Class

Abolfazl Madani

GitHub: abmadani

email: abolfazl.madani71@gmail.com

It's all objects…

- Everything in Python is really an object.
  - We've seen hints of this already…

    ```
    "hello".upper()
    list3.append('a')
    dict2.keys()
    ```

    - These look like Java or C++ method calls.
    - New object classes can easily be defined in addition to these built-in data-types.
- In fact, programming in Python is typically done in an object oriented fashion.

Defining a Class

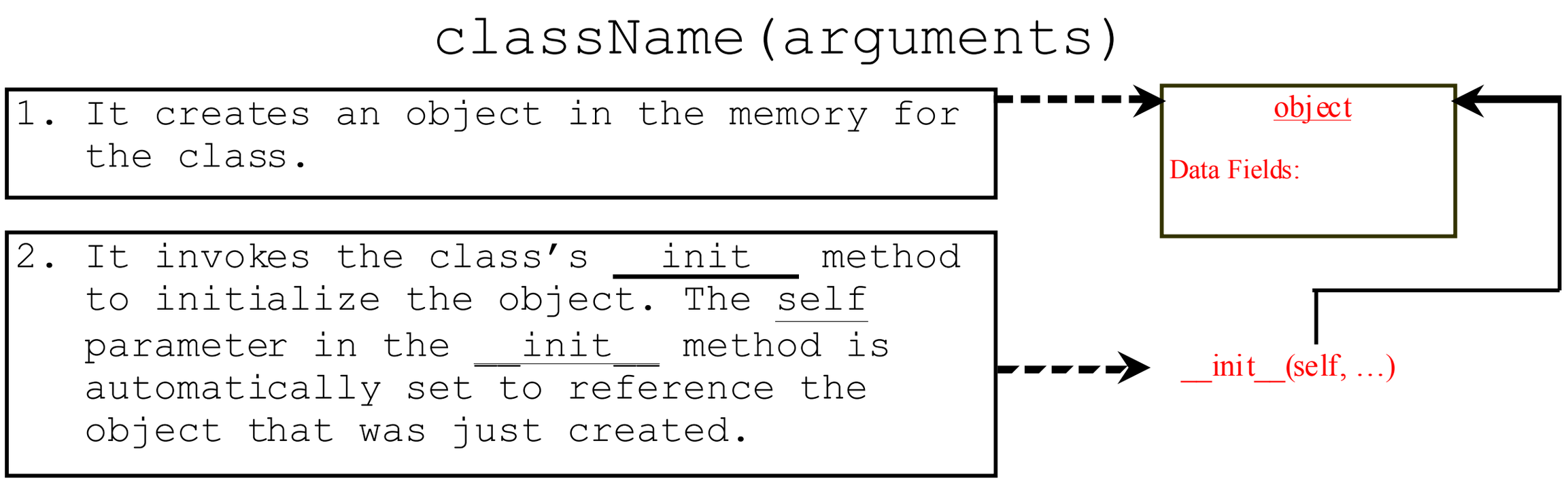
- A *class* is a special data type which defines how to build a certain kind of object.
- The *class* also stores some data items that are shared by all the instances of this class
- *Instances* are objects that are created which follow the definition given inside of the class
- Python dosen't use separate class interface definitions as in some languages
- You just define the class and then use it

Methods in Classes

- Define a *method* in a *class* by including function definitions within the scope of the class block
- There must be a special first argument **`self`** in *all* of method definitions which gets bound to the calling instance
- There is usually a special method called **`__init__`** in most classes
- We'll talk about both later…

Constructing Objects

Once a class is defined, you can create objects from the class by using the following syntax, called a *constructor*:



A simple class def: student

```
class student:
 """A class representing a
 student """
    def __init__(self,n,a):
        self.full_name = n
        self.age = a
    def get_age(self):
        return self.age
```

OOP principles

- ***encapsulation***: hiding design details to make the program clearer and more easily modified later
- ***modularity***: the ability to make objects stand alone so they can be reused (our modules). Like the math module
- ***inheritance***: create a new object by inheriting (like father to son) many object characteristics while creating or over-riding for this object
- ***polymorphism***: (hard) Allow one message to be sent to any object and have it respond appropriately based on the type of object it is.

Class versus instance

- One of the harder things to get is what a class is and what an instance of a class is.

- The analogy of the cookie cutter and a cookie.

Why a class

- We make classes because we need more complicated, user-defined data types to construct instances we can use.

- Each class has potentially two aspects:
  - the data (types, number, names) that each instance might contain
  - the messages that each instance can respond to

Classes

A Python class uses variables to store data fields and defines methods to perform actions. Additionally, a class provides a special type method, known as *initializer*, which is invoked to create a new object. An initializer can perform any action, but initializer is designed to perform initializing actions, such as creating the data fields of objects.

```
class ClassName:
    initializer
    methods
```

Circle

TestCircle

Run

Overriding a Method

- __init__ is frequently overridden because many subclasses need to both (a) let their superclass initialize their data, and (b) initialize their own data, usually in that order

```
class Stack(list):
    push = list.append

class Calculator(Stack):
    def __init__(self):
        Stack.__init__(self)
        self.accumulator = 0
    def __str__(self):
        return
str(self.accumulator)
    def push(self, value):
        Stack.push(self, value)
        self.accumulator = value

c = Calculator()
c.push(10)
print c
```

Every class should have `__init__`

- By providing the constructor, we ensure that every instance, at least at the point of construction, is created with the same contents

- This gives us some control over each instance.

Standard Class Names

The standard way to name a class in Python is called ***CapWords***:

- Each word of a class begins with a Capital letter
- no underlines
- sometimes called ***CamelCase***
- makes recognizing a class easier

Part of the Object Scope Rule

The first two rules in object scope are:

- First, look in the object itself

- If the attribute is not found, look up to the class of the object and search for the attribute there.

OO Programming Concepts

Object-oriented programming (OOP) involves programming using objects. An *object* represents an entity in the real world that can be distinctly identified.

For example, a student, a desk, a circle, a button, and even a loan can all be viewed as objects.

An object has a unique identity, state, and behaviors. The *state* of an object consists of a set of *data fields* (also known as *properties*) with their current values. The *behavior* of an object is defined by a set of methods.

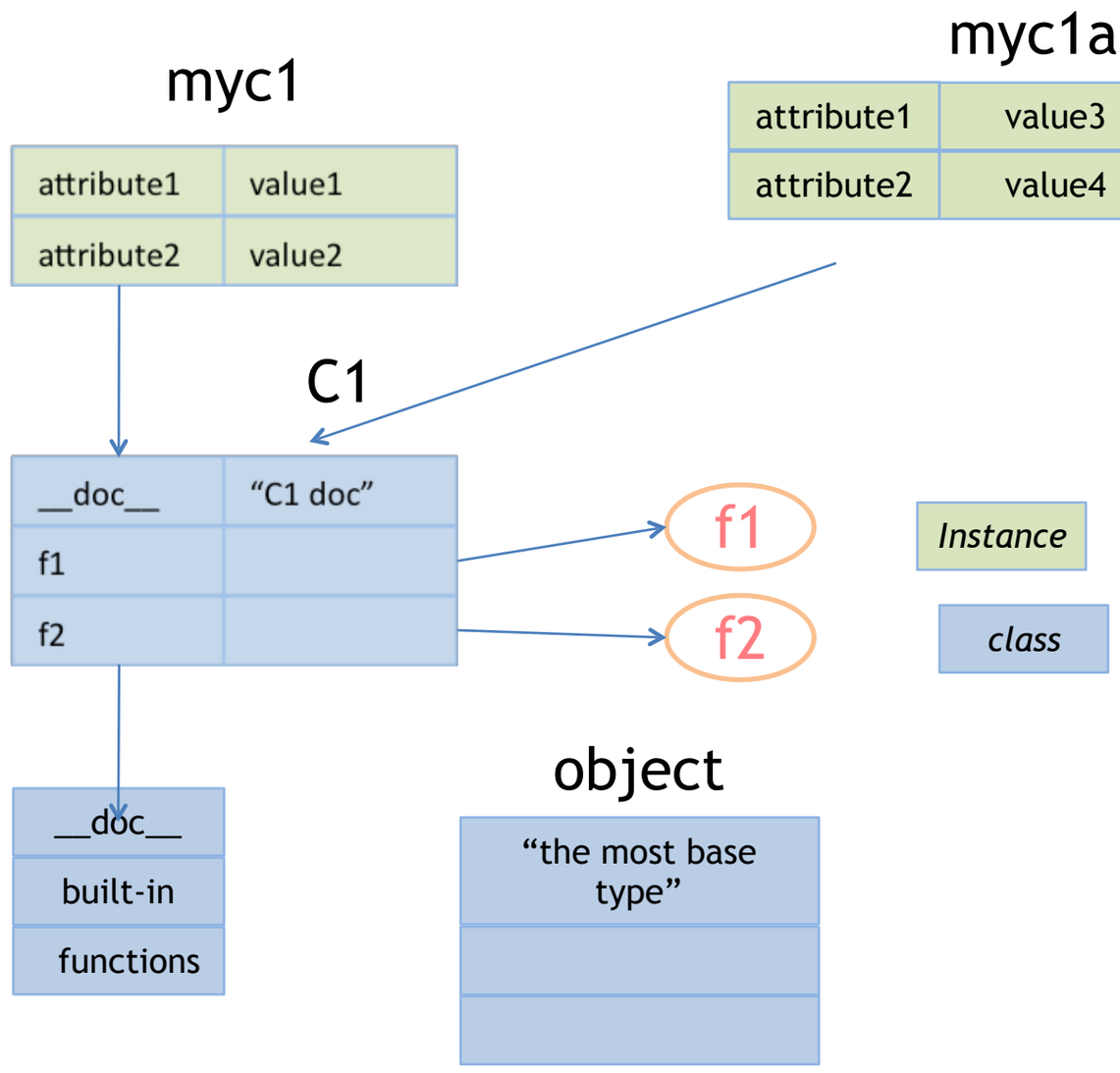
OOP with Classes

- Suppose we have class **C1** and instances **myc1** and **myc1a**

```
class C1(object):
    "C1 doc"
    def f1(self):
        # do something with
self
    def f2(self):
        # do something with
self

# create C1 instances
myc1 = C1()
myc1a = C1()

# call f2 method on one
instance
myc1.f2()
```



OOP with Classes (cont.)

- The **object** class is created automatically by Python
- Executing the “class” statement creates the **C1** class
  - Note **C1** is actually a variable: a reference to a **class** object; this is analogous to the “import” statement where the result is a variable referring to a **module** object
  - Note also that the class object contains data, eg **__doc__**, as well as method references, eg **f1** and **f2**

```
class C1(object):
    “C1 doc”
    def f1(self):
        # do something with
self
    def f2(self):
        # do something with
self

# create a C1 instance
myc1 = C1()
myc1a = C1()

# call f2 method
myc1.f2()
```

OOP with Classes (cont.)

- Creating an instance creates a new attribute namespace
- Each instance has its own attribute namespace, but they all share the same *class* namespace(s)
- Both instance and class attributes may be accessed using the *instance.attribute* syntax

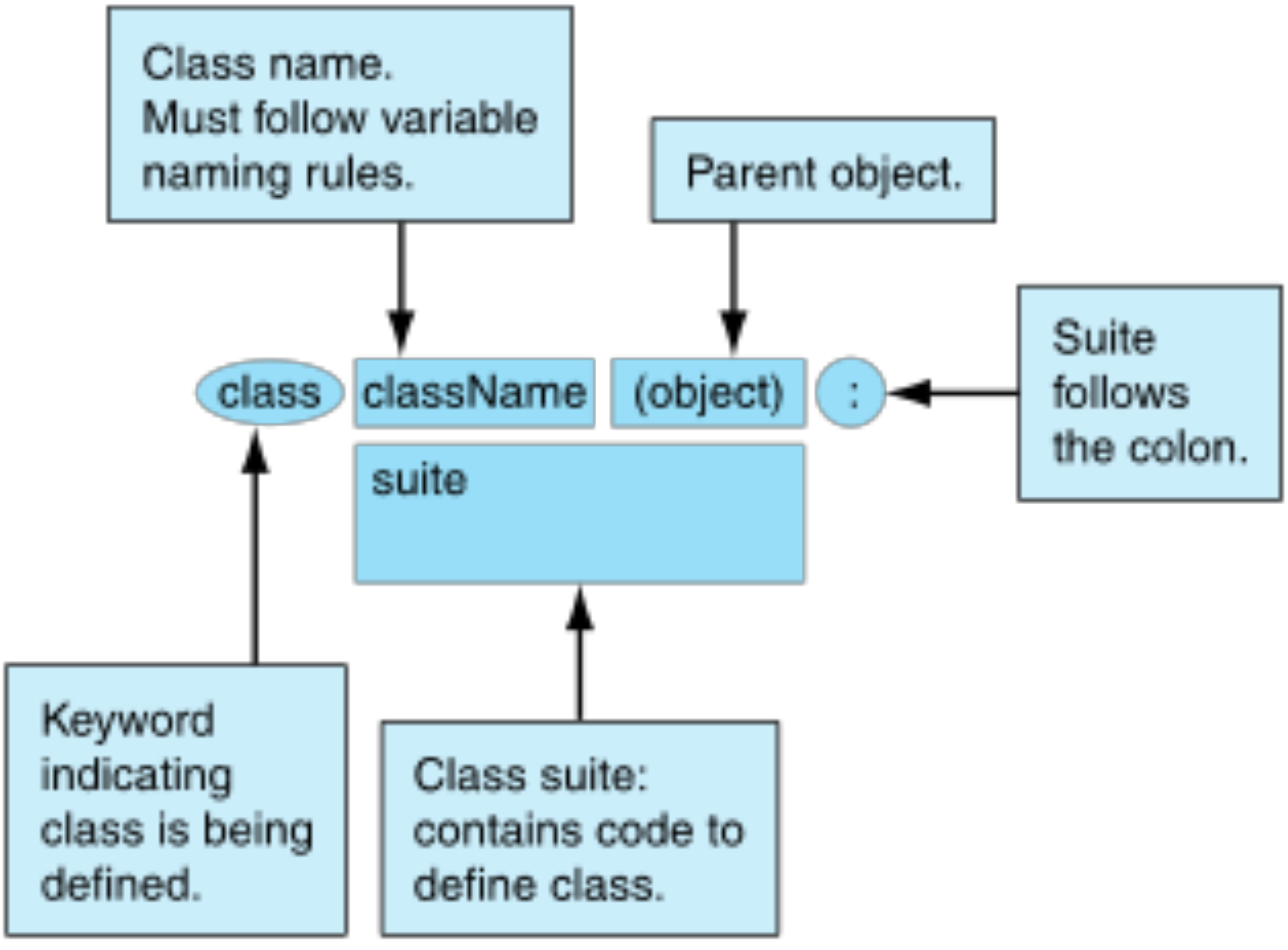
Accessing Attributes (cont.)

- Setting and looking up class attributes
  - Class attributes may be looked up via the instances, but they cannot be modified using the *instance.attribute* syntax
  - To access and manipulate class attributes, use the class variable

```
>>> C1.count = 12
>>> print C1.count
12
>>> C1.f1
<unbound method C1.f1>
>>> C1.f1(myc1)
>>> print C1.__doc__
C1 doc
>>> C1.__doc__ = "new
documentation"
Traceback (most recent call
last):
  File "<stdin>", line 1, in
<module>
AttributeError: attribute
'__doc__' of 'type' objects is
not writable
>>> help(C1)
…
```

Objectives

- To describe objects and classes, and use classes to model objects
- To define classes
- To construct an object using a constructor that invokes the initializer to create and initialize data fields
- To access the members of objects using the dot operator (.)
- To reference an object itself with the self parameter
- To use UML graphical notation to describe classes and objects
- To distinguish between immutable and mutable
- To hide data fields to prevent data corruption and make classes easy to maintain
- To apply class abstraction and encapsulation to software development
- To explore the differences between the procedural paradigm and the object-oriented paradigm


Class name.
Must follow variable
naming rules.
Parent object.
Suite
follows
the colon.
class
className
(object)
:
suite
Keyword
indicating
class is being
defined.
Class suite:
contains code to
define class.

Why self?

self is a parameter that represents an object.

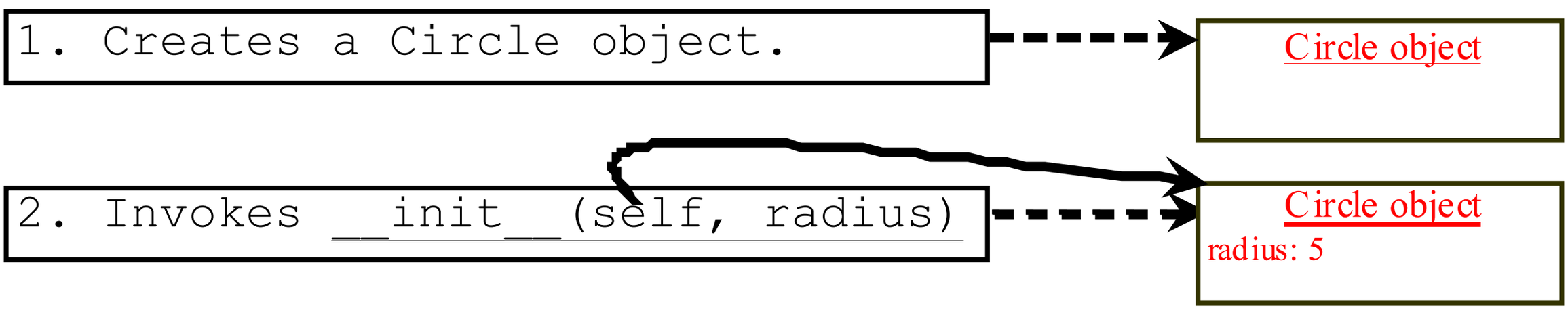
Using self, you can access instance variables in an object. Instance variables are for **storing data** fields.

Each object is an instance of a class. Instance variables are tied to specific objects.

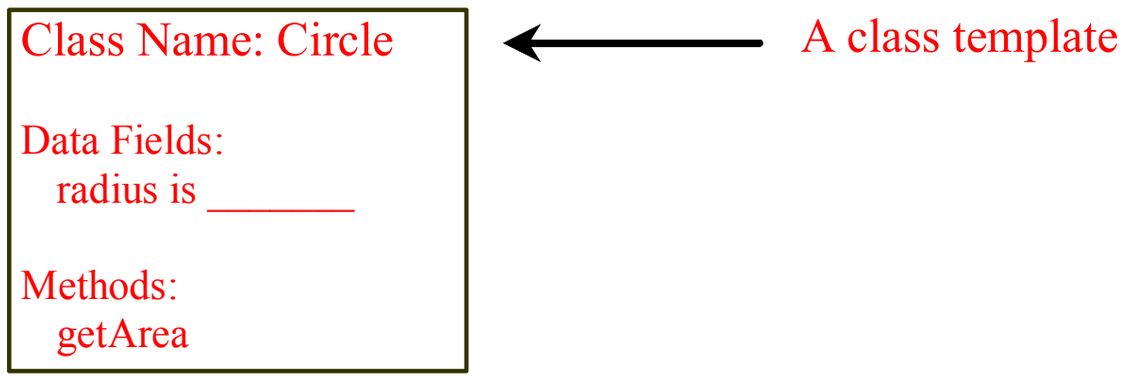
Each object has its own instance variables. You can use the syntax self.x to access the instance variable x for the object self in a method.

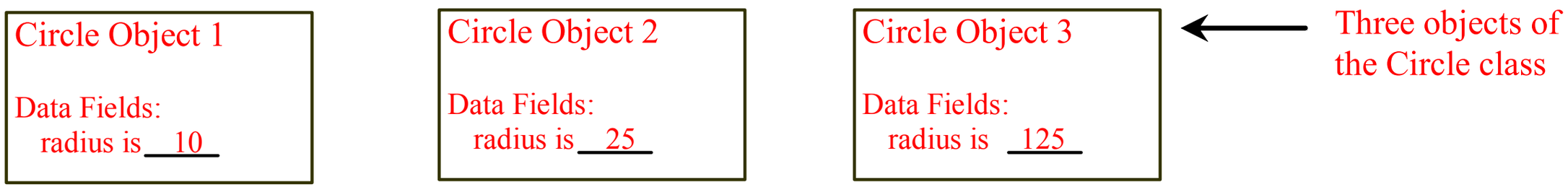
Constructing Objects

The effect of constructing a Circle object using Circle(5) is shown below:



Objects


Class Name: Circle
Data Fields:
radius is _____
Methods:
getArea
A class template


Circle Object 1
Data Fields:
radius is 10
Circle Object 2
Data Fields:
radius is 25
Circle Object 3
Data Fields:
radius is 125
Three objects of the Circle class

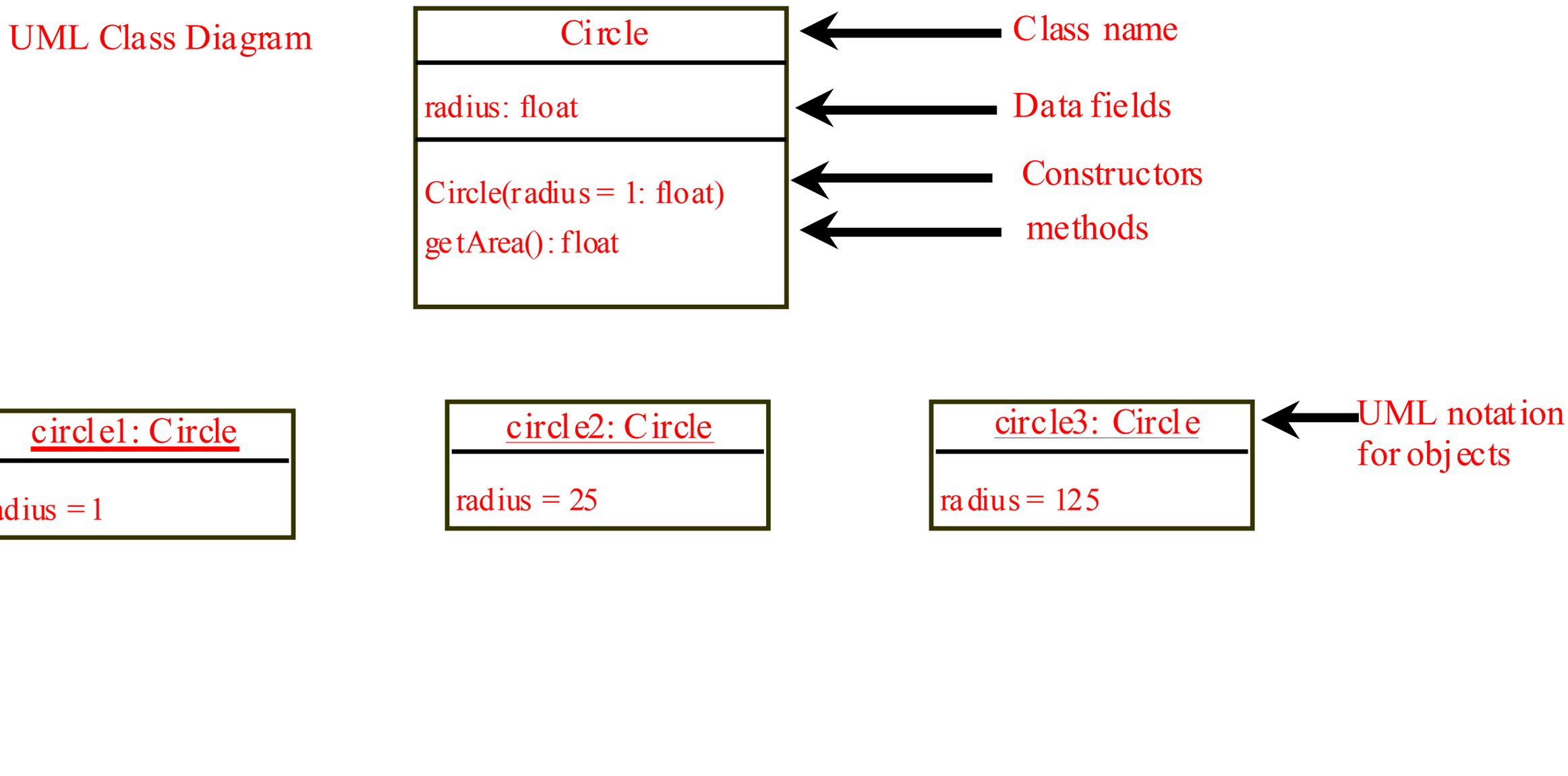
Accessing Objects

After an object is created, you can access its data fields and invoke its methods using the dot operator (.), also known as the *object member access operator*.

For example, the following code accesses the radius data field and invokes the getPerimeter and getArea methods.

```
>>> from Circle import Circle
   >>> c = Circle(5)
  >>> c.getPerimeter()
   31.415926535897932
   >>> c.radius = 10
    >>> c.getArea()
   314.15926535897932
```

UML (Unified Modeling Language) Class Diagram



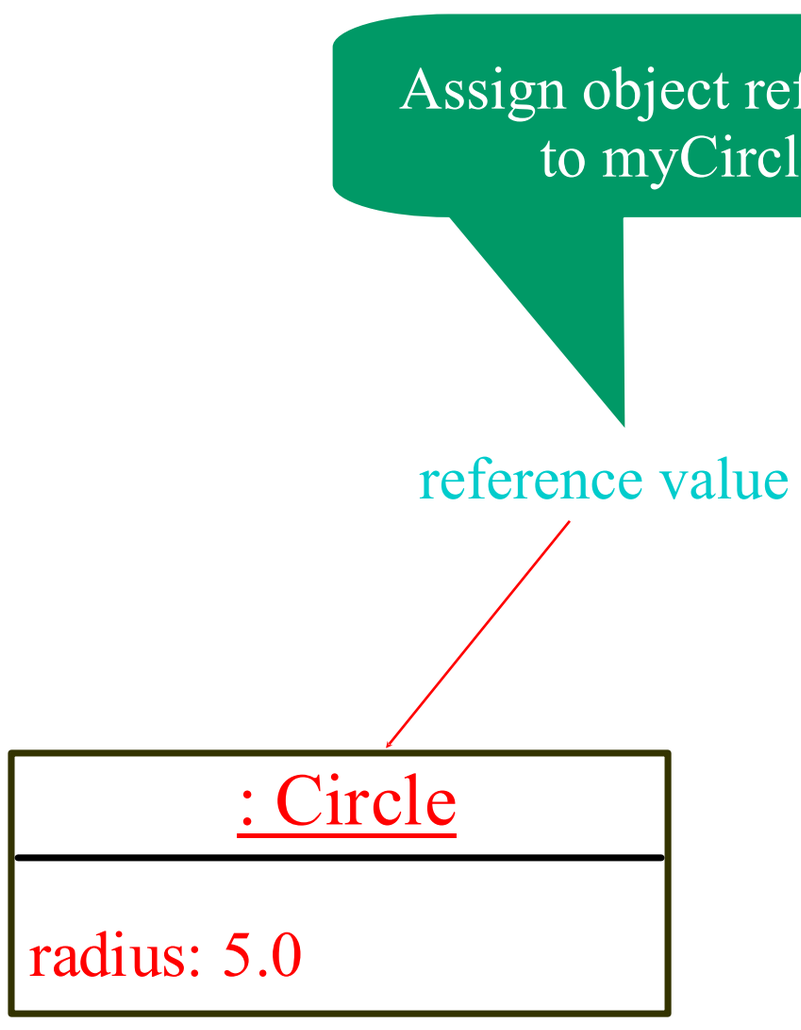
Trace Code

myCircle = Circle(5.0)

yourCircle = Circle()

yourCircle.radius = 100

Assign object reference to myCircle

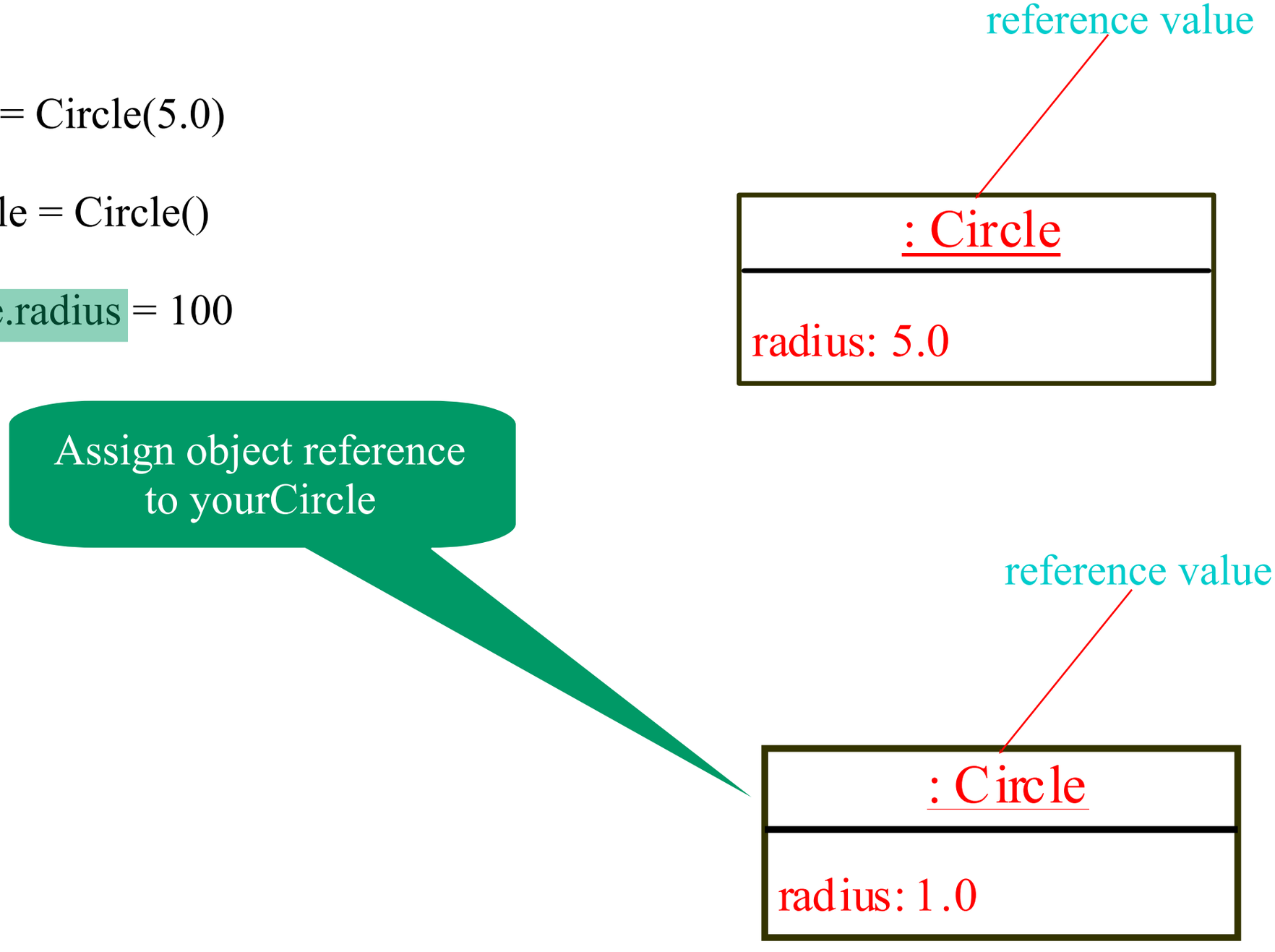


Trace Code

myCircle = Circle(5.0)

yourCircle = Circle()

yourCircle.radius = 100

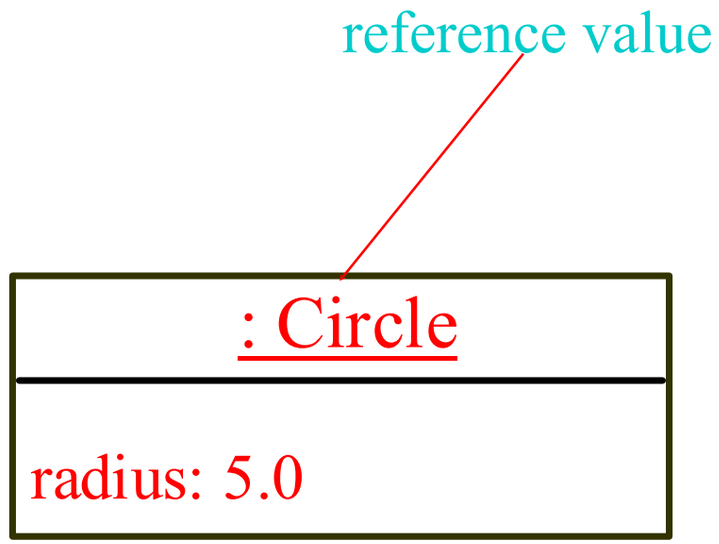


Trace Code

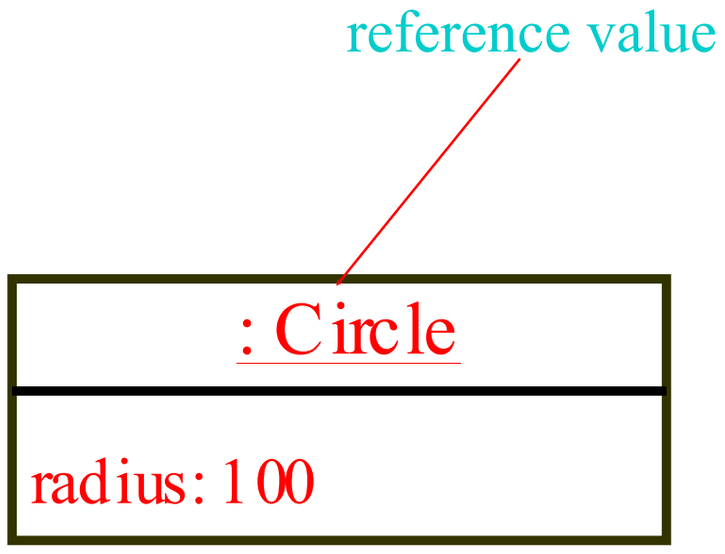
myCircle = Circle(5.0)

yourCircle = Circle()

yourCircle.radius = 100



Modify radius in yourCircle



Example: Defining Classes and Creating Objects

TV	
channel: int	The current channel (1 to 120) of this TV.
volumeLevel: int	The current volume level (1 to 7) of this TV.
on: bool	Indicates whether this TV is on/off.
TV()	Constructs a default TV object.
turnOn(): None	Turns on this TV.
turnOff(): None	Turns off this TV.
getChannel(): int	Returns the channel for this TV.
setChannel(channel: int): None	Sets a new channel for this TV.
getVolume(): int	Gets the volume level for this TV.
setVolume(volumeLevel: int): None	Sets a new volume level for this TV.
channelUp(): None	Increases the channel number by 1.
channelDown(): None	Decreases the channel number by 1.
volumeUp(): None	Increases the volume level by 1.
volumeDown(): None	Decreases the volume level by 1.

TV

TestTV

Run

Design Guide

If a class is designed for other programs to use, to prevent data from being tampered with and to make the class easy to maintain, define data fields private. If a class is only used internally by your own program, there is no need to encapsulate the data fields.

Why Encapsulate?

- By defining a specific interface you can keep other modules from doing anything incorrect to your data
- By limiting the functions you are going to support, you leave yourself free to change the internal data without messing up your users
  - Write to the Interface, not the the Implementation
  - Makes code more modular, since you can change large parts of your classes without affecting other parts of the program, so long as they only use your public functions

Classes that look like arrays

- Overload __getitem__(self,index) to make a class act like an array

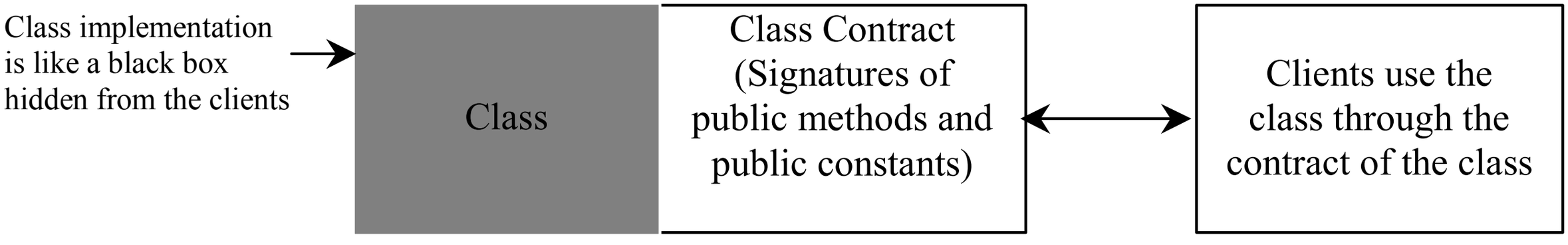
```
class molecule:
   def __getitem__(self,index):
      return self.atomlist[index]

>>> mol = molecule('Water') #defined as before
>>> for atom in mol:        #use like a list!
      print atom
>>> mol[0].translate(1.,1.,1.)
```

- An example of focusing on the interface!

Class Abstraction and Encapsulation

Class abstraction means to separate class implementation from the use of the class. The creator of the class provides a description of the class and let the user know how the class can be used. The user of the class does not need to know how the class is implemented. The detail of implementation is encapsulated and hidden from the user.



Object-Oriented Thinking

This book's approach is to teach problem solving and fundamental programming techniques before object-oriented programming.

This section will show how procedural and object-oriented programming differ.

You will see the benefits of object-oriented programming and learn to use it effectively.

We will use several examples in the rest of the chapter to illustrate the advantages of the object-oriented approach.

The examples involve designing new classes and using them in applications.

Procedural vs. Object-Oriented

In procedural programming, data and operations on the data are separate, and this methodology requires sending data to methods. Object-oriented programming places data and the operations that pertain to them in an object. This approach solves many of the problems inherent in procedural programming.

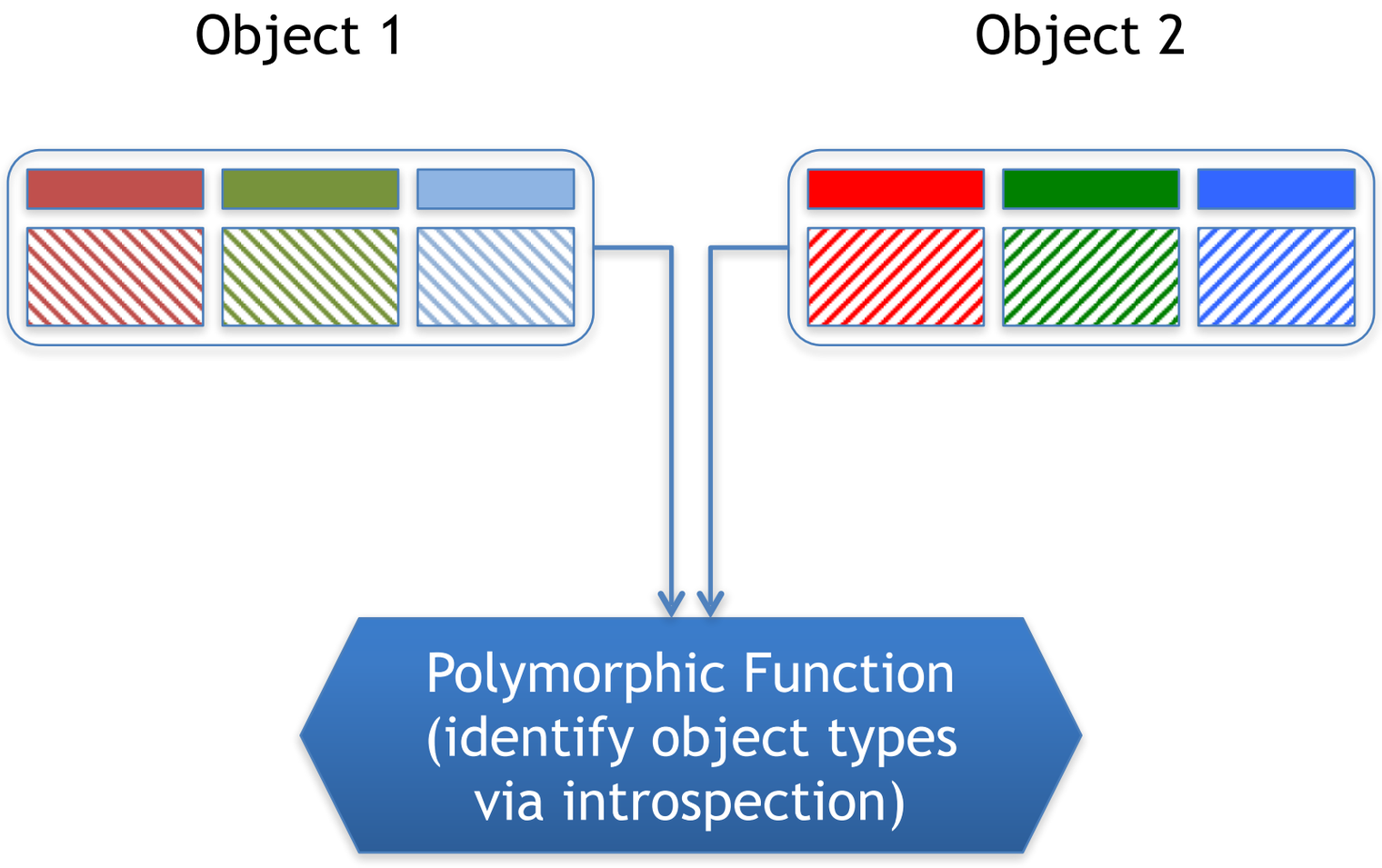
Procedural vs. Object-Oriented

The object-oriented programming approach organizes programs in a way that mirrors the real world, in which all objects are associated with both attributes and activities. Using objects improves software reusability and makes programs easier to develop and easier to maintain. Programming in Python involves thinking in terms of objects; a Python program can be viewed as a collection of cooperating objects.

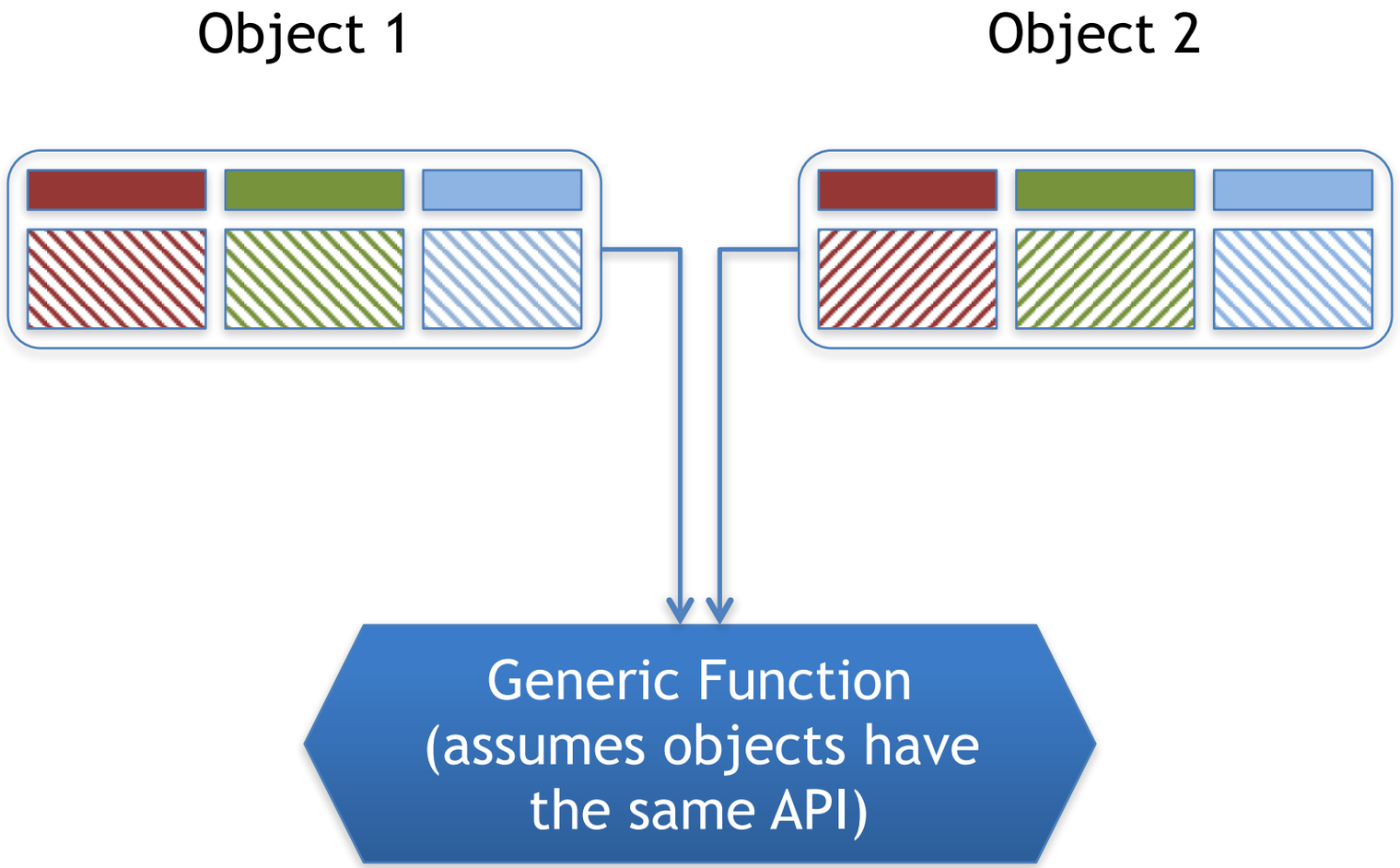
Polymorphism

- “Functions that can work with several types are called **polymorphic**.” – Downey, *Think Python*
- “The primary usage of **polymorphism** in industry (object-oriented programming theory) is the ability of objects belonging to different types to respond to method, field, or property calls of the same name, each one according to an appropriate type-specific behavior.
  - The programmer (and the program) does not have to know the exact type of the object in advance, and so the exact behavior is determined at run time (this is called **late binding** or **dynamic binding**).” - *Wikipedia*

Polymorphic Function



Polymorphic Classes



Polymorphism (cont.)

- The critical feature of polymorphism is a shared **interface**
  - Using the Downey definition, we present a common interface where the same function may be used regardless of the argument type
  - Using the Wikipedia definition, we require that polymorphic objects share a common interface that may be used to manipulate the objects regardless of type (class)

Polymorphism (cont.)

- Why is polymorphism useful?
  - By reusing the same interface for multiple purposes, polymorphism reduces the number of “things” we have to remember
  - It becomes possible to write a “generic” function that perform a particular task, eg sorting, for many different classes (instead of one function for each class)

Polymorphism (cont.)

- To define a polymorphic function that accepts multiple types of data requires the function either:
  - be able to distinguish among the different types that it should handle, or
  - be able to use other polymorphic functions, methods or syntax to manipulate any of the given types

Polymorphic Classes

- Classes that share a common interface
  - A function implemented using only the common interface will work with objects from any of the classes
- Although Python does not require it, a simple way to achieve this is to have the classes derive from a common superclass
  - To maintain polymorphism, methods overridden in the subclasses ***must*** keep the same arguments as the method in the superclass

Polymorphic Classes (cont.)

- The superclass defining the interface often has no implementation and is called an **abstract base class**
- Subclasses of the abstract base class override interface methods to provide class-specific behavior
- A generic function can manipulate all subclasses of the abstract base class

```
class InfiniteSeries(object):
     def next(self):
          raise
NotImplementedError(”next")
class Fibonacci(InfiniteSeries):
     def __init__(self):
          self.n1, self.n2 = 1, 1
     def next(self):
          n = self.n1
          self.n1, self.n2 = self.n2,
self.n1 + self.n2
          return n
class Geometric(InfiniteSeries):
     def __init__(self, divisor=2.0):
          self.n = 1.0 / divisor
          self.nt = self.n / divisor
          self.divisor = divisor
     def next(self):
          n = self.n
          self.n += self.nt
          self.nt /= self.divisor
          return n
def print_series(s, n=10):
     for i in range(n):
          print "%.4g" % s.next(),
     print
```

Polymorphic Classes (cont.)

- In our example, all three subclasses overrode the **next** method of the base class, so they each have different behavior
- If a subclass does ***not*** override a base class method, then it **inherits** the base class behavior
  - If the base class behavior is acceptable, the writer of the subclass does not need to do anything
  - There is only one copy of the code so, when a bug is found it the inherited method, only the base class needs to be fixed
- ***instance.method()*** is preferable over ***class.method(instance)***
  - Although the code still works, the explicit naming of a class in the statement suggests that the method is defined in the class when it might actually be inherited from a base class

Polymorphic Syntax

- Python uses the same syntax for a number of data types, so we can implement polymorphic functions for these data types if we use the right syntax

```
def histogram(s):
        d = dict()
    for c in s:
                d[c] = d.get(c,
0) + 1
    return d

print histogram("aabc")
print histogram([1, 2, 2, 5])
print histogram(("abc", "abc",
"xyz"))
```

method versus function

- discussed before, a method and a function are closely related. They are both “small programs” that have parameters, perform some operation and (potentially) return a value

- main difference is that methods are functions tied to a particular object

difference in definition

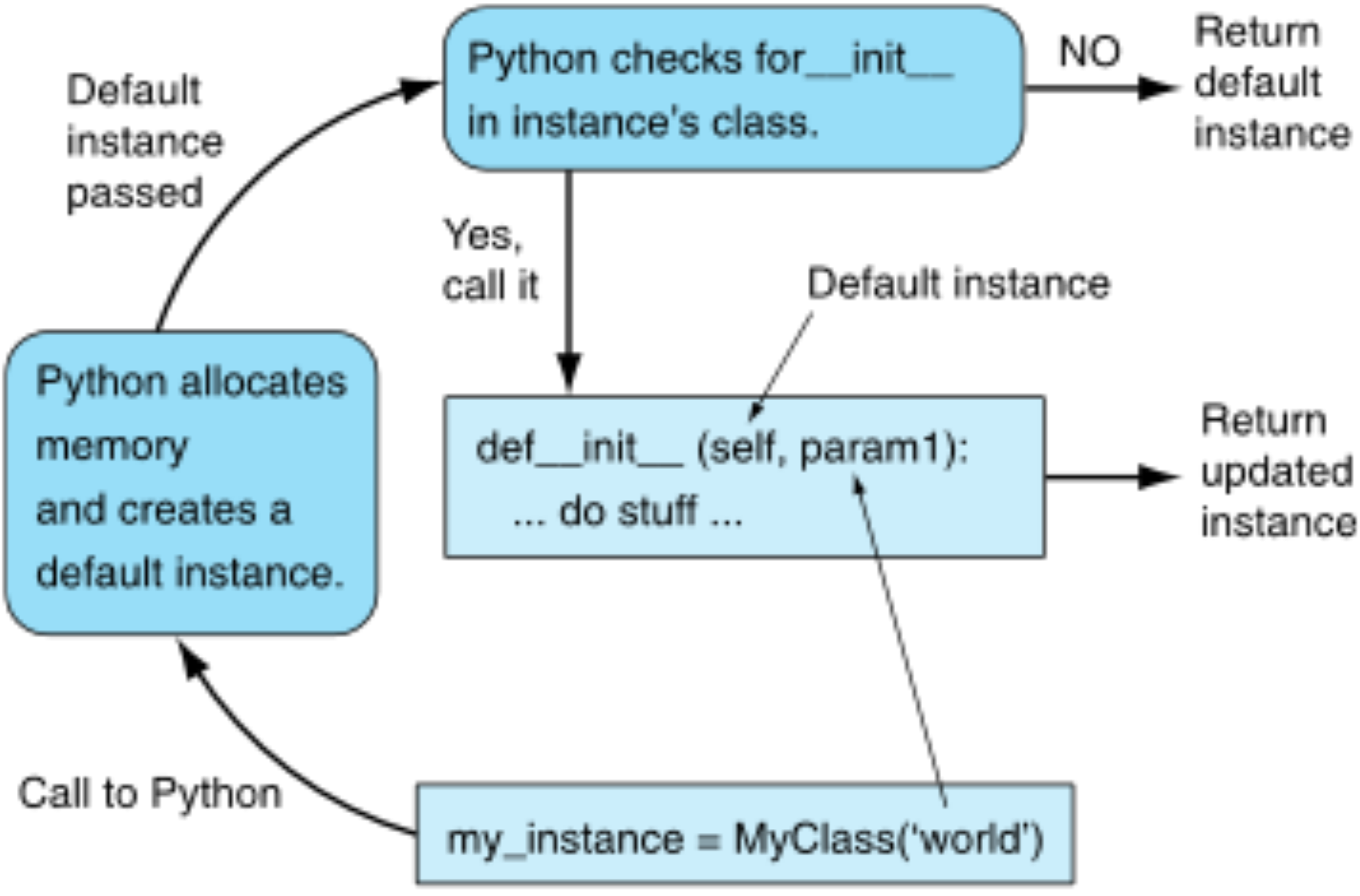
- methods are defined ***inside*** the suite of a class
- methods always bind the first parameter in the definition to the object that called it
- This parameter can be named anything, but traditionally it is named ***self***

```
class MyClass(object):
    def my_method(self,param1):
        suite
```

more on self

- `self` is an important variable. In any method it is bound to the object that called the method

- through `self` we can access the instance that called the method (and all of its attributes as a result)


Default instance passed
Python checks for__init__ in instance's class.
NO
Return default instance
Yes, call it
Default instance
Python allocates memory and creates a default instance.
def__init__ (self, param1):
... do stuff ...
Return updated instance
Call to Python
my_instance = MyClass('world')

Other things to overload

- __setitem__(self,index,value)
  - Another function for making a class look like an array/dictionary
  - a[index] = value

- __add__(self,other)
  - Overload the "+" operator
  - molecule = molecule + atom

- __mul__(self,number)
  - Overload the "*" operator
  - zeros = 3*[0]

- __getattr__(self,name)
  - Overload attribute calls
  - We could have done atom.symbol() this way

Other things to overload, cont.

- __del__(self)
  - Overload the default destructor
  - del temp_atom

- __len__(self)
  - Overload the len() command
  - natoms = len(mol)

- __getslice__(self,low,high)
  - Overload slicing
  - glycine = protein[0:9]

- __cmp__(self,other):
  - On comparisons (<, ==, etc.) returns -1, 0, or 1, like C's strcmp

Is More Complex Better?

- Advantages
  - Each class corresponds to a real concept
  - It should be possible to write a polymorphic function to play cards using only Game and Hand interfaces
  - It should be easier to implement other card games
- Disadvantages
  - More classes means more things to remember
  - Need multiple inheritance (although in this case it should not be an issue because the class hierarchy is simple)

Function

```
def see():
    def inner1():
        print("hello")
    def inner2():
        print("hewwwww")
    #y = int(input(""))
    d = math.sin(12)
    return d
```

- Fact :
  - Whenever you call `See()`, the inner functions `inner1()` and `inner2()` are also called. But because of their local scope, they aren't available outside of the `see()` function.

Showing all of them

- By adding 4 lines of code we can share 2 functions that we want directly.

```
if x==1:
        return inner1
    else:
        return d
```

So syntax

- First looking of Decorators:

```
import math
def see(x):
    def inner1():
        print("hello")
    def inner2():
        print("hewwww")
    #y = int(input(""))
def see1():
    print("wheel")

see1 = see(see1)
print(see1)
```

Wow

- So, `@my_decorator` is just an easier way of saying `say_whee = my_decorator(say_whee)`. It's how you apply a decorator to a function.

```
def function(attr):
    def inner():
        <body>

    return inner

@function
def dec_function():
    print("we catch it")
```