# Chapter 6

## Object Oriented Programming & Console Interaction


CLASSES
INSTANCES
ATTRIBUTES

# Object-Oriented Programming

- **Object:** It as a reflection of the **real world**. Objects can have specific properties (variables) and behaviors (methods).

- **Class:** It is the template or blueprint we use to **create objects**. The class defines the properties that objects will have and the functions they can perform.

- **Method:** These are the functions defined within a class. These functions represent **the actions or behaviors** that an object can perform. For example, in a ‘Car’ class, there can be methods like ‘drive’ or ‘stop’.

- **Instance:** A **specific object created** from a **class**. In other words, if a class is a general template, an instance is the materialized form of this template. Like in the example of ‘Toyota Corolla’, it is an instance of the ‘Car’ class.

# Example:

- The **Bicycle class** has an __init__ method to initialize properties like color and bike_type.
- It has **two methods**, pedal() and brake(), to simulate the actions of pedaling and braking.
- In the __main__ block, an **instance of Bicycle** is created, properties are set during initialization, and methods are called.

```
class Bicycle:
    # Constructor to initialize properties
    def __init__(self, color, bike_type):
        self.color = color
        self.bike_type = bike_type

    # Method to simulate pedaling
    def pedal(self):
        print(f"The {self.color} {self.bike_type} is pedaling!")

    # Method to simulate braking
    def brake(self):
        print(f"The {self.color} {self.bike_type} has stopped.")

# Main code to create an instance and use its methods
if __name__ == "__main__":
    # Instance creation
    my_bike = Bicycle("Blue", "Mountain Bike")

    # Using object methods
    my_bike.pedal()
    my_bike.brake()
```

# Classes in Python

```python
class Student(object):
    """Simple Student class."""
    def __init__(self,first='', last='', id=0): # initializer
        self.first_name_str = first
        self.last_name_str = last
        self.id_int = id

    def __str__(self):  # string representation, e.g. for printing
        return "{} {}, ID:{}".format\
            (self.first_name_str, self.last_name_str, self.id_int)
```

# What is a class?

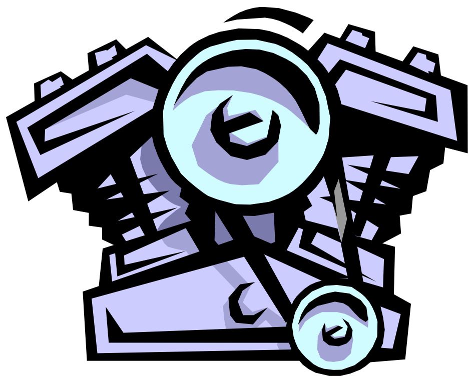
- If you have done anything in computer science before, you likely will have heard the term **object oriented programming (OOP)**
- What is OOP, and why should I care?

# Short answer

- The short answer is that object-oriented programming is a way to think about "objects" in a program (such as **variables, functions**, etc.)
- A program becomes less a list of instruction and more a set of objects and how they interact.

# Responding to “messages”

- As a set of interacting objects, each object responds to “messages” sent to it
- The interaction of objects via messages makes a high-level description of what the program is doing.



# Everything in Python is an object

- In case you hadn't noticed, everything in Python is an object
- Thus, Python embraces OOP at a fundamental level

# type vs class

There is a strong similarity between a type and a Python class

- seen many types already: `list, dict, str`, ...
- suitable for representing different data
- respond to different messages regarding the manipulation of that data

# OOP helps for software engineering

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

# More refactoring

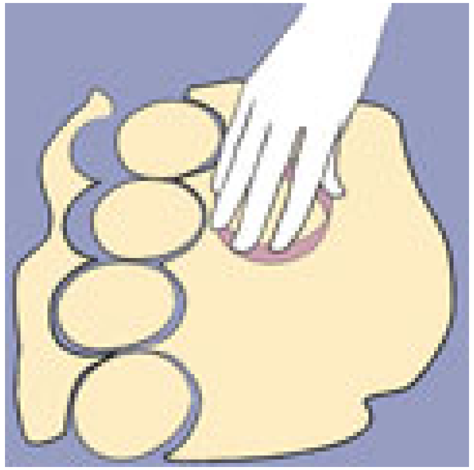
- Hiding the details of what the message entails means that changes can be made to the object and the flow of messages (and their results) can stay the same
- Thus the implementation of the message can change but its intended effect stay the same.
- This is ***encapsulation***

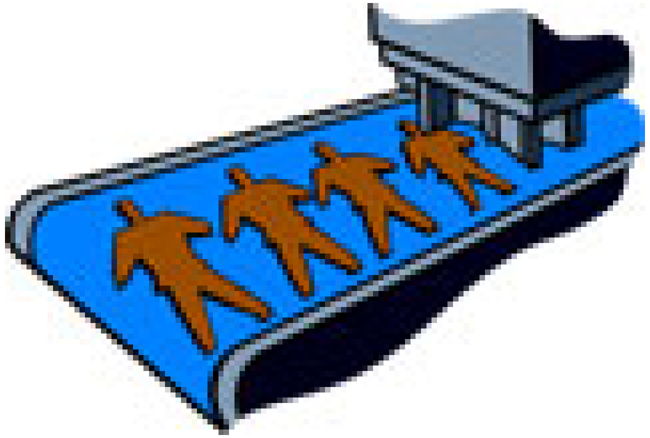
# OOP principles

- ***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 vs 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.

# Why a class

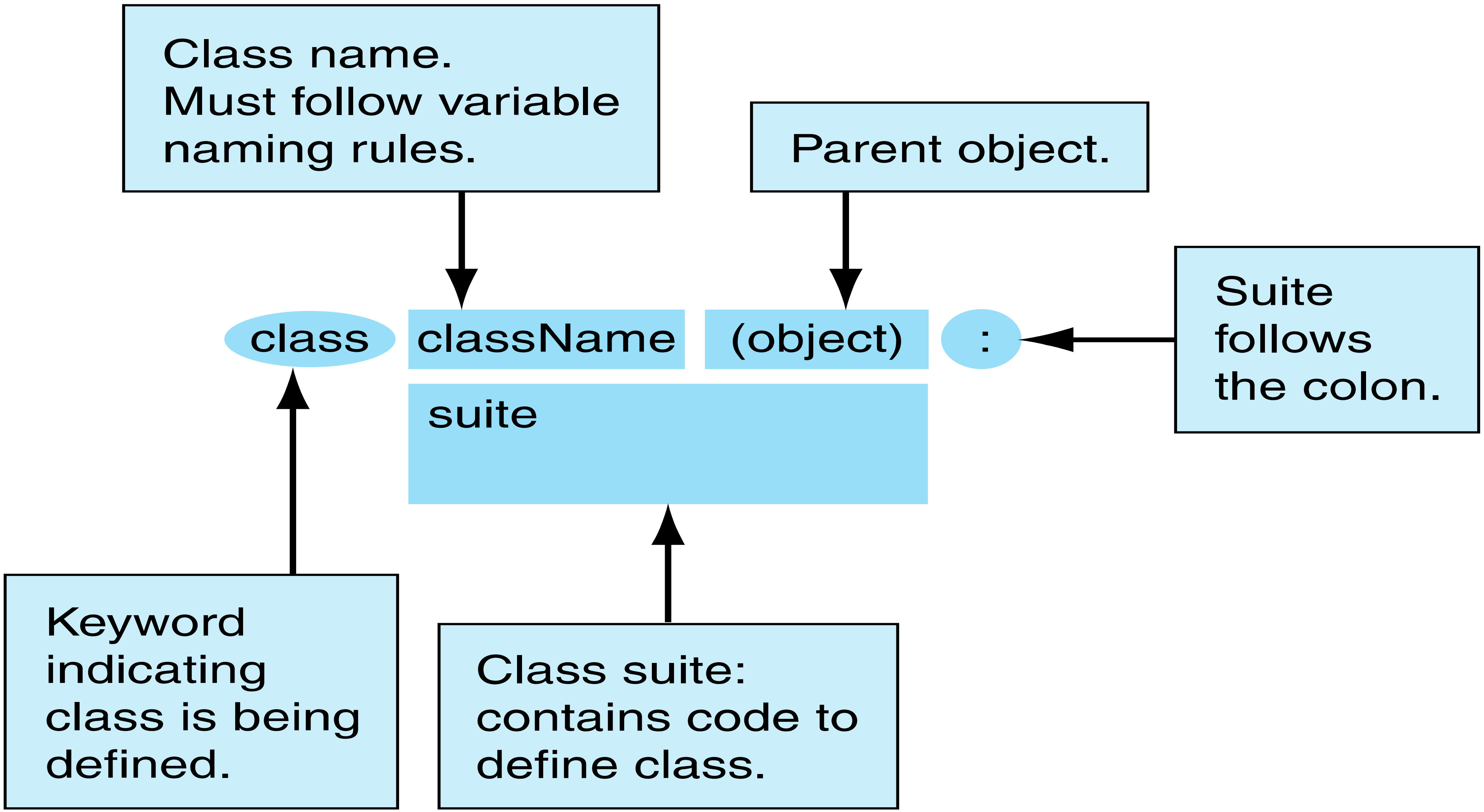
- We make classes because we need more complicated, user-defined data types to construct instances we can use.
- Each class has potentially two aspects:
  - the data (types, number, names) that each instance might contain
  - the messages that each instance can respond to.

# A First Class

# 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




The basic format of a class definition.

# 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.

# pass keyword

Remember, 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

```
>>> class MyClass (object):
        pass

>>> dir(MyClass)
'__class__', '__delattr__', '__dict__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
'__init__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__']

>>> my_instance = MyClass()
>>> dir(my_instance)
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
'__init__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__']

>>> type(my_instance)
<class '__main__.MyClass'>
```

# 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

# examples

`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

# How to make an object-local value

- once an object is made, the data is made the same way as in any other Python situation, by assignment
- Any object can thus be augmented by adding a variable

```
my_instance.attribute = 'hello'
```

# New attribute shown in dir

```
dir(my_instance)
```

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

# Class instance relationship

# 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 (object):
        pass

>>> my_instance = MyClass()
>>> MyClass.class_attribute = 'hello'
>>> my_instance.instance_attribute = 'world'
>>> dir(my_instance)
['__class__', ... , 'class_attribute', 'instance_attribute']
>>> print(my_instance.__class__)
<class '__main__.MyClass'>
>>> type(my_instance)
<class '__main__.MyClass'>
>>> print(my_instance.instance_attribute)
world
>>> print(my_instance.class_attribute)
hello
>>> print(MyClass.instance_attribute)

Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    print MyClass.instance_attribute
AttributeError: type object 'MyClass' has
no attribute 'instance_attribute'
```

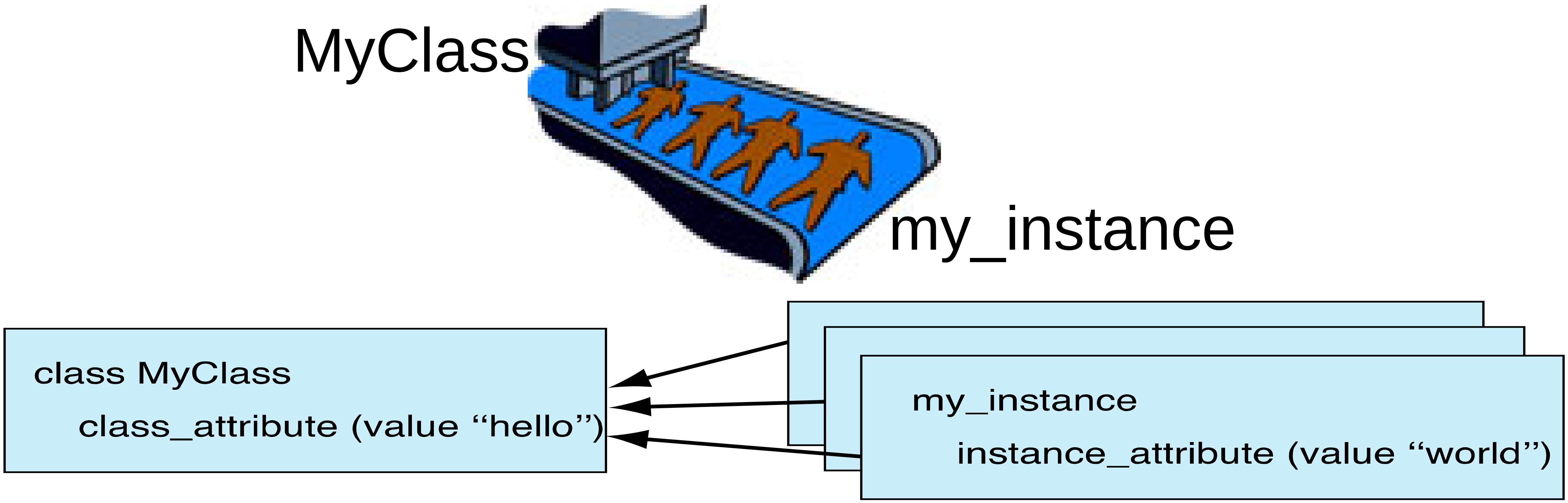



**FIGURE 11.3** The instance-of relationship.

# Scope

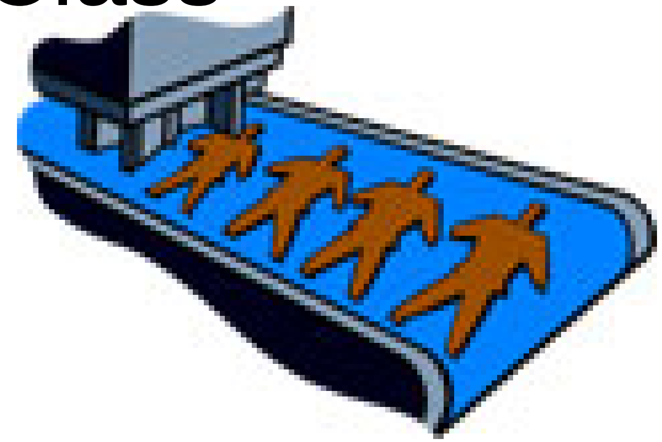
- Introduced the idea of scope in Chapter 7
- It works differently in the class system, taking advantage of the ***instance-of*** relationship

# Part of the Object Scope Rule

The first two rules in object scope are:

1. First, look in the object itself
2. If the attribute is not found, look up to the class of the object and search for the attribute there.

MyClass



inst1
inst2
inst3

```
>>> class MyClass (object):
        pass

>>> inst1 = MyClass()
>>> inst2 = MyClass()
>>> inst3 = MyClass()
>>> MyClass.class_attribute = 27
>>> inst1.class_attribute = 72
>>> print(inst1.class_attribute)
72
>>> print(inst2.class_attribute)
27
>>> print(inst3.class_attribute)
27
>>> MyClass.class_attribute = 999
>>> print(inst1.class_attribute)
72
>>> print(inst2.class_attribute)
999
>>> print(inst3.class_attribute)
999
```

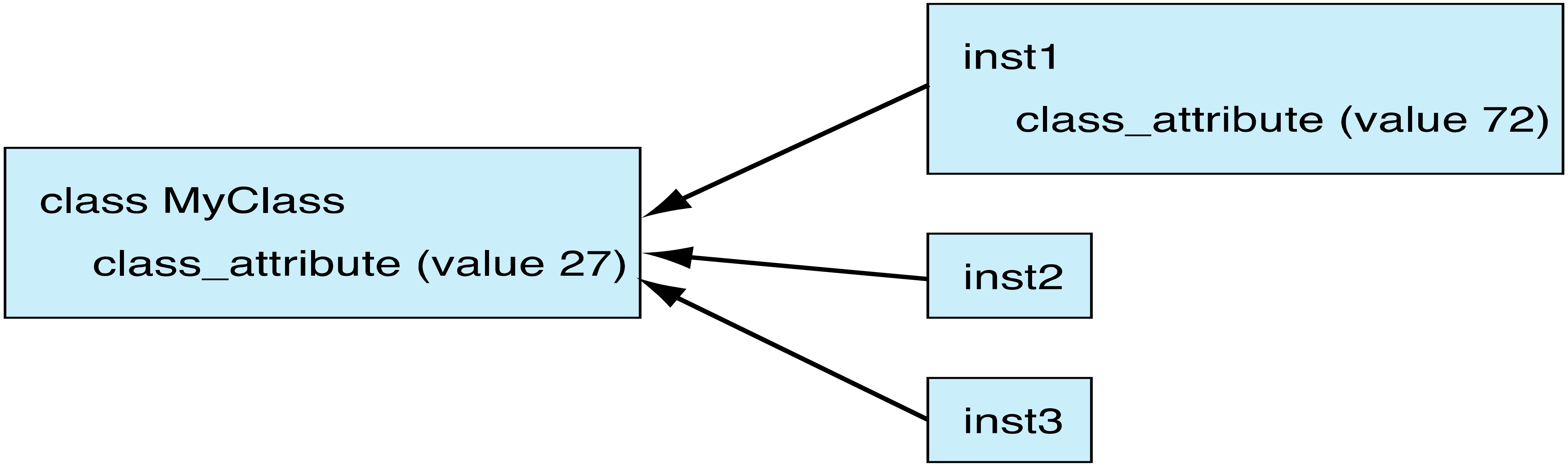



**FIGURE 11.4** A mixture of local and instance-of attribute relationships.

# Methods

```python
class MyClass (object):
    class_attribute = 'world'

    def my_method (self, param1):
        print('\nhello {}'.format(param1))
        print('The object that called this method is: {}'.\
              format(str(self)))
        self.instance_attribute = param1


my_instance = MyClass()
print("output of dir(my_instance):")
print(dir(my_instance))
my_instance.my_method('world')              # adds the instance_attribute
print("Instance has new attribute with value: {}".\
      format(my_instance.instance_attribute))
print("output of dir(my_instance):")
print(dir(my_instance))
```

# 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 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
```

# 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)

# Back to the example

```
class MyClass (object):
    class_attribute = 'world'

    def my_method (self, param1):
        print('\nhello {}'.format(param1))
        print('The object that called this method is: {}'.\
              format(str(self)))
        self.instance_attribute = param1
```

# 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`**.

# self is bound for us

- when a dot method call is made, the object that called the method is **automatically** assigned to `self`
- we can use `self` to remember, and therefore refer, to the calling object
- to reference any part of the calling object, we must always precede it with `self`.
- The method can be written generically, dealing with calling objects through `self`

# Writing a class

```python
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)
```

# Python Standard Methods

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

# 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

# Student constructor

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

- 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.

# example

```
s1 = Student()
print(s1.last_name_str)

s2 = Student(last='Python', first='Monty')
print(s2.first_name_str, s2.last_name_str)

Python
```

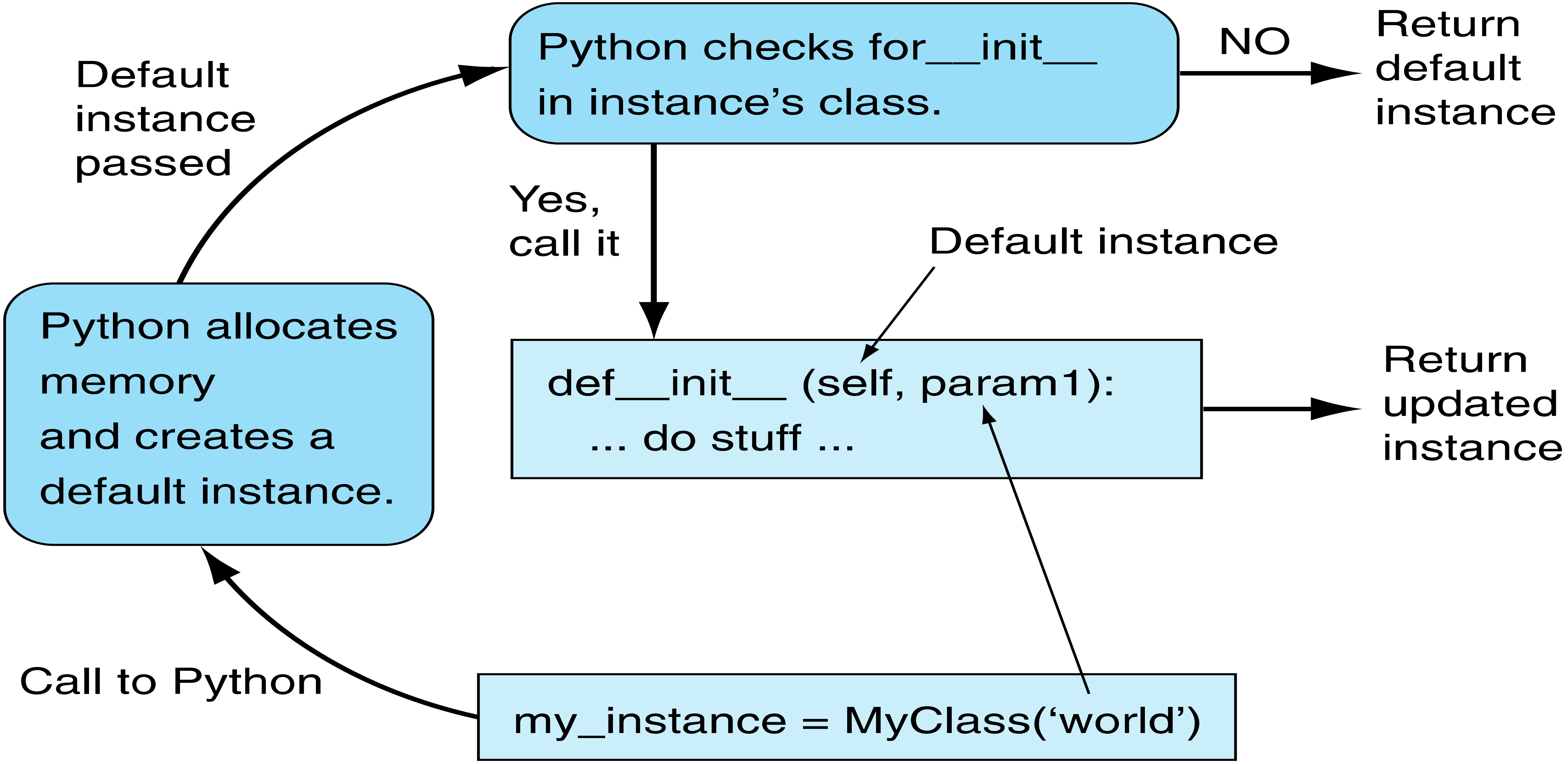



**FIGURE 11.6** How an instance is made in Python.

# default constructor

- if you don't provide a constructor, then only the default constructor is provided
- the default constructor does system stuff to create the instance, nothing more
- you cannot pass arguments to the default constructor.

# 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.

# __str__, printing

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

- When `print(my_inst)` called, it is assumed, by Python, to be a call to “convert the instance to a string”, which is the __str__ method
- In the method, `my_inst` is bound to `self`, and printing then occurs using that instance.
- __str__ ***must return a string***!

# Now there are three

There are now three groups in our coding scheme:

- user
- programmer, class user
- programmer, class designer

# Class designer

- The class designer is creating code to be used by other programmers
- In so doing, the class designer is making a kind of library that other programmers can take advantage of

```python
import math  # need sqrt (square root)

# a Point is a Cartesion point (x,y)
# all values are float unless otherwise stated
class Point(object):
    def __init__(self, x_param = 0.0, y_param = 0.0):
        '''Create x and y attributes. Defaults are 0.0'''
        self.x = x_param
        self.y = y_param

    def distance (self,param_pt):
        """Distance between self and a Point"""
        x_diff = self.x - param_pt.x  # (x1 — x2)
        y_diff = self.y - param_pt.y  # (y1 — y2)
        # square differences, sum, and take sqrt
        return math.sqrt(x_diff**2 + y_diff**2)


    def sum (self,param_pt):
        """Vector Sum of self and a Point
           return a Point instance"""
        # new_pt = Point()
        # new_pt.x = self.x + param_pt.x
        # new_pt.y = self.x + param_pt.x
        return Point(self.x + param_pt.x, self.x + param_pt.x)

    def __str__(self):
        """Print as a coordinate pair."""
        # print("called the __str__ method")
        return "({:.2f}, {:.2f})".format(self.x,self.y)
```

# Rule 9

Make sure your new class does the right thing

- we mean that a class should behave in a way familiar to a Python programmer
  - for example, we should be able to call the `print` function on it

# OOP helps software engineering

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

# More refactoring

- Hiding the details of what the message entails means that changes can be made to the object and the flow of messages (and their results) can stay the same
- Thus the implementation of the message can change but its intended effect stay the same.
- This is encapsulation

# OOP principles (again)

- ***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.

# We are still at encapsulation

- We said that encapsulation:
- hid details of the implementation so that the program was easier to read and write
- modularity, make an object so that it can be reused in other contexts
- providing an interface (the methods) that are the approved way to deal with the class

# Private values

# class namespaces are dicts

- the namespaces in every object and module is indeed a dictionary
- that dictionary is bound to the special variable `__dict__`
- it lists all the local attributes (variables, functions) in the object

# private variables in an instance

- many OOP approaches allow you to make a variable or function in an instance ***private***
- private means not accessible by the class user, only the class developer.
- there are advantages to controlling who can access the instance values

# privacy in Python

- Python takes the approach "We are all adults here". No hard restrictions.
- Provides naming to avoid accidents. Use __ (double underlines) in front of any variable
- this ***mangles*** the name to include the class, namely __var becomes _class__var
- still fully accessible, and the __dict__ makes it obvious

# privacy example

```
class NewClass (object):
    def __init__(self, attribute='default', name='Instance'):
        self.name = name                    # public attribute
        self.__attribute = attribute        # a "private" attribute
    def __str__(self):
        return '{} has attribute {}'.format(self.name, self.__attribute)
```

```
>>> inst1 = NewClass(name='Monty', attribute='Python')
>>> print(inst1)
Monty has attribute Python
>>> print(inst1.name)
Monty
>>> print(inst1.__attribute)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    print(inst1.__attribute)
AttributeError: 'newClass' object has no attribute '__attribute'
>>> dir(inst1)
'_NewClass__attribute', '__class__', ... , 'name']

>>> print(inst1._NewClass__attribute)
Python
```

# Reminder, rules so far

1. Think before you program!
2. A program is a human-readable essay on problem solving that also happens to execute on a computer.
3. The best way to improve your programming and problem-solving skills is to practice!
4. A foolish consistency is the hobgoblin of little minds
5. Test your code, often and thoroughly
6. If it was hard to write, it is probably hard to read. Add a comment.
7. All input is evil, unless proven otherwise.
8. A function should do one thing.
9. Use the right data structure for the job
10. Make sure your class does the right thing.