Module 6

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

Module 6 Study Guide and Deliverables

Theme: Classes in Detail

Readings:
- Chapters 11, 12, and 13
- Module Lecture Notes

Topics: Introduction to Classes, Assignment and Copy, Static vs. Instance Variables, Data Encapsulation, Overloading, Inheritance and Polymorphism, Multiple Inheritance and Abstract Classes

Assignments Assignment 6 due on Tuesday, April 27 at 6:00 PM ET

Assessments Quiz 6:
- Available Friday, April 23 at 6:00 AM ET
- Due on Tuesday, April 27 at 6:00 PM ET

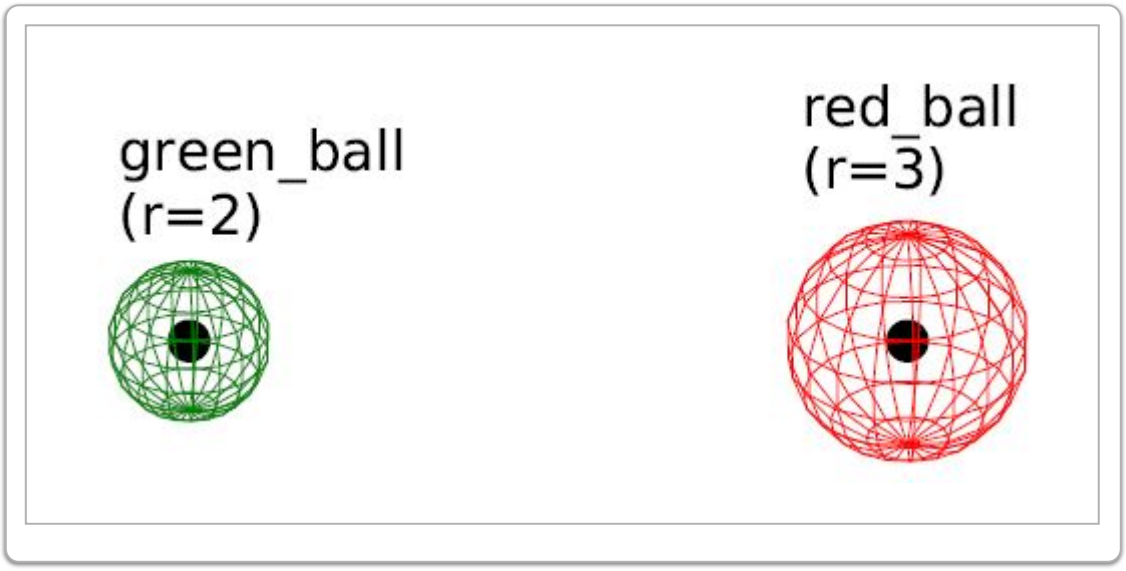
Live Classrooms:
- Tuesday, April 20, 8:00 - 9:30 PM ET
- Thursday, April 22, 6:00 - 7:30 PM ET
- Facilitator Session: Friday, April 23, at 8:00 PM ET

Introduction to Classes

Introduction

- Class: User defined data type (data and methods)
- Object: Instance of a class
- Python programs are built with objects
- Classes define objects
- Defined using *class* keyword
- New classes can be derived and meethods inherited

Example: Class *Sphere*

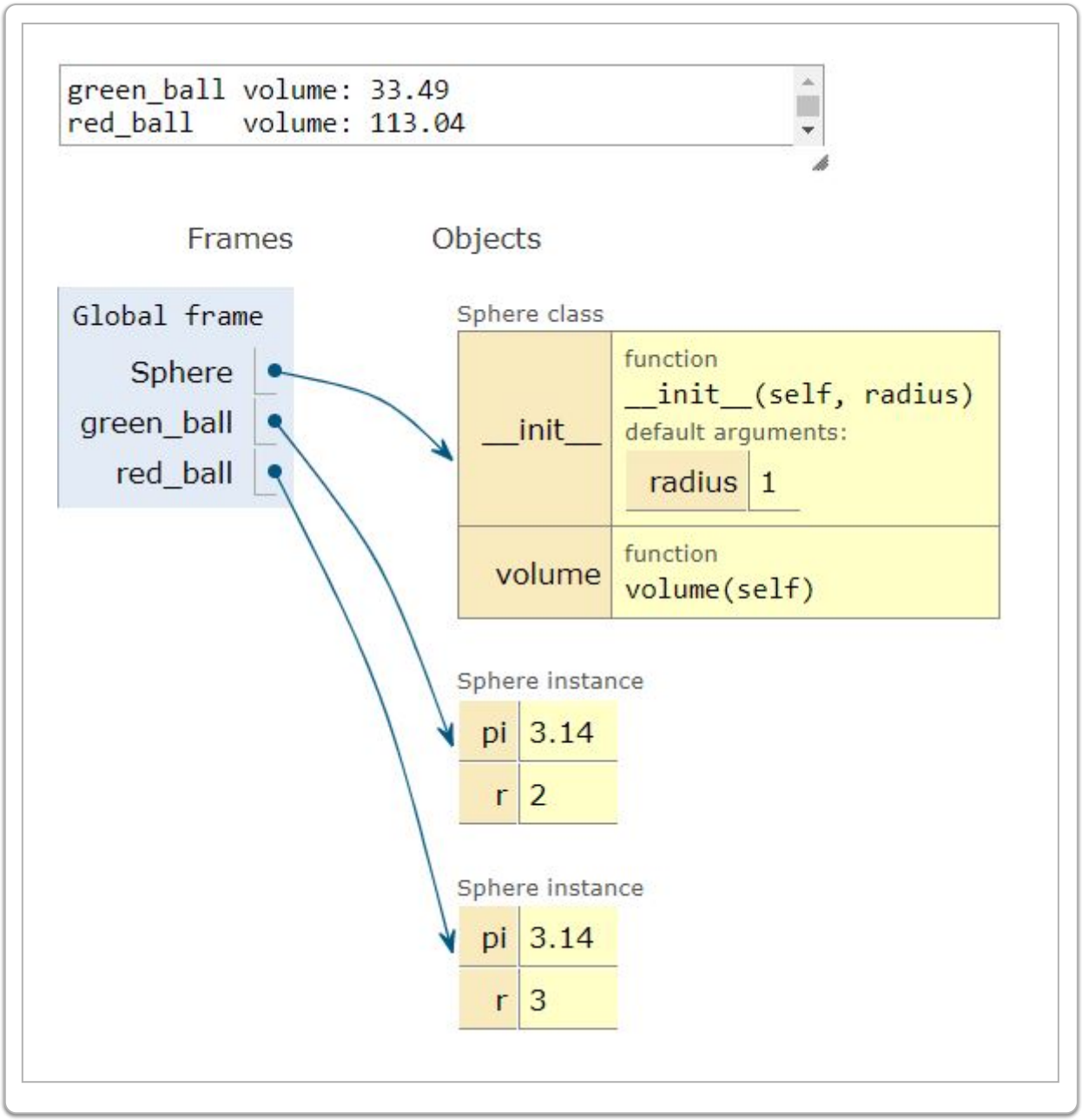


- *sphere* - class
- green_ball - an instance with radius 2
- red_ball - an instance with radius 3
- instances are distinct

```
class Sphere():
    def __init__(self, radius = 1):
        self.pi = 3.14
        self.r = radius

    def volume(self):
        return 4 * self.pi * self.r**3 / 3

green_ball = Sphere(2)
red_ball = Sphere(3)
print ('green_ball volume:', green_ball.volume())
print ('red_ball volume:', red_ball.volume())
```

- *_init_*() is a constructor
- *volume*() is a method
- *r* and *pi* are object variables
- volume for sphere: \(4\pi r^3/3\)

Details for Class *Sphere*



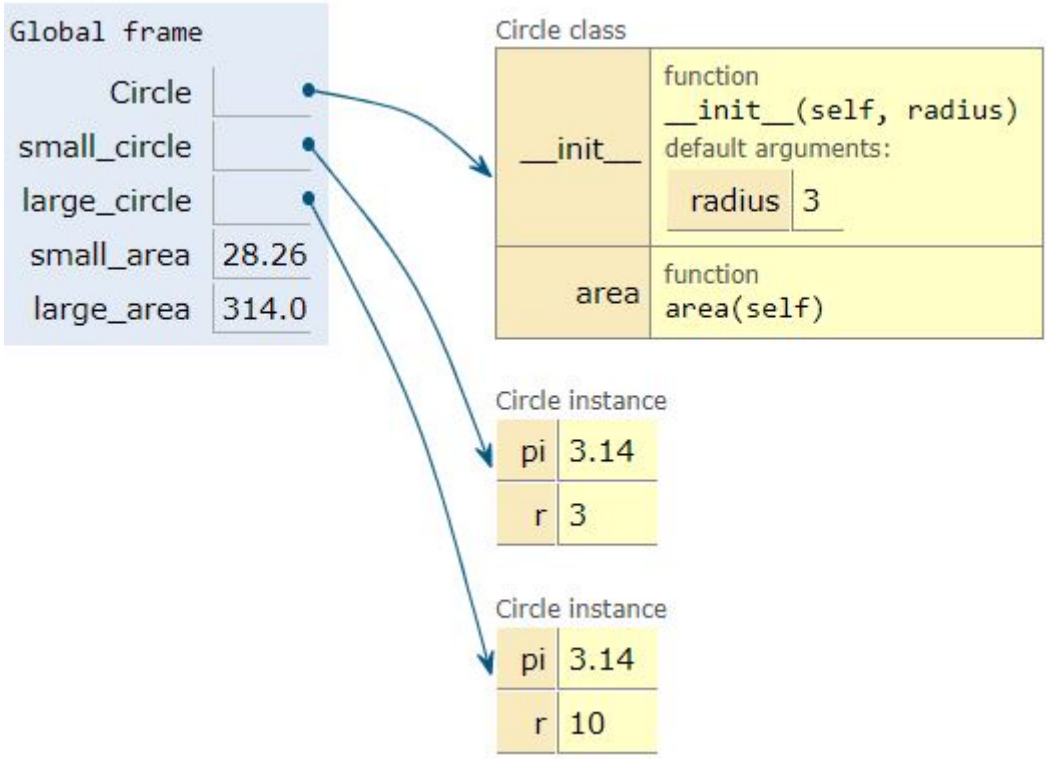
Test Yourself: 6.1.01

Define a class *Circle* according to the following:

- takes *radius* as constructor
- default radius is 3
- has a single method *area*()

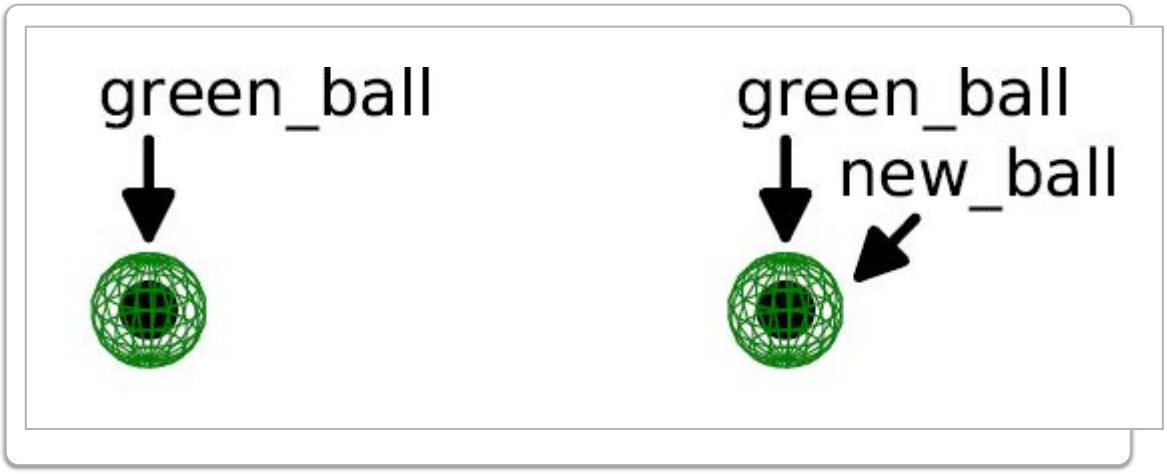
Solution:

```
class Circle():
    def __init__(self, radius = 3):
        self.pi = 3.14
        self.r = radius

    def area(self):
        return self.pi * self.r **2

small_circle = Circle() # radius 3
large_circle = Circle(10) # radius 10
small_area = small_circle.area()
large_area = large_circle.area()
```



Assignment and Copy

Object Assignment

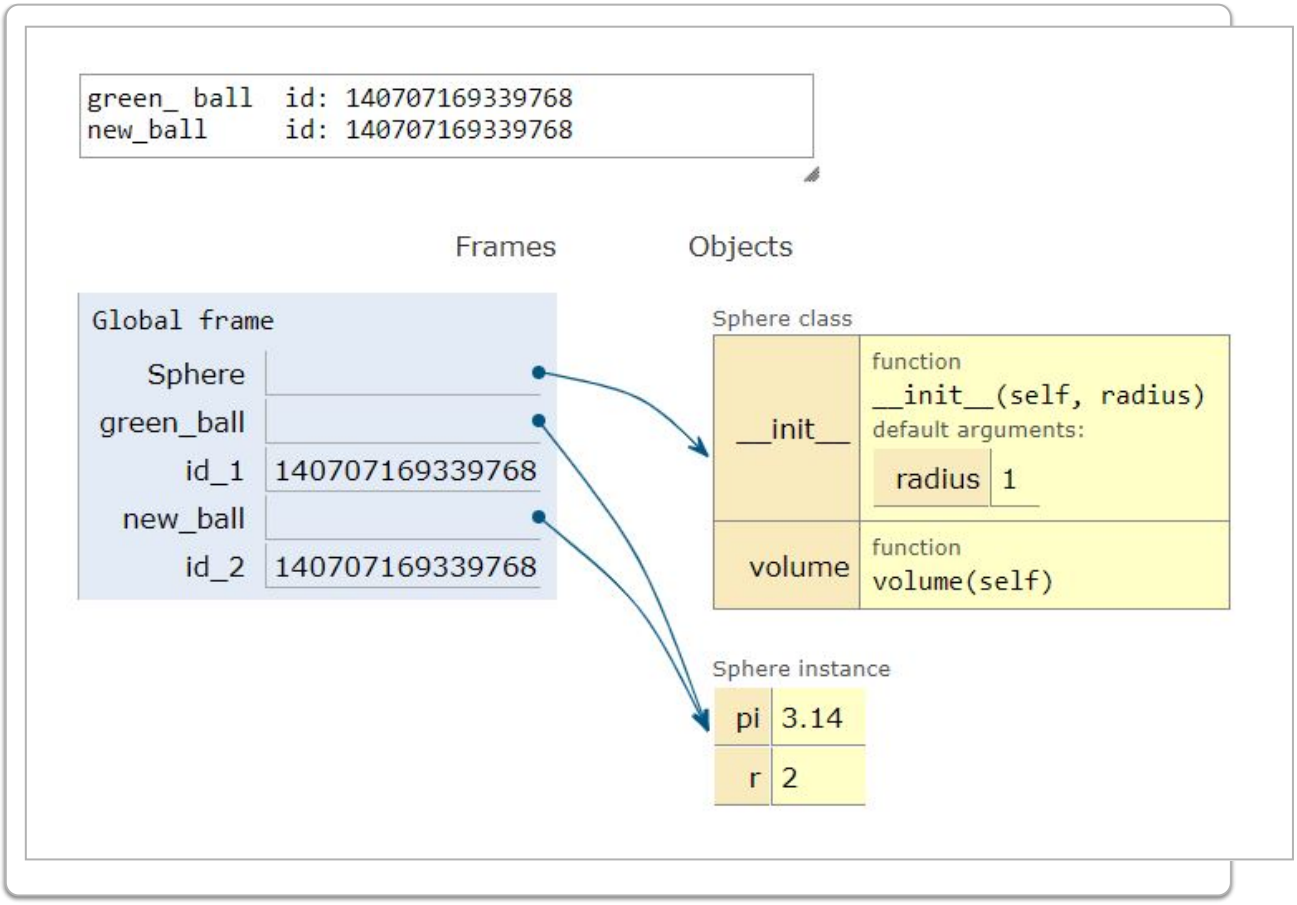



```
green_ball = Sphere(2)
id_1 = id(green_ball)

new_ball = green_ball
id_2 = id(new_ball)

print('green_ball id:', id_1)
print('new_ball id:', id_2)
```

```
green_ball  id: 140707169339768
new_ball    id: 140707169339768
```

- simply "retagging"

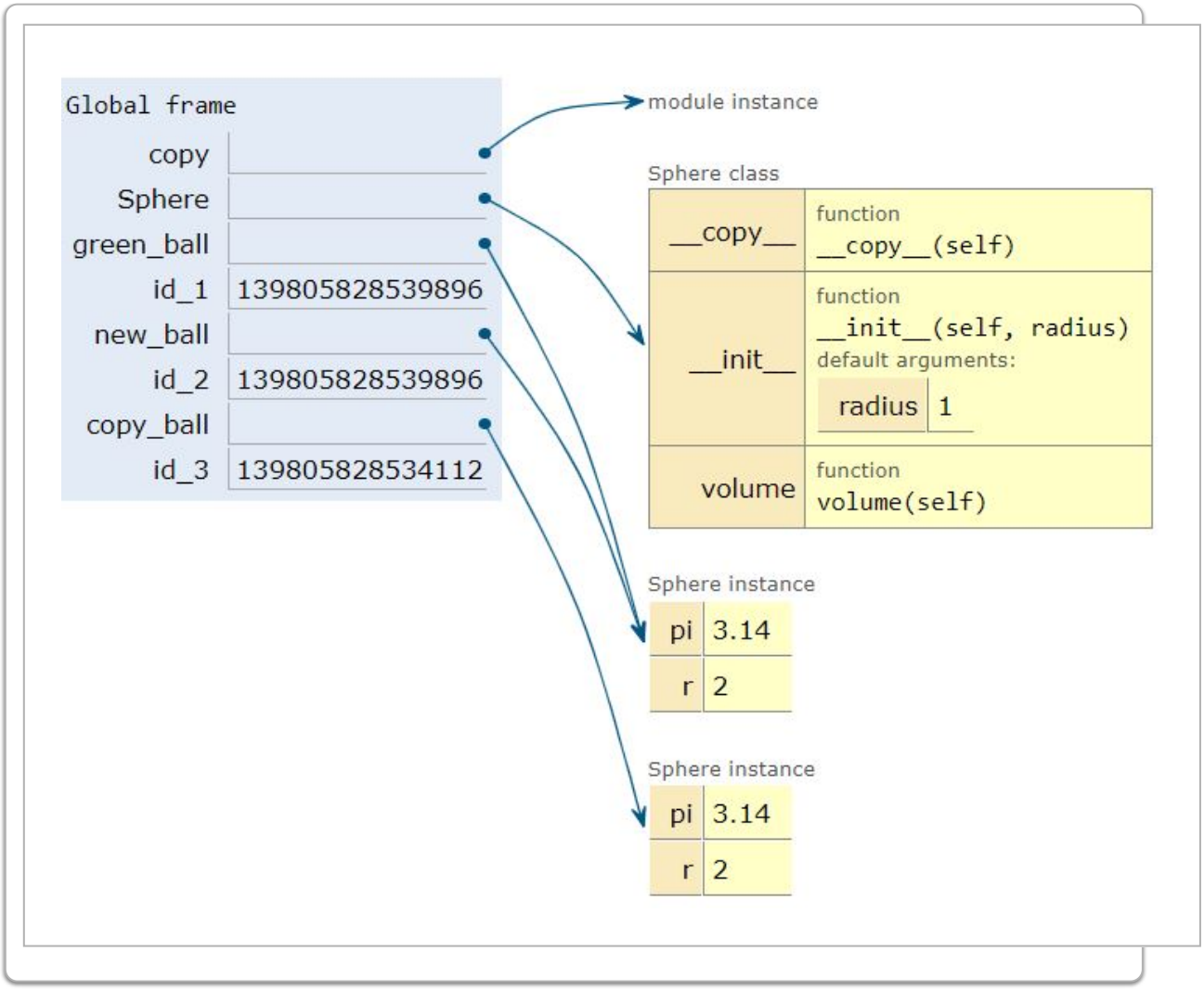
Assignment and Copy



- to copy an object, need to implement *_copy_*() method

Copying Objects

```
import copy
class Sphere():
    def __init__(self, radius = 1):
        self.pi = 3.14
        self.r = radius

    def volume(self):
        return 4 * self.pi * self.r**3 / 3

    def __copy__(self):
        return Sphere(self.r)

green_ball = Sphere(2)
id_1 = id(green_ball)
new_ball = green_ball
id_2 = id(new_ball)
copy_ball = copy.copy(green_ball)
id_3 = id(copy_ball)

print('green_ ball id:', id_1)
print('new_ball id:', id_2)
print('copy_ball id:', id_3)
```



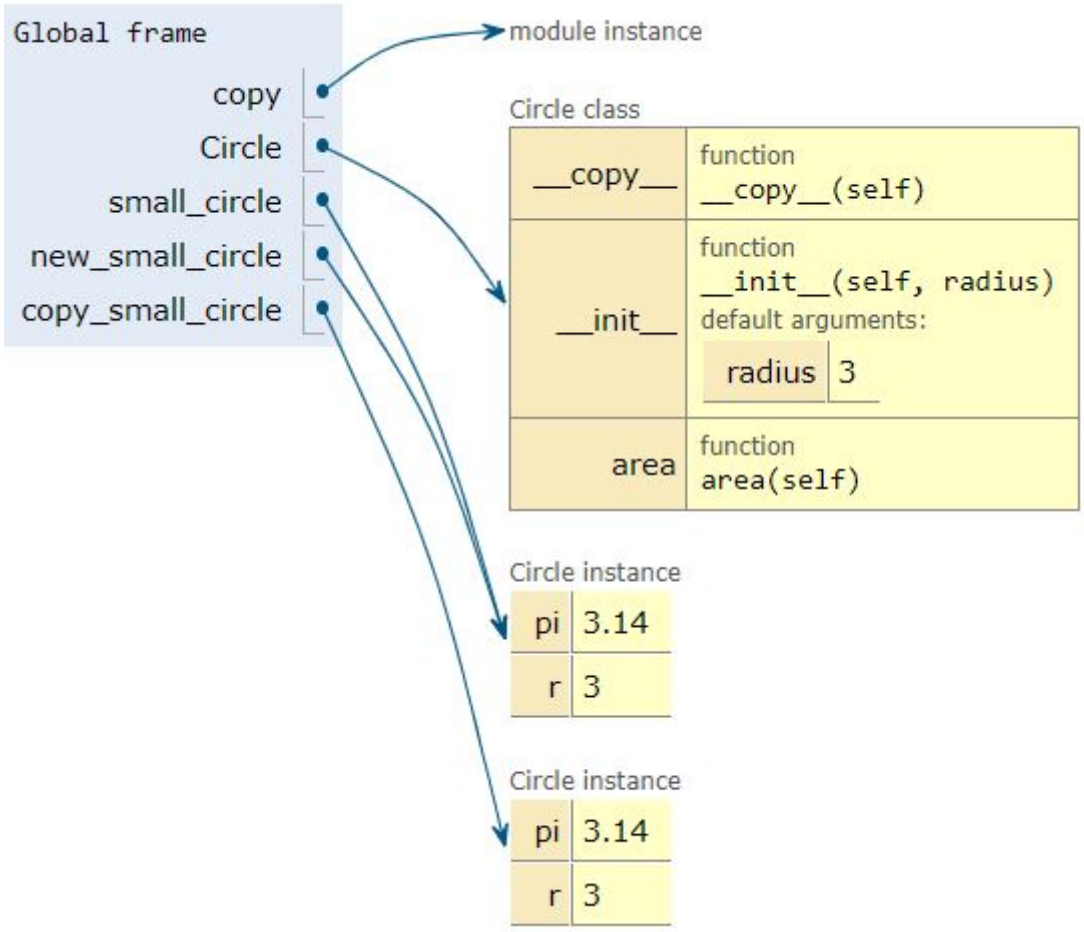
- "shallow" copy only

Test Yourself: 6.2.01

Add a *copy* method to allow copying for the *Circle* class.

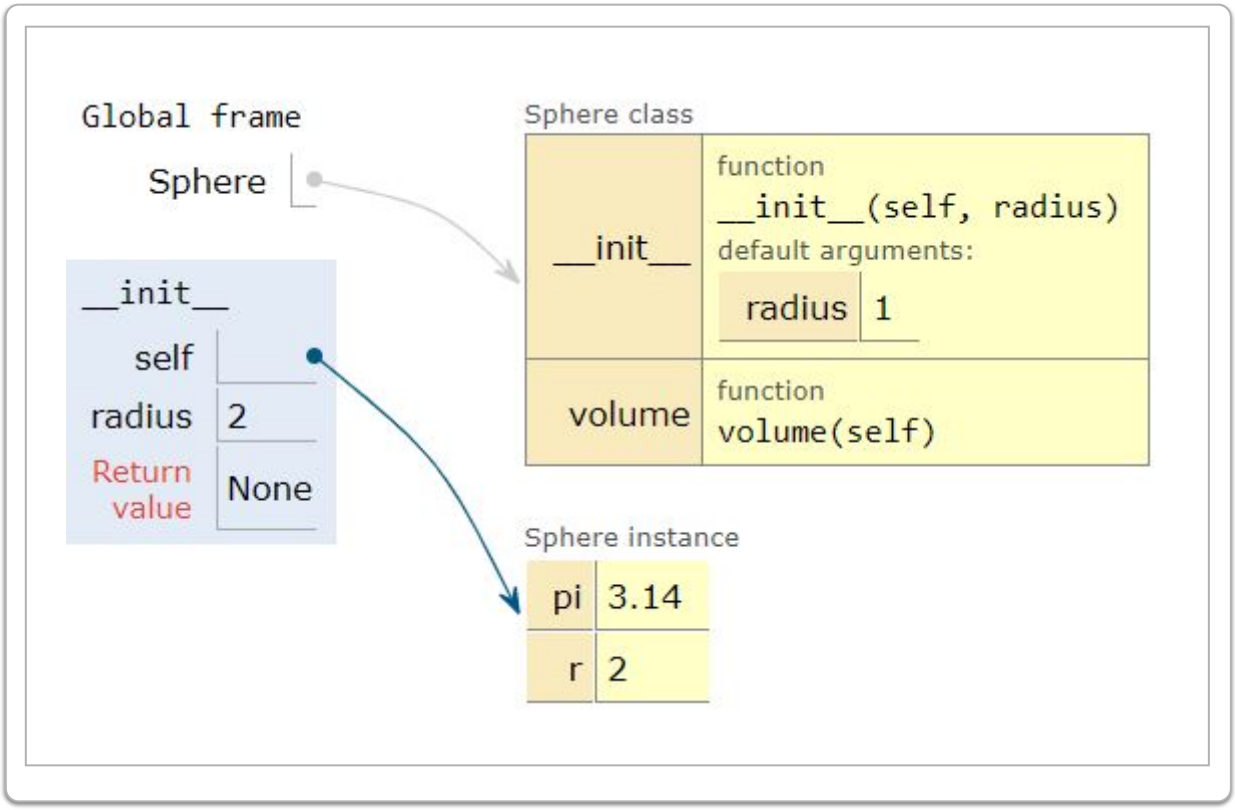
Solution:

```
import copy

class Circle():
    def __init__(self, radius = 3):
        self.pi = 3.14
        self.r = radius

    def area(self):
        return self.pi * self.r **2

    def __copy__(self):
        return Circle(self.r)

small_circle = Circle()
new_small_circle = small_circle
copy_small_circle = copy.copy(small_circle)
```



Static vs. Instance Variables

self Parameter



- each instance is passed *self* parameter
- similar to *this* in Java/C++
- allows object to keep its own data

Static vs. Instance Variables

- *r*, *pi*: instance variables
- each instance has its own copy
- *pi* is the same across instances
- need to make *pi* static
- static: single copy per class
- how: define before methods
- access as Class.Name

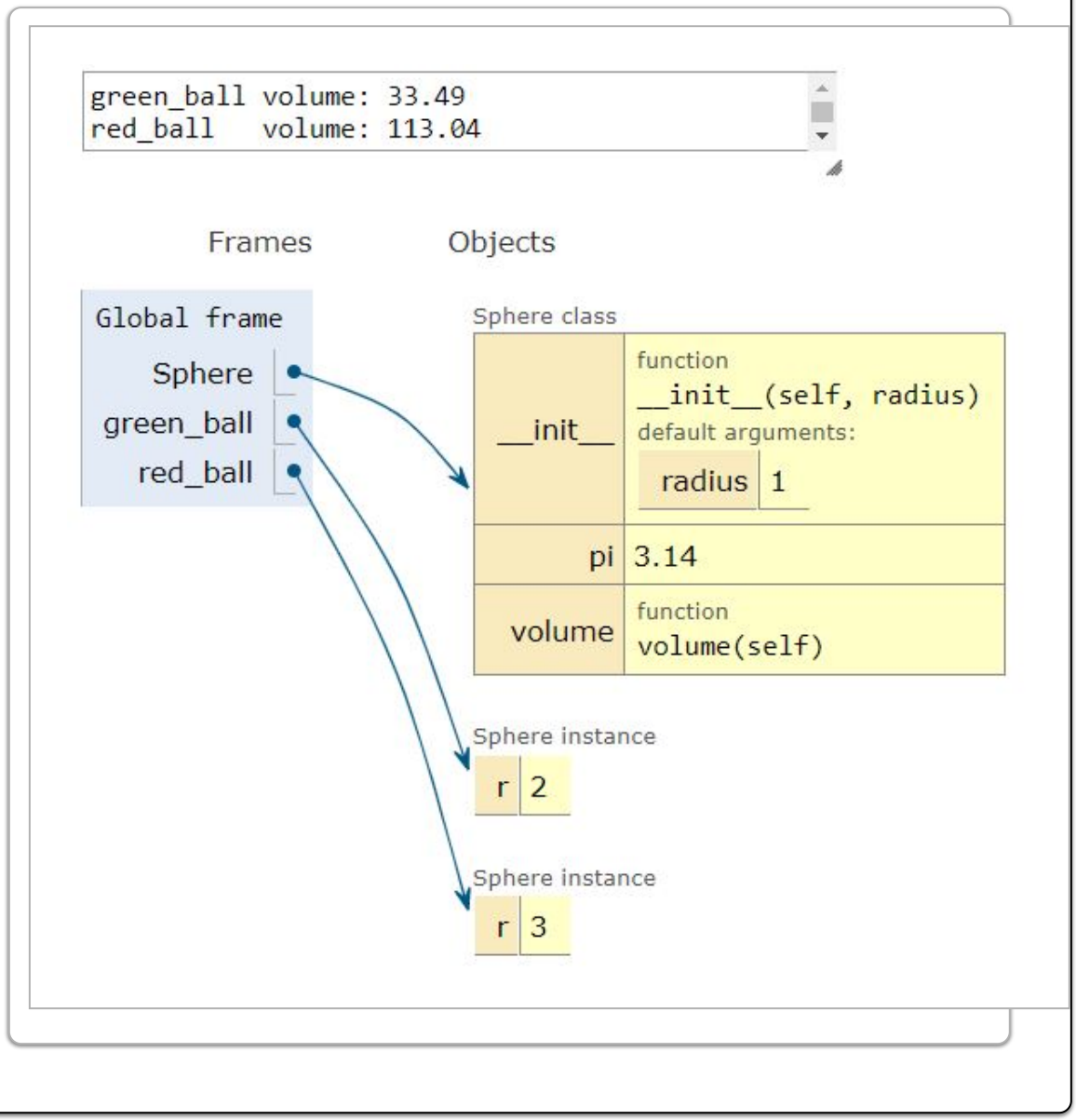
Static Variables

```
class Sphere():
    pi = 3.14 # static
```

```
    def __init__(self, radius = 1):
        self.r = radius # instance

    def volume(self):
        return 4 * Sphere.pi * self.r**3 / 3

green_ball = Sphere(2)
red_ball = Sphere(3)
print('green_ball volume:', green_ball.volume())
print('red_ball volume:', red_ball.volume())
```

- a single copy of *pi* is shared

Example



Static vs. *Non-Static*

- non-static

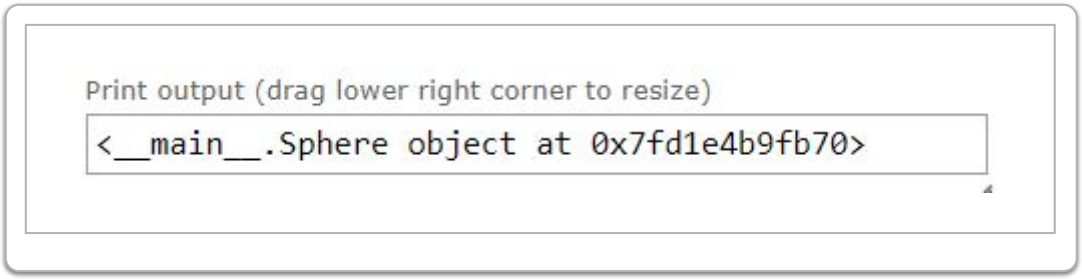
```
class Sphere():
    def __init__(self, radius = 1):
        self.pi = 3.14 # instance
        self.r = radius # instance

    def volume(self):
        return 4 * self.pi * self.r**3 / 3
```

- static

```
class Sphere():
    pi = 3.14 # static

    def __init__(self, radius = 1):
        self.r = radius # instance

    def volume(self):
        return 4 * Sphere.pi * self.r**3 / 3
```

Describing Objects

```
class Sphere():
    pi = 3.14

    def __init__(self, radius = 1):
        self.r = radius

    def volume(self):
        return 4 * Sphere.pi * self.r**3 / 3

green_ball = Sphere(2)
print(green_ball)
```

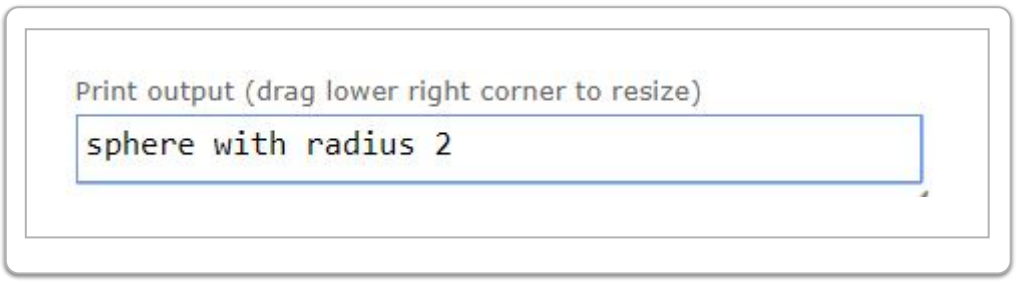


- want to give "human" description

__str__() Method

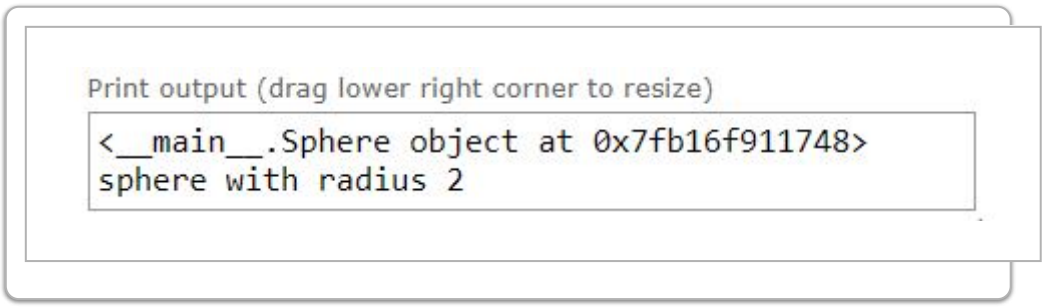
- user-defined description

```
class Sphere():
    pi = 3.14

    def __init__(self, radius = 1):
        self.r = radius

    def __str__(self):
        return 'sphere with radius {}'\
            .format(self.r)

    def volume(self):
        return 4 * Sphere.pi * self.r**3 / 3

green_ball = Sphere(2)
print(green_ball)
```



*str*() and _repr_()

```
green_ball = Sphere(2)
print(repr(green_ball))
print(green_ball)
```



- _*repr*_(): "official" object description
- _*str*_(): "human" object description
- _*str*_() uses _*repr*_() as a fall-back

class Template

```
class Sphere(): # class name

    pi = 3.14 # static data field (s)

    def __init__(): # constructor

    def __str__(self): # representation

    def volume(): # method (s)
```

- classes use "dot" notation

```
green_ball = Sphere(2)
volume_1 = green_ball.volume()
```

- all variables are public

Test Yourself: 6.3.01

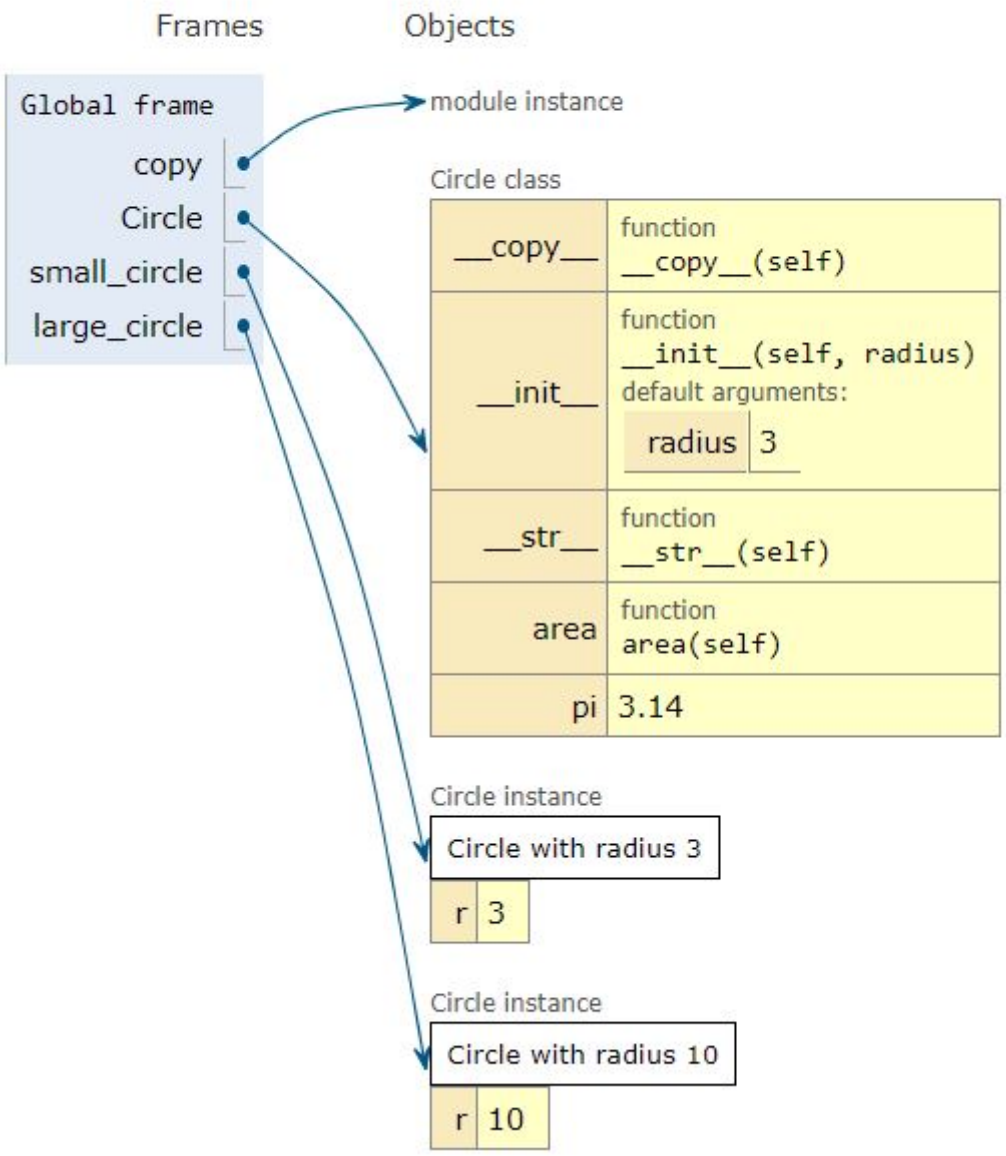
Make *pi* to be static in your class:

- write *_str_*() method for the *Circle* class

Solution:

```
import copy

class Circle():
    pi = 3.14
    def __init__(self, radius = 3):
        self.r = radius

    def area(self):
        return Circle.pi * self.r**2

    def __copy__(self):
        return Circle(self.r)

    def __str__(self):
        return "Circle with radius {}"\
                .format(self.r)

small_circle = Circle()
large_circle = Circle(10)

print (small_circle)
print (large_circle)
```

```
Circle with radius 3
Circle with radius 10
```



Data Encapsulation

Data Privacy

- No mechanism for privacy

```
green_ball = Sphere(2)
print('green_ball volume:', green_ball.volume())

Sphere.pi = 0
```

```
print('set pi = 0 ')
print('green_ball volume:', green_ball.volume())
```

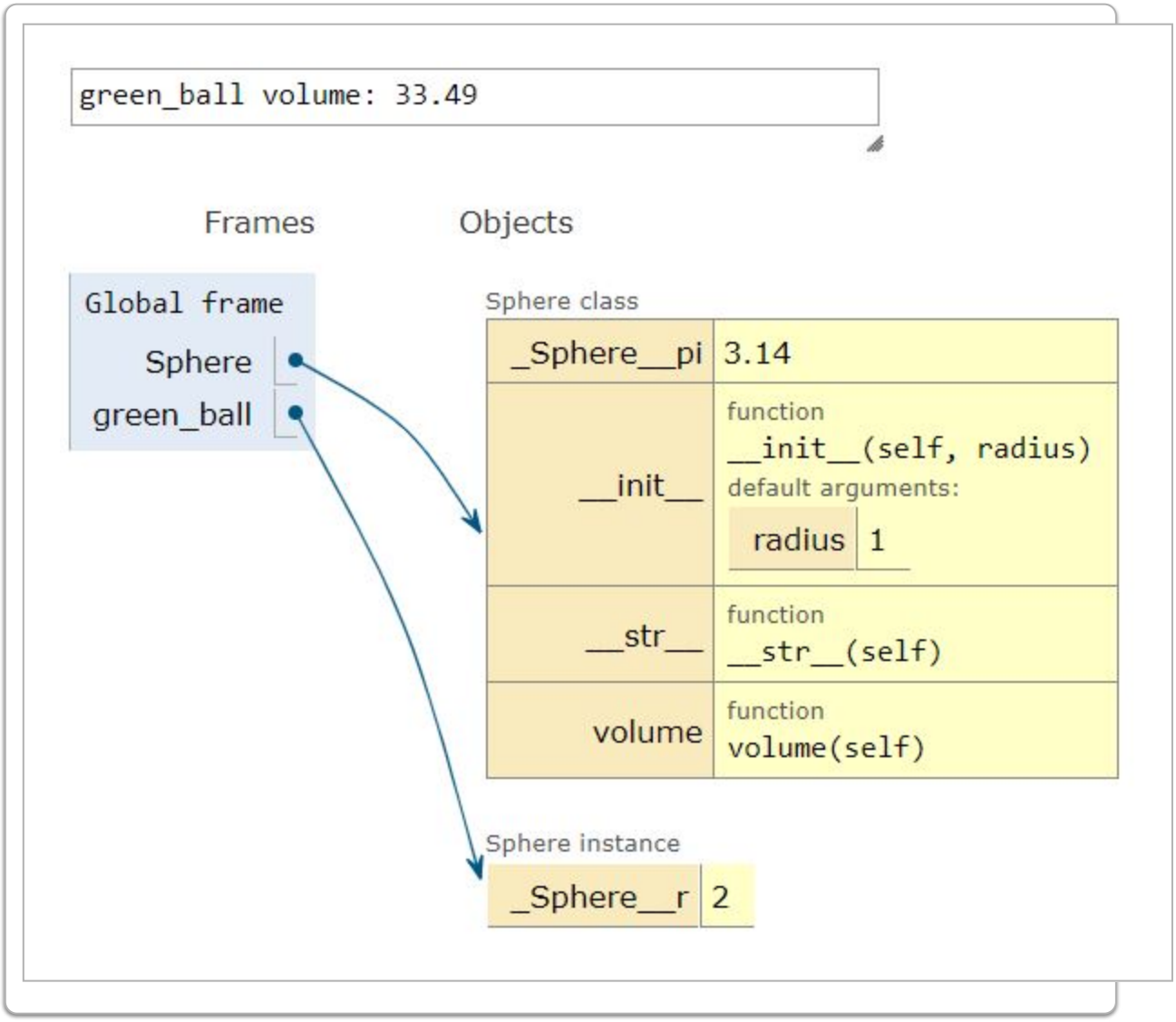
```
green_ball volume: 33.49
set pi = 0
green_ball volume: 0.0
```

- Solution: "name mangling"
- How? *pi* \(\mapsto\) *_pi* & *r* \(\mapsto\) *_r*
- Python creates new names:

  _Sphere_pi & *_Sphere_r*

Name Mangling

```
class Sphere():
    __pi = 3.14 # name mangling

    def __init__(self, radius = 1):
        self.__r = radius

    def __str__(self):
        return 'sphere with radius {} '\
            .format(self.__r)

    def volume(self):
        return 4*Sphere.__pi * self.__r**3/3

green_ball = Sphere(2)
print('green_ball volume:', round(green_ball.volume))

# code below will now generate an error
Sphere.pi = 0

# in theory , can still set it to 0
Sphere._Sphere__pi = 0
```



- prevents accidental change

Accessing and Setting Class Variables

- data encapsulation
- expose instance variables by methods:
    1. *accessors*: return values
    2. *mutators*: set or change variables

Modified Class Example

```
class Sphere():
    __pi = 3.14 # name mangling

    def __init__(self, radius = 1):
        self.__r = radius

    def __str__(self):
        return 'sphere with radius {}'\
            .format(self.__r)

    def set_radius(self, r): # mutator
        self.__r = r # (setter)

    def get_radius(self): # accessor
        return self.__r # (getter)

    def volume(self):
        return 4 * Sphere.__pi * self.__r **3 / 3
```

Test Yourself: 6.4.01

Apply "name mangling" to class variables in *Circle* class.

Solution:

```
import copy
class Circle():
    __pi = 3.14
    def __init__(self, radius = 3):
        self.__r = radius

    def __str__(self):
        return "Circle with radius {}"\
            .format(self.__r)

    def __copy__(self):
        return Circle(self.__r)

    def set_radius(self, r): # mutator
        self.__r = r # (setter)

    def get_radius(self): # accessor
        return self.__r # mutator
```

Overloading

Operator Overloading

- Built-in types have common operators (+, <, ==)

- Can override buil-in methods
- How: special functions ("magic" methods)
- Such functions start and end with __ (double undersocre)
- Example: to use '+', need to define __*add*__()

Overloading +

- need to define __add__():

```
def __add__(self, other):
    return Sphere(self.__r + \
            other.__r)
```

- add or compare objects like any data types

```
green_ball = Sphere(2)
red_ball = Sphere(3)
blue_ball = green_ball + red_ball
print(blue_ball)
```