Object Oriented Programming

Haleh Ashki 2015, updated Peter Beerli 2017

Traditionally, a program has been seen as a recipe – a set of instructions that you follow from start to finish in order to complete a task. That approach is sometimes known as **procedural programming.**

Broadly, object oriented programming is the concept that, in programming, the **objects** that we're manipulating are more important than the logic needed to manipulate those objects.

Putting the **objects** at the **center** of the process, instead of simply using them as necessary containers for information as part of our procedural instructions

First, we define the **objects** we want to manipulate and how they relate to each other, and then we start to flesh it out with **logic** to make the program actually work.

The roots go back to 1960s. The first programming language to use objects was Simula 67.

There are those who glorify OOP and think that anything which is not programmed in an object oriented way can't be good. On the other hand there are well known computer scientists and specialists who criticize OOP.

OOP helps for software engineering (SE)

- (SE) is the discipline of managing code to ensure its long-term use
- remember SE via refactoring:
  - takes existing code and modifies it
  - makes the overall code simpler, easier to understand
  - doesn't change the functionality, only the form!

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.

An object oriented program is based on **classes** and there exists a collection of interacting objects, as opposed to the conventional model, in which a program consists of functions and routines. In OOP, each object can receive messages, process data, and send messages to other objects.

An OOP approach is to define the objects we’re going to be using. The way we do this is to first define the properties that it possesses using a class. You can think of a class as a sort of template; a guide for the way an object should be structured. Each object belongs to a class and inherits the properties of that class, but acts individually to the other objects of that class.

An object is sometimes referred to as an ‘instance’ of a class.

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.

Template vs exemplar

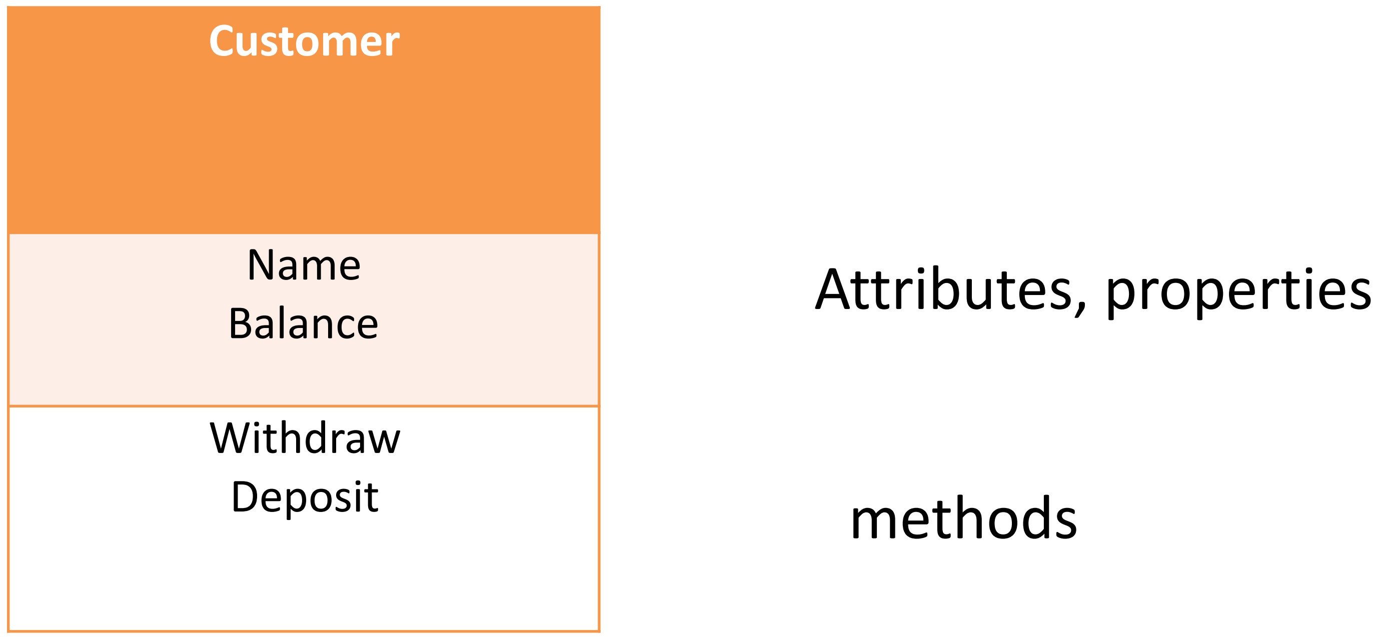
- The cutter is a template for stamping out cookies, the cookie is what is made each time the cutter is used
- One template can be used to make an infinite number of cookies, each one just like the other.
- No one confuses a cookie for a cookie cutter, do they?

Same in OOP

- You define a class as a way to generate new instances of that class.
- Both the instances and the classes are themselves objects
- The structure of an instance starts out the same, as dictated by the class.
- The instances respond to the messages defined as part of the class.

A class contains **variables** and **methods**. If you bake a cake you need ingredients and instructions to bake the cake.

There are **class variables**, which have the same value in all methods and their are **instance variables**, which have normally different values for different objects. A class also has to define all the necessary **methods**, which are needed to access the data.



A simple example:

you might have a **class** named ‘person’ with an age and a name property.

An **instance** of that class (an **object**) would be a single person. That person might have a name of “Andy” and an age of 23.

You could **simultaneously** have another person belonging to the same class with the name of “Lucy” and an age of 18.

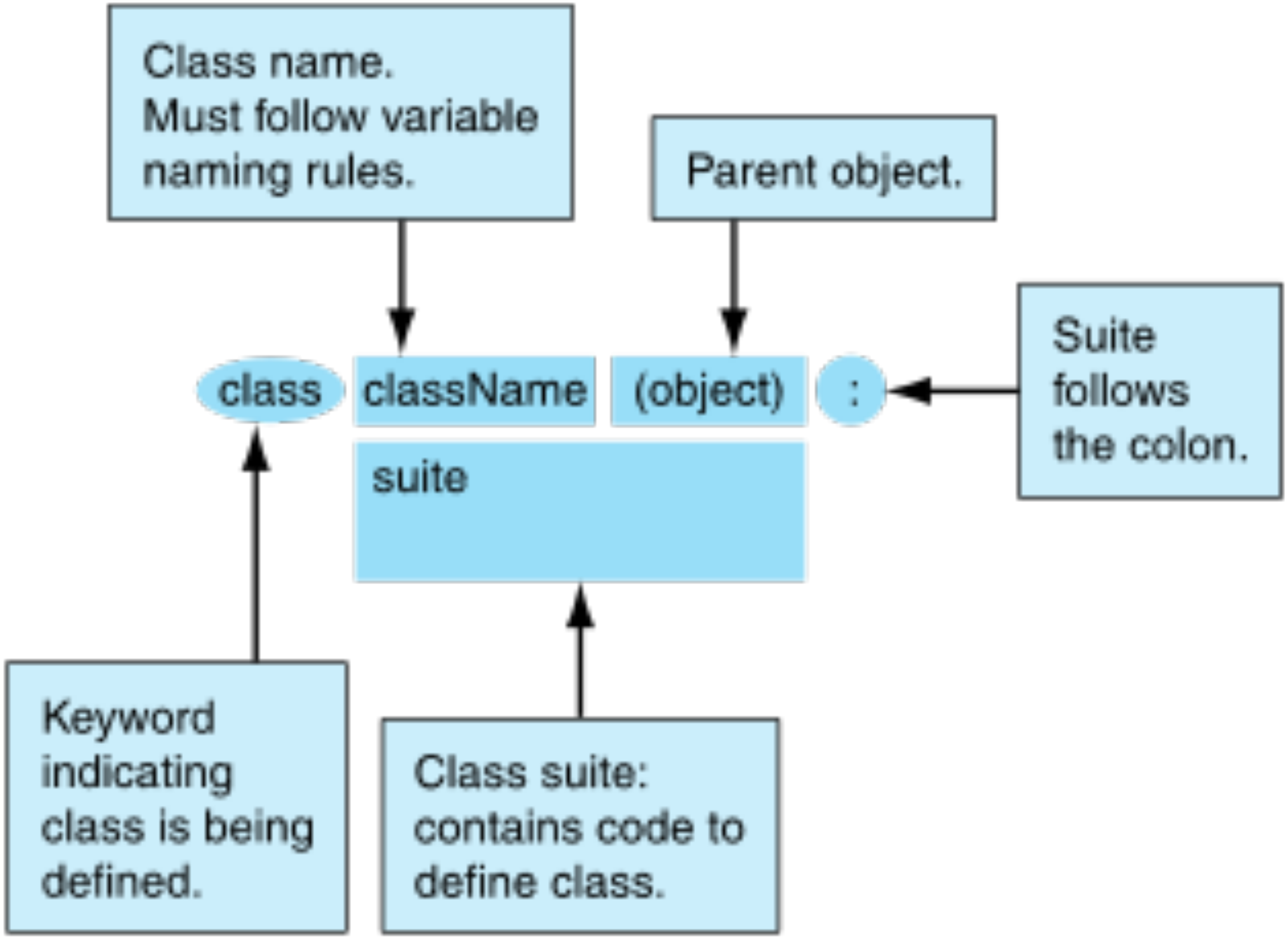

Class name.
Must follow variable naming rules.
Parent object.
Suite follows the colon.
class
className
(object)
:
suite
Keyword indicating class is being defined.
Class suite: contains code to define class.

FIGURE 11.2 The basic format of a class definition.

```
class pet:
    number_of_legs = 0

doug = pet()
```

Classes in python start with **keyword “class”**

Class pet is used to create pet object.
Doug is an object, known as instant, is the realized version of the pet class.

We can create as many pet objects as we’d like.
There is still, however, only one Pet class, regardless of how many instances of the class we create.

```python
class Account(object):
    def __init__(self, holder, number, balance,credit_line=1500):
        self.Holder = holder
        self.Number = number
        self.Balance = balance
        self.CreditLine = credit_line

    def deposit(self, amount):
        self.Balance = amount

    def withdraw(self, amount):
        if(self.Balance - amount < -self.CreditLine):
            # coverage insufficient
            return False
        else:
            self.Balance -= amount
            return True

    def balance(self):
        return self.Balance

    def transfer(self, target, amount):
    if(self.Balance - amount < -self.CreditLine):
            # coverage insufficient
            return False
        else:
            self.Balance -= amount
            target.Balance += amount
            return True
```

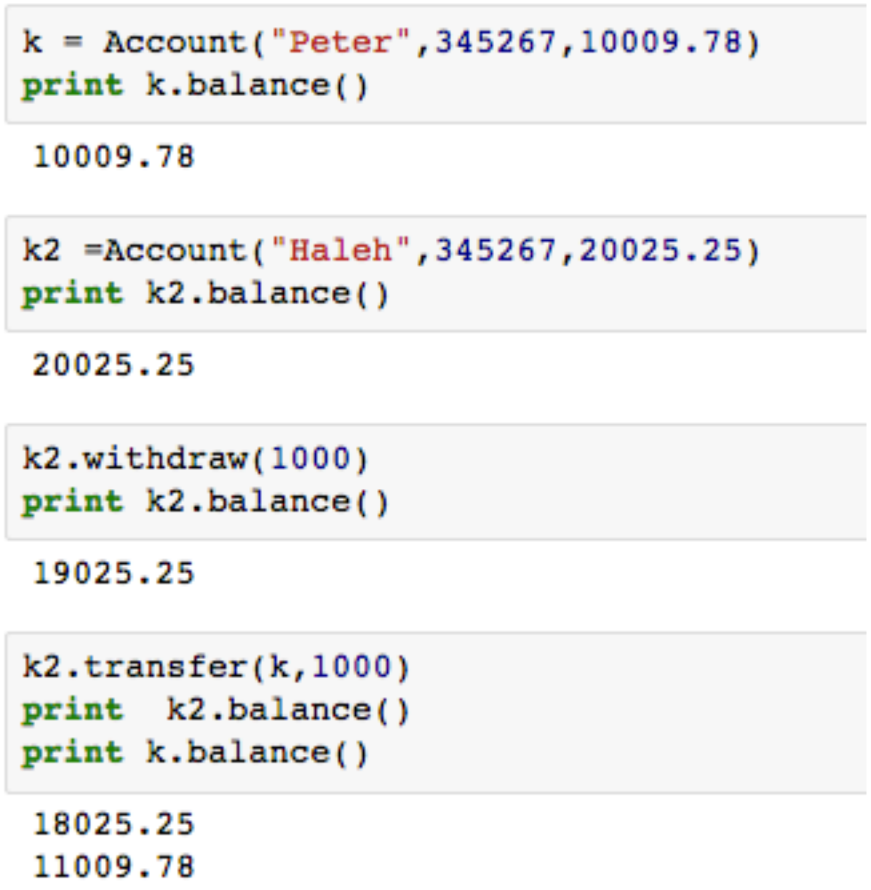
dot reference

- we can refer to the attributes of an object by doing a dot reference, of the form:
  `object.attribute`
- the attribute can be a variable or a function
- it is part of the object, either directly or by that object being part of a class

```
Example:

print(my_instance.my_val)
```
print a variable associated with the object `my_instance`
```
my_instance.my_method()
```
call a method associated with the object `my_instance`

variable versus method, you can tell by the parenthesis at the end of the reference



```
k = Account("Peter",345267,10009.78)
print k.balance()
```

```
10009.78
```

```
k2 =Account("Haleh",345267,20025.25)
print k2.balance()
```

```
20025.25
```

```
k2.withdraw(1000)
print k2.balance()
```

```
19025.25
```

```
k2.transfer(k,1000)
print  k2.balance()
print k.balance()
```

```
18025.25
11009.78
```

dir() function

The `dir()` function lists all the attributes of a class

- you can think of these as keys in a dictionary stored in the class.

```python
class Myclass(object):
    pass
```

```python
dir(Myclass)
```

```
['__class__',
 '__delattr__',
 '__dict__',
 '__doc__',
 '__format__',
 '__getattribute__',
 '__hash__',
 '__init__',
 '__module__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'class_attribute']
```

```python
inst=Myclass()
dir(inst)
```

```
['__class__',
 '__delattr__',
 '__dict__',
 '__doc__',
 '__format__',
 '__getattribute__',
 '__hash__',
 '__init__',
 '__module__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'class_attribute']
```

Instance knows its class

- Because each instance has as its type the class that it was made from, an instance remembers its class
- This is often called the ***instance-of*** relationship
- stored in the `__class__` attribute of the instance

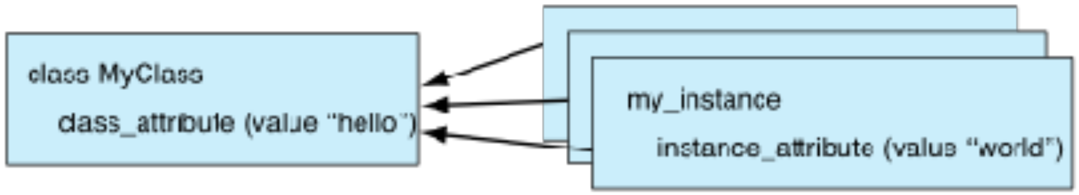

class MyClass
class_attribute (value "hello")
my_instance
instance_attribute (value "world")

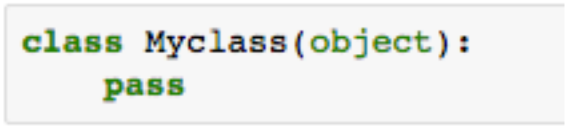
FIGURE 11.3 The instance-of relationship.

Pass Keyword:

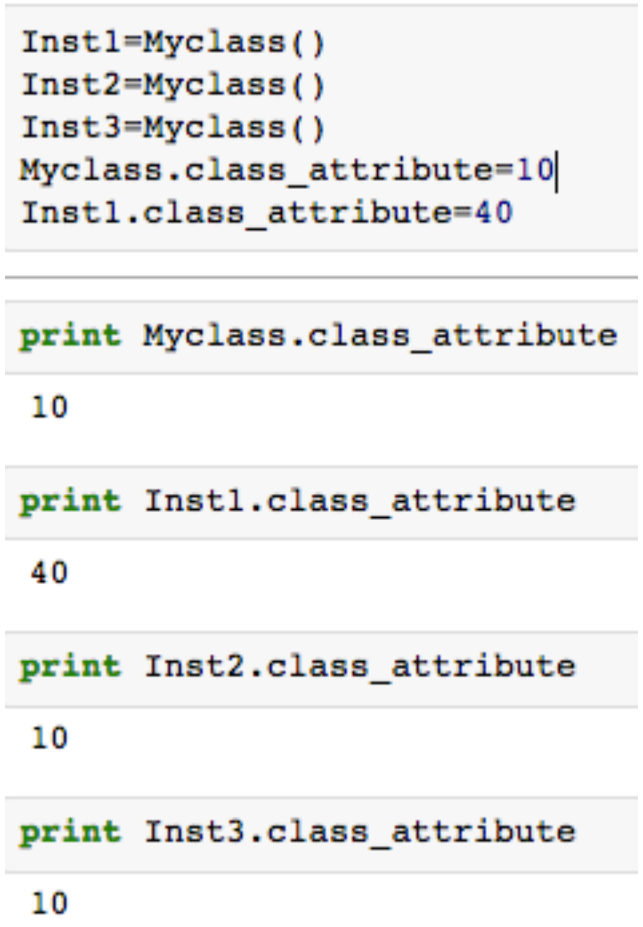
The `pass` keyword is used to signify that you have *intentionally* left some part of a definition (of a function, of a class) undefined.

by making the suite of a class undefined, we get only those things that Python defines for us automatically

Scope in Class and Instance



```
class Myclass(object):
    pass
```



```
Inst1=Myclass()
Inst2=Myclass()
Inst3=Myclass()
Myclass.class_attribute=10
Inst1.class_attribute=40
```

```
print Myclass.class_attribute
```

```
10
```

```
print Inst1.class_attribute
```

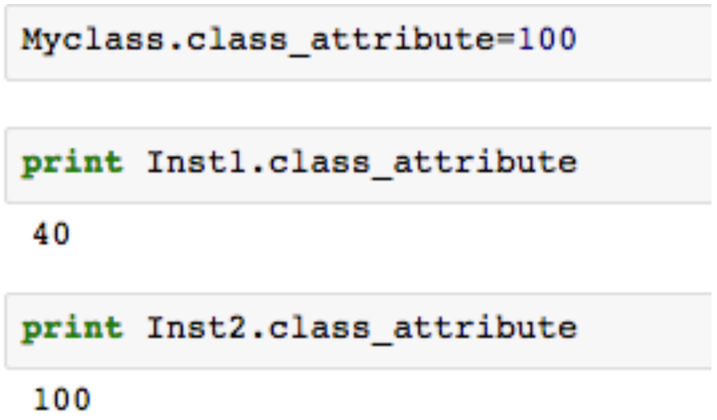
```
40
```

```
print Inst2.class_attribute
```

```
10
```

```
print Inst3.class_attribute
```

```
10
```



```
Myclass.class_attribute=100
```

```
print Inst1.class_attribute
```

```
40
```

```
print Inst2.class_attribute
```

```
100
```

Methods:

Essentially, a method is a function, but it's a special kind of function which belongs to a class.

It is defined within a class, and works on the instance and class data of this class.

It is defined in exactly the same way as function, but the difference is it is inside a class, and it belongs to that class.

Methods can only be called through instances of a class or a subclass, i.e. the class name followed by a dot and the method name.

method versus function

- 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 calling

functions are called, methods are called in the context of an object:

- function:

```
do_something(param1)
```

- method:

```
an_object.do_something(param1)
```

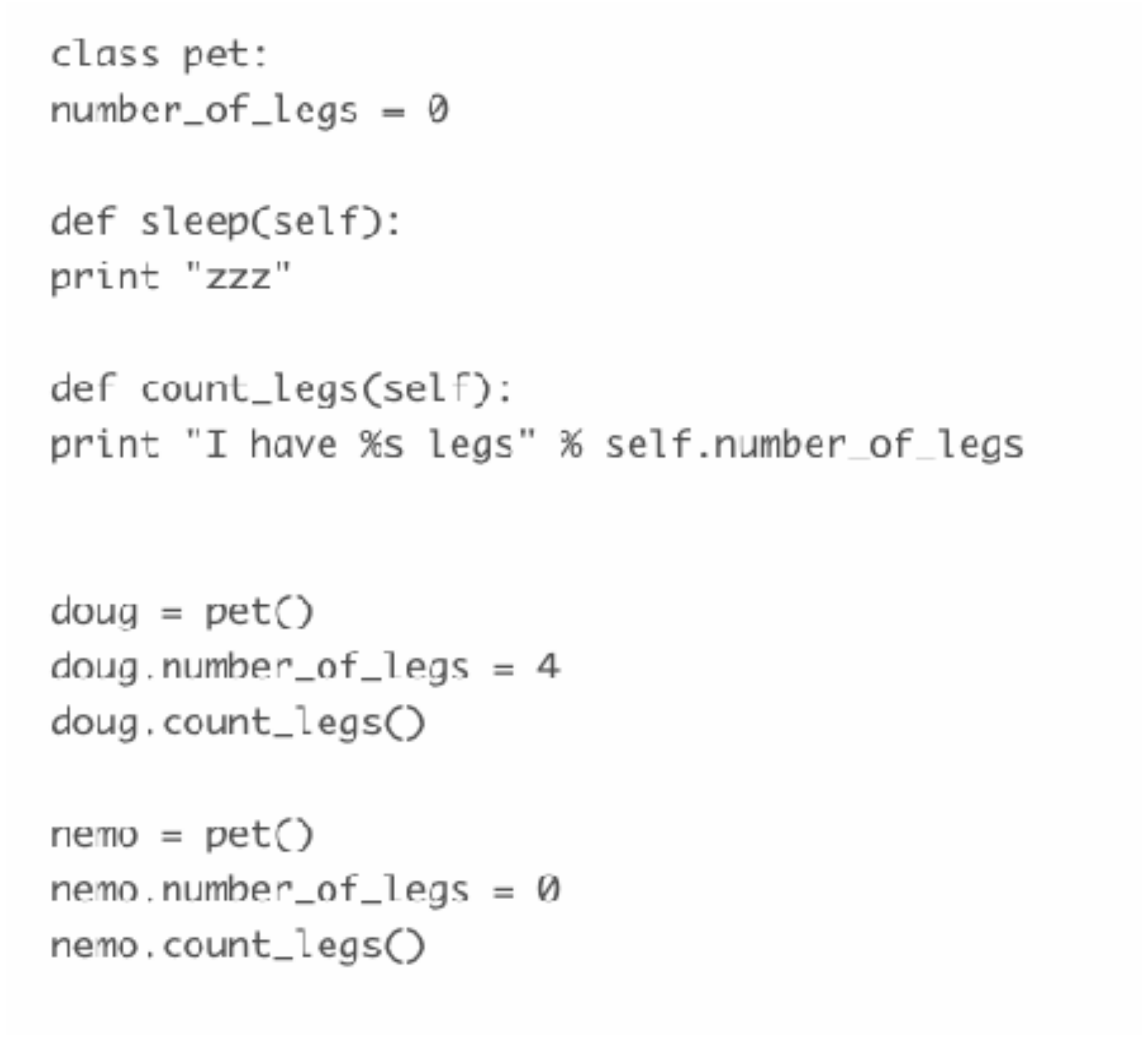
This means that the object that the method is called on is *always implicitly a parameter*!

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
```

Example of using classes and methods



```
class pet:
number_of_legs = 0

def sleep(self):
print "zzz"

def count_legs(self):
print "I have %s legs" % self.number_of_legs


doug = pet()
doug.number_of_legs = 4
doug.count_legs()

nemo = pet()
nemo.number_of_legs = 0
nemo.count_legs()
```

There is another difference between a method and a function:

a method always, always, always has to have an argument, called **‘self’** between the parentheses.

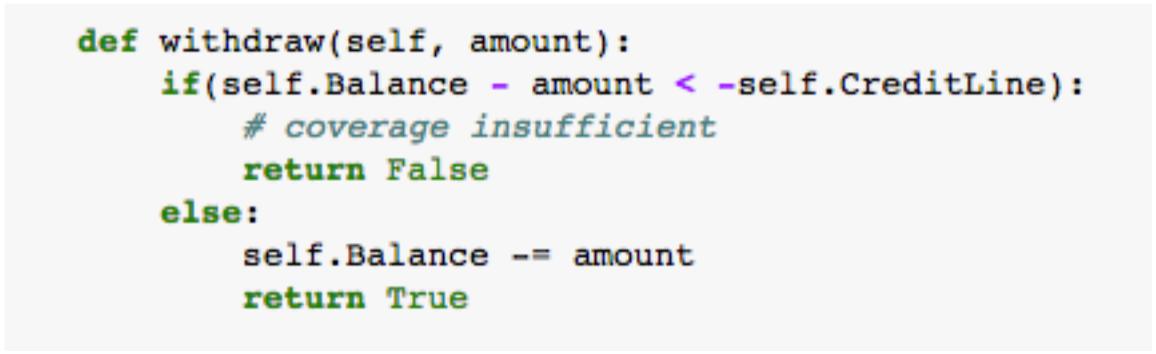
When Python calls a method, what it does is passes the current object to that method as the first argument. In other words, when we call doug.sleep(), Python is actually going to pass the object ‘doug’ as an argument to the sleep method.

Self

- Self is passing parameter to all of the Customer methods

- So when we say def withdraw(self, amount):, we're saying, "here's how you withdraw money from a Customer object (which we'll call self) and a dollar figure (which we'll call amount).

- self is the *instance* of the Customer that withdraw is being called on.



```
def withdraw(self, amount):
    if(self.Balance - amount < -self.CreditLine):
        # coverage insufficient
        return False
    else:
        self.Balance -= amount
        return True
```

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)

Binding self

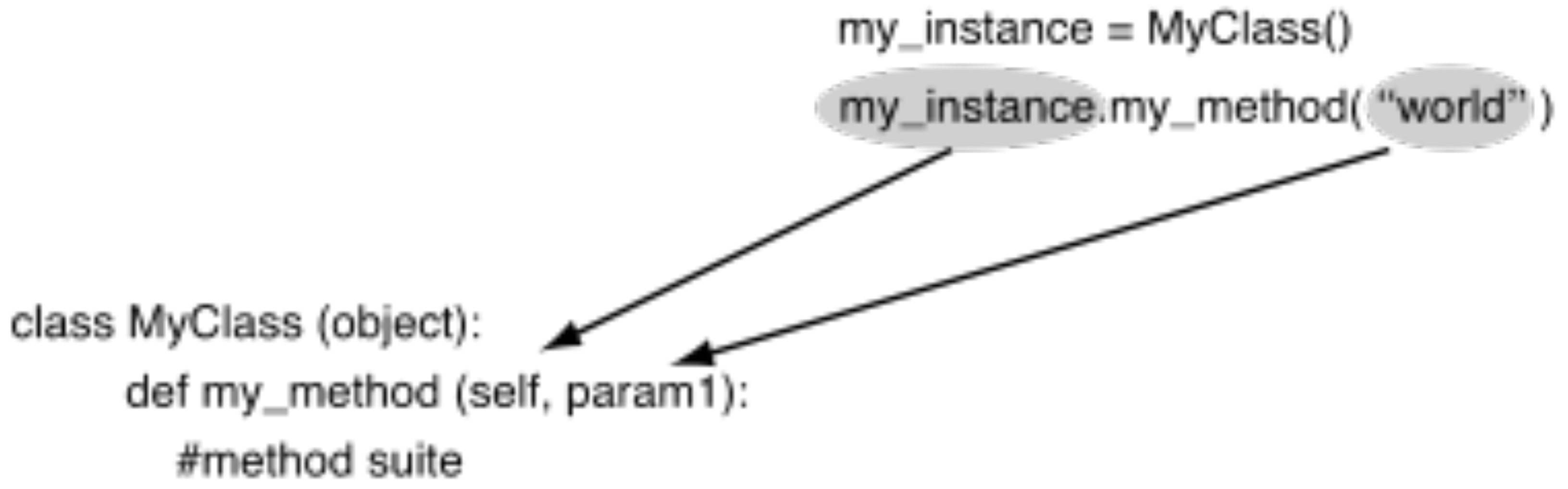

my_instance = MyClass()
my_instance.my_method("world")
class MyClass (object):
def my_method (self, param1):
#method suite

FIGURE 11.5 How the calling object maps to **self**.

Python Standard Methods

Python provides a number of standard methods which, if the class designer provides, can be used in a normal "Python" way

- many of these have the double underlines in front and in back of their name
- by using these methods, we "fit in" to the normal Python flow

```
class Student(object):
    def __init__(self, first='', last='', id=0):
        # print 'In the __init__ method'
        self.first_name_str = first
        self.last_name_str = last
        self.id_int = id

    def update(self, first='', last='', id=0):
        if first:
            self.first_name_str = first
        if last:
            self.last_name_str = last
        if id:
            self.id_int = id

    def __str__(self):
        # print "In __str__ method"
        return "{} {}, ID:{}".\
            format(self.first_name_str, self.last_name_str, self.id_int)
```

Standard Method: Constructor

- Constructor is called when an instance is made, and provides the class designer the opportunity to set up the instance with variables, by assignment

calling a constructor

As mentioned, a constructor is called by using the name of the class as a function call (by adding () after the class name)

```
student_inst = student()
```

- creates a new instance using the constructor from class `Student`

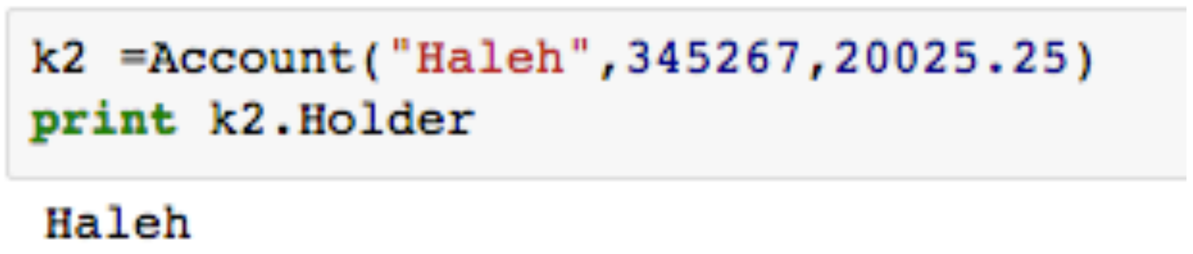
defining the constructor

- one of the special method names in a class is the constructor name, `__init__`
- by assigning values in the constructor, every instance will start out with the same variables
- you can also pass arguments to a constructor through its init method

Account constructor

```
def __init__(self, holder, number,
  balance,credit_line=1500):
        self.Holder = holder
        self.Number = number
        self.Balance = balance
        self.CreditLine = credit_line
```

- `self` is bound to the default instance as it is being made
- If we want to add an attribute to that instance, we modify the attribute associated with self.



```
k2 =Account("Haleh",345267,20025.25)
print k2.Holder
```

```
Haleh
```

```python
class Customer(object):
    """A customer of ABC Bank with a checking account. Customers have the
    following properties:

    Attributes:
        name: A string representing the customer's name.
        balance: A float tracking the current balance of the customer's account.
    """

    def __init__(self, name, balance=0.0):
        """Return a Customer object whose name is *name* and starting
        balance is *balance*."""
        self.name = name
        self.balance = balance

    def withdraw(self, amount):
        """Return the balance remaining after withdrawing *amount*
        dollars."""
        if amount > self.balance:
            raise RuntimeError('Amount greater than available balance.')
        self.balance -= amount
        return self.balance
```

- This is why when we call __init__, we *initialize* objects by saying things like self.name = name.

- Remember, since self *is* the instance, this is equivalent to saying , self.balance = balance is the same as jeff.balance = 1000.0.

- After these two lines, we consider the Customer object "initialized" and ready for use.

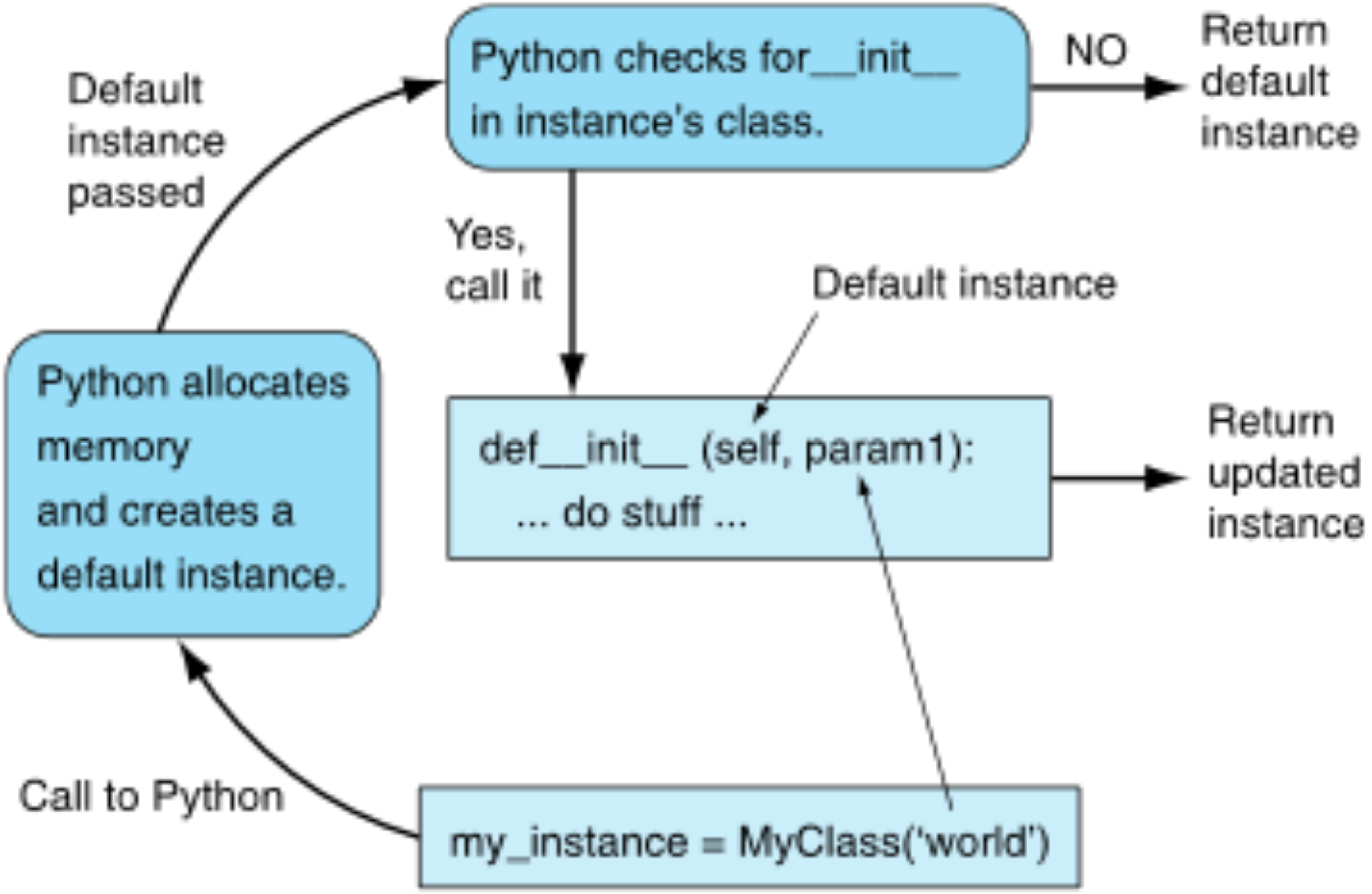

Python checks for__init__ in instance's class.
NO
Return default instance
Default instance passed
Yes, call it
Default instance
Python allocates memory and creates a default instance.
def__init__ (self, param1):
... do stuff ...
Return updated instance
Call to Python
my_instance = MyClass('world')

FIGURE 11.6 How an instance is made in Python.