be equal to 70.



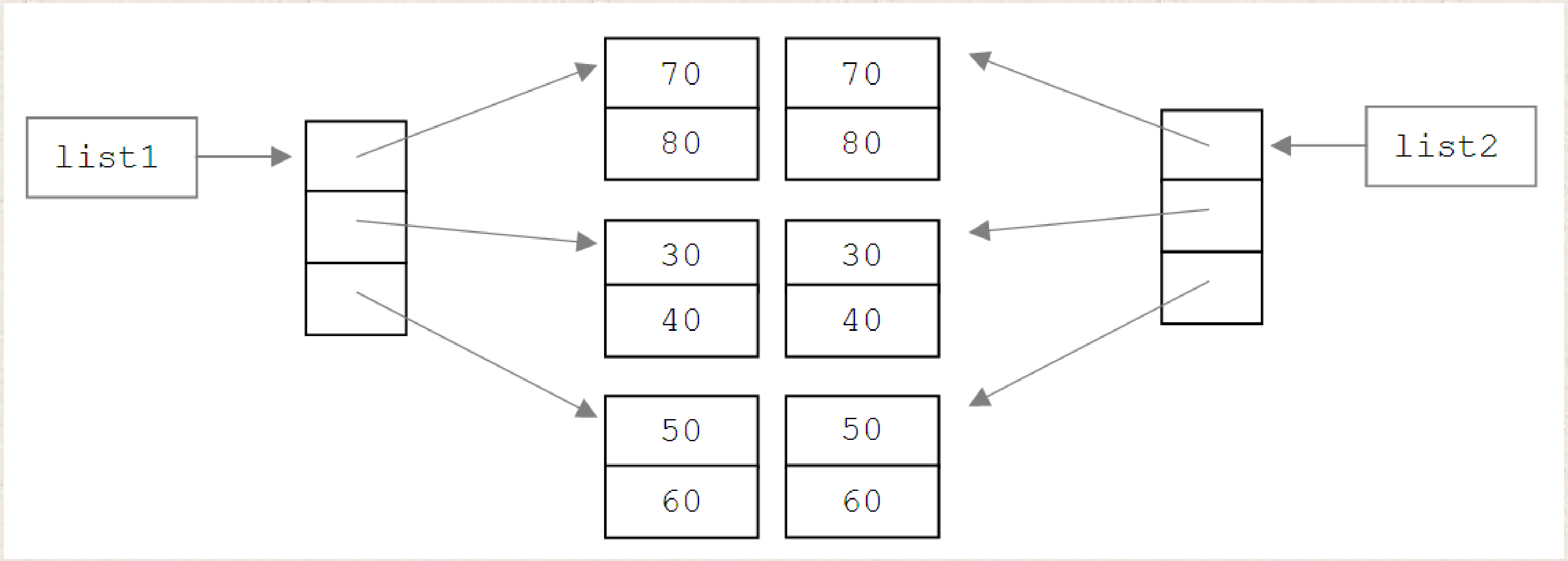
A **deep copy operation of a list (structure) makes a copy of the *complete* structure, including sublists**. (Since immutable types cannot be altered, immutable parts of the structure may not be copied.)

Such an operation can be performed with the deepcopy method of the copy module,



```
>>> import copy
>>> list2 = copy.deepcopy(list1)
```

The result of this form of copy is shown below.



As a result, the reassignment of any part (top level or sublist) of one list will not result in a change in the other

Let’s Try It

From the Python Shell, enter the following and observe the results.

```
>>> import copy

>>> list1 = [10, 20, 30, 40]
>>> list2 = list1
>>> id(list1) == id(list2)
???
>>> list1[0] = 60
>>> list1
???
>>> list2
???

>>> list1 = [10, 20, 30, [40]]
>>> list2 = list(list1)
>>> id(list1) == id(list2)
???
>>> list1[0] = 60
>>> list1[3][0] = 90
>>> list1
???
>>> list2
???
```

```
>>> list1 = [10, 20, 30, [40]]
>>> list2 = copy.deepcopy(list1)
>>> id(list1) == id(list2)
???
>>> list1[0] = 60
>>> list1[3][0] = 90
>>> list1
???
>>> list2
???

>>> list1 = [10, 20, 30, (40)]
>>> list2 = copy.deepcopy(list1)
>>> list1[3][0] = 90
???
>>> list1[3] = (100,)
>>> list1
???
>>> list2
???
```

Self-Test Questions

1. All objects have a set of ______________ and _______________.

2. The ____________ operator is used to select members of a given object.

3. Functions that are part of an object are called _____________________.

4. There are two values associated with every object in Python, the ___________ value and the _________________ value.

5. When memory locations are *deallocated*, it means that,
(a) The memory locations are marked as unusable for the rest of the program execution.
(b) The memory locations are marked as available for reuse during the remaining program execution.

6. Garbage collection is the process of automatically identifying which areas of memory can be deallocated. (TRUE/FALSE)

7. Indicate which of the following are true,
(a) When one variable is assigned to another holding an integer value, if the second variable is assigned a new value, the value of the first variable will change as well.
(b) When one variable is assigned to another holding a list of integer values, if the second variable assigns a new integer value to an element in the list, the list that the first variable is assigned to will be changed as well.

ANSWERS: [redacted]

Self-Test Questions

1. All objects have a set of _____________ and ______________.

2. The ___________ operator is used to select members of a given object.

3. Functions that are part of an object are called ___________________.

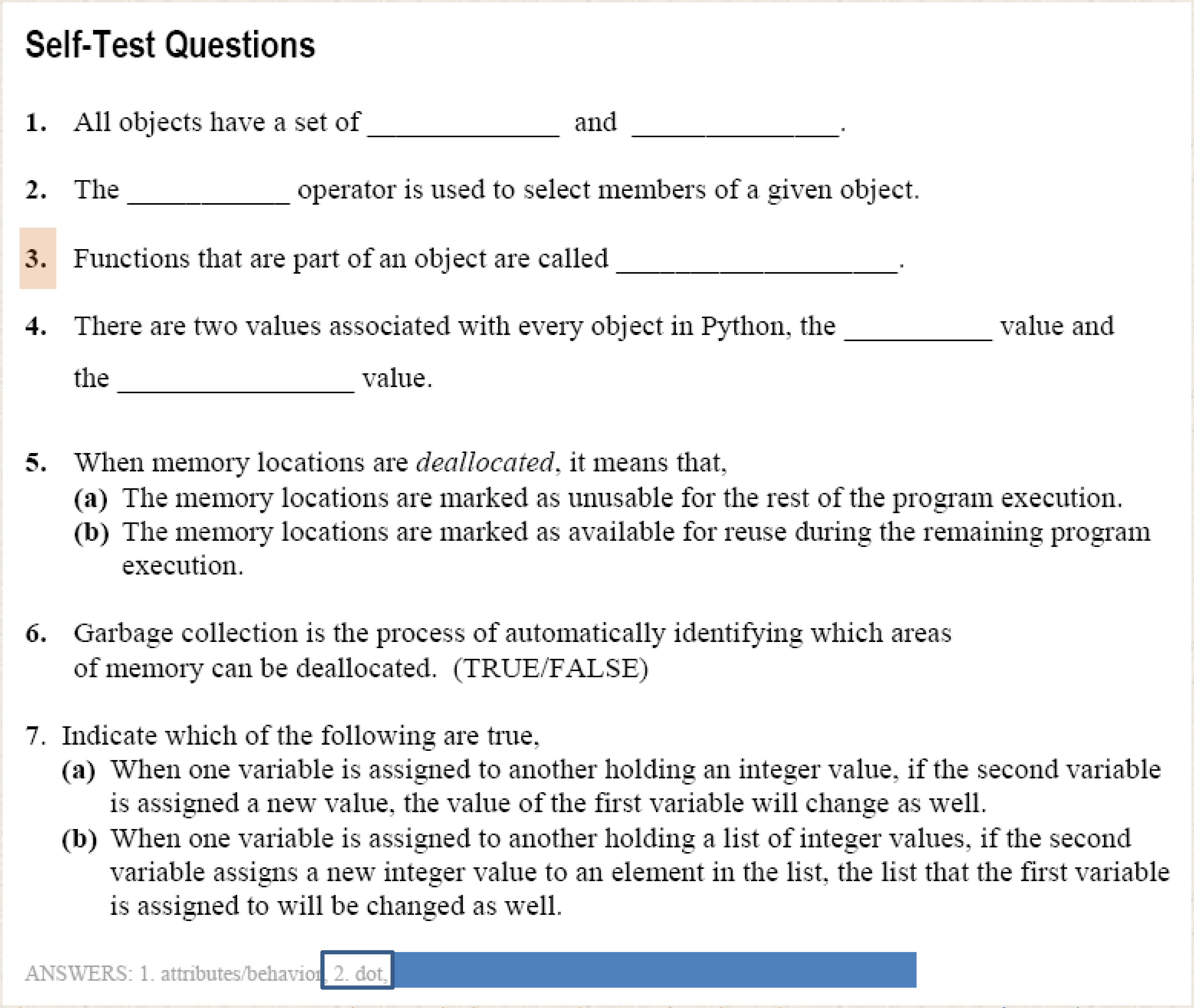
4. There are two values associated with every object in Python, the __________ value and the ________________ value.

5. When memory locations are *deallocated*, it means that,
(a) The memory locations are marked as unusable for the rest of the program execution.
(b) The memory locations are marked as available for reuse during the remaining program execution.

6. Garbage collection is the process of automatically identifying which areas of memory can be deallocated. (TRUE/FALSE)

7. Indicate which of the following are true,
(a) When one variable is assigned to another holding an integer value, if the second variable is assigned a new value, the value of the first variable will change as well.
(b) When one variable is assigned to another holding a list of integer values, if the second variable assigns a new integer value to an element in the list, the list that the first variable is assigned to will be changed as well.

ANSWERS 1. attributes/behavior,



Self-Test Questions

1. All objects have a set of ______________ and _______________.

2. The ____________ operator is used to select members of a given object.

3. Functions that are part of an object are called _____________________.

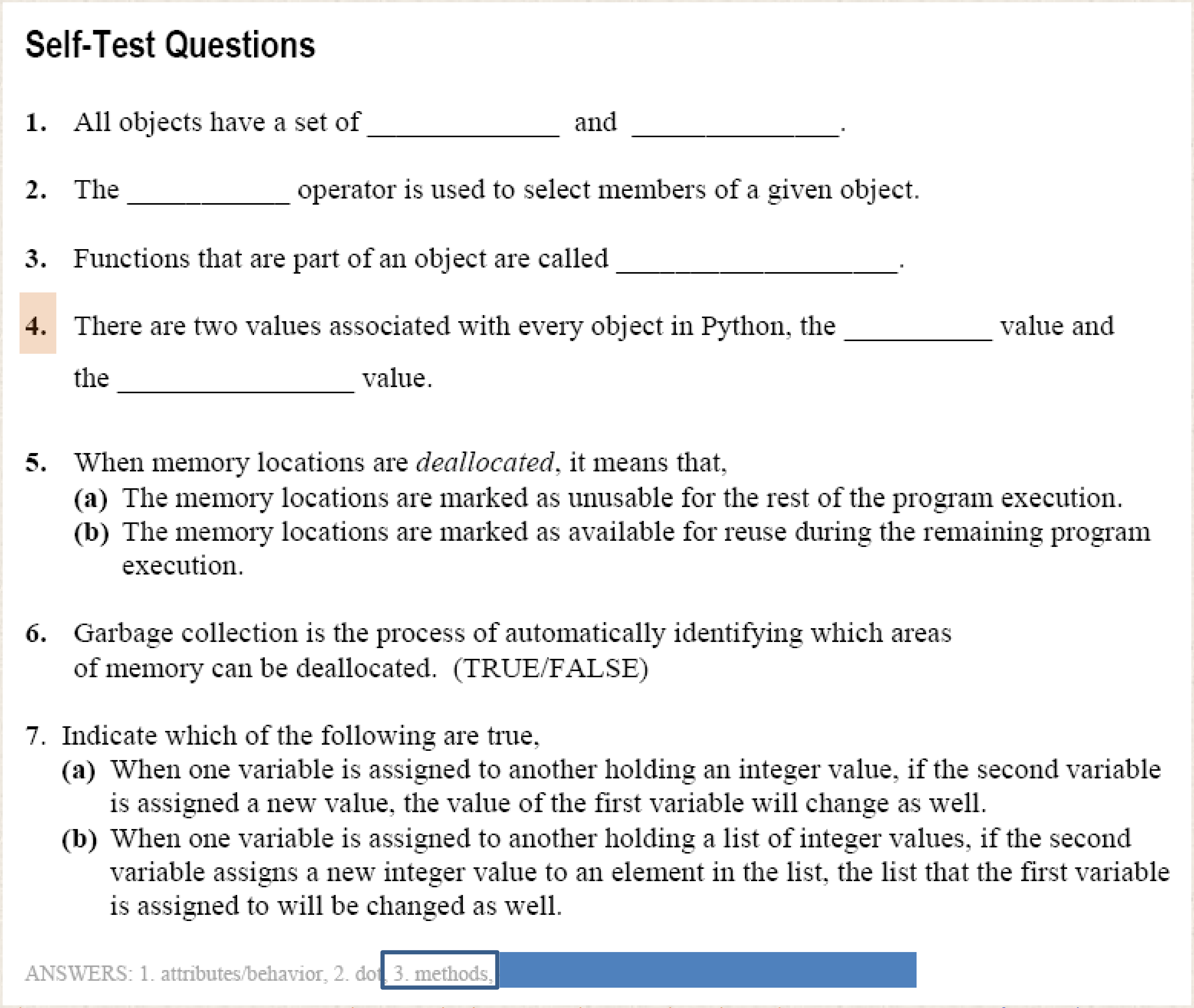
4. There are two values associated with every object in Python, the ___________ value and the _________________ value.

5. When memory locations are *deallocated*, it means that,
(a) The memory locations are marked as unusable for the rest of the program execution.
(b) The memory locations are marked as available for reuse during the remaining program execution.

6. Garbage collection is the process of automatically identifying which areas of memory can be deallocated. (TRUE/FALSE)

7. Indicate which of the following are true,
(a) When one variable is assigned to another holding an integer value, if the second variable is assigned a new value, the value of the first variable will change as well.
(b) When one variable is assigned to another holding a list of integer values, if the second variable assigns a new integer value to an element in the list, the list that the first variable is assigned to will be changed as well.

ANSWERS: 1. attributes/behavio 2. dot, [redacted]



Self-Test Questions

1. All objects have a set of ____________ and _____________.

2. The __________ operator is used to select members of a given object.

3. Functions that are part of an object are called __________________.

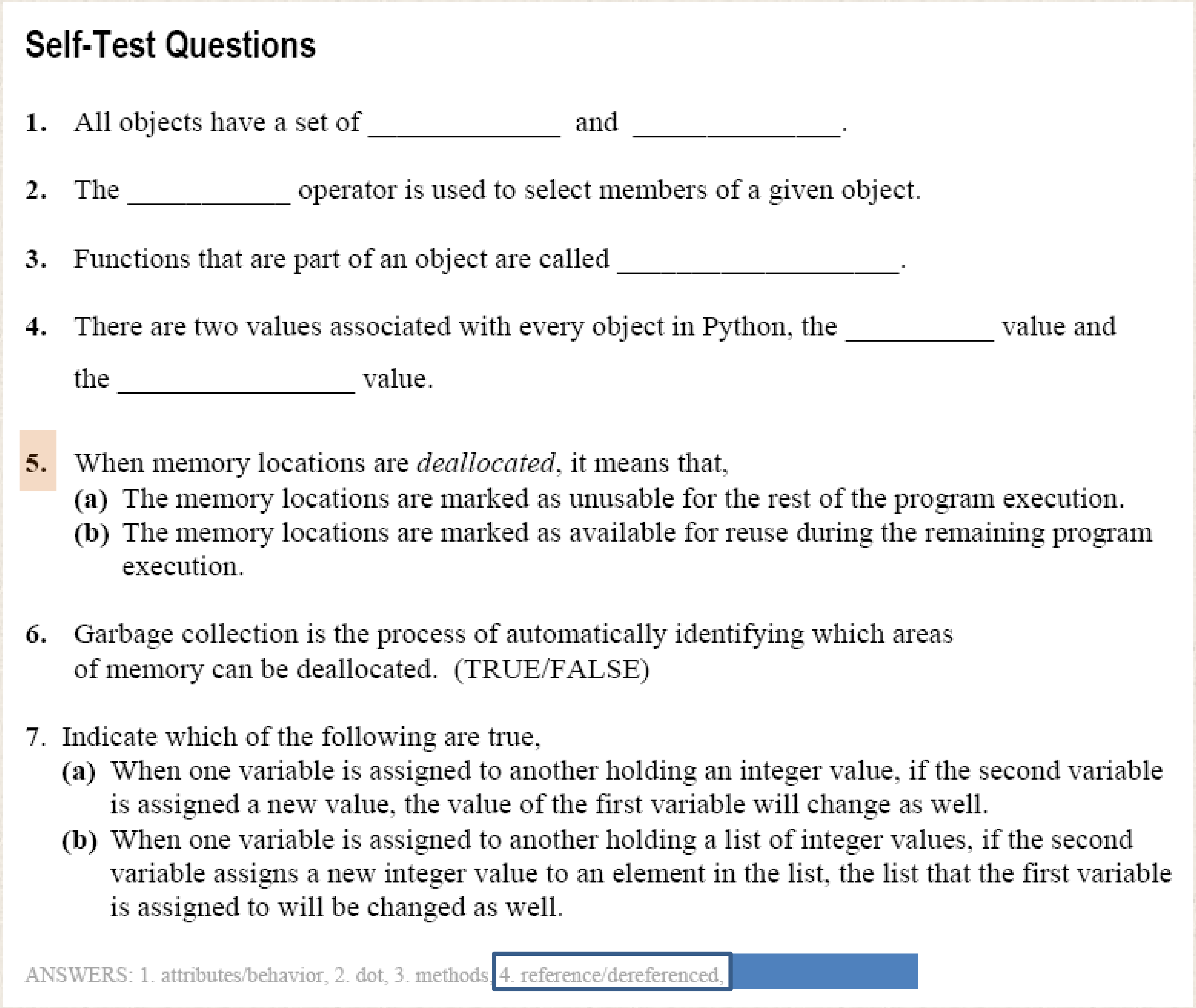
4. There are two values associated with every object in Python, the _________ value and the _______________ value.

5. When memory locations are *deallocated*, it means that,
(a) The memory locations are marked as unusable for the rest of the program execution.
(b) The memory locations are marked as available for reuse during the remaining program execution.

6. Garbage collection is the process of automatically identifying which areas of memory can be deallocated. (TRUE/FALSE)

7. Indicate which of the following are true,
(a) When one variable is assigned to another holding an integer value, if the second variable is assigned a new value, the value of the first variable will change as well.
(b) When one variable is assigned to another holding a list of integer values, if the second variable assigns a new integer value to an element in the list, the list that the first variable is assigned to will be changed as well.

ANSWERS: 1. attributes/behavior, 2. dot, 3. methods, [redacted]



Self-Test Questions

1. All objects have a set of _____________ and ______________.

2. The ___________ operator is used to select members of a given object.

3. Functions that are part of an object are called ___________________.

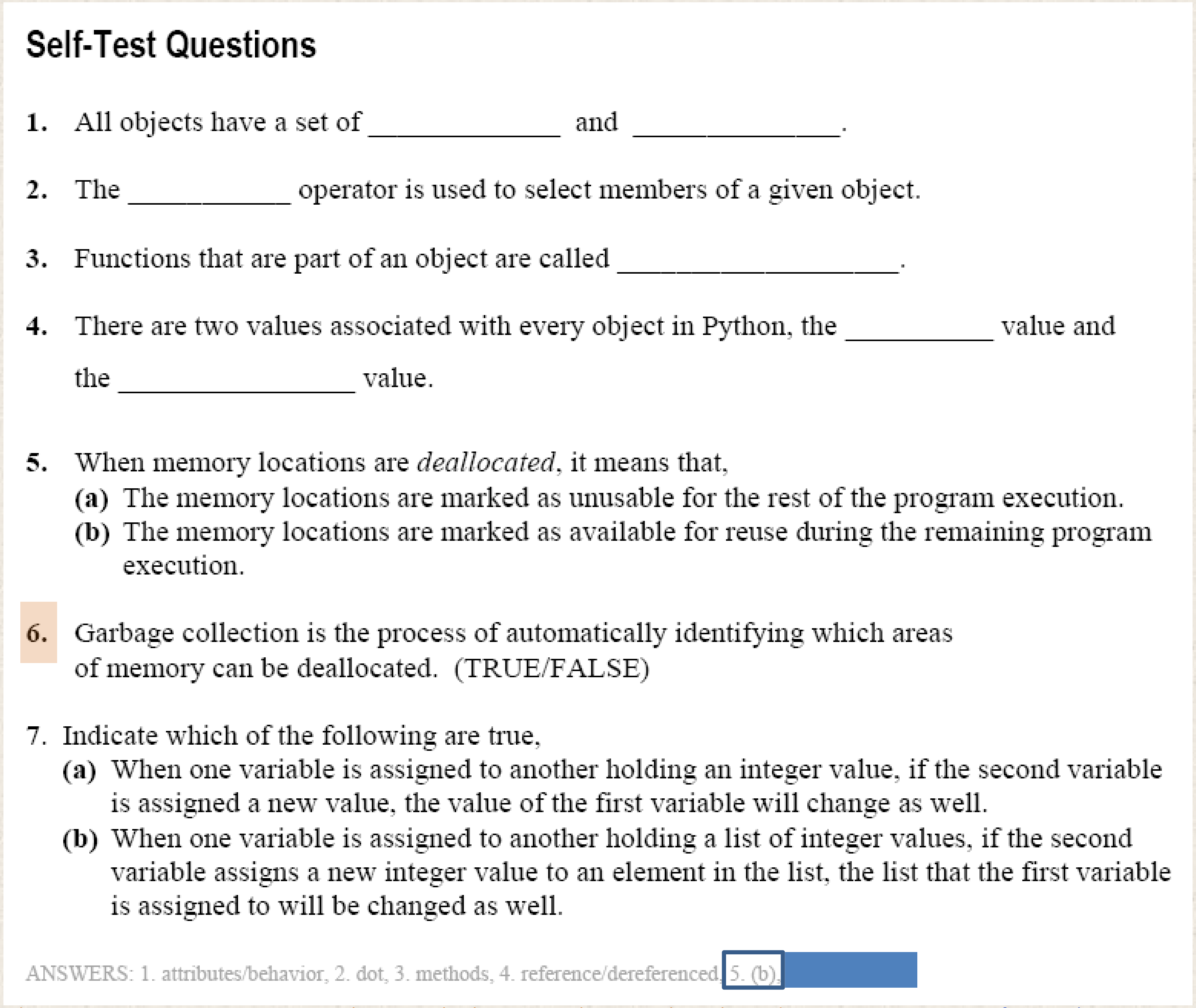
4. There are two values associated with every object in Python, the __________ value and the ________________ value.

5. When memory locations are *deallocated*, it means that,
(a) The memory locations are marked as unusable for the rest of the program execution.
(b) The memory locations are marked as available for reuse during the remaining program execution.

6. Garbage collection is the process of automatically identifying which areas of memory can be deallocated. (TRUE/FALSE)

7. Indicate which of the following are true,
(a) When one variable is assigned to another holding an integer value, if the second variable is assigned a new value, the value of the first variable will change as well.
(b) When one variable is assigned to another holding a list of integer values, if the second variable assigns a new integer value to an element in the list, the list that the first variable is assigned to will be changed as well.

ANSWERS: 1. attributes/behavior, 2. dot, 3. methods, 4. reference/dereferenced, [redacted]



Self-Test Questions

1. All objects have a set of ______________ and ______________.

2. The ____________ operator is used to select members of a given object.

3. Functions that are part of an object are called ____________________.

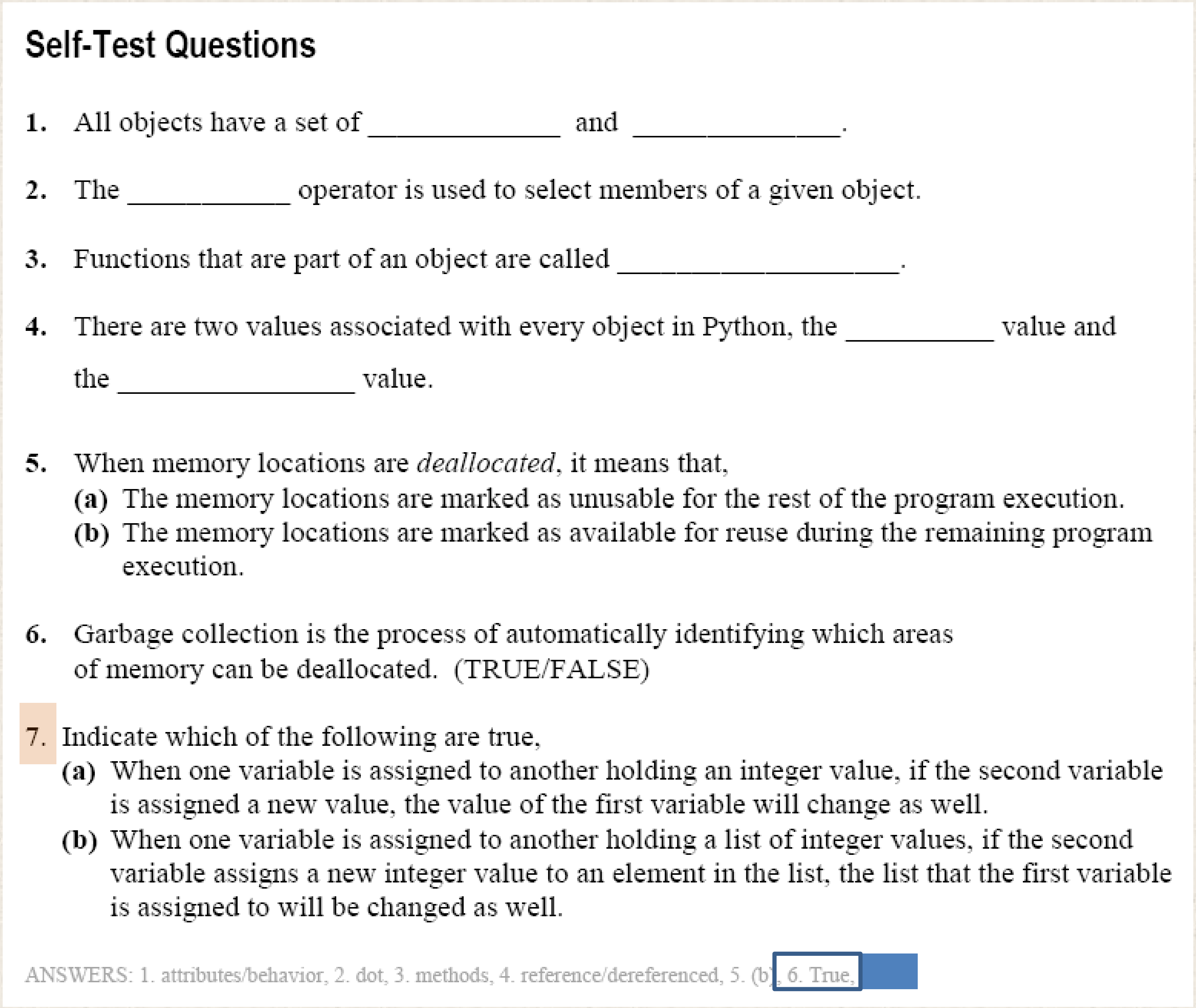
4. There are two values associated with every object in Python, the ___________ value and the _________________ value.

5. When memory locations are *deallocated*, it means that,
(a) The memory locations are marked as unusable for the rest of the program execution.
(b) The memory locations are marked as available for reuse during the remaining program execution.

6. Garbage collection is the process of automatically identifying which areas of memory can be deallocated. (TRUE/FALSE)

7. Indicate which of the following are true,
(a) When one variable is assigned to another holding an integer value, if the second variable is assigned a new value, the value of the first variable will change as well.
(b) When one variable is assigned to another holding a list of integer values, if the second variable assigns a new integer value to an element in the list, the list that the first variable is assigned to will be changed as well.

ANSWERS: 1. attributes/behavior, 2. dot, 3. methods, 4. reference/dereferenced, 5. (b), [redacted]



Self-Test Questions

1. All objects have a set of ______________ and _______________.

2. The ____________ operator is used to select members of a given object.

3. Functions that are part of an object are called _____________________.

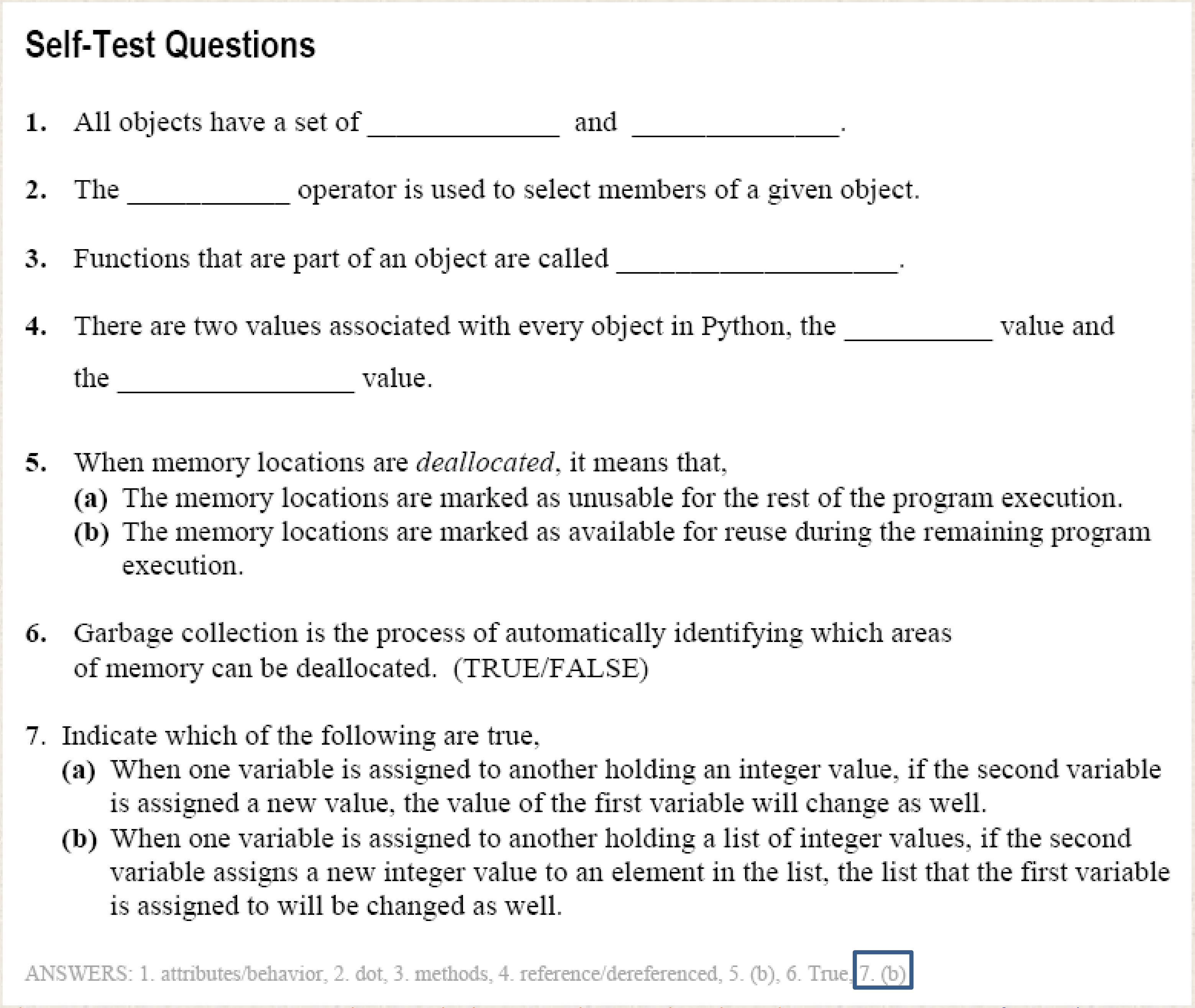
4. There are two values associated with every object in Python, the ___________ value and the __________________ value.

5. When memory locations are *deallocated*, it means that,
   (a) The memory locations are marked as unusable for the rest of the program execution.
   (b) The memory locations are marked as available for reuse during the remaining program execution.

6. Garbage collection is the process of automatically identifying which areas of memory can be deallocated. (TRUE/FALSE)

7. Indicate which of the following are true,
   (a) When one variable is assigned to another holding an integer value, if the second variable is assigned a new value, the value of the first variable will change as well.
   (b) When one variable is assigned to another holding a list of integer values, if the second variable assigns a new integer value to an element in the list, the list that the first variable is assigned to will be changed as well.

ANSWERS: 1. attributes/behavior, 2. dot, 3. methods, 4. reference/dereferenced, 5. (b), 6. True, [redacted]



Self-Test Questions

1. All objects have a set of ______________ and _______________.

2. The ____________ operator is used to select members of a given object.

3. Functions that are part of an object are called _____________________.

4. There are two values associated with every object in Python, the ___________ value and the _________________ value.

5. When memory locations are *deallocated*, it means that,
(a) The memory locations are marked as unusable for the rest of the program execution.
(b) The memory locations are marked as available for reuse during the remaining program execution.

6. Garbage collection is the process of automatically identifying which areas of memory can be deallocated. (TRUE/FALSE)

7. Indicate which of the following are true,
(a) When one variable is assigned to another holding an integer value, if the second variable is assigned a new value, the value of the first variable will change as well.
(b) When one variable is assigned to another holding a list of integer values, if the second variable assigns a new integer value to an element in the list, the list that the first variable is assigned to will be changed as well.

ANSWERS: 1. attributes/behavior, 2. dot, 3. methods, 4. reference/dereferenced, 5. (b), 6. True, $\boxed{\text{7. (b)}}$

Turtle Graphics

Turtle graphics refers to a means of controlling a graphical entity (a "turtle") in a graphics window with x,y coordinates. A turtle can draw lines as it travels, thus creating various graphical designs. Turtle graphics was part of a language named Logo developed in the 1960s for teaching children how to program.

Python provides the capability of turtle graphics in the `turtle` Python standard library module.

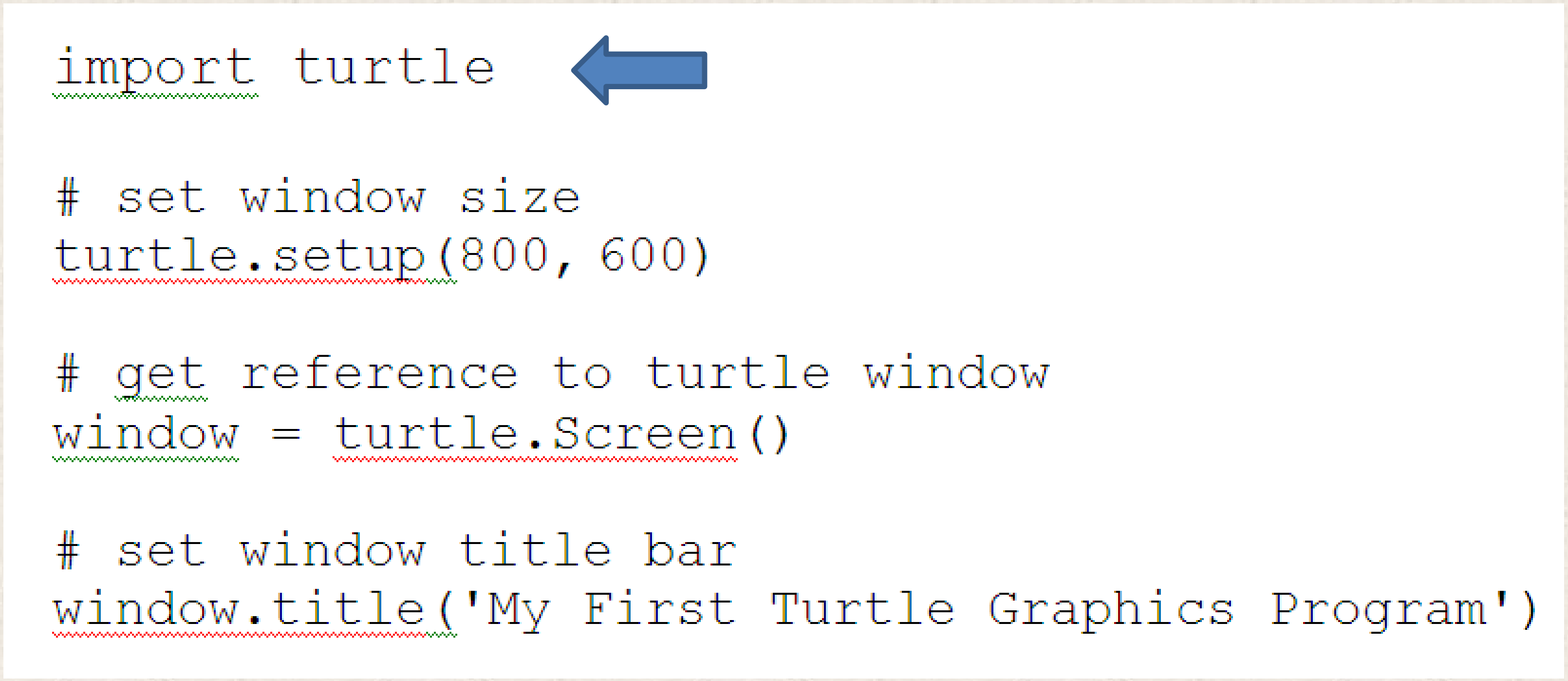
Turtle Graphics (cont.)

There may be more than one turtle on the screen at once. Each turtle is represented by a distinct object. Thus, each can be individually controlled by the methods available for turtle objects.

We introduce turtle graphics here for two reasons — first, to provide a means of better understanding objects in programming, and second, to have some fun!

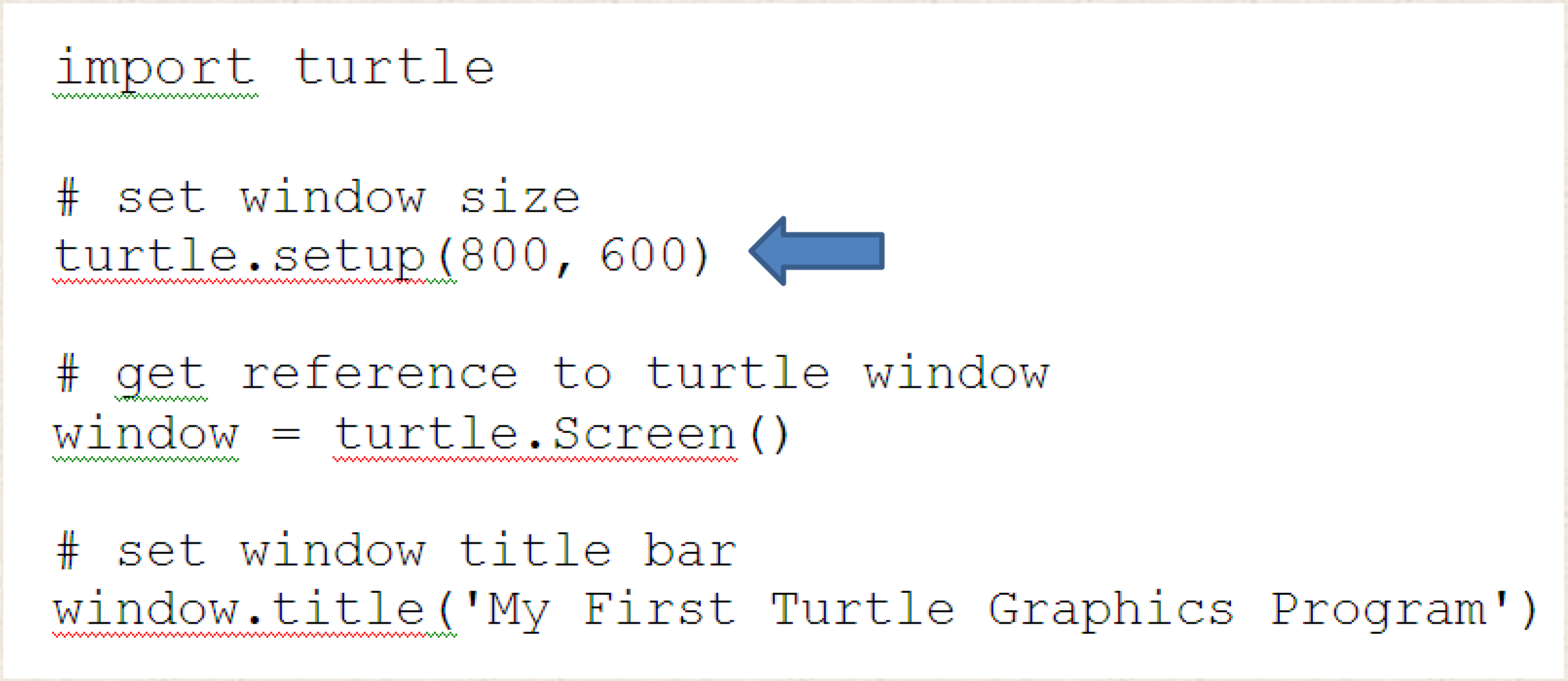
Creating a Turtle Graphics Window

The first step in the use of turtle graphics is the creation of a turtle graphics window (a **turtle screen**).



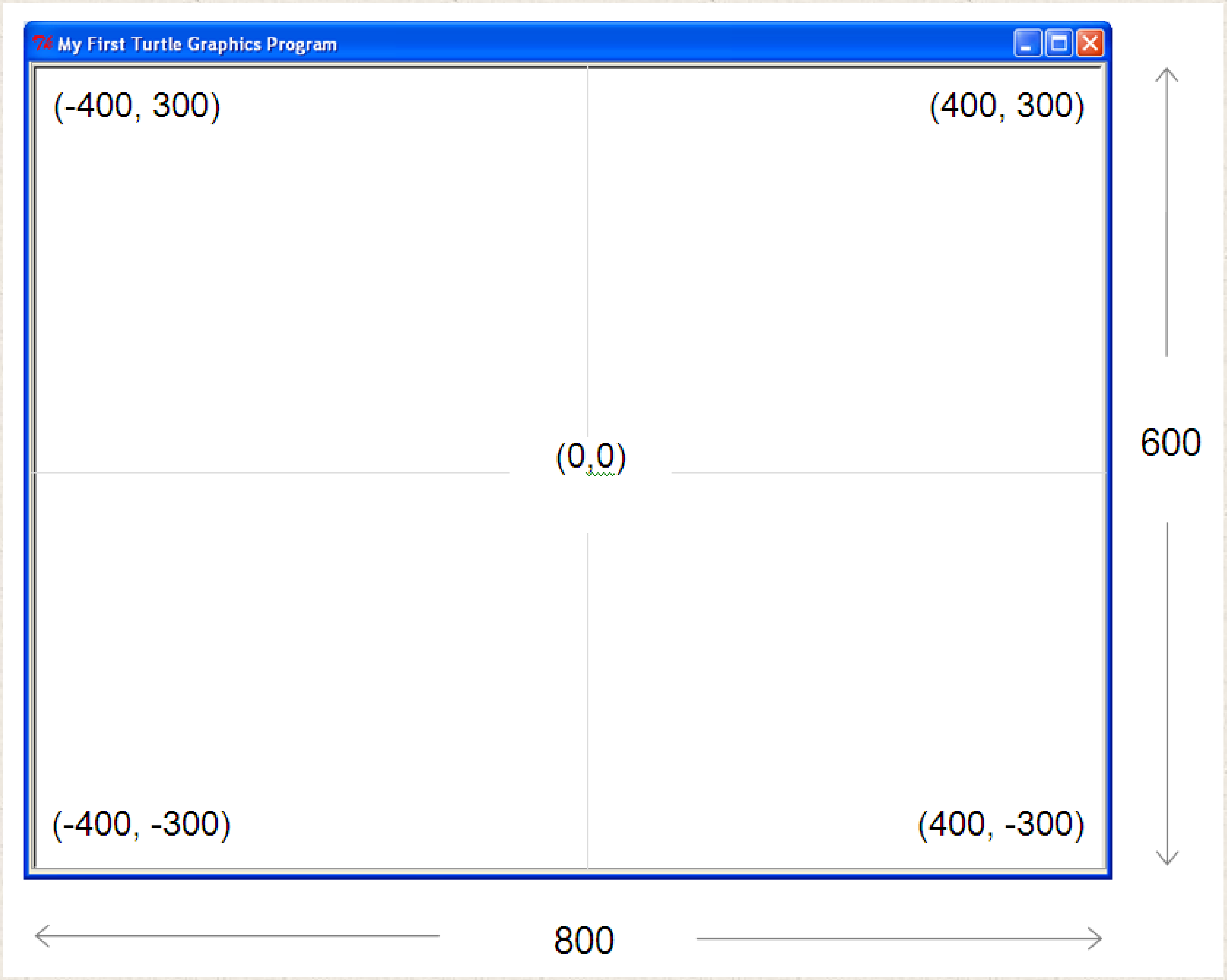
```
import turtle

# set window size
turtle.setup(800, 600)

# get reference to turtle window
window = turtle.Screen()

# set window title bar
window.title('My First Turtle Graphics Program')
```

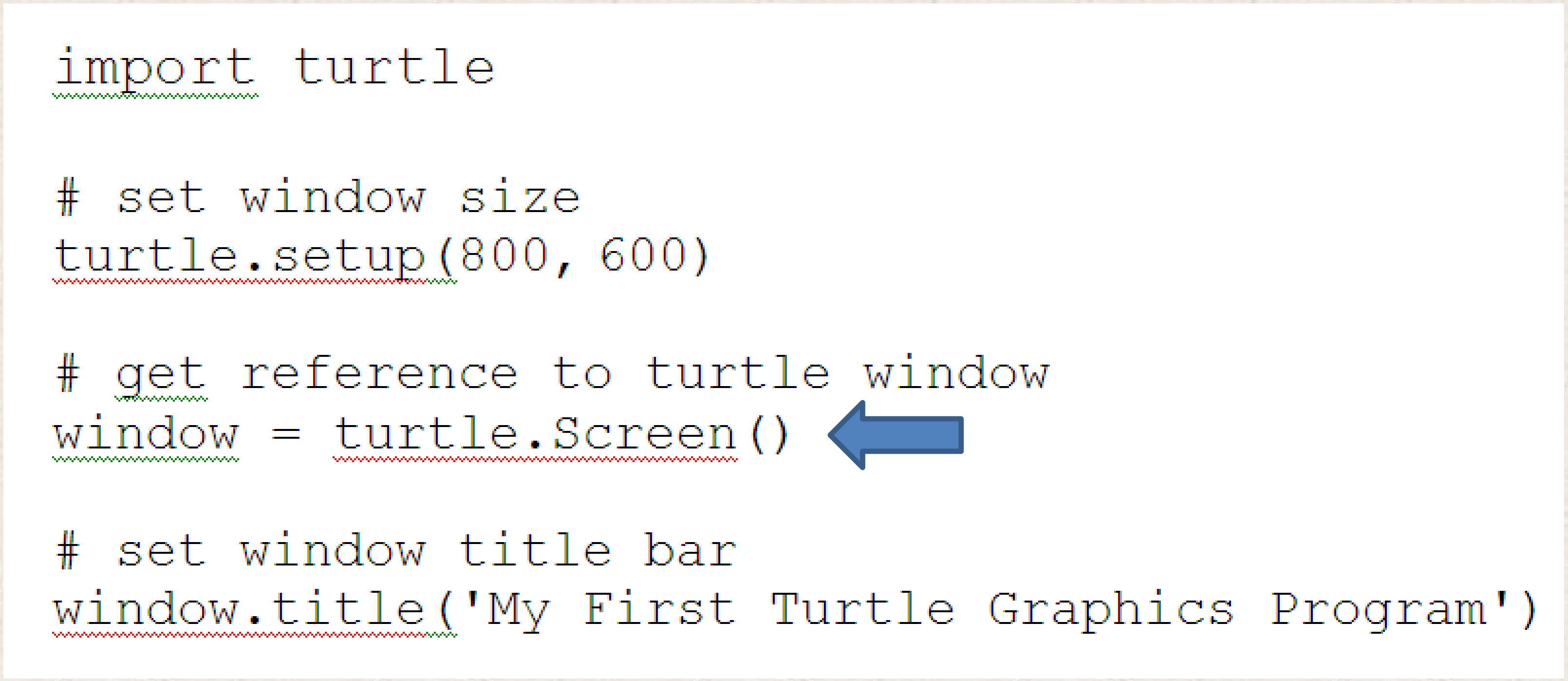
Assuming that the `import moduleName` form of import is used, **each of the turtle graphics methods must be called in the form turtle.*methodname***.



```
import turtle

# set window size
turtle.setup(800, 600)

# get reference to turtle window
window = turtle.Screen()

# set window title bar
window.title('My First Turtle Graphics Program')
```

The first method called, `setup`, **creates a graphics window** of the specified size (in pixels). In this case, a window of size 800 pixels wide by 600 pixels high is created.


My First Turtle Graphics Program
(-400, 300)
(400, 300)
(0,0)
600
(-400, -300)
(400, -300)
800

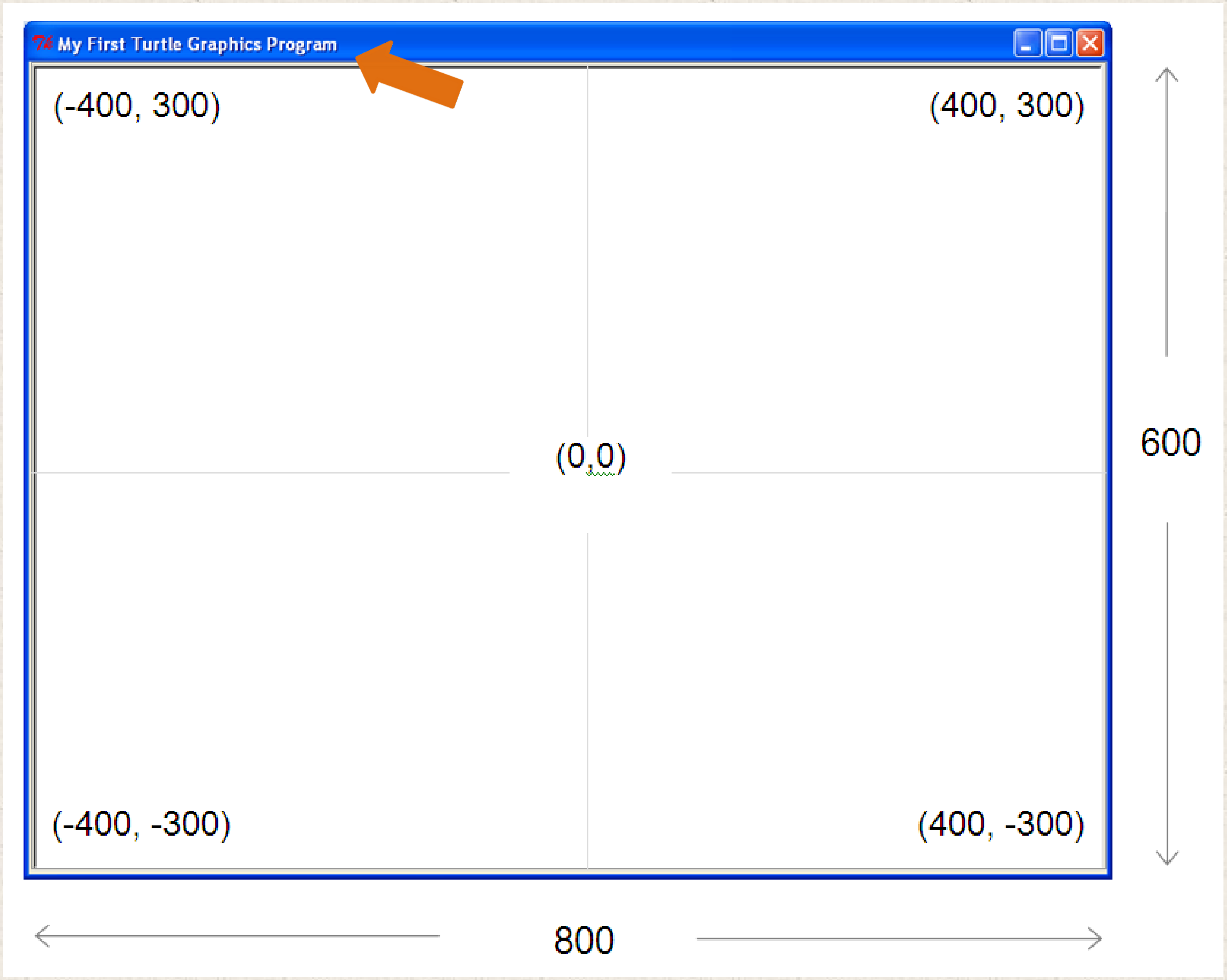


```
import turtle

# set window size
turtle.setup(800, 600)

# get reference to turtle window
window = turtle.Screen()

# set window title bar
window.title('My First Turtle Graphics Program')
```

A call to **`turtle.Screen()`** returns the reference to the screen created.

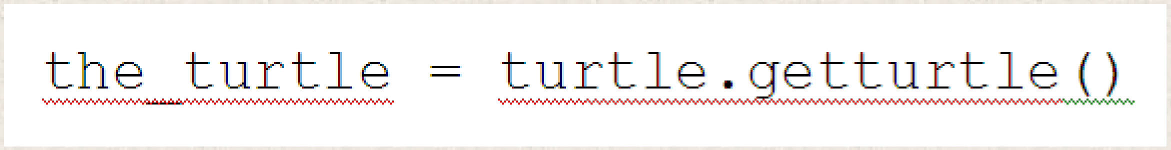
```
import turtle

# set window size
turtle.setup(800, 600)

# get reference to turtle window
window = turtle.Screen()

# set window title bar
window.title('My First Turtle Graphics Program')
```

A call to method **`title`** on `window` sets the top title line for the `window` object created.


My First Turtle Graphics Program
(-400, 300)
(400, 300)
(0,0)
600
(-400, -300)
(400, -300)
800

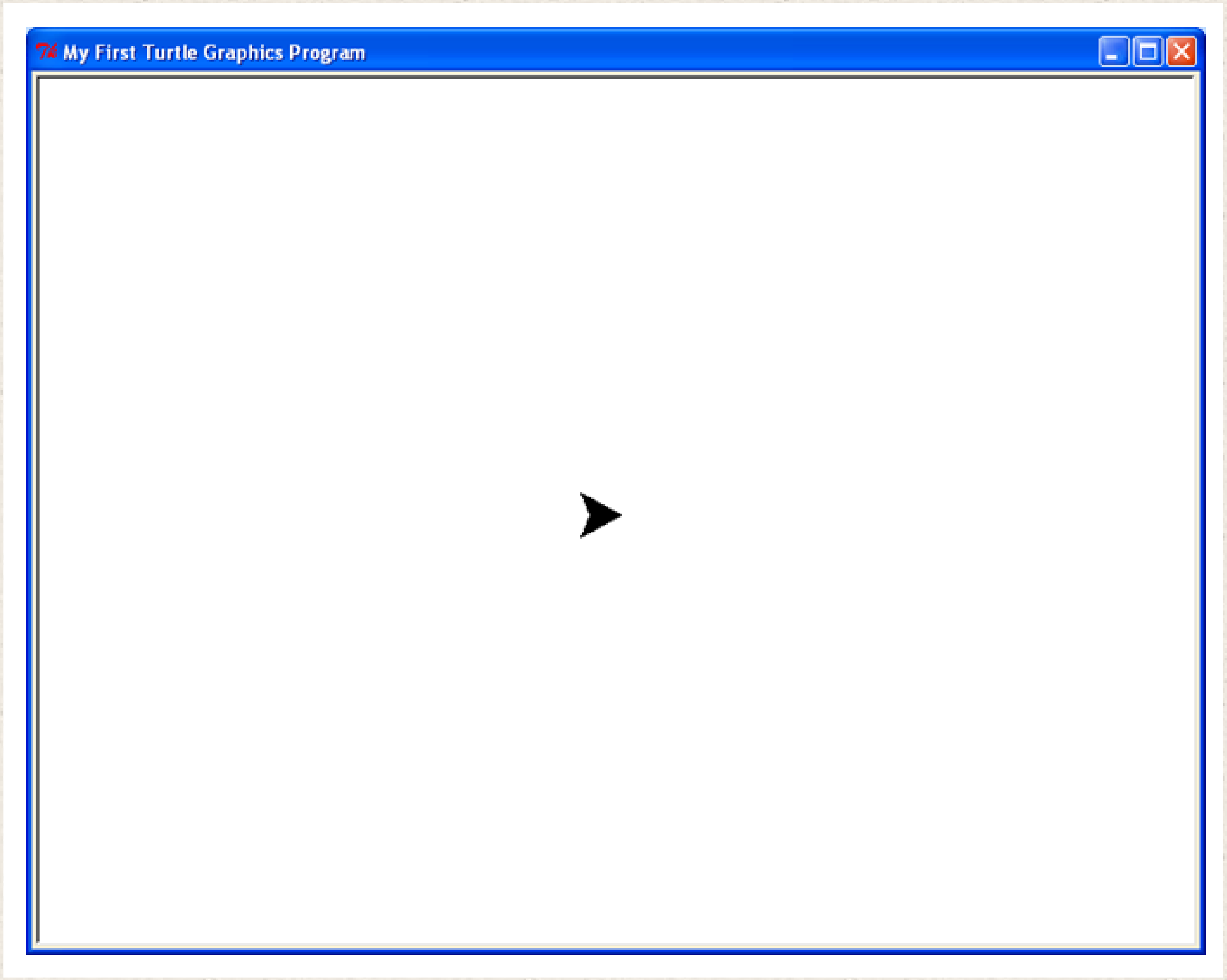
The “Default” Turtle

A “default” turtle **is created when the `setup` method is called**. The reference to this turtle object can be obtained by,



```
the_turtle = turtle.getturtle()
```

A call to `getturtle` causes the default turtle to appear in the window. The initial position of all turtles is the center of the screen at coordinate (0, 0)



The default turtle shape is an arrowhead. (The size of the turtle shape was enlarged from its default size for clarity.)

Fundamental Turtle Attributes and Behavior

Turtle objects have three fundamental attributes:

- *position attributes*
- *heading* (orientation) atributes
- *pen* attributes

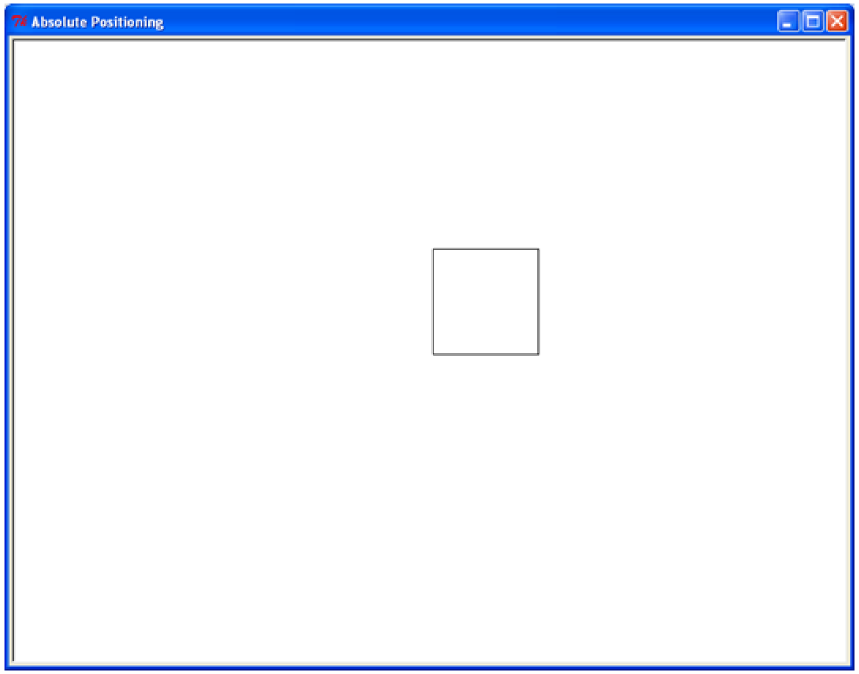
A call to `getturtle` causes the default turtle to appear in the window. The initial position of all turtles is the center of the screen at coordinate (0, 0)

Absolute Positioning

Method **`position`** returns a turtle's current position. Thus, for a newly-created turtle `t.position()` returns the tuple `(0, 0)`.

A turtle's position can be changed using *absolute positioning* by moving the turtle to a specific x,y coordinate by use of method `setposition`.

Creating a Square from Four Line Segments Using Absolute Positioning



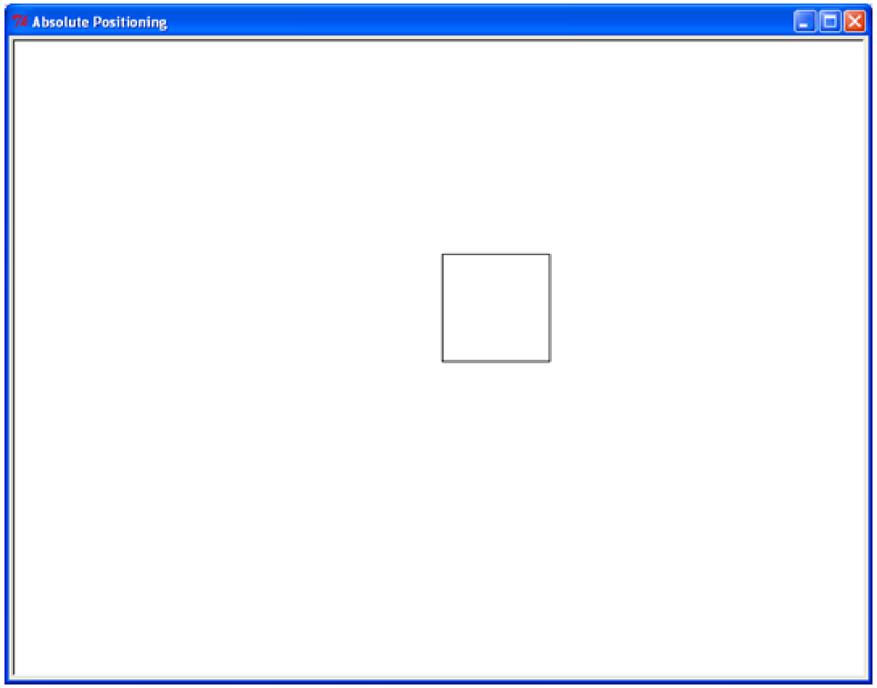
```
# set window title
window = turtle.Screen()
window.title('Absolute Positioning')

# get default turtle and hide
the_turtle = turtle.getturtle()
the_turtle.hideturtle()

# create square (absolute positioning)
the_turtle.setposition(100, 0)
the_turtle.setposition(100, 100)
the_turtle.setposition(0, 100)
the_turtle.setposition(0, 0)
```

The turtle starts at location (0,0), with pen down. It is then positioned at (100, 0), thus drawing a line from (0, 0) to (100, 0) (the bottom line of the square). The turtle is then positioned at (100, 100), then (0, 100), and then back to (0, 0), drawing all four lines of the square.

Turtle Heading and Relative Positioning

A turtle's position can also be changed through *relative positioning*. In this case, the location that a turtle moves to is determined by its second fundamental attribute, its heading.

Creating a Square from Four Line Segments Using Relative Positioning



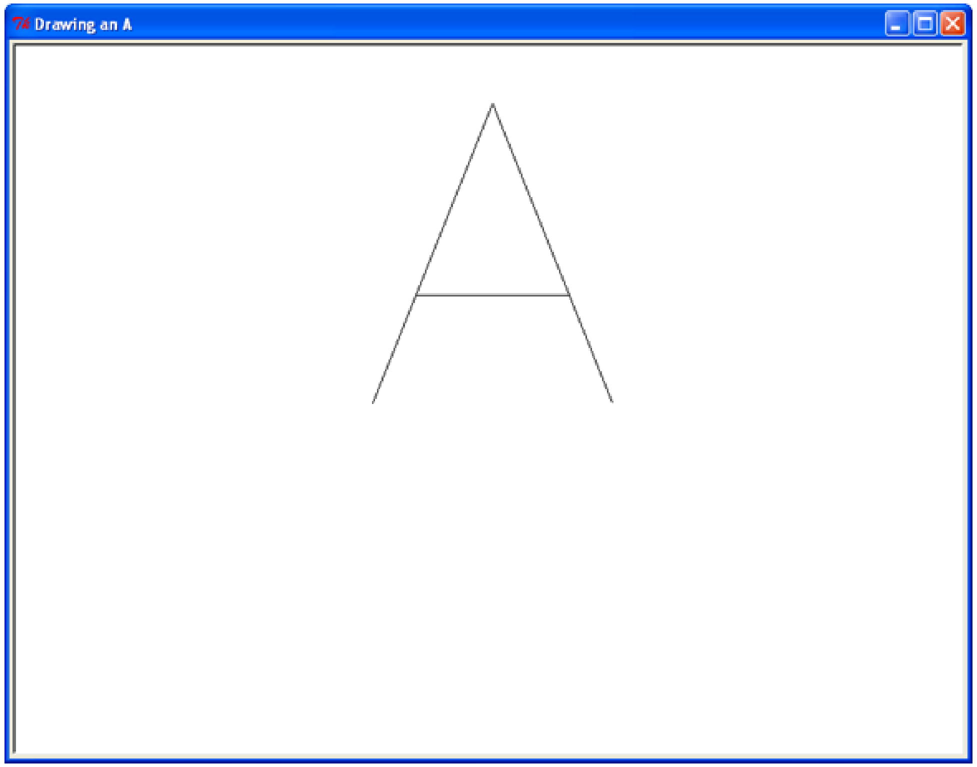
```
# set window title
window = turtle.Screen()
window.title('Relative Positioning')

# get default turtle and hide
the_turtle = turtle.getturtle()
the_turtle.hideturtle()

# create box (relative positioning)
the_turtle.forward(100)
the_turtle.left(90)
the_turtle.forward(100)
the_turtle.left(90)
the_turtle.forward(100)
the_turtle.left(90)
the_turtle.forward(100)
```

Turtles have an initial heading of 0 degrees (facing right). Thus, the turtle is moved forward 100 pixels, drawing the bottom line of the square. By the following pairs of instructions left(90), forward(100), the turtle turns left and move forward 90 degrees, doing this a total of three times, thus completing the square.

Pen Attributes: Pen up and Down

The pen attribute of a turtle object is related to its drawing capabilities. The most fundamental of these attributes is whether the pen is currently “up” or “down,” controlled by methods **penup()** and **pendown()**. When the pen attribute value is “up,” the turtle can be moved to another location without lines being drawn. This is needed when creating drawings with disconnected parts.

Drawing Disconnected Lines



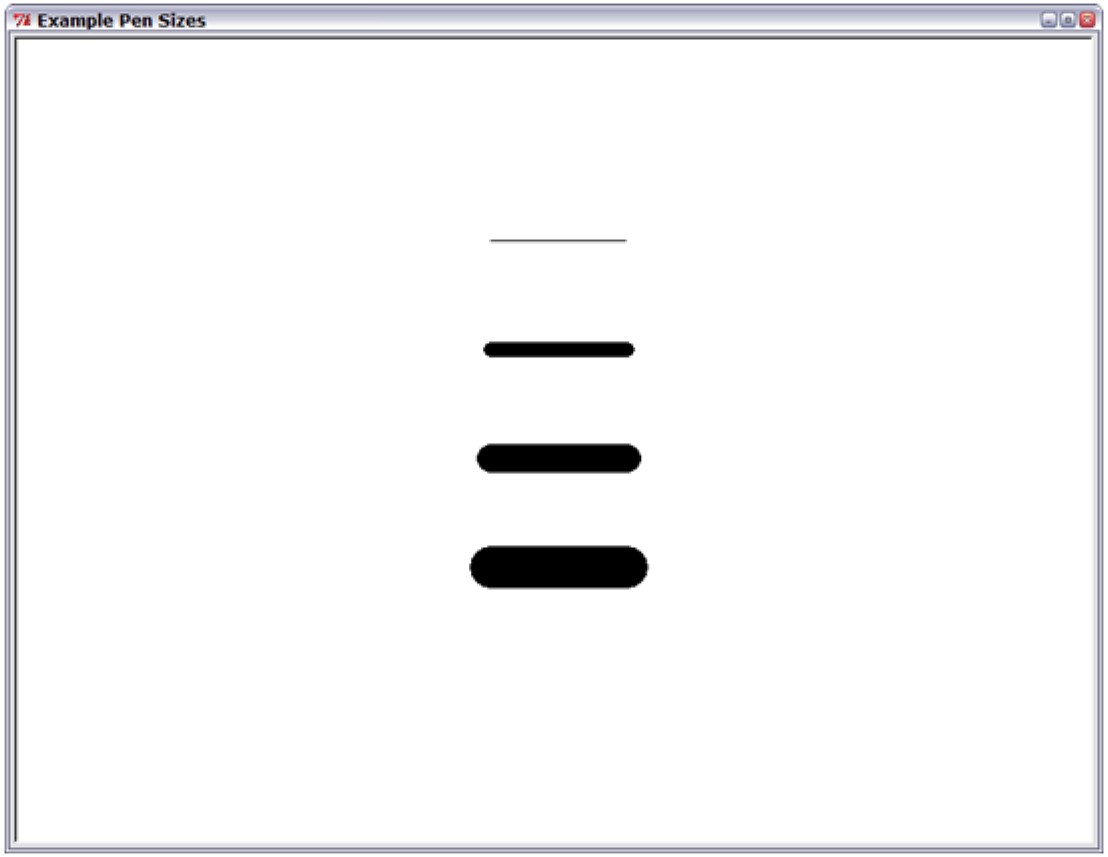
```
turtle.setup(800, 600)
window = turtle.Screen()
window.title('Drawing an A')

the_turtle = turtle.getturtle()
the_turtle.hideturtle()
the_turtle.penup()
the_turtle.setposition(-100, 0)
the_turtle.pendown()
the_turtle.setposition(0, 250)
the_turtle.setposition(100, 0)
the_turtle.penup()
the_turtle.setposition(-64, 90)
the_turtle.pendown()
the_turtle.setposition(64, 90)
```

The **penup** and **pendown** methods are used to allow the turtle to be repositioned.

Pen Attributes: Width of Lines

The pen size of a turtle determines the width of the lines drawn when the pen attribute is "down." The **`pensize`** method is used to control this, **`the_turtle.pensize(5)`**. The line width is given in pixels, and is limited only by the size of the turtle screen.

Setting Line Width



```
pensize(1)

pensize(11)

pensize(21)

pensize(41)
```

Pen Attributes: Pen Color

The **pen color** can be selected by use of the **`pencolor`** method, **the_turtle.pencolor('blue')**. The name of any common color can be used, for example 'white', 'red', 'blue', 'green', 'yellow', 'gray', and 'black'.

Colors can also be specified in RGB (red/green/blue) component values. These values can be specified in the range 0–255 if the color mode attribute of the turtle window is set as given below,

```
turtle.colormode(255)
the_turtle.pencolor(238, 130, 238)  # violet
```

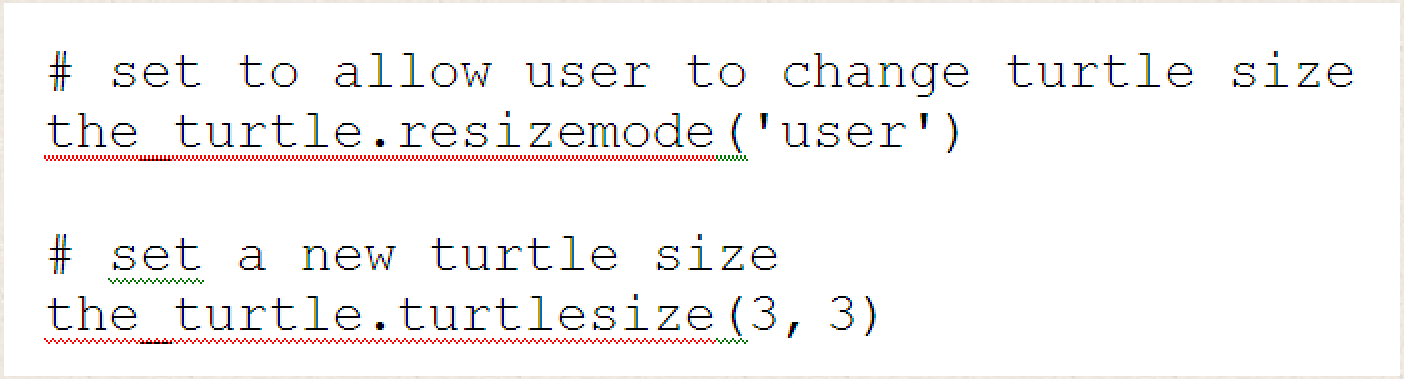
This allows for a full spectrum of colors to be displayed.

Additional Turtle Attributes

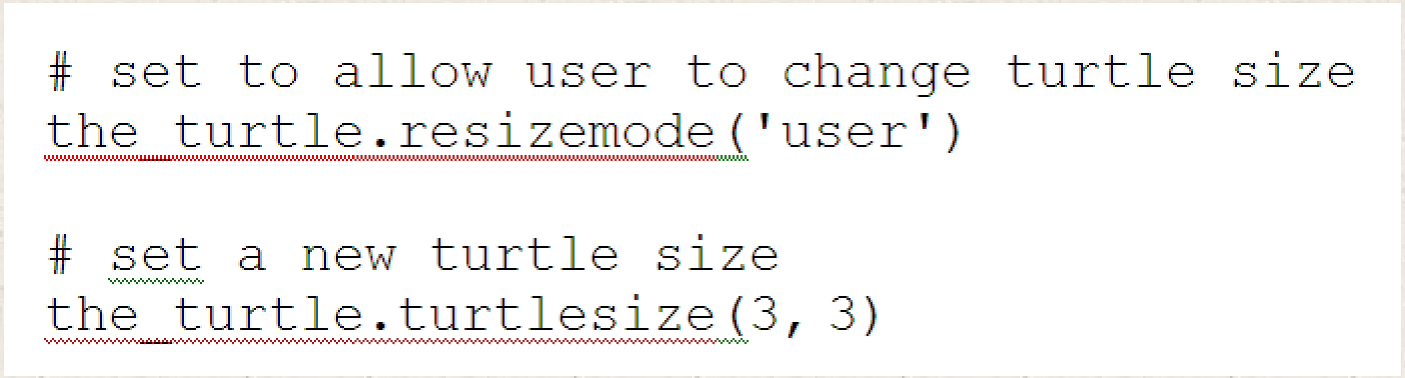
In addition to the fundamental turtle attributes already discussed, there are **other attributes of a turtle** that may be controlled. This includes the size, shape, and fill color of the turtle, the turtle's speed, and the tilt of the turtle.

Turtle Size

The **size of a turtle shape** can be controlled with methods **`resizemode`** and **`turtlesize`**.

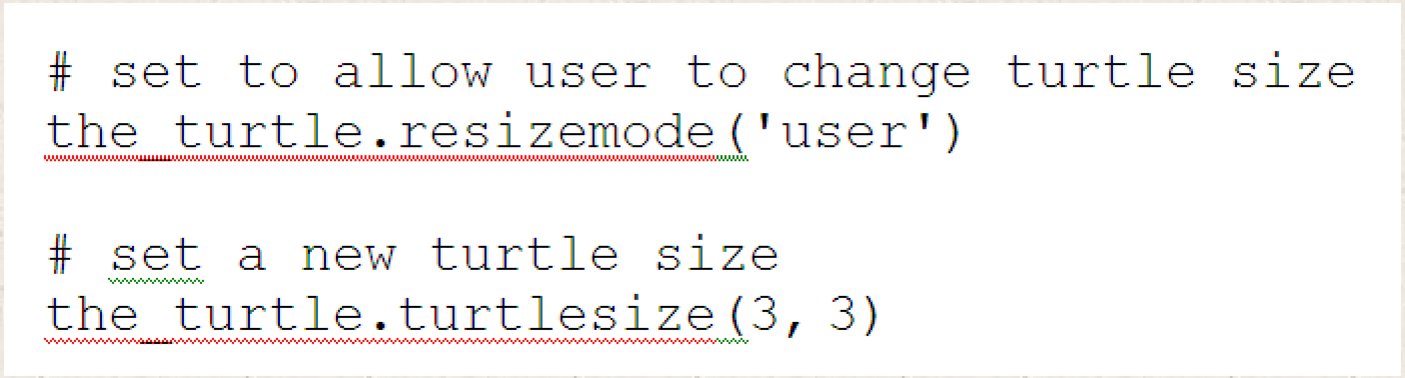


```
# set to allow user to change turtle size
the_turtle.resizemode('user')

# set a new turtle size
the_turtle.turtlesize(3, 3)
```



```
# set to allow user to change turtle size
the_turtle.resizemode('user')

# set a new turtle size
the_turtle.turtlesize(3, 3)
```

First, set the resize attribute of the turtle to **'user'**. This allows the user (programmer) to change the size of the turtle by use of method **`turtlesize`**. Otherwise, a call to the method will have no effect.



```
# set to allow user to change turtle size
the_turtle.resizemode('user')

# set a new turtle size
the_turtle.turtlesize(3, 3)
```

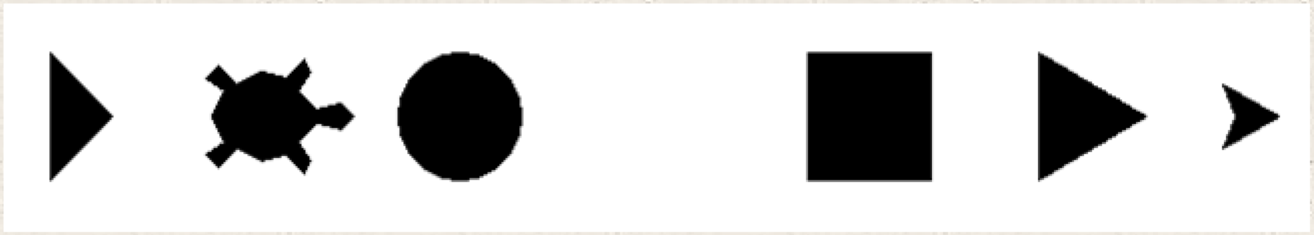
Method **turtlesize** is passed two parameters: the *width* of the shape and its *length*. Each provides a factor by which the size should be changed. **the_turtle.turtlesize(3, 3)** stretches each by a factor of 3. (An optional third parameter determines the thickness of the shape's outline.)

Method resizemode may also be set to **'auto'** which causes the size of the turtle to change with changes to the the pen size, and **'noresize'** , which causes the turtle to remain the same size.

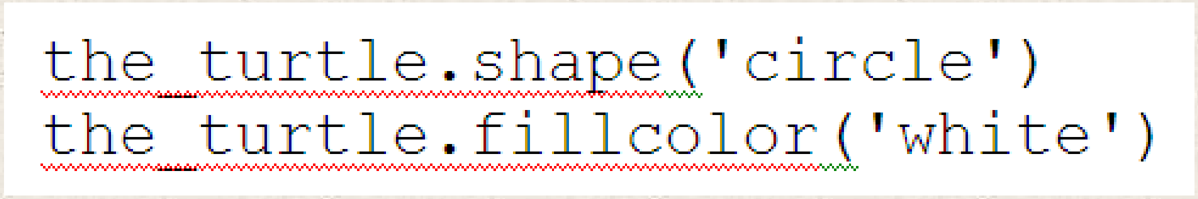
Turtle Shape

There are **a number of ways that a turtle's shape (and fill color) may be defined** to something other than the default shape (the arrowhead) and fill color (black).

A turtle may be assigned one of the following provided shapes: `'arrow'`, `'turtle'`, `'circle'`, `'square'`, `'triangle'`, and `'classic'` (the default arrowhead shape)



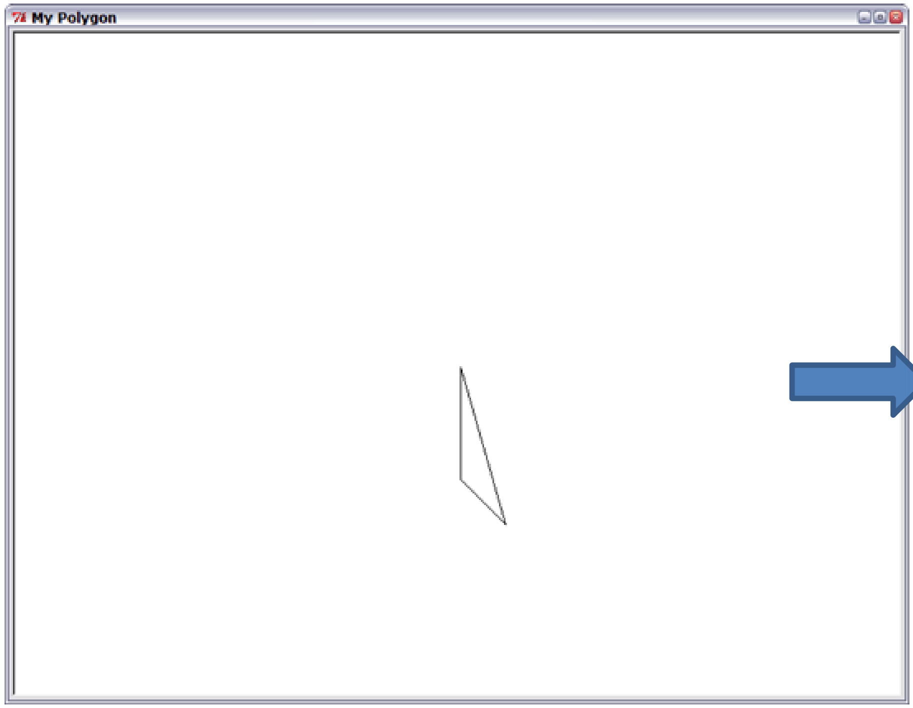
The shape and fill colors are set by use of the `shape` and `fillcolor` methods.



```
the_turtle.shape('circle')
the_turtle.fillcolor('white')
```

New shapes may be created and registered with (added to) the turtle screen's *shape dictionary*. One way of creating a new shape is by providing a set of coordinates denoting a polygon.

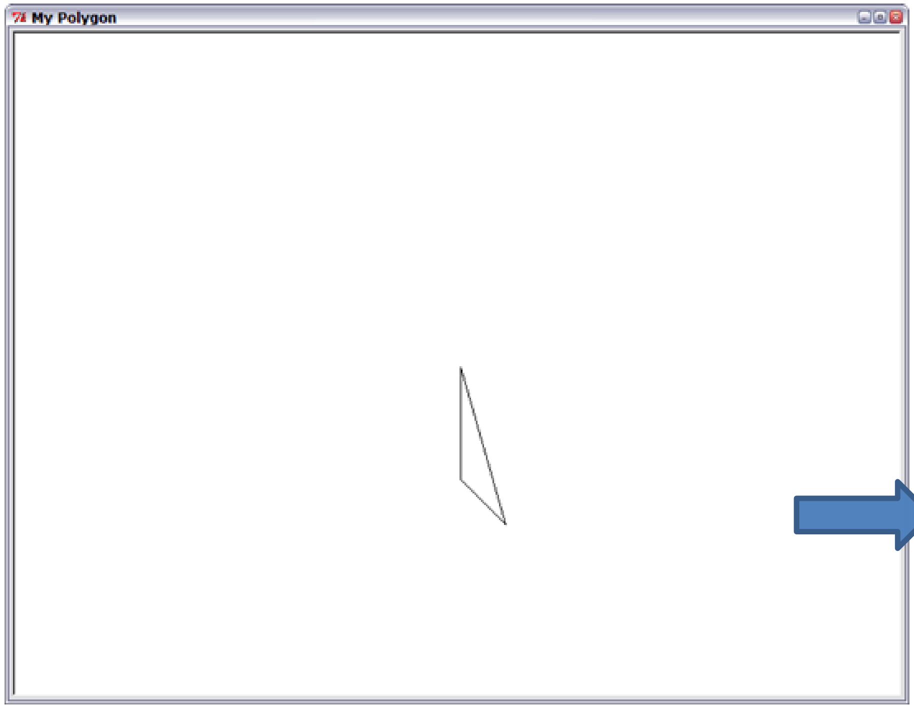
Creating a New Polygon Turtle Shape



```
turtle.setup(800, 600)
window = turtle.Screen()
window.title('My Polygon')
the_turtle = turtle.getturtle()

turtle.register_shape('mypolygon',
((0, 0), (100, 0), (140, 40)))
the_turtle.shape('mypolygon')
the_turtle.fillcolor('white')
```

Method **register_shape** is used to register the new turtle shape with the name `mypolygon`. The new shape is provided by the tuple of coordinates in the second argument, defining the polygon shown in the figure. Once the new shape is defined, a turtle can be set to that shape by calling the shape method with the desired shape's name.

Creating a New Polygon Turtle Shape



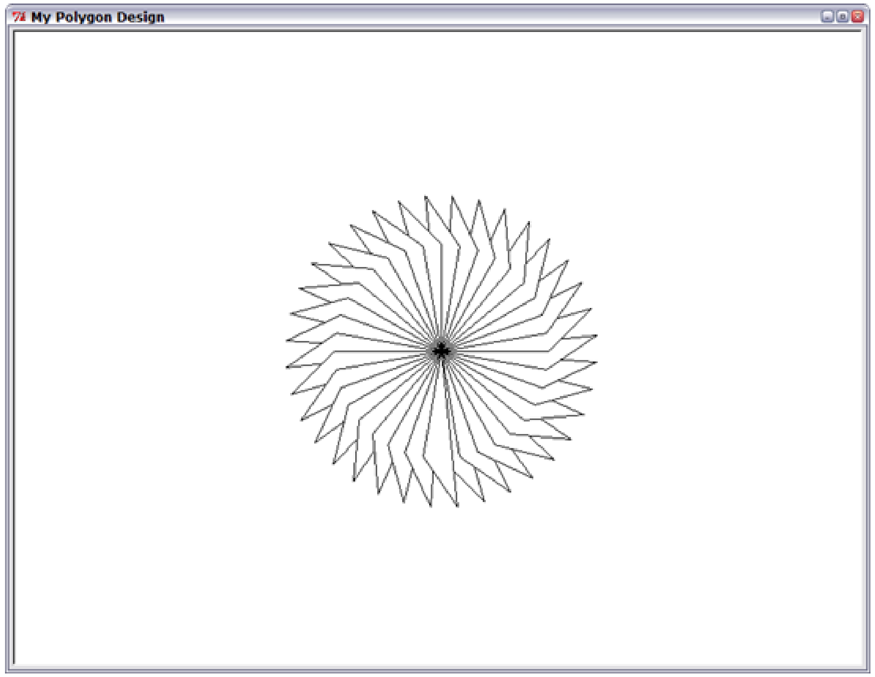
```
turtle.setup(800, 600)
window = turtle.Screen()
window.title('My Polygon')
the_turtle = turtle.getturtle()

turtle.register_shape('mypolygon',
((0, 0), (100, 0), (140, 40)))
the_turtle.shape('mypolygon')
the_turtle.fillcolor('white')
```

The `fillcolor` method is then called to make the fill color of the polygon white (with the edges remaining black). **It is also possible to create turtle shapes composed of various individual polygons**, called *compound shapes*.

Stamping a Turtle on the Screen

A turtle is able to stamp its shape on the screen by use of the **stamp** method, which remains there even after the turtle is repositioned (or relocated). That means that we can create all sorts interesting graphic patterns by appropriately repositioning the turtle.

Stamping a Turtle Shape on the Screen

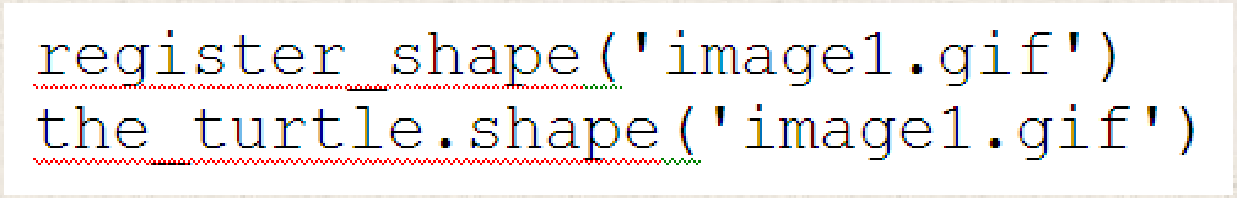


```
turtle.setup(800, 600)
window = turtle.Screen()
window.title('My Polygon Design')
the_turtle = turtle.getturtle()

turtle.register_shape('mypolygon',
((0, 0), (100, 0), (140, 40)))
the_turtle.shape('mypolygon')
the_turtle.fillcolor('white')

for angle in range(0, 360, 10):
    the_turtle.setheading(angle)
    the_turtle.stamp()
```

Only a few lines of code are needed to generate this design. The for loop iterates variable **`angle`** from 0 to 360 degrees (by increments of 10 degrees). Within the loop the **`stamp()`** method is called to stamp the polygon shape at the turtle's current position. By varying the shape of the polygon and the angles that the turtle is set to, a wide range of such designs may be produced.

Creating a Turtle Shape from an Image

Another way that **a turtle shape can be created is by use of an existing image**. The image file used must be a "gif file" (with file extension .gif). The name of the file is then registered and the shape of the turtle set to the registered name,



```
register_shape('image1.gif')
the_turtle.shape('image1.gif')
```

Controlling the Turtle Speed

A turtle's speed can be set to a value from 0 to 10, with a "normal" speed being around 6. The **`speed`** method is used for this, **`the_turtle.speed(6)`**. The following speed values can be set using a descriptive, rather than a numerical value.

```
10: 'fast'    6: 'normal'    3: 'slow'    1: 'slowest'    0: 'fastest'
```

Thus, a normal speed can be set by **the_turtle.speed('normal')**. When using the turtle for line drawing only, the turtle will move more quickly if it is made invisible (by use of the **`hideturtle`** method).

Creating Multiple Turtles

It is possible to create and control any number of turtle objects. To create a new turtle, the **Turtle()** method is used.

```
turtle1 = turtle.Turtle()
turtle2 = turtle.Turtle()
etc.
```

Any number of turtles may be maintained in a list.

```
turtles = []
turtles.append(turtle.Turtle())
turtles.append(turtle.Turtle())
etc.
```

Each turtle has its own distinct set of attributes.

Let’s Apply It

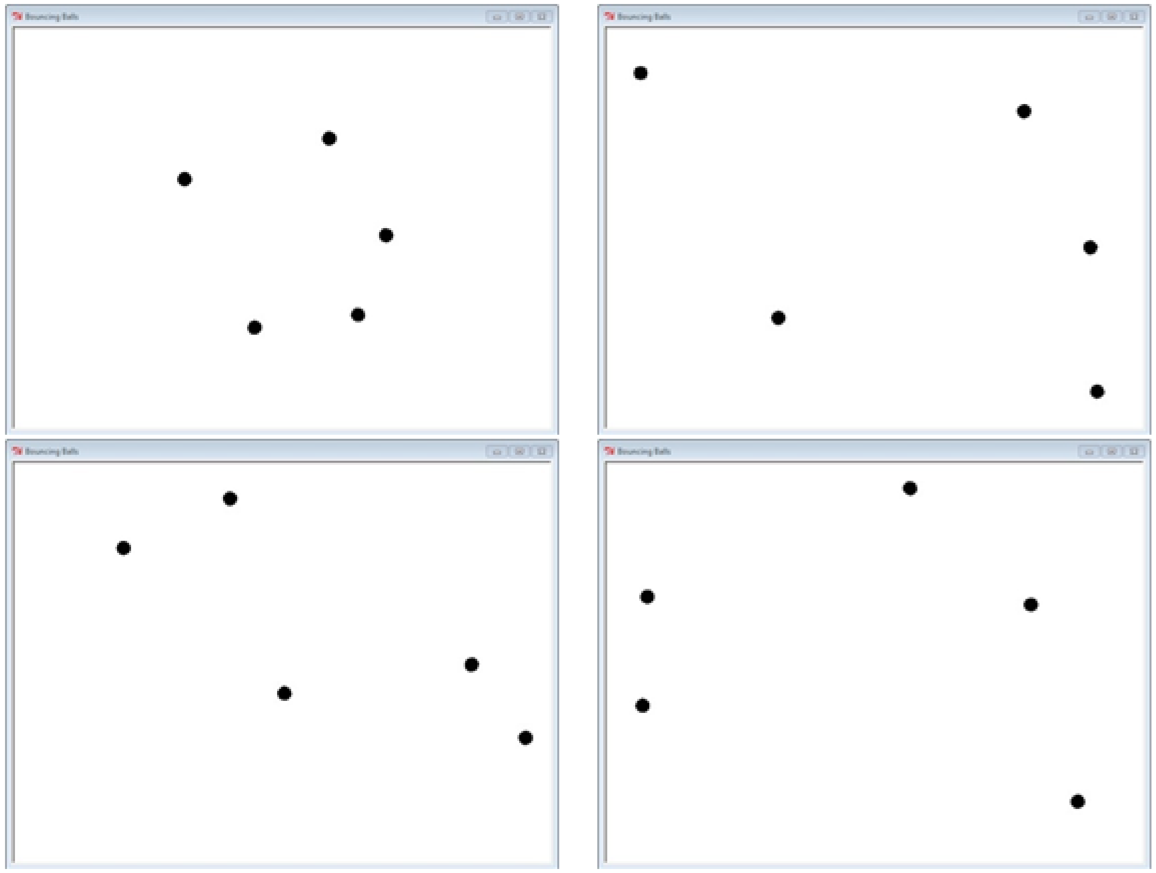
Bouncing Balls Program

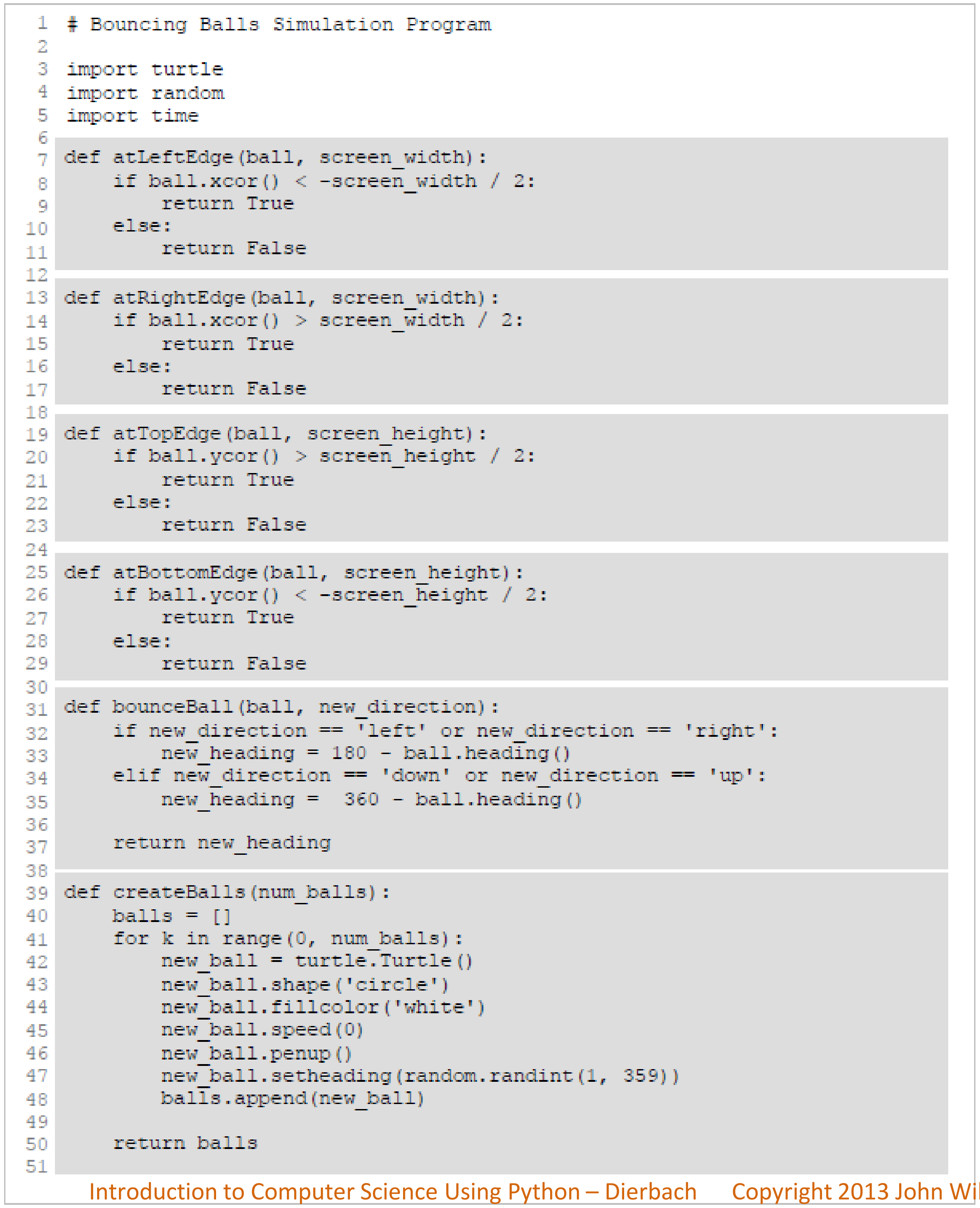
The following program displays one or more bouncing balls within a turtle screen window. This program utilizes the following programming features.

► turtle module ► time module

Program Execution ….

```
This program simulates one or more bouncing balls on a turtle screen.
Enter number of seconds to run: 30
Enter number of balls in simulation: 5
```





```
 1 # Bouncing Balls Simulation Program
 2
 3 import turtle
 4 import random
 5 import time
 6
 7 def atLeftEdge(ball, screen_width):
 8     if ball.xcor() < -screen_width / 2:
 9         return True
10     else:
11         return False
12
13 def atRightEdge(ball, screen_width):
14     if ball.xcor() > screen_width / 2:
15         return True
16     else:
17         return False
18
19 def atTopEdge(ball, screen_height):
20     if ball.ycor() > screen_height / 2:
21         return True
22     else:
23         return False
24
25 def atBottomEdge(ball, screen_height):
26     if ball.ycor() < -screen_height / 2:
27         return True
28     else:
29         return False
30
31 def bounceBall(ball, new_direction):
32     if new_direction == 'left' or new_direction == 'right':
33         new_heading = 180 - ball.heading()
34     elif new_direction == 'down' or new_direction == 'up':
35         new_heading =  360 - ball.heading()
36
37     return new_heading
38
39 def createBalls(num_balls):
40     balls = []
41     for k in range(0, num_balls):
42         new_ball = turtle.Turtle()
43         new_ball.shape('circle')
44         new_ball.fillcolor('white')
45         new_ball.speed(0)
46         new_ball.penup()
47         new_ball.setheading(random.randint(1, 359))
48         balls.append(new_ball)
49
50     return balls
51
```

In addition to the turtle graphics module, the `time` and `random` Python standard library modules are imported to allow control of how long the simulation is executed, and to generate the random motion of the bouncing balls.

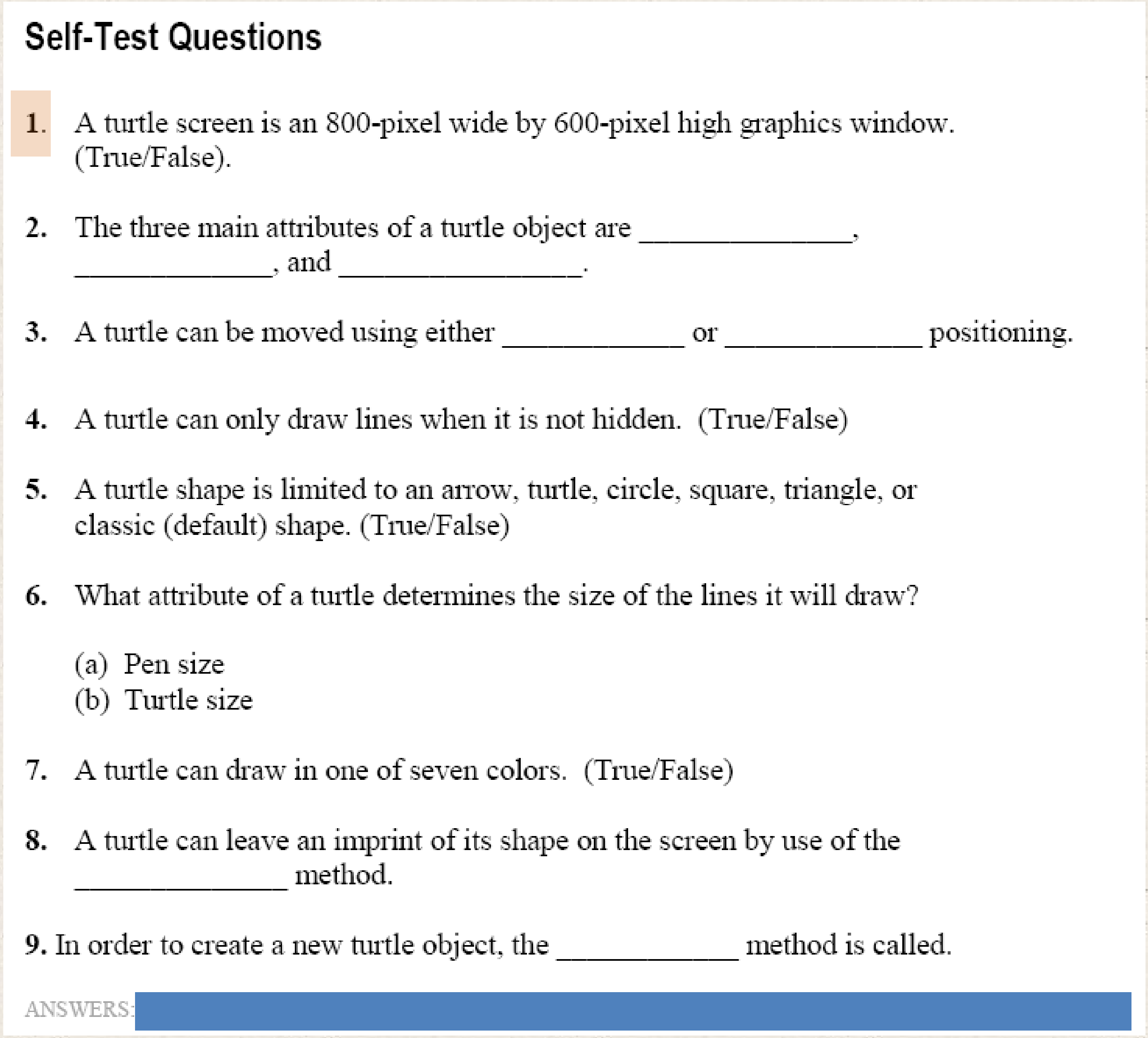
Functions `atLeftEdge`, `atRightEdge`, `atTopEdge`, and `atBottomEdge` **(lines 7-37)** are used to determine when a ball should be bounced off of a wall. Function `bounceBall` is called to do the bouncing.

Function `createBalls` **(lines 39–50)** initializes an empty list named `balls` and creates the requested number of balls one-by-one, each appended to the list. Each ball is created with shape 'circle', fill color of 'white', speed of 0 (fastest speed), and pen attribute 'up'. In addition, the initial heading of each ball turtle is set to a random angle between 1 and 359 (**line 47**).

```
52 # ---- main
53 # program greeting
54 print('This program simulates one or more bouncing balls on a turtle
55 screen.')
56
57 # init screen size
58 screen_width = 800
59 screen_height = 600
60 turtle.setup(screen_width, screen_height)
61
62 # create turtle window
63 window = turtle.Screen()
64 window.title('Bouncing Balls')
65
66 # prompt user for execution time and number of balls
67 num_seconds = int(input('Enter number of seconds to run: '))
68 num_balls = int(input('Enter number of balls in simulation: '))
69
70 # create balls
71 balls = createBalls(num_balls)
72
73 # set start time
74 start_time = time.time()
75
76 # begin simulation
77 terminate = False
78
79 while not terminate:
80     for k in range(0, len(balls)):
81         balls[k].forward(15)
82
83         if atLeftEdge(balls[k], screen_width):
84             balls[k].setheading(bounceBall(balls[k], 'right'))
85         elif atRightEdge(balls[k], screen_width):
86             balls[k].setheading(bounceBall(balls[k], 'left'))
87         elif atTopEdge(balls[k], screen_height):
88             balls[k].setheading(bounceBall(balls[k], 'down'))
89         elif atBottomEdge(balls[k], screen_height):
90             balls[k].setheading(bounceBall(balls[k], 'up'))
91
92         if time.time() - start_time > num_seconds:
93             terminate = True
94
95 # exit on close window
96 turtle.exitonclick()
```

The while loop on **line 79** begins the simulation. The loop iterates as long as Boolean variable `terminate` is not `True` (initialized to `False` on **line 77**). The for loop within this loop at **line 80** moves each of the specified number of balls a small distance until reaching one of the four edges of the window (left, right, top, or bottom), using Boolean functions `atLeftEdge`, `atRightEdge`, `atTopEdge`, and `atBottomEdge`.

Function `bounceBall` is called to bounce the ball in the opposite direction heading, and returns the new heading of the ball, passed as the argument to that ball's `setheading` method. Finally, on **line 92** a check is made to determine whether the user-requested simulation time has been exceeded. If so, the Boolean variable `terminate` is set to `True`, and the program terminates. Because of the call to `exitonclick()` on **line 96**, the program will properly shut down when the close button of the turtle window is clicked on.



Self-Test Questions

1. A turtle screen is an 800-pixel wide by 600-pixel high graphics window. (True/False).

2. The three main attributes of a turtle object are ______________, ______________, and _________________.

3. A turtle can be moved using either ____________ or _____________ positioning.

4. A turtle can only draw lines when it is not hidden. (True/False)

5. A turtle shape is limited to an arrow, turtle, circle, square, triangle, or classic (default) shape. (True/False)

6. What attribute of a turtle determines the size of the lines it will draw?

   (a) Pen size
   (b) Turtle size

7. A turtle can draw in one of seven colors. (True/False)

8. A turtle can leave an imprint of its shape on the screen by use of the _______________ method.

9. In order to create a new turtle object, the ____________ method is called.

ANSWERS: [redacted]

Self-Test Questions

1. A turtle screen is an 800-pixel wide by 600-pixel high graphics window. (True/False).

2. The three main attributes of a turtle object are ______________, ______________, and _________________.

3. A turtle can be moved using either _____________ or ______________ positioning.

4. A turtle can only draw lines when it is not hidden. (True/False)

5. A turtle shape is limited to an arrow, turtle, circle, square, triangle, or classic (default) shape. (True/False)

6. What attribute of a turtle determines the size of the lines it will draw?

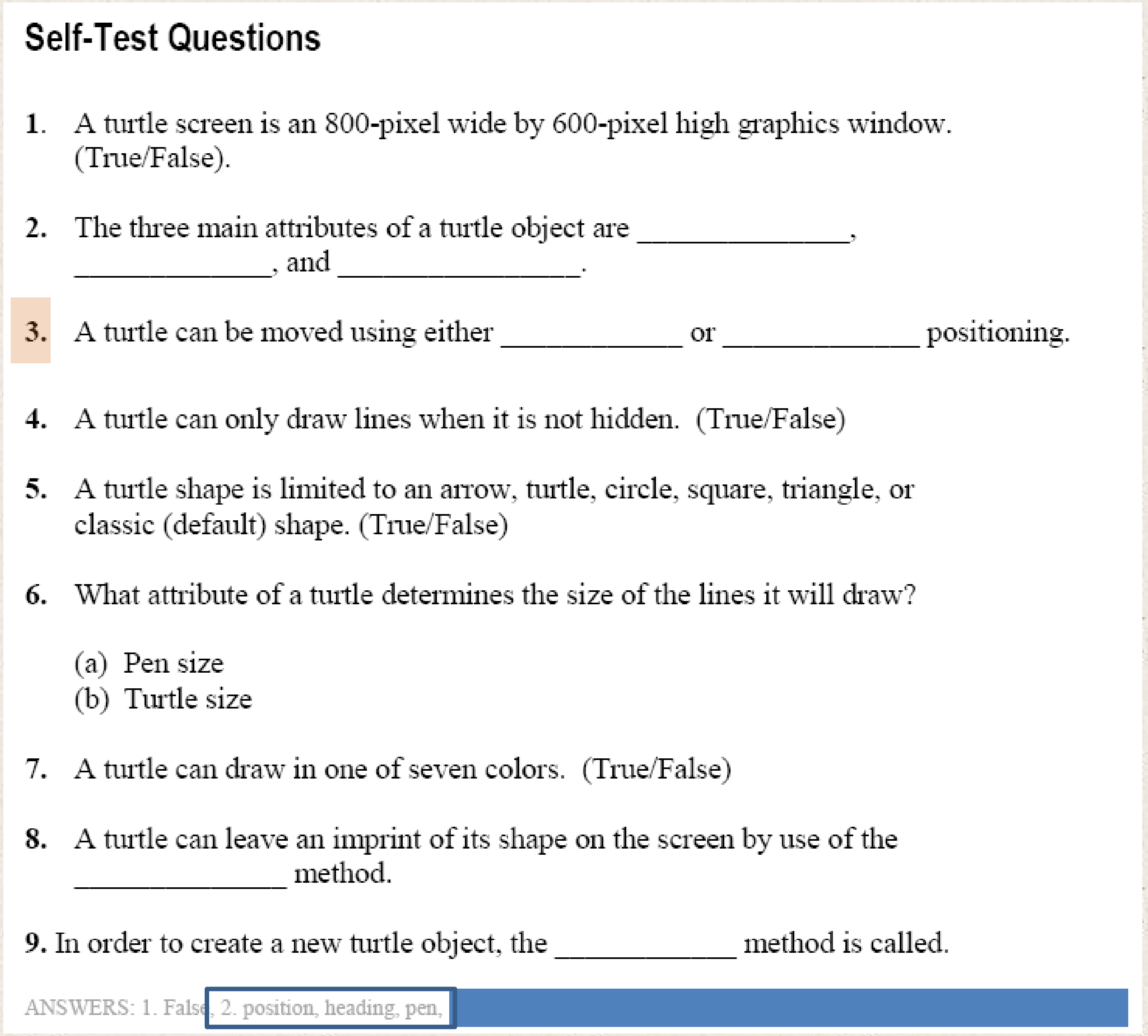
   (a) Pen size
   (b) Turtle size

7. A turtle can draw in one of seven colors. (True/False)

8. A turtle can leave an imprint of its shape on the screen by use of the _______________ method.

9. In order to create a new turtle object, the _____________ method is called.

ANSWERS: 1. False,



Self-Test Questions

1. A turtle screen is an 800-pixel wide by 600-pixel high graphics window. (True/False).

2. The three main attributes of a turtle object are _______________, _______________, and __________________.

3. A turtle can be moved using either _____________ or ______________ positioning.

4. A turtle can only draw lines when it is not hidden. (True/False)

5. A turtle shape is limited to an arrow, turtle, circle, square, triangle, or classic (default) shape. (True/False)

6. What attribute of a turtle determines the size of the lines it will draw?

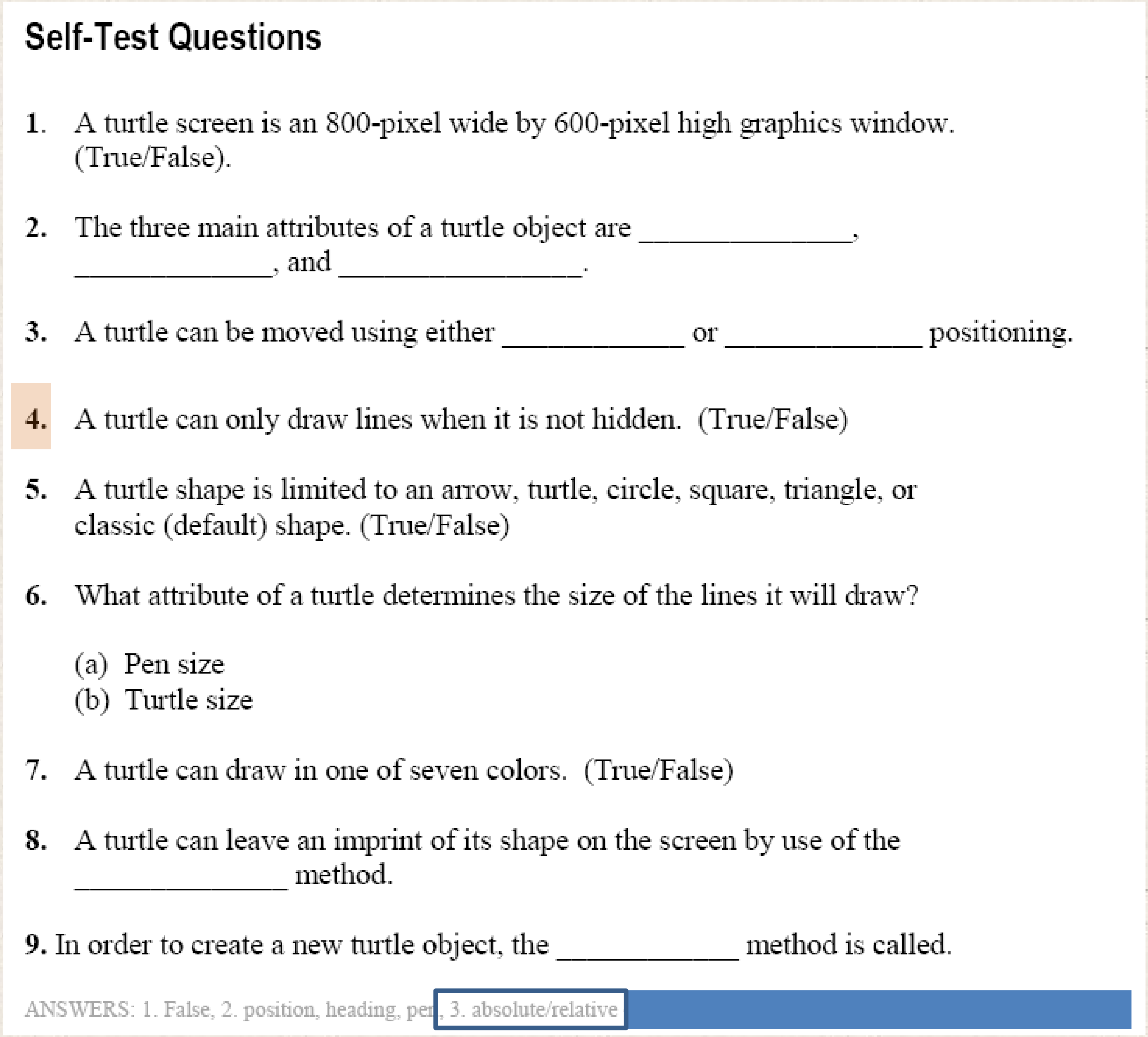
   (a) Pen size
   (b) Turtle size

7. A turtle can draw in one of seven colors. (True/False)

8. A turtle can leave an imprint of its shape on the screen by use of the ________________ method.

9. In order to create a new turtle object, the _____________ method is called.

ANSWERS: 1. False, 2. position, heading, pen, [redacted]



Self-Test Questions

1. A turtle screen is an 800-pixel wide by 600-pixel high graphics window. (True/False).

2. The three main attributes of a turtle object are ______________, ______________, and _________________.

3. A turtle can be moved using either _____________ or ______________ positioning.

4. A turtle can only draw lines when it is not hidden. (True/False)

5. A turtle shape is limited to an arrow, turtle, circle, square, triangle, or classic (default) shape. (True/False)

6. What attribute of a turtle determines the size of the lines it will draw?

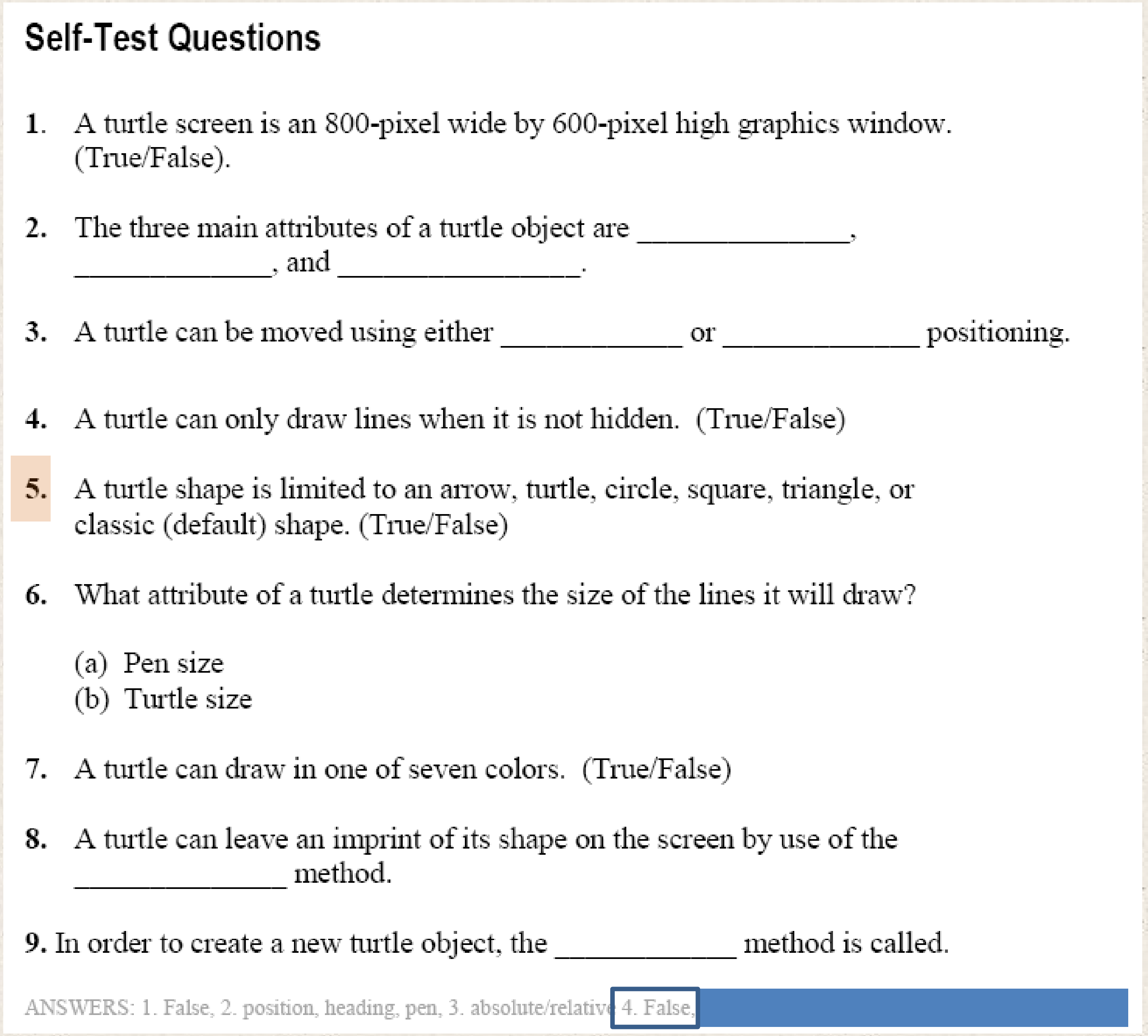
   (a) Pen size
   (b) Turtle size

7. A turtle can draw in one of seven colors. (True/False)

8. A turtle can leave an imprint of its shape on the screen by use of the _______________ method.

9. In order to create a new turtle object, the _____________ method is called.

ANSWERS: 1. False, 2. position, heading, pen, 3. absolute/relative [redacted]



Self-Test Questions

1. A turtle screen is an 800-pixel wide by 600-pixel high graphics window. (True/False).

2. The three main attributes of a turtle object are ______________, ______________, and _________________.

3. A turtle can be moved using either _____________ or ______________ positioning.

4. A turtle can only draw lines when it is not hidden. (True/False)

5. A turtle shape is limited to an arrow, turtle, circle, square, triangle, or classic (default) shape. (True/False)

6. What attribute of a turtle determines the size of the lines it will draw?

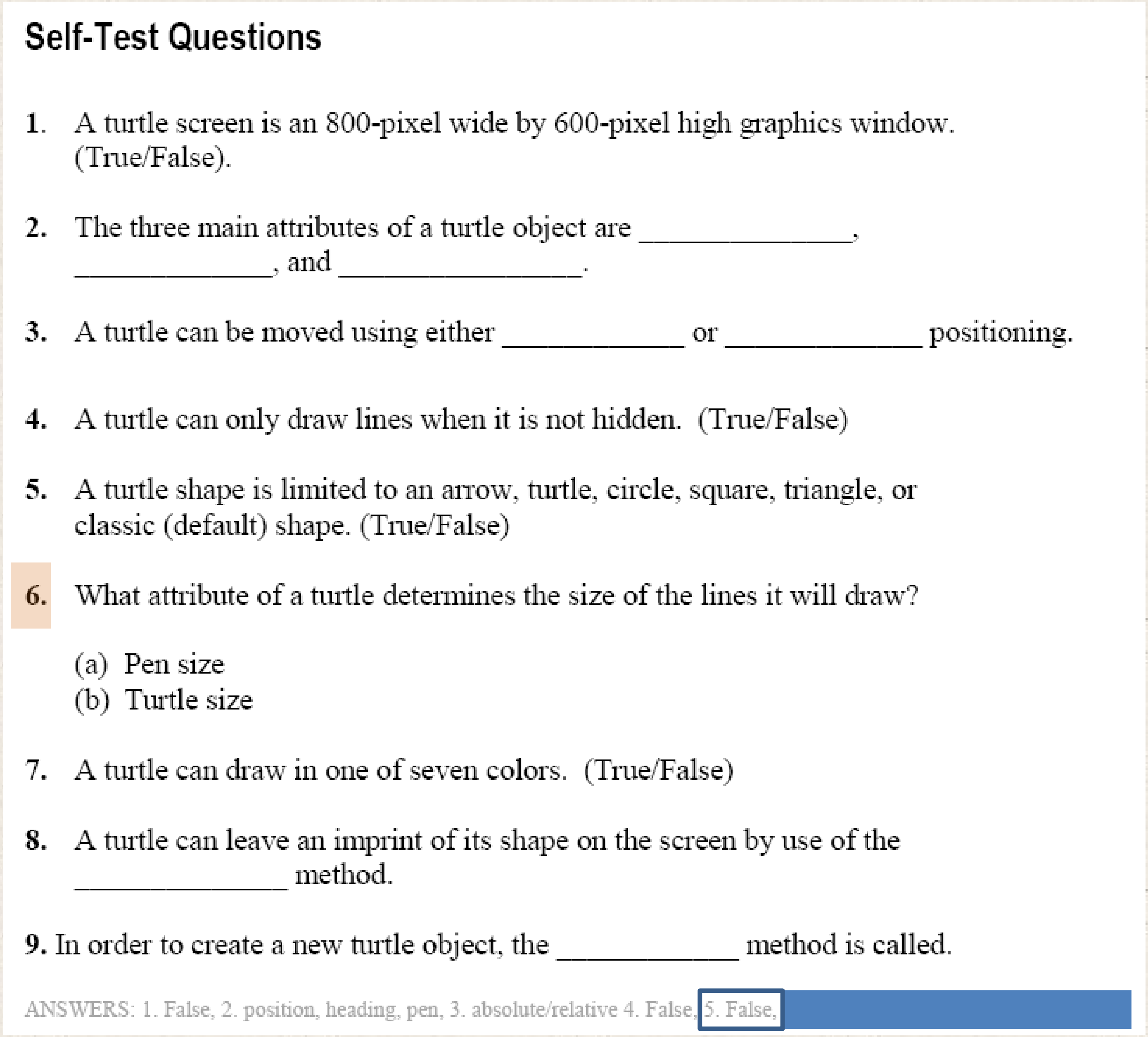
   (a) Pen size
   (b) Turtle size

7. A turtle can draw in one of seven colors. (True/False)

8. A turtle can leave an imprint of its shape on the screen by use of the _______________ method.

9. In order to create a new turtle object, the _____________ method is called.

ANSWERS: 1. False, 2. position, heading, pen, 3. absolute/relative 4. False,



Self-Test Questions

1. A turtle screen is an 800-pixel wide by 600-pixel high graphics window. (True/False).

2. The three main attributes of a turtle object are ______________, ______________, and ________________.

3. A turtle can be moved using either ____________ or _____________ positioning.

4. A turtle can only draw lines when it is not hidden. (True/False)

5. A turtle shape is limited to an arrow, turtle, circle, square, triangle, or classic (default) shape. (True/False)

6. What attribute of a turtle determines the size of the lines it will draw?

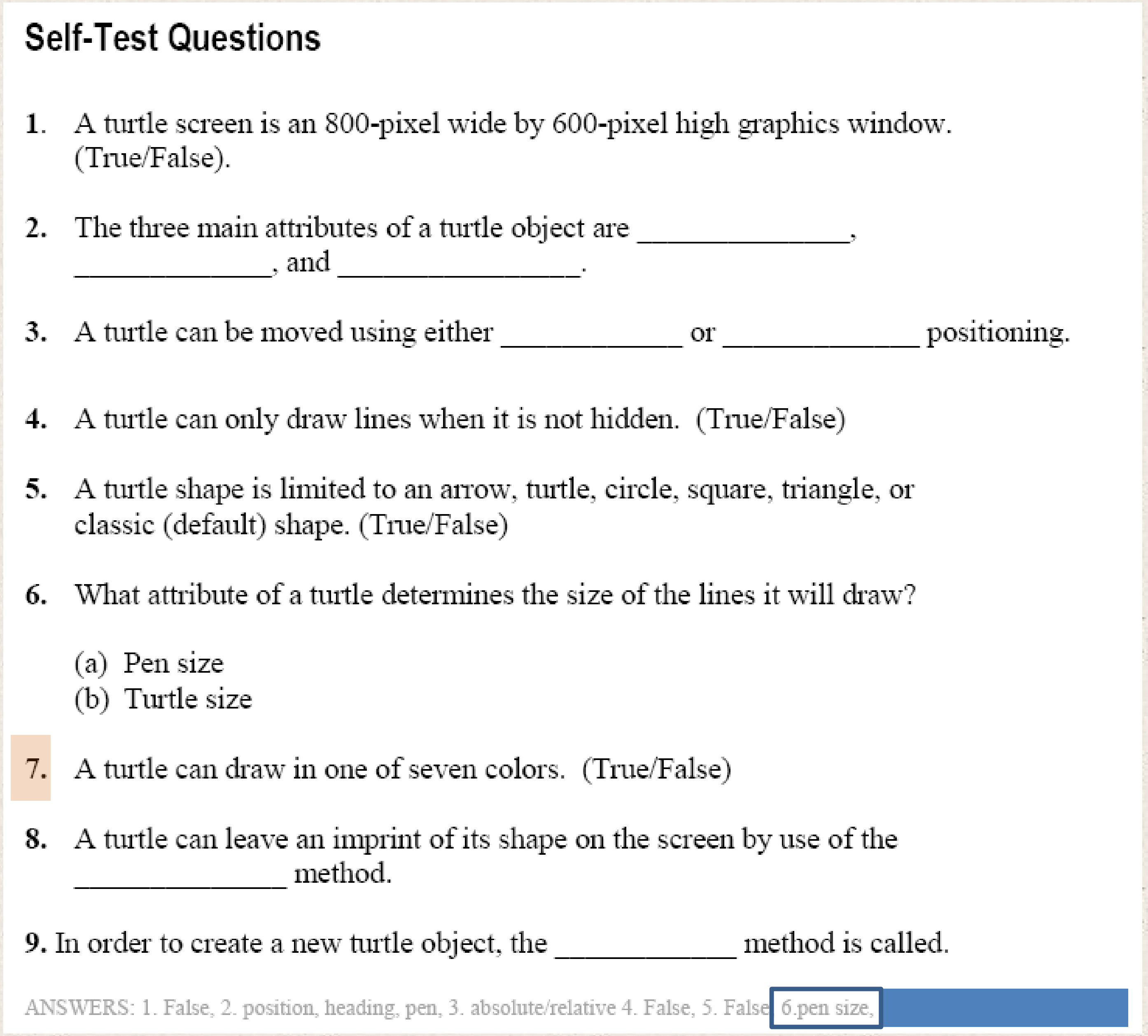
(a) Pen size
(b) Turtle size

7. A turtle can draw in one of seven colors. (True/False)

8. A turtle can leave an imprint of its shape on the screen by use of the ______________ method.

9. In order to create a new turtle object, the ____________ method is called.

ANSWERS: 1. False, 2. position, heading, pen, 3. absolute/relative 4. False, 5. False, [redacted]



Self-Test Questions

1. A turtle screen is an 800-pixel wide by 600-pixel high graphics window. (True/False).

2. The three main attributes of a turtle object are ______________, ______________, and ________________.

3. A turtle can be moved using either ____________ or _____________ positioning.

4. A turtle can only draw lines when it is not hidden. (True/False)

5. A turtle shape is limited to an arrow, turtle, circle, square, triangle, or classic (default) shape. (True/False)

6. What attribute of a turtle determines the size of the lines it will draw?

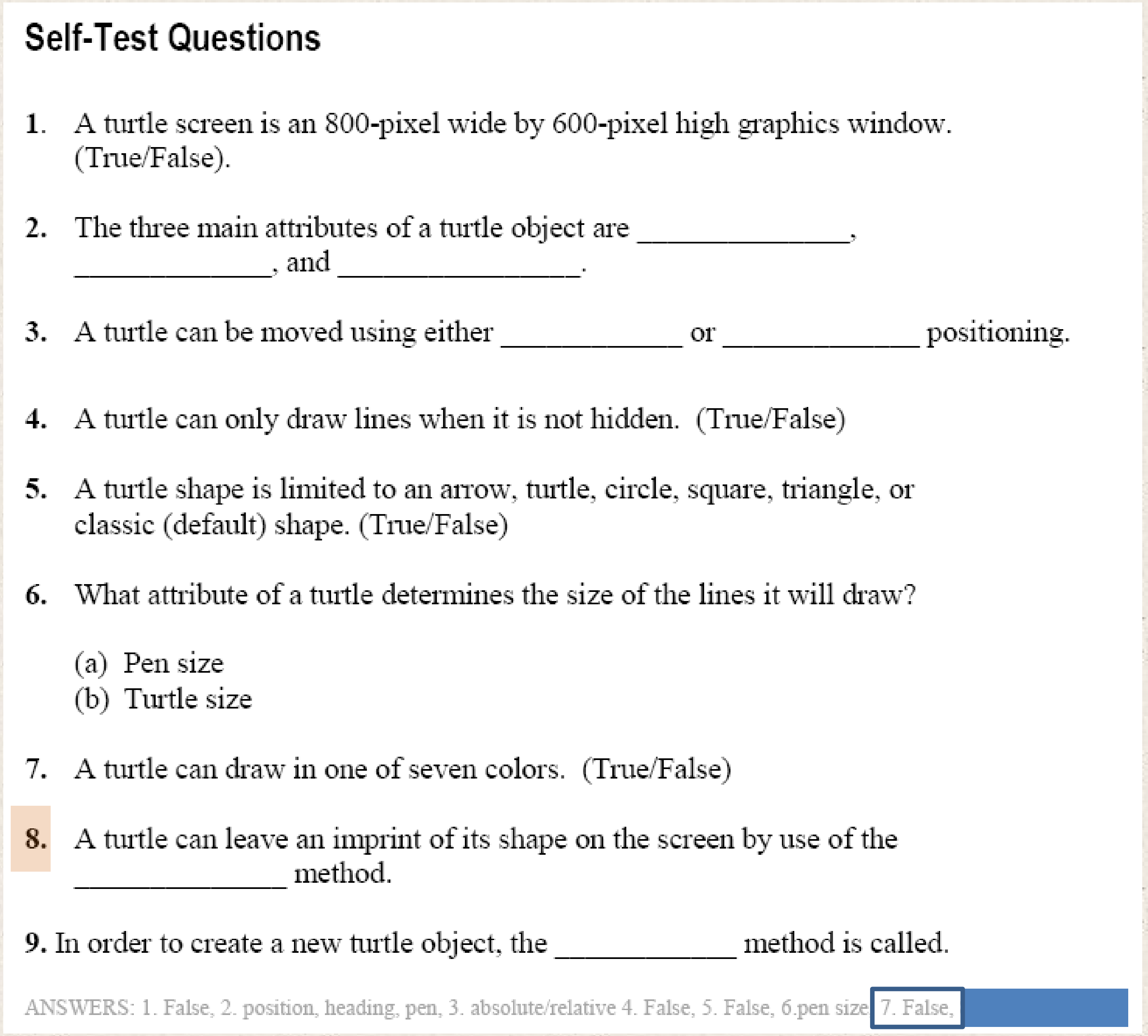
   (a) Pen size
   (b) Turtle size

7. A turtle can draw in one of seven colors. (True/False)

8. A turtle can leave an imprint of its shape on the screen by use of the ______________ method.

9. In order to create a new turtle object, the ____________ method is called.

ANSWERS: 1. False, 2. position, heading, pen, 3. absolute/relative 4. False, 5. False 6.pen size, [redacted]



Self-Test Questions

1. A turtle screen is an 800-pixel wide by 600-pixel high graphics window. (True/False).

2. The three main attributes of a turtle object are _______________, _______________, and __________________.

3. A turtle can be moved using either ______________ or _______________ positioning.

4. A turtle can only draw lines when it is not hidden. (True/False)

5. A turtle shape is limited to an arrow, turtle, circle, square, triangle, or classic (default) shape. (True/False)

6. What attribute of a turtle determines the size of the lines it will draw?

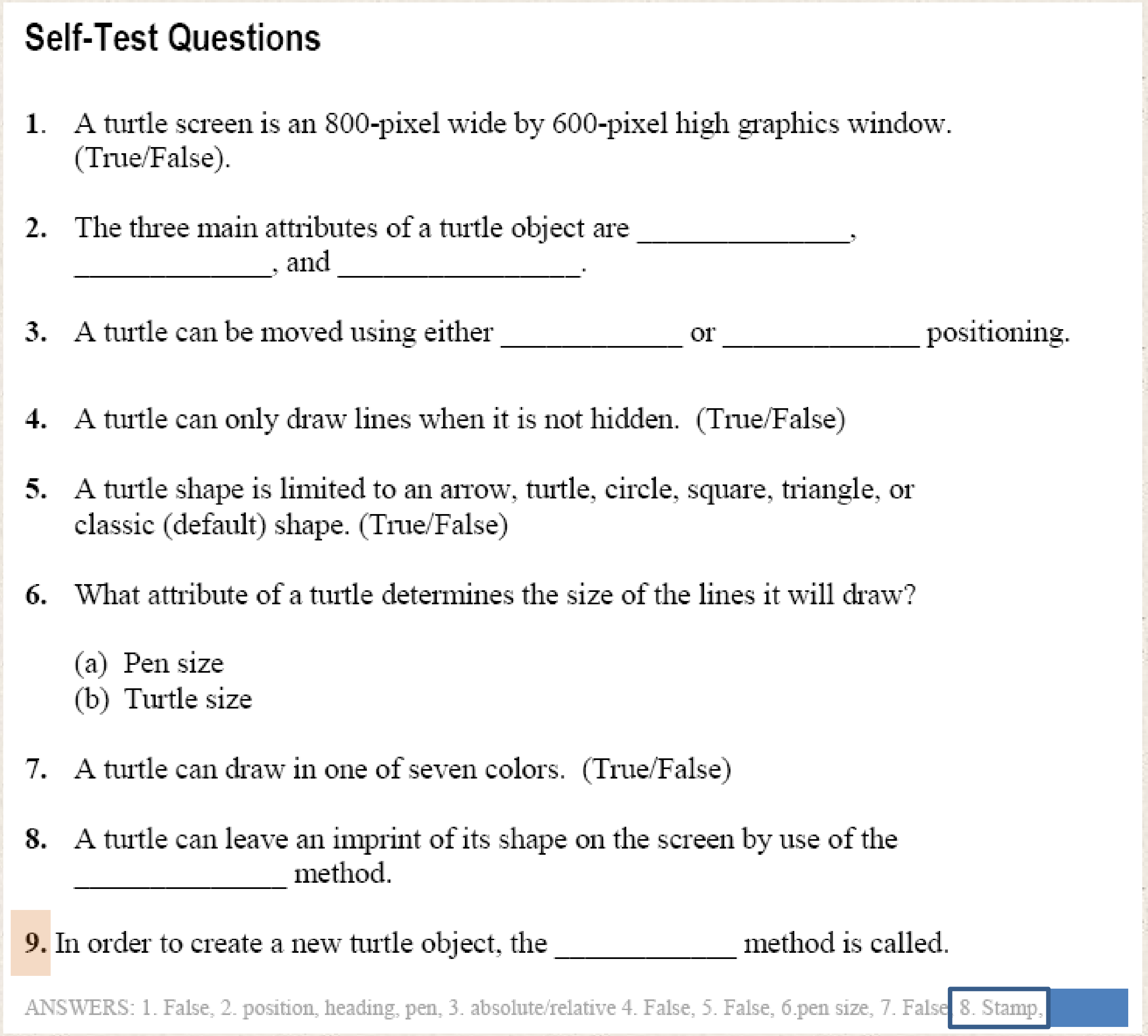
   (a) Pen size
   (b) Turtle size

7. A turtle can draw in one of seven colors. (True/False)

8. A turtle can leave an imprint of its shape on the screen by use of the ________________ method.

9. In order to create a new turtle object, the ______________ method is called.

ANSWERS: 1. False, 2. position, heading, pen, 3. absolute/relative 4. False, 5. False, 6.pen size 7. False, [redacted]



Self-Test Questions

1. A turtle screen is an 800-pixel wide by 600-pixel high graphics window. (True/False).

2. The three main attributes of a turtle object are _______________, _______________, and __________________.

3. A turtle can be moved using either _____________ or _______________ positioning.

4. A turtle can only draw lines when it is not hidden. (True/False)

5. A turtle shape is limited to an arrow, turtle, circle, square, triangle, or classic (default) shape. (True/False)

6. What attribute of a turtle determines the size of the lines it will draw?

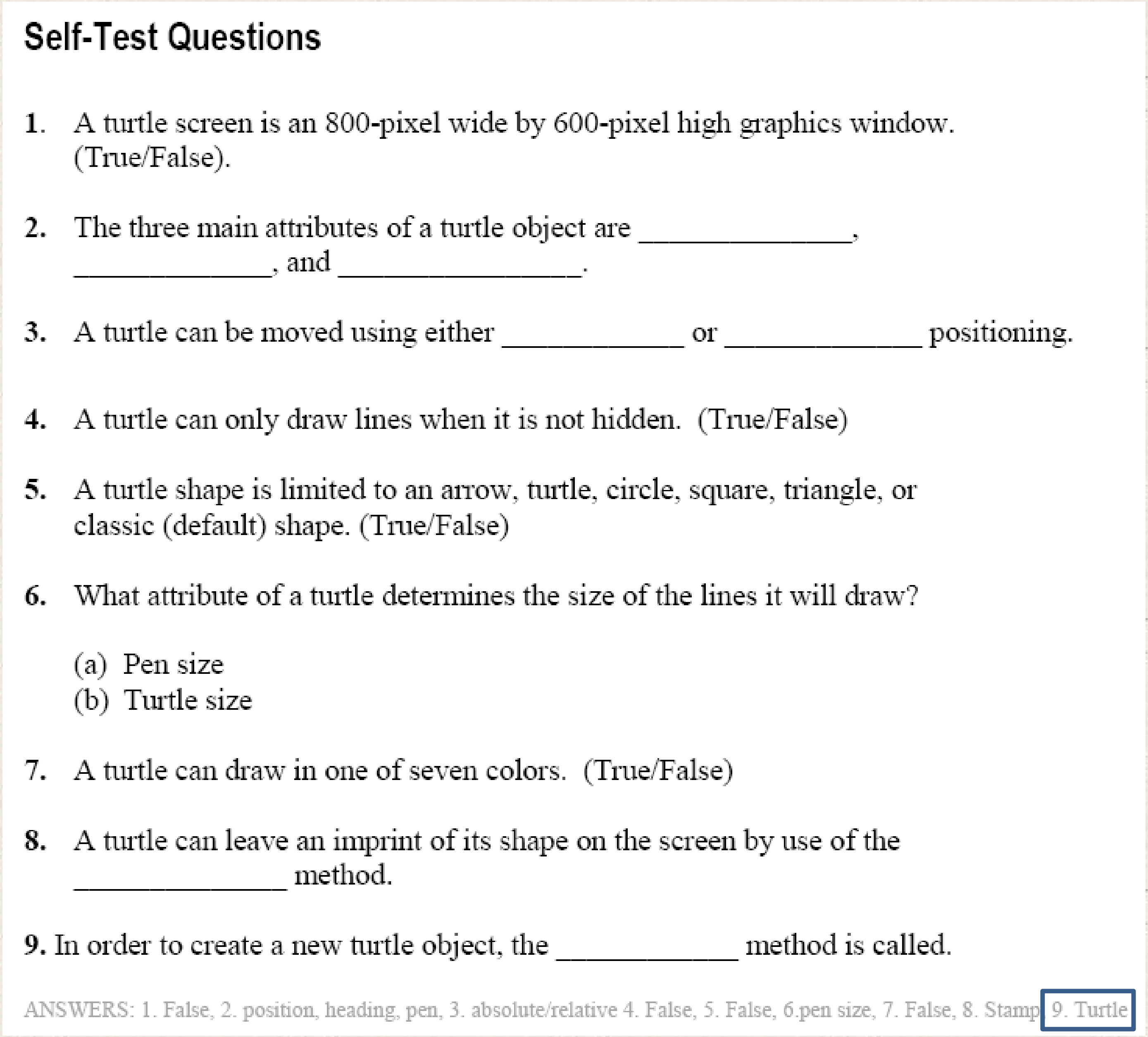
   (a) Pen size
   (b) Turtle size

7. A turtle can draw in one of seven colors. (True/False)

8. A turtle can leave an imprint of its shape on the screen by use of the ________________ method.

9. In order to create a new turtle object, the _____________ method is called.

ANSWERS: 1. False, 2. position, heading, pen, 3. absolute/relative 4. False, 5. False, 6.pen size, 7. False, 8. Stamp,



Self-Test Questions

1. A turtle screen is an 800-pixel wide by 600-pixel high graphics window. (True/False).

2. The three main attributes of a turtle object are ______________, ______________, and _________________.

3. A turtle can be moved using either _____________ or ______________ positioning.

4. A turtle can only draw lines when it is not hidden. (True/False)

5. A turtle shape is limited to an arrow, turtle, circle, square, triangle, or classic (default) shape. (True/False)

6. What attribute of a turtle determines the size of the lines it will draw?

   (a) Pen size
   (b) Turtle size

7. A turtle can draw in one of seven colors. (True/False)

8. A turtle can leave an imprint of its shape on the screen by use of the _______________ method.

9. In order to create a new turtle object, the _____________ method is called.

ANSWERS: 1. False, 2. position, heading, pen, 3. absolute/relative 4. False, 5. False, 6.pen size, 7. False, 8. Stamp $\boxed{\text{9. Turtle}}$

Horse Race Simulation Program

We design, implement and test a program that will simulate a horse race.

Horse Race Simulation

The Problem

The problem is to create a visualization of a horse race in which horses are moved ahead a random distance at fixed time intervals until there is a winner.

Horse Race Simulation
Problem Analysis

The program needs a source of random numbers for advancing the horses a random distance in the race. We can use a random number generator of the Python standard library module `random`.

There must also be a way to control the pace of the race, so that the horses don’t move across the screen too quickly. For this we can make use of Python standard library module `time`.

The remaining part of the problem is the creation of appropriate graphics for producing a visualization of a horse race. We shall make use of the turtle graphics module from the Python standard library.

Horse Race Simulation
Program Design

- Meeting the Program Requirements
- Data Desciption
- Algorithmic Approach

Meeting the Program Requirements

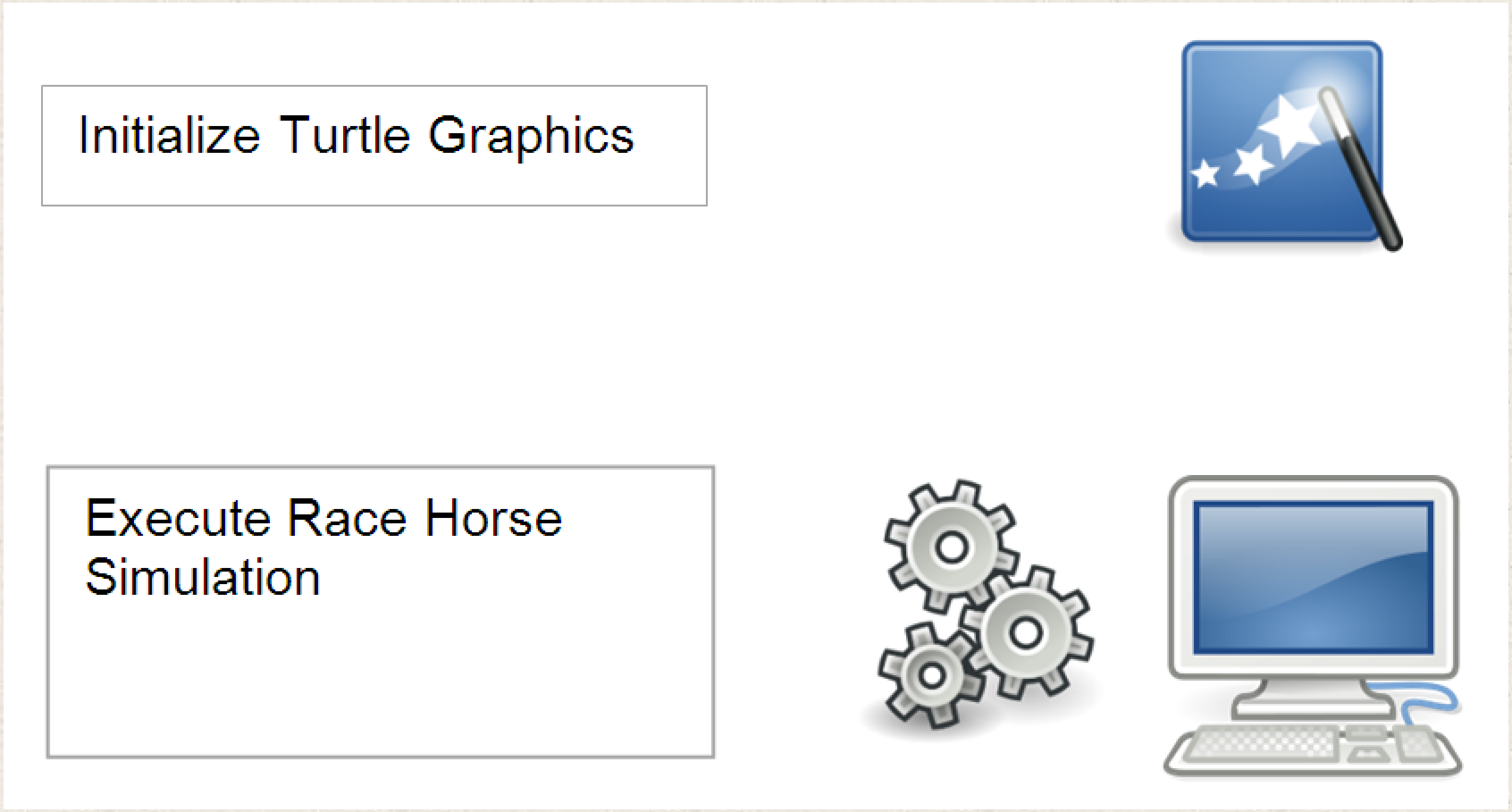
There are no specific requirements for this problem, other than to create an appropriate simulation of a horse race. Therefore, the requirement is essentially the generation of horse races in which the graphics look sufficiently compelling, and each horse has an equal chance of winning a given race. Since a specific number of horses was not specified, we will design the program for ten horses in each race.

Data Description

The essential information for this program is the current location of each of the ten horses in a given race. Each turtle is an object, whose attributes include its shape and its coordinate position on the turtle screen. Therefore, we will maintain a list of ten turtle objects with the shape attribute of a horse image for this purpose. Thus, suitable horse images must be found or created for this purpose.

Algorithmic Approach

There is no algorithm, per se, needed in this program other than to advance each horse a random distance at fixed time intervals until one of the horses reaches a certain point on the turtle screen (the “finish line”).



The Overall Steps of the Program

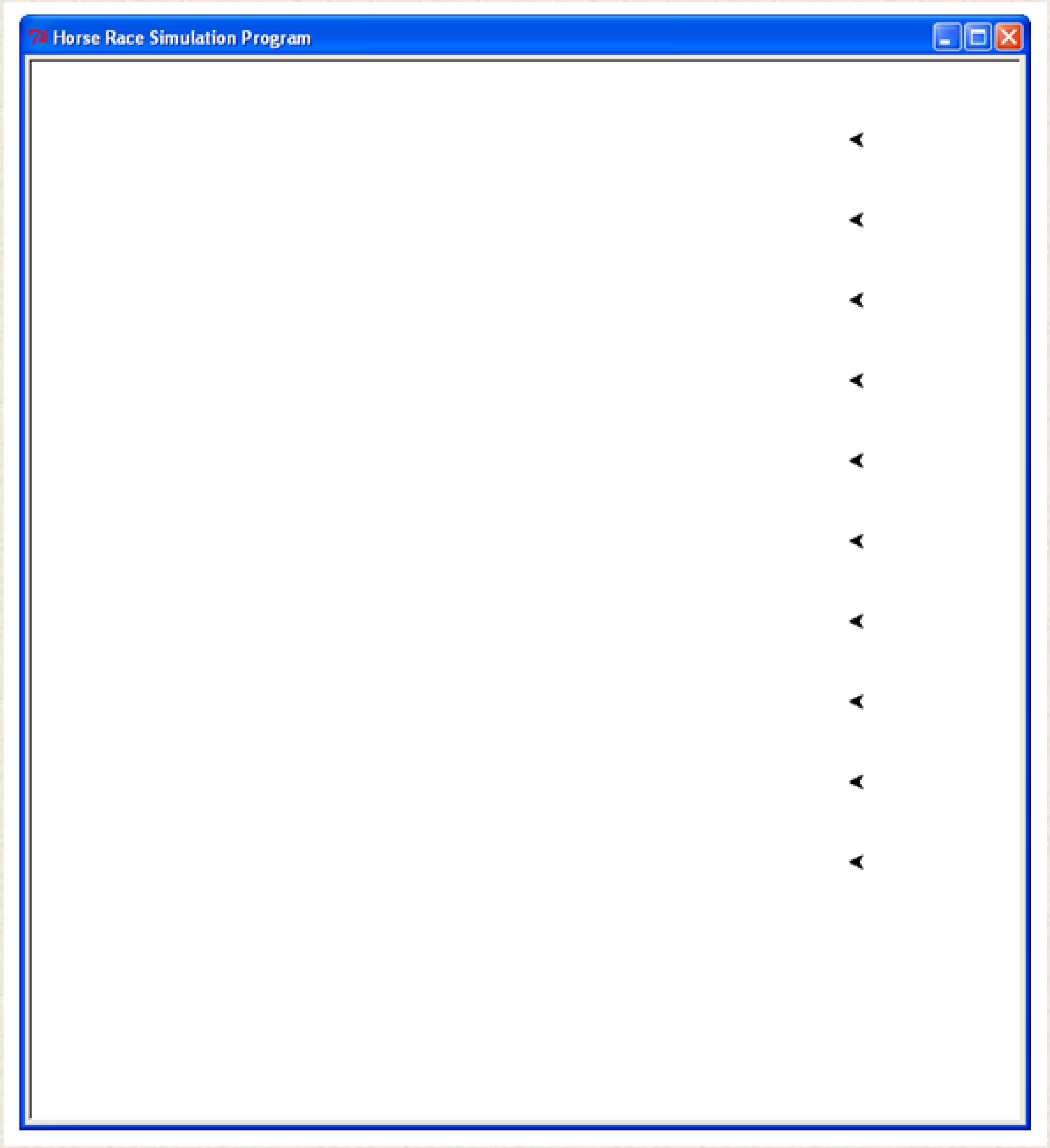
Horse Race Simulation

Program Implementation

Stage 1— Creating an Initial Turtle Screen Layout

We first develop and then test an initial program that lays out the starting positions of the horses on the turtle graphics screen.

The extent of this version of the program is to ensure that the turtle screen is appropriately sized and that the initial layout of horse locations is achieved. Therefore, this version simply uses the default turtle image. In the next version we will focus on generating a set of horse images on the turtle screen


Horse Race Simulation Program

```
 1  # Horse Racing Program (Stage 1)
 2
 3  import turtle
 4
 5  def newHorse():
 6      horse = turtle.Turtle()
 7      return horse
 8
 9  def generateHorses(num_horses):
10      horses = []
11      for k in range(0, num_horses):
12          horse = newHorse()
13          horses.append(horse)
14
15      return horses
16
17  def placeHorses(horses, loc, separation):
18      for k in range(0, len(horses)):
19          horses[k].hideturtle()
20          horses[k].penup()
21          horses[k].setposition(loc[0], loc[1] + k * separation)
22          horses[k].setheading(180)
23          horses[k].showturtle()
24
25  # ---- main
26
27  # init number of horses
28  num_horses = 10
29
30  # set window size
31  turtle.setup(750, 800)
32
33  # get turtle window
34  window = turtle.Screen()
35
36  # set window title bar
37  window.title('Horse Race Simulation Program')
38
39  # init screen layout parameters
40  start_loc = (240, -200)
41  track_separation = 60
42
43  # generate and init horses
44  horses = generateHorses(num_horses)
45
46  # place horses at starting line
47  placeHorses(horses, start_loc, track_separation)
48
49  # terminate program when close window
50  turtle.exitonclick()
```

On **line 3** the turtle module is imported. Since the `import module_name` form of import is used, each call to a method of this module must be prefixed with the module name.

On **line 40**, the coordinates of the first (lowest) horse displayed are set. The vertical separation of horses is assigned to `track_separation`.

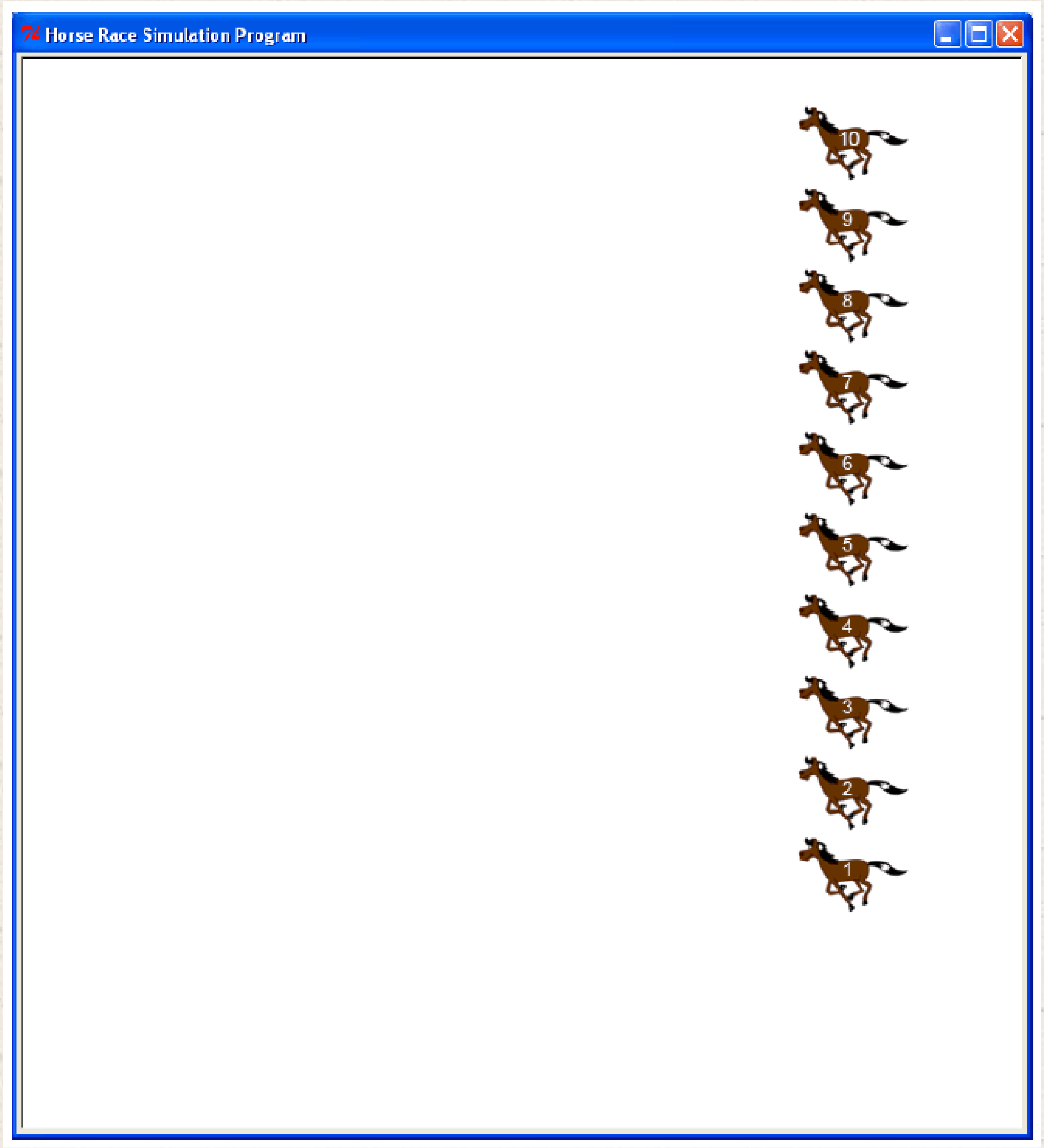
Function `generateHorses`, called on **line 44**, returns a list of ten new turtle objects, assigned to variable `horses`. Function `newHorse` (**lines 5–7**) is called to create each new turtle object (at this stage returning a regular turtle shape).

Function `placeHorses` (**lines 17–23**) is passed the list of turtle objects, the location of the first turtle, and the amount of separation between each and determines the position of each (established as 60 pixels on **line 41**). Each horse is initially hidden with pen up (**lines 19–20**), placed at its starting position (**line 21**), heading left (**line 22**), and made visible (**line 23**). Finally, method `exitonclick()` (**line 50**) is called so that the program will terminate when the user clicks on the program window's close box.

Program Implementation

Stage 2 – Adding the Appropriate Shapes and Images

We next develop and test the program with additional code that adds the horse shapes (images) needed.


Horse Race Simulation Program
10
9
8
7
6
5
4
3
2
1

```
 1 # Horse Racing Program (Stage 2)
 2
 3  import turtle
 4
 5 def getHorseImages(num_horses):
 6     # init empty list
 7     images = []
 8
 9     # get all horse images
10     for k in range(0, num_horses):
11         images = images + ['horse_' + str(k + 1) + '_image.gif']
12
13     return images
14
15 def registerHorseImages(images):
16     for k in range(0, len(images)):
17         turtle.register_shape(images[k])
18
19 def newHorse(image_file):
20     horse = turtle.Turtle()
21     horse.hideturtle()
22     horse.shape(image_file)
23
24     return horse
25
26 def generateHorses(images, num_horses):
27     horses = []
28     for k in range(0, num_horses):
29         horse = newHorse(images[k])
30         horses.append(horse)
31
32     return horses
33
34 def placeHorses(horses, loc, separation):
35     for k in range(0, len(horses)):
36         horses[k].hideturtle()
37         horses[k].penup()
38         horses[k].setposition(loc[0], loc[1] + k * separation)
39         horses[k].setheading(180)
40         horses[k].showturtle()
41
```

On **line 3** the turtle module is imported. Since the `import` *`module_name`* form of import is used, each call to a method of this module must be prefixed with the module name. (The use of Python modules is covered in Chapter 7).

We add in this stage of the program functions `getHorseImages` (**lines 5-15**) and `registerHorseImages` (**lines 15-17**) (called from **lines 61- 62** of main). Function `getHorseImages` returns a list of GIF image files, each image the same horse image, with a unique number from 1 to 10 added. Function `registerHorseImages` does the required registering of images.

Function `generateHorses` (**lines 26–32**) is the same as in stage 1, except that it is passed a list of horse images rather than the number of horses to generate..

Function `newHorse` (**lines 19–24**) is altered as well to be passed a particular horse image to set the shape of the turtle object that this horse created, `horse.shape(image_file)`.

```
42 # ---- main
43
44 # init number of horses
45 num_horses = 10
46
47 # set window size
48 turtle.setup(750, 800)
49
50 # get turtle window
51 window = turtle.Screen()
52
53 # set window title bar
54 window.title('Horse Race Simulation Program')
55
56 # init screen layout parameters
57 start_loc = (240, -200)
58 track_separation = 60
59
60 # register images
61 horse_images = getHorseImages()
62 registerHorseImages(horse_images)
63
64 # generate and init horses
65 horses = generateHorses(horse_images)
66
67 # place horses at starting line
68 placeHorses(horses, start_loc, track_separation)
69
70 # terminate program when close window
71 turtle.exitonclick()
```

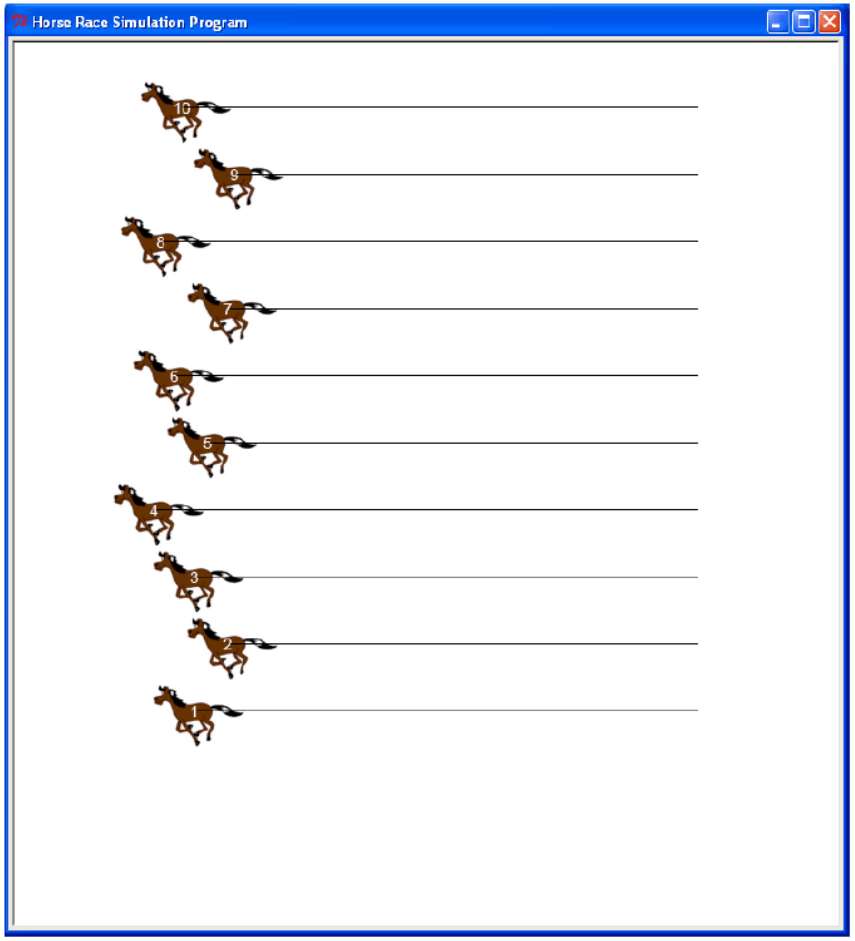
In this stage of the program we add functions `getHorseImages` and `registerHorseImages`, called from **lines 61** and **62**. Function `getHorseImages` returns a list of GIF image files. Each image contains the same horse image, each with a unique number from 1 to 10. Function `registerHorseImages` does the required registering of images in by calling `turtle.register_shape` on each.

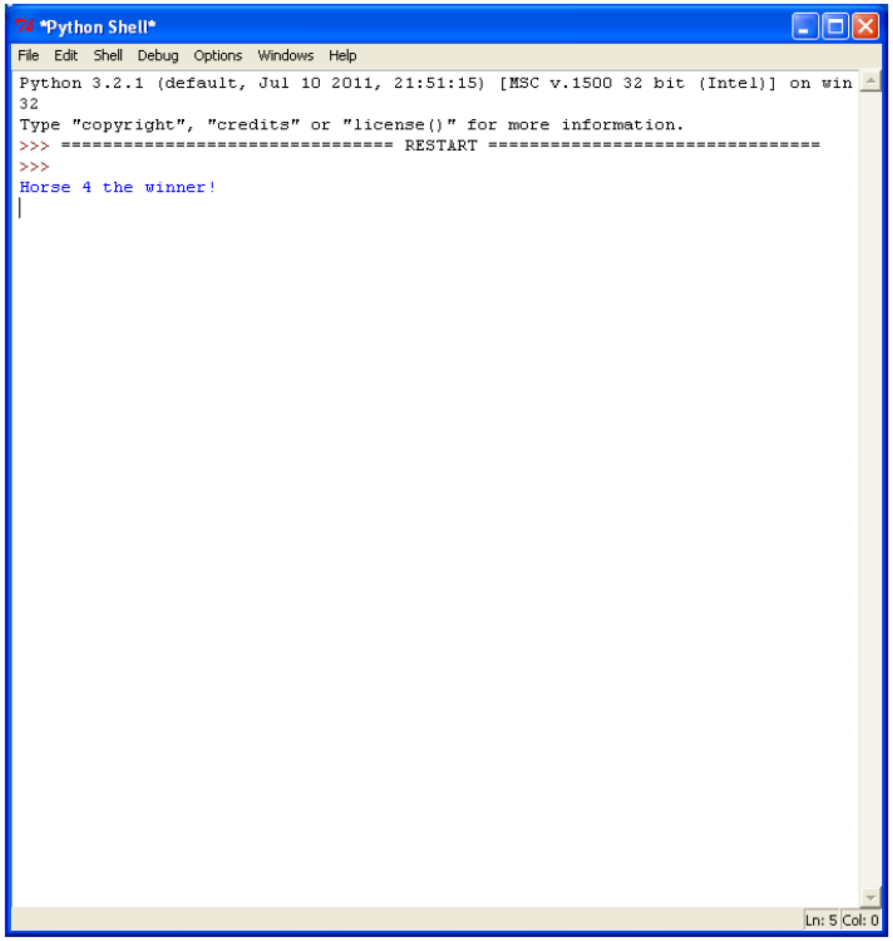
Function `generateHorses` (**lines 26–32**) is implemented the same as in stage 1 (to return a list of turtle objects), except that it is passed a list of horse images, rather than the number of horses to generate. Thus, `generateHorses` in **line 65** is passed the list of images in variable `horse_images`.

Program Implementation

Stage 3 – Animating the Horses

Next we develop and test the program with additional code that animates the horses so that they are randomly advanced until the first horse crosses the finish line. The number of the winning horse is displayed in the Python shell.





```
Python 3.2.1 (default, Jul 10 2011, 21:51:15) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
Horse 4 the winner!
```

```
1  # Horse Racing Program (Stage 3)
2
3  import turtle
4  import random
5
6  def getHorseImages(num_horses):
7      # init empty list
8      images = []
9
10     # get all horse images
11     for k in range(0, num_horses):
12         images = images + ['horse_' + str(k+1) + '_image.gif']
13
14     return images
15
16 def registerHorseImages(images):
17     for k in range(0, len(images)):
18         turtle.register_shape(images[k])
19
20 def newHorse(image_file):
21     horse = turtle.Turtle()
22     horse.hideturtle()
23     horse.shape(image_file)
24
25     return horse
26
27 def generateHorses(images, num_horses):
28     horses = []
29     for k in range(0, num_horses):
30         horse = newHorse(images[k])
31         horses.append(horse)
32
33     return horses
34
35 def placeHorses(horses, loc, separation):
36     for k in range(0, len(horses)):
37         horses[k].hideturtle()
38         horses[k].penup()
39         horses[k].setposition(loc[0], loc[1] + k * separation)
40         horses[k].setheading(180)
41         horses[k].showturtle()
42         horses[h].pendown()
43
44 def startHorses(horses, finish_line, forward_incr):
45     # init
46     have_winner = False
47
48     k = 0
49     while not have_winner:
50         horse = horses[k]
51         horse.forward(random.randint(1, 3) * forward_incr)
52
53         # check for horse over finish line
54         if horse.position()[0] < finish_line:
55             have_winner = True
56         else:
57             k = (k + 1) % len(horses)
58     return k
59
```

Two new functions are added in this version of the program, `startHorses` and `displayWinner`.

Function `startHorses` (**lines 44–58**) is passed the list of horse turtle objects, the location of the finish line, and the fundamental increment amount. Each horse is randomly advanced by one to three times this amount. The while loop for incrementally moving the horses is on **line 48**. The loop iterates until a winner is found (until the `have_winner` is `True`).

Since each horse in turn is advanced some amount during the race, variable `k` is incremented by one, modulo the number of horses. When `k` becomes equal to `num_horses – 1` (9), it is reset to 0 (for horse 1).

```
 60 def displayWinner(winning_horse):
 61     print('Horse', winning_horse, 'the winner!')
 62
 63 # ---- main
 64
 65 # init number of horses
 66 num_horses = 10
 67
 68 # set window size
 69 turtle.setup(750, 800)
 70
 71 # get turtle window
 72 window = turtle.Screen()
 73
 74 # set window title bar
 75 window.title('Horse Race Simulation Program')
 76
 77 # init screen layout parameters
 78 start_loc = (240, -200)
 79 finish_line = -240
 80 track_separation = 60
 81 forward_incr = 6
 82
 83 # register images
 84 horse_images = getHorseImages(num_horses)
 85 registerHorseImages(horse_images)
 86
 87 # generate and init horses
 88 horses = generateHorses(horse_images, num_horses)
 89
 90 # place horses at starting line
 91 placeHorses(horses, start_loc, track_separation)
 92
 93 # start horses
 94 winner = startHorses(horses, finish_line, forward_incr)
 95
 96 # display winning horse
 97 displayWinner(winner + 1)
 98
 99 # terminate program when close window
100 turtle.exitonclick()
```

The amount that each horse is advanced is a factor of one to three, randomly determined by call to method `randint(1, 3)` of the Python standard library module `random` on **line 51**. Variable `forward_incr` is multiplied by this factor to move the horses forward an appropriate amount. The value of `forward_incr` is initialized in the main program section. This value can be adjusted to speed up or slow down the overall speed of the horses.

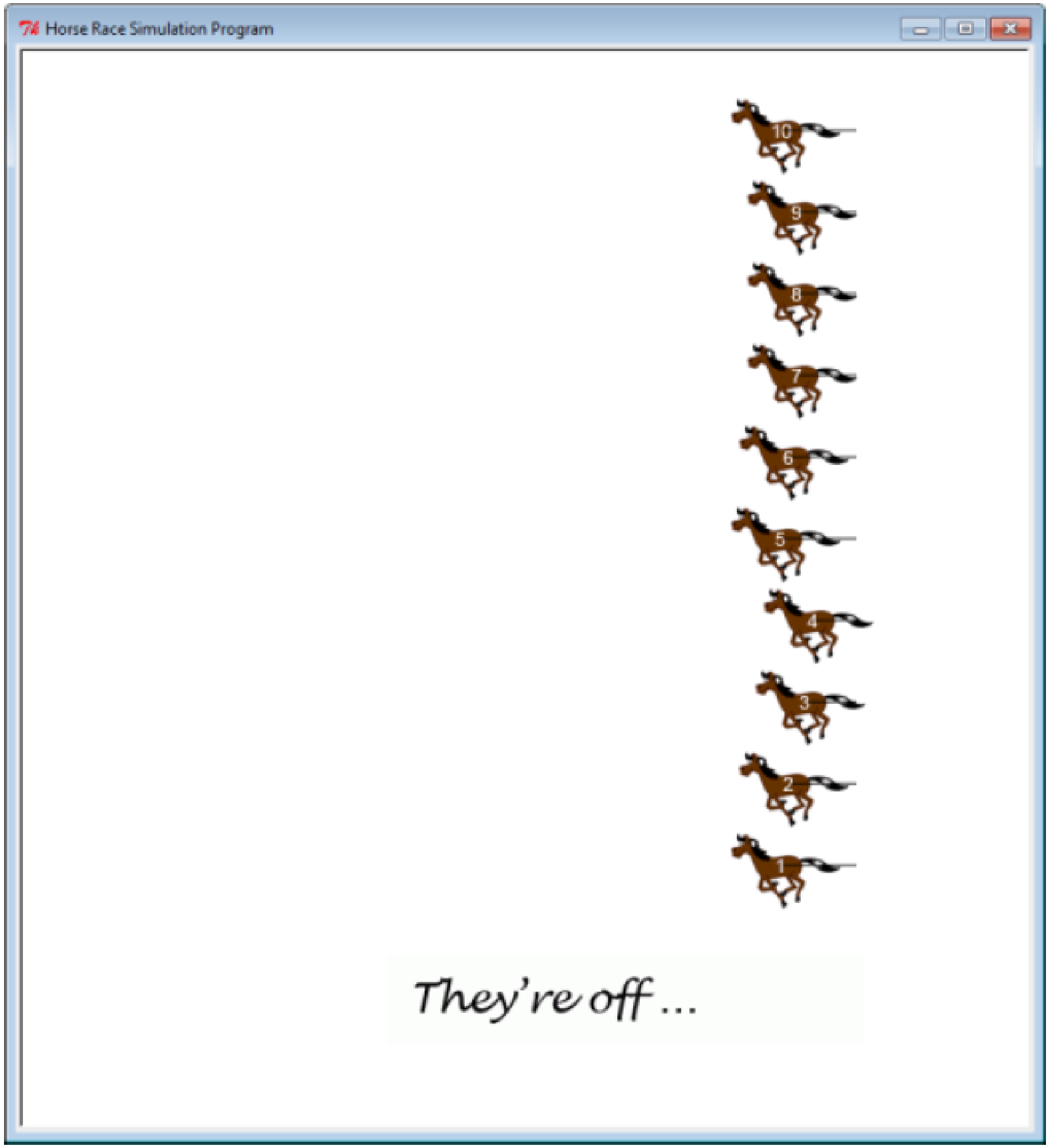
Function `displayWinner` displays the winning horse number in the Python shell (**lines 60–61**). This function will be rewritten in the next stage of program development to display a "winner" banner image in the turtle screen. Thus, this implementation of the function is for testing purposes only.

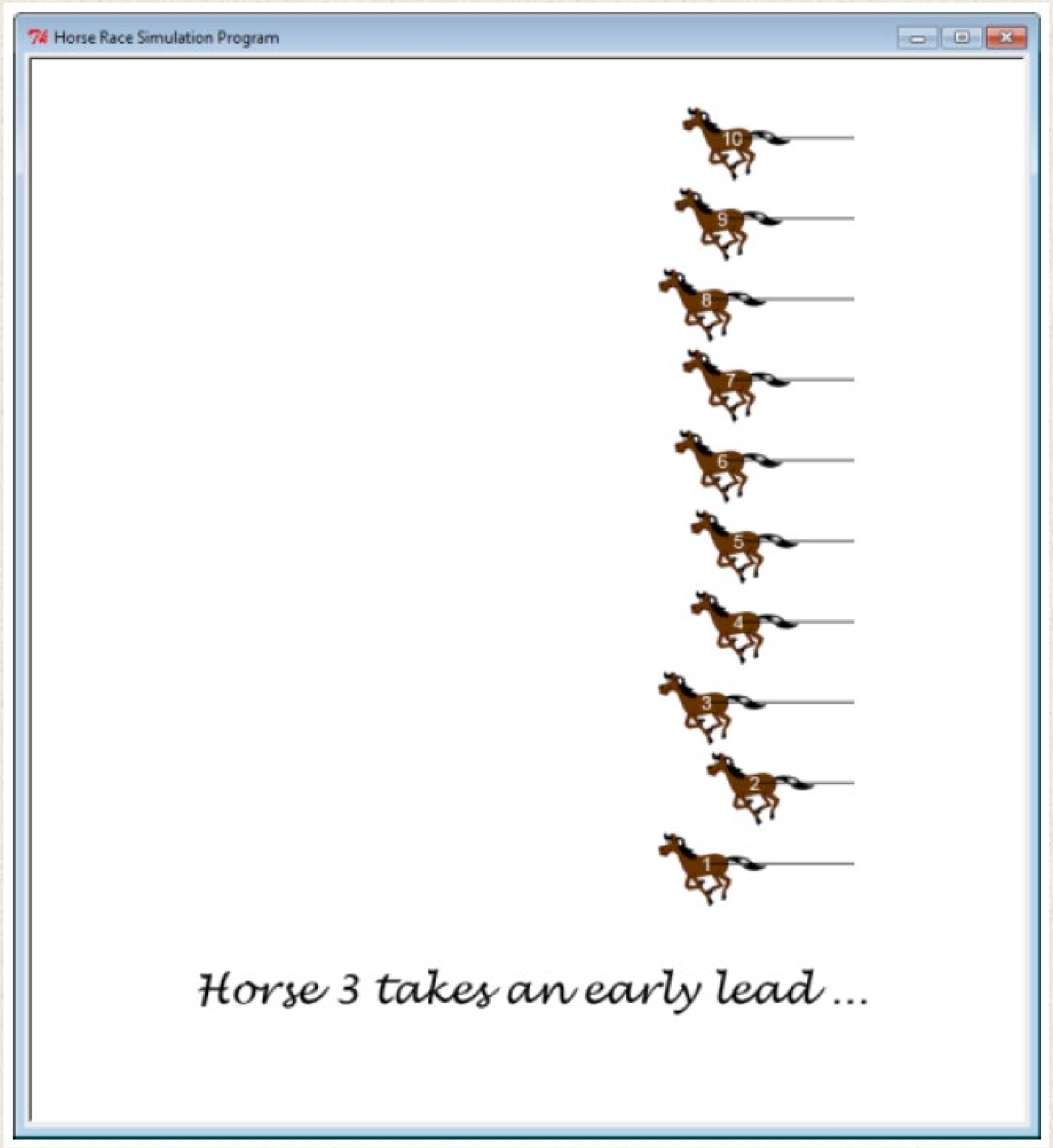
The main program section (**lines 63–100**) is the same as in the previous stage of program development, except for the inclusion of the calls to functions `startHorses` and `displayWinner` on **lines 94** and **97**.

Program Implementation

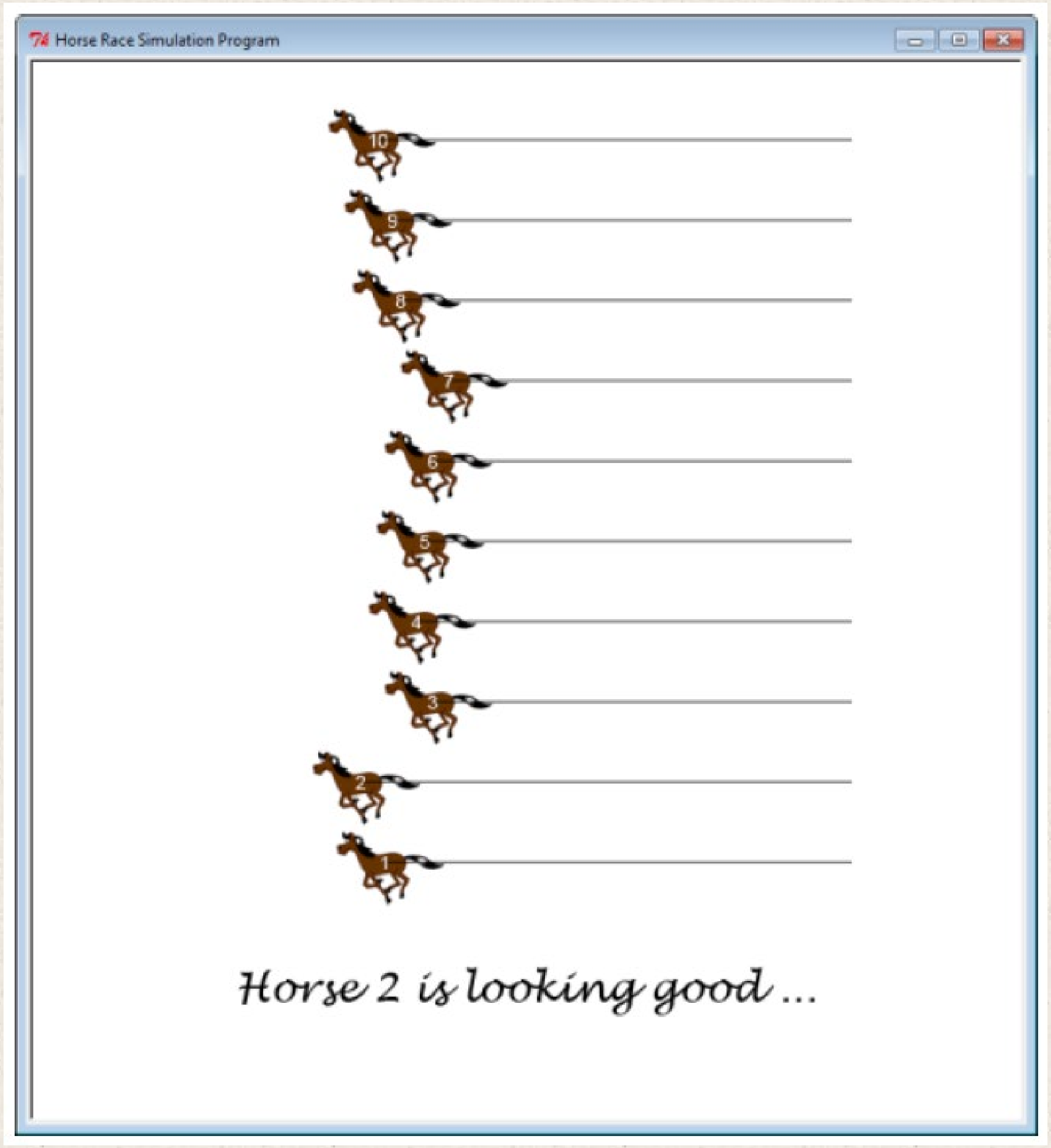
Final Stage – Adding Race Banners

Finally, **we add the code for** displaying banners at various points in the race. In addition, the winning horse is made to blink.

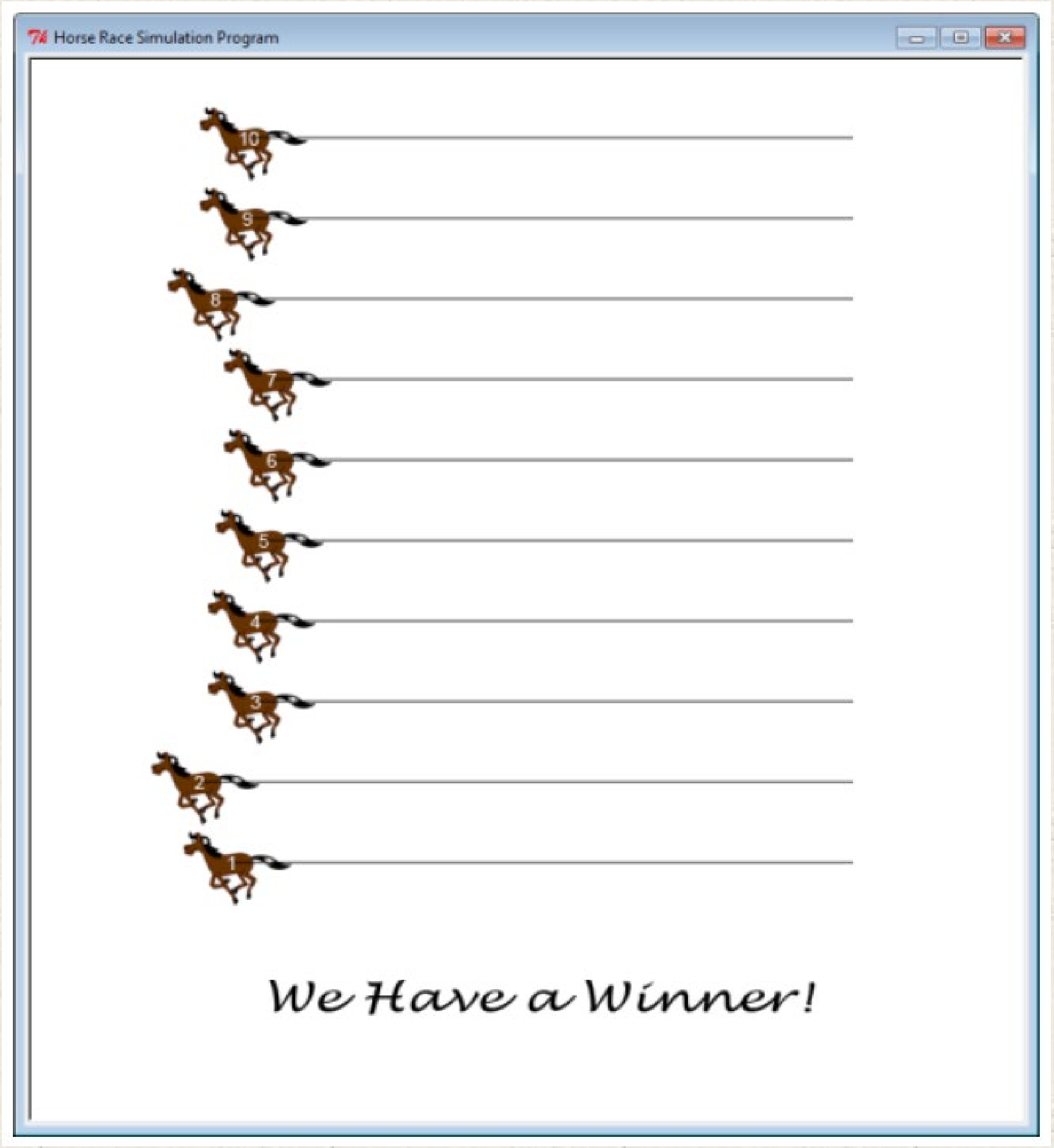

Horse Race Simulation Program
10
9
8
7
6
5
4
3
2
1
They're off …



Horse 3 takes an early lead ...



Horse 2 is looking good ...


Horse Race Simulation Program
10
9
8
7
6
5
4
3
2
1
We Have a Winner!

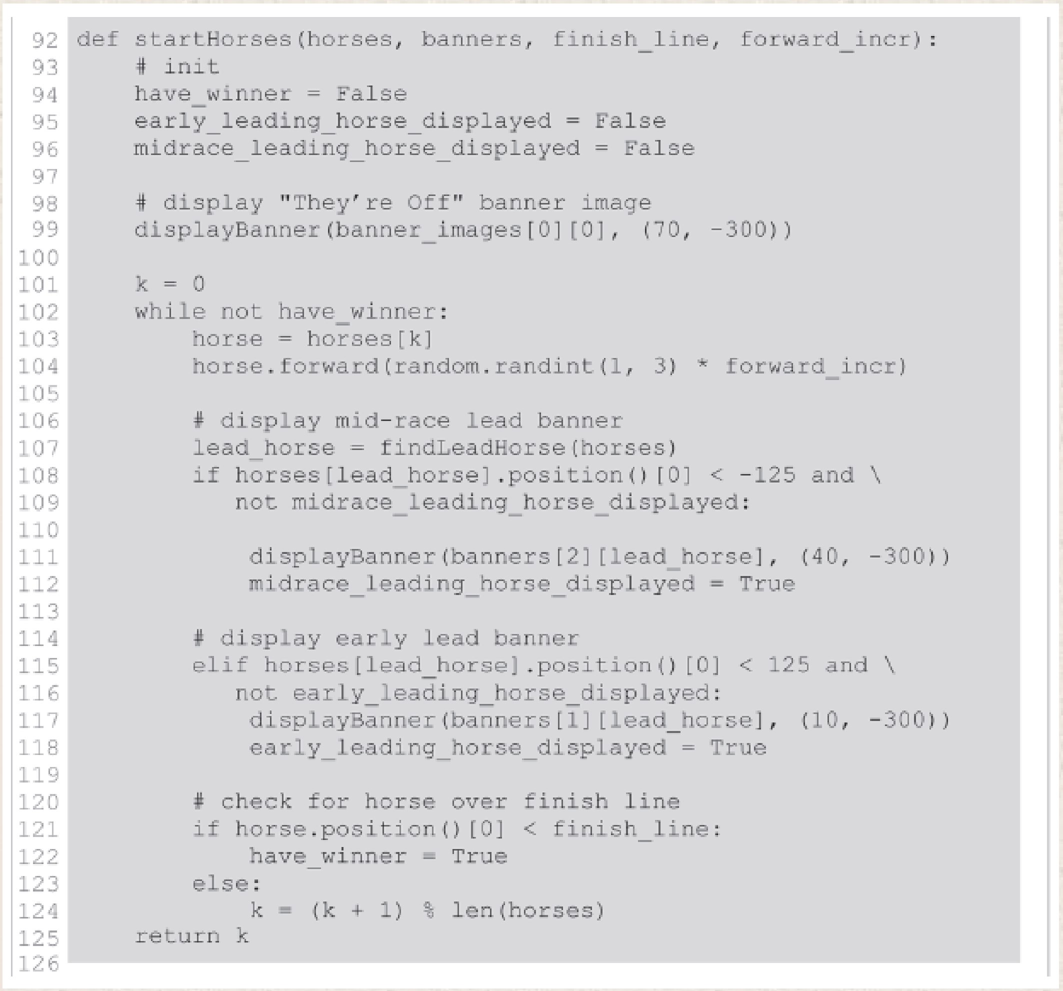
```
 1  # Horse Racing Program (Final Stage)
 2
 3  import turtle
 4  import random
 5  import time
 6
 7  def getHorseImages(num_horses):
 8      # init empty list
 9      images = []
10
11      # get all horse images
12      for k in range(0, num_horses):
13          images = images + ['horse_' + str(k + 1) + '_image.gif']
14
15      return images
16
17  def getBannerImages(num_horses):
18      # init empty list
19      all_images = []
20
21      # get "They're Off" banner image
22      images = ['theyre_off_banner.gif']
23      all_images.append(images)
24
25      # get early lead banner images
26      images = []
27      for k in range(0, num_horses):
28          images = images + ['lead_at_start_' + str(k + 1) + '.gif']
29      all_images.append(images)
30
31      # get mid-way lead banner images
32      images = []
33      for k in range(0, num_horses):
34          images = images + ['looking_good_' + str(k + 1) + '.gif']
35      all_images.append(images)
36
37      # get "We Have a Winner" banner image
38      images = ['winner_banner.gif']
39      all_images.append(images)
40
41      return all_images
42
43  def registerHorseImages(images):
44      for k in range(0, len(images)):
45          turtle.register_shape(images[k])
46
```

The final version of the program imports one additional module, Python Standard Library module `time` (**line 5**). The program uses method `sleep` from the `time` module to control the blinking rate of the image of the winning horse

Function `getBannerImages` (**line 17**), along with functions `registerBannerImages` (**line 47**), and `displayBanner` (**line 84**) incorporate the banner images into the program the same way that the horse images were incorporated in the previous program version.

```
47 def registerBannerImages(images):
48     for k in range(0, len(images)):
49         for j in range(0, len(images[k])):
50             turtle.register_shape(images[k][j])
51
52 def newHorse(image_file):
53     horse = turtle.Turtle()
54     horse.hideturtle()
55     horse.shape(image_file)
56
57     return horse
58
59 def generateHorses(images, num_horses):
60     horses = []
61     for k in range(0, num_horses):
62         horse = newHorse(images[k])
63         horses.append(horse)
64
65     return horses
66
67 def placeHorses(horses, loc, separation):
68     for k in range(0, len(horses)):
69         horses[k].hideturtle()
70         horses[k].penup()
71         horses[k].setposition(loc[0], loc[1] + k * separation)
72         horses[k].setheading(180)
73         horses[k].showturtle()
74         horses[k].pendown()
75
76 def findLeadHorse(horses):
77     # init
78     lead_horse = 0
79
80     for k in range(1, len(horses)):
81         if horses[k].position()[0] <  \
82            horses[lead_horse].position()[0]:
83             lead_horse = k
84     return lead_horse
85
86 def displayBanner(banner, position):
87     the_turtle = turtle.getturtle()
88     the_turtle.setposition(position[0], position[1])
89     the_turtle.shape(banner)
90     the_turtle.stamp()
91
```

Added functions `getBannerImages`, `registerBannerImages` (**lines 47–50**), and `displayBanner` (**lines 86–90**) incorporate the banner images into the program the same way that the horse images were incorporated in the previous program version. When the banners appear during a race appear is based on the location of the currently leading horse.



```
 92 def startHorses(horses, banners, finish_line, forward_incr):
 93     # init
 94     have_winner = False
 95     early_leading_horse_displayed = False
 96     midrace_leading_horse_displayed = False
 97 
 98     # display "They're Off" banner image
 99     displayBanner(banner_images[0][0], (70, -300))
100 
101     k = 0
102     while not have_winner:
103         horse = horses[k]
104         horse.forward(random.randint(1, 3) * forward_incr)
105 
106         # display mid-race lead banner
107         lead_horse = findLeadHorse(horses)
108         if horses[lead_horse].position()[0] < -125 and \
109            not midrace_leading_horse_displayed:
110 
111             displayBanner(banners[2][lead_horse], (40, -300))
112             midrace_leading_horse_displayed = True
113 
114         # display early lead banner
115         elif horses[lead_horse].position()[0] < 125 and \
116            not early_leading_horse_displayed:
117             displayBanner(banners[1][lead_horse], (10, -300))
118             early_leading_horse_displayed = True
119 
120         # check for horse over finish line
121         if horse.position()[0] < finish_line:
122             have_winner = True
123         else:
124             k = (k + 1) % len(horses)
125     return k
126 
```

Function startHorses was modified to take another parameter, banners, containing the list of registered banners displayed during the race, passed to it from the main program section.

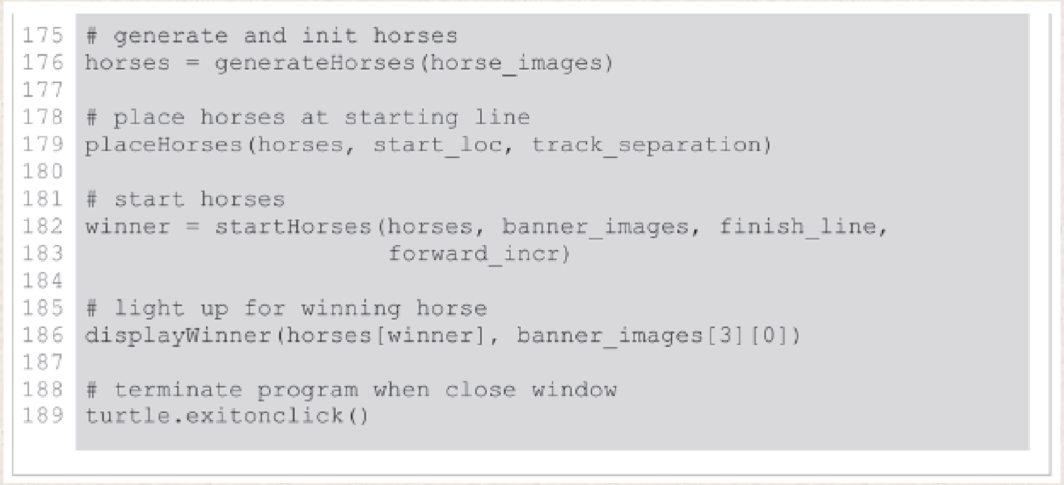
While the race progresses within the while loop at **line 102** , checks for the location of the lead horse are made in two places—before and after the halfway mark of the race (on **line 108).** If the x coordinate location of the lead horse is less then 125, the “early lead banner” is displayed on **line 117** by a call to function displayBanner.

```
127 def displayWinner(winning_horse, winner_banner):
128     # display "We Have a Winner" banner
129     displayBanner(winner_banner, (20, -300))
130 
131     # blink winning horse
132     show = False
133     blink_counter = 5
134     while blink_counter != 0:
135         if show:
136             winning_horse.showturtle()
137             show = False
138             blink_counter = blink_counter - 1
139         else:
140             winning_horse.hideturtle()
141             show = True
142 
143         time.sleep(.4)
144 
145 # ---- main
146 
147 # init number of horses
148 num_horses = 10
149 
150 # set window size
151 turtle.setup(750, 800)
152 
153 # get turtle window
154 window = turtle.Screen()
155 
156 # set window title
157 window.title('Horse Race Simulation Program')
158 
159 # hide default turtle and keep from drawing
160 the_turtle.hideturtle()
161 the_turtle.penup()
162 
163 # init screen layout parameters
164 start_loc = (240, -200)
165 finish_line = -240
166 track_separation = 60
167 forward_incr = 6
168 
169 # register images
170 horse_images = getHorseImages()
171 banner_images = getBannerImages()
172 registerHorseImages(horse_images)
173 registerBannerImages(banner_images)
174 
```

This version of `displayWinner` (**line 127**) replaces the previous version that simply displayed the winning horse number in the shell window. A "count-down" variable, `blink_counter`, is set to 5 on **line 133**, decrementing it to zero in the while loop, causing the winning horse to blink five times.

Boolean variable `show`, initialized to False on **line 132** , is used to alternately show and hide the turtle. The `sleep` method, called on **line 143**, causes the program to suspend execution for four-tenths of a second so that the showing/hiding of the winning horse image switch slowly enough to cause a blinking effect.

The default turtle is utilized in function `displayBanners` and in the main section. It is used to display the various banners at the bottomof the screen. To do this, the turtle's "shape" is changed to the appropriate banner images stored in list `banner_images`. To prevent the turtle from drawing lines when moving from the initial (0, 0) coordinate location to the location where banners are displayed, the default turtle is hidden and its pen attribute is set to "up" (**lines 160–161**).



```
175 # generate and init horses
176 horses = generateHorses(horse_images)
177
178 # place horses at starting line
179 placeHorses(horses, start_loc, track_separation)
180
181 # start horses
182 winner = startHorses(horses, banner_images, finish_line,
183                      forward_incr)
184
185 # light up for winning horse
186 displayWinner(horses[winner], banner_images[3][0])
187
188 # terminate program when close window
189 turtle.exitonclick()
```

The only change in the main module of the program is related to the display of banner images. Added **lines 171** and **173** register and display the banners. The calls to `startHorses` and `displayWinner` on **lines 182** and **186** are changed (and the corresponding function definitions) to each pass one more argument consisting of banner images.