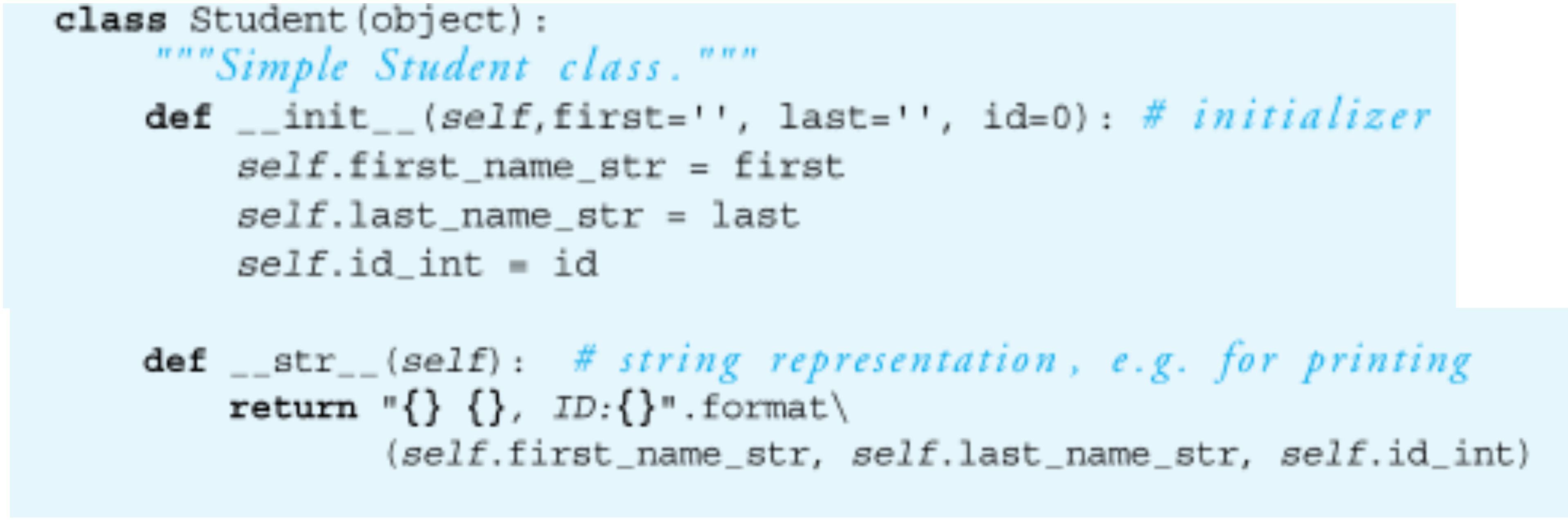
Object-Oriented Programming



```
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)
```

```
>>>stu1 = Student(‘Terry’, ‘Jones’, 12345)
>>>print(stu1)
Terry Jones, ID:12345
```

What is a class?

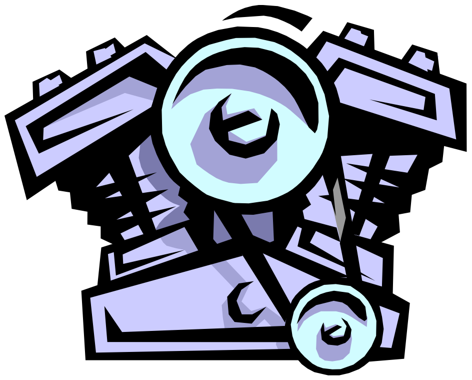
- If you have learned other programming languages before, you likely will have heard the term object oriented programming (OOP)
- What is OOP, and why should I care?

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)
- <u>A program </u>becomes less a list of instruction and more <u>a set of objects </u>and <u>how they interact</u>
- These objects have two characteristics:
  - Each object has some numbers of *attributes* (e.g., make, color) that are stored with the object.
  - The object responds to some *methods*, which are also attributes, that are particular for that kind of object (e.g., move_forward, print)

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 had not 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
- 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!

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 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.

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

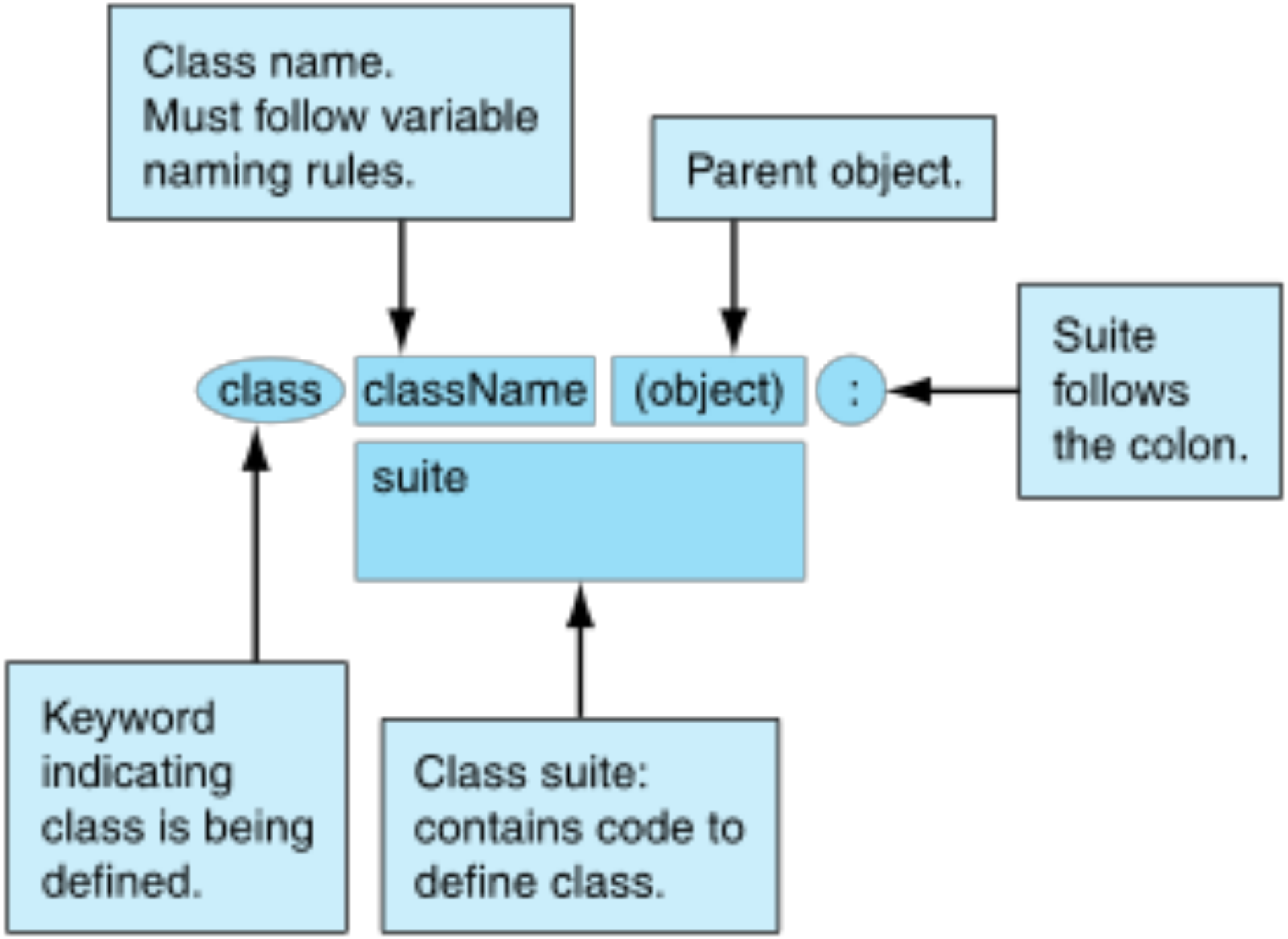

Class name.
Must follow variable naming rules.
Parent object.
class
className
(object)
:
Suite follows the colon.
suite
Keyword indicating class is being defined.
Class suite: contains code to define class.

FIGURE 11.2 The basic format of a class definition.

```
>>> class 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'>
```

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

Constructor

- When a class is defined, a function is made *with the same name as the class*
- This function is called the *constructor*. By calling it, you can create an instance of the class
- We can affect this creation (more later), but by default Python can make an instance.

Built-In Class and Instance

- We can apply our new terminology, class and instructor, to the programming we have been doing.
- Any of the built-in data structure are defined as a class: a list is a class, so is a string, set, tuple, or dictionary.
- These class can be used to make individual instances using either the constructor (respectively list, str, set, tuple, dict) or their shortcuts, where available.
- The constructor creates a new object, which is an instance of the class.
- These instance have internal attributes and associated methods that may applied those instance.

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'
```

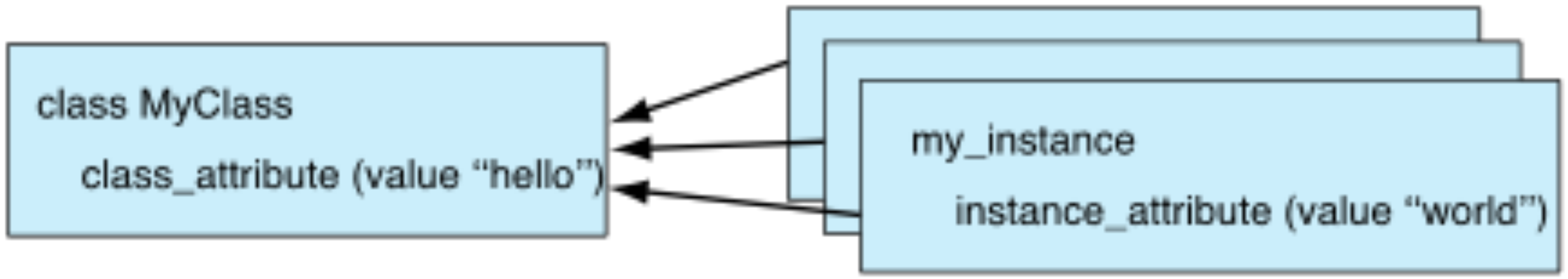

class MyClass
class_attribute (value “hello”)
my_instance
instance_attribute (value “world”)

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 instance (object) itself
2. If the attribute is not found, look up to the class of the object and search for the attribute there.

```
>>> 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
```

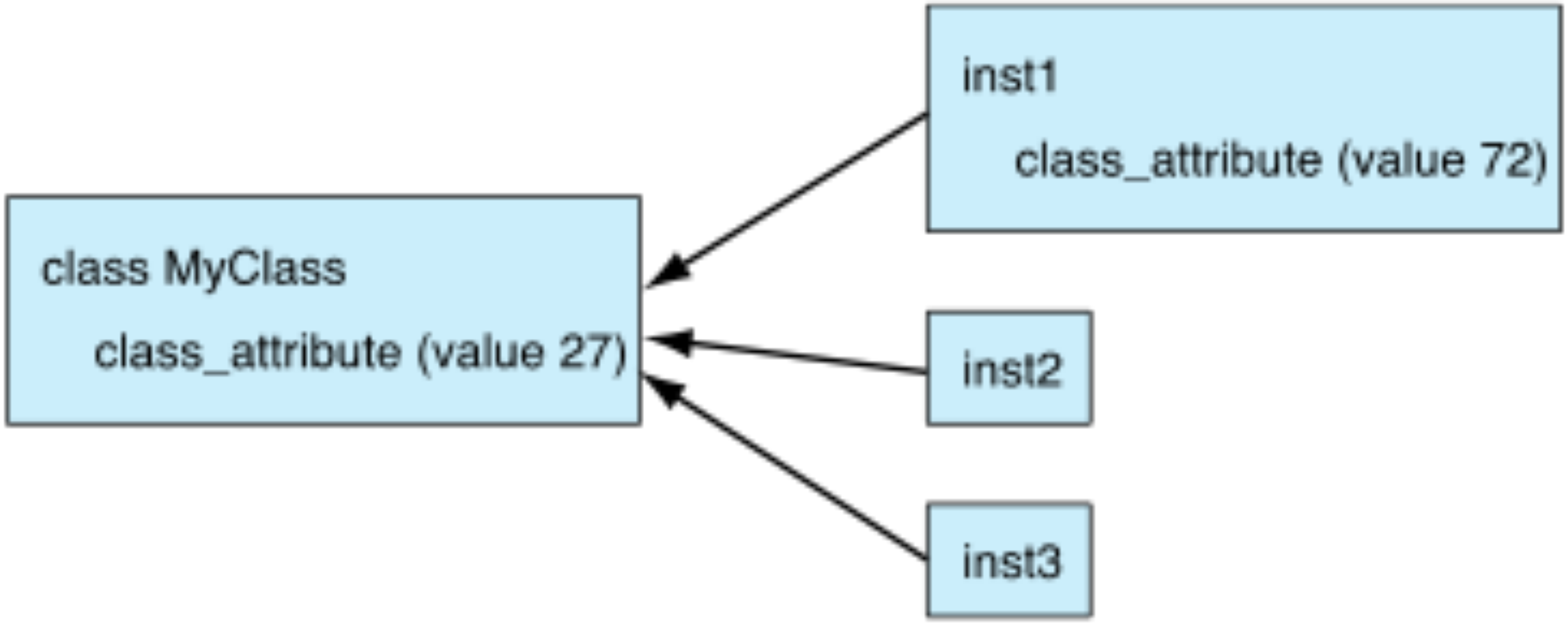


FIGURE 11.4 A mixture of local and instance-of attribute relationships.

Methods

```
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

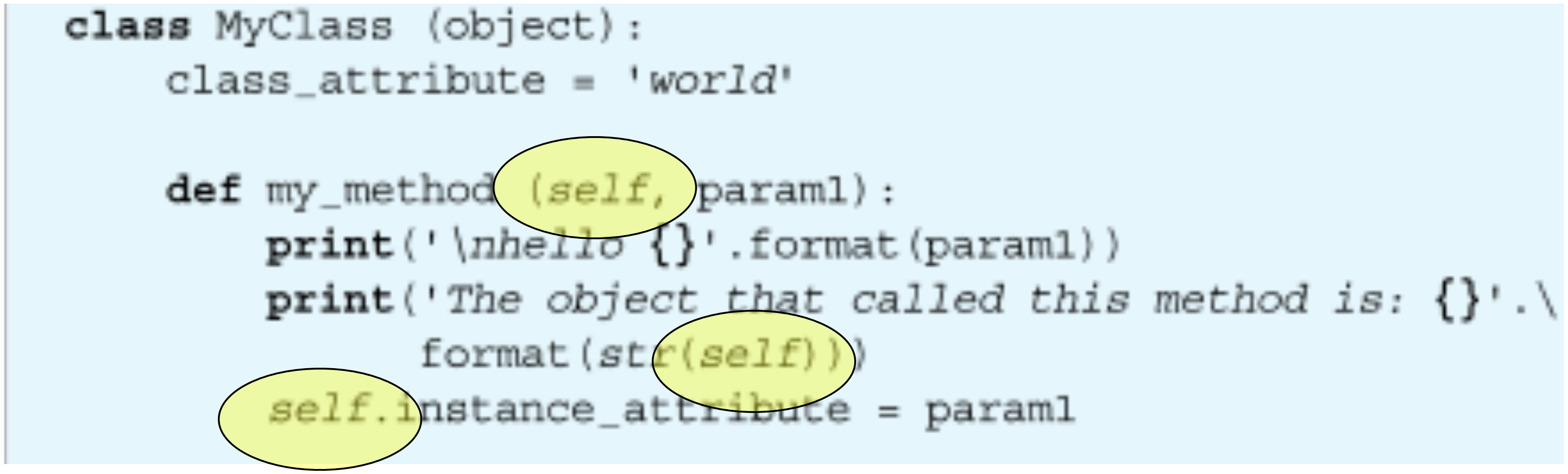
- methods are defined ***inside*** the suite of a class
- methods always bind the first parameter in the definition to the object that called it
- This parameter can be named anything, but traditionally it is named ***self***

```
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)

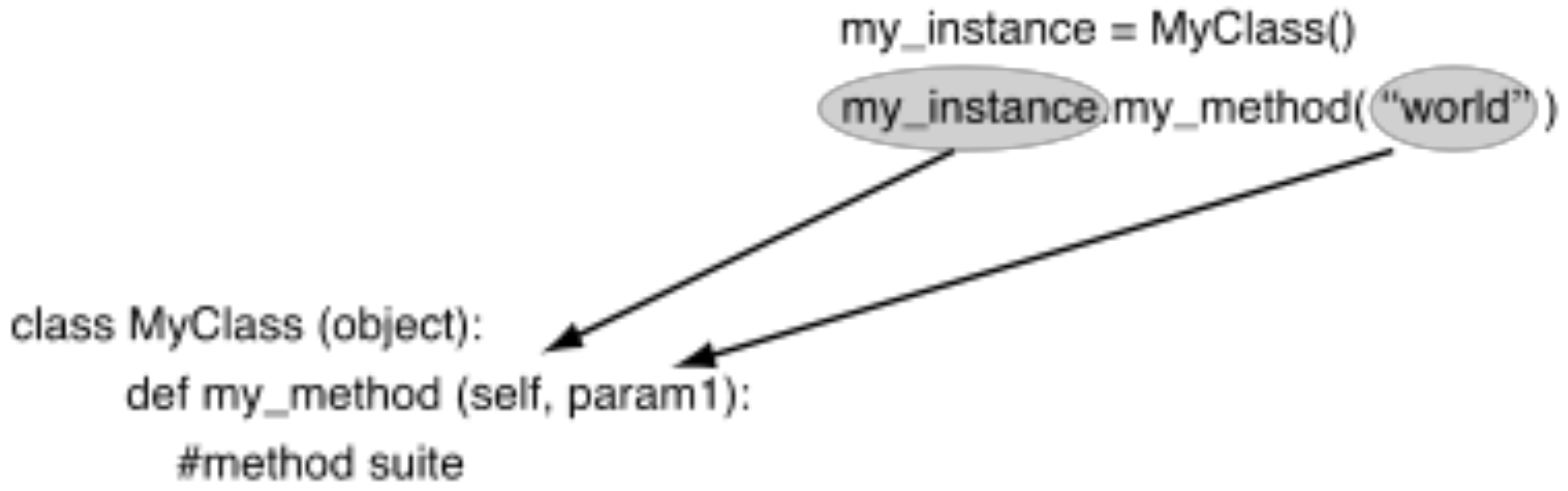
Back to the example



```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
```

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`

Two Rules

- All methods takes self as first argument, but self is left our in method calls.
- All data attributes and method calls with the class must be prefixed by 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 "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

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.last_name_str)

Outputs are:

Python
```

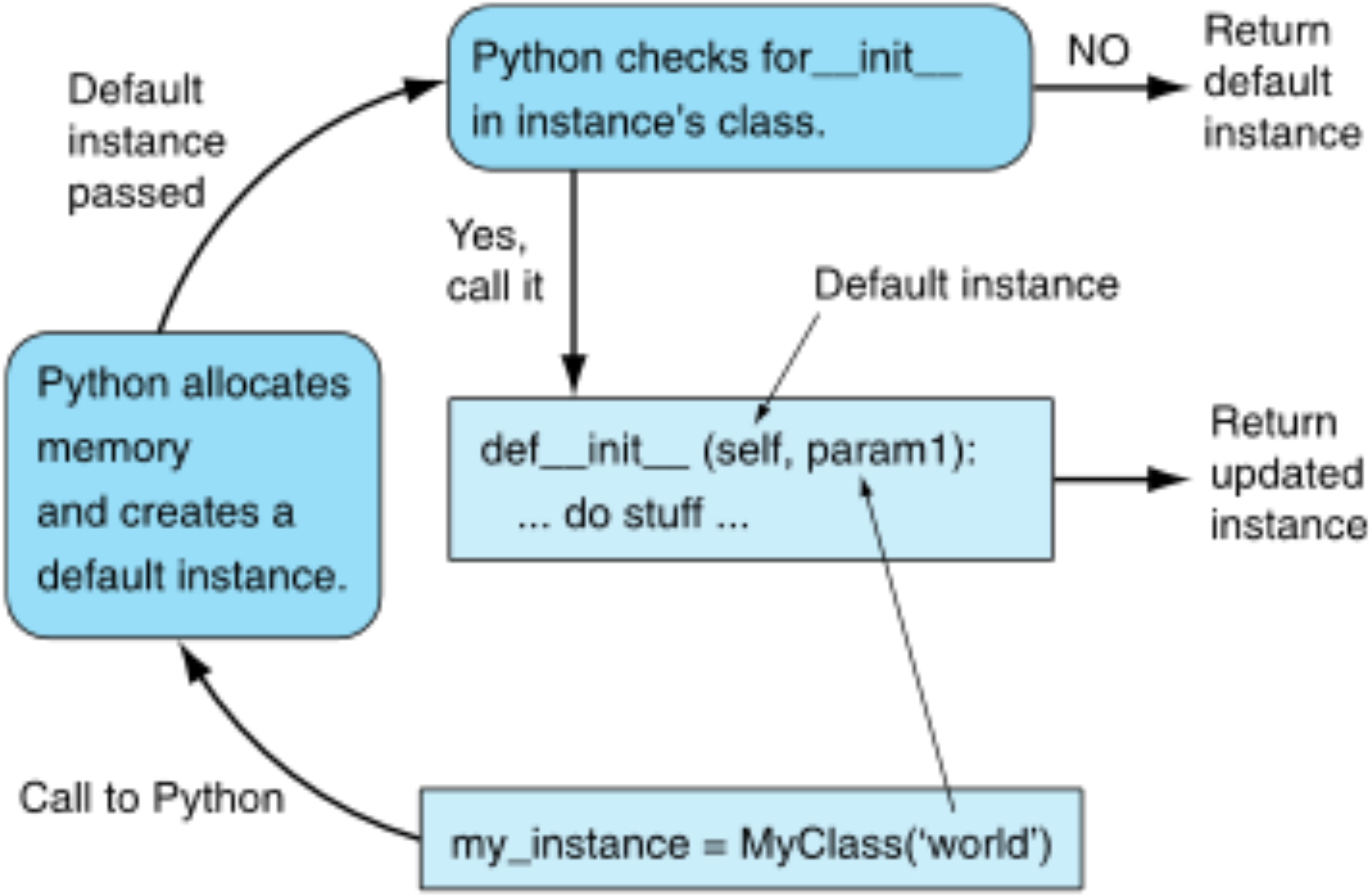

Python checks for__init__ in instance's class.
NO
Return default instance
Default instance passed
Yes, call it
Default instance
Python allocates memory and creates a default instance.
def__init__ (self, param1):
... do stuff ...
Return updated instance
Call to Python
my_instance = MyClass('world')

FIGURE 11.6 How an instance is made in Python.

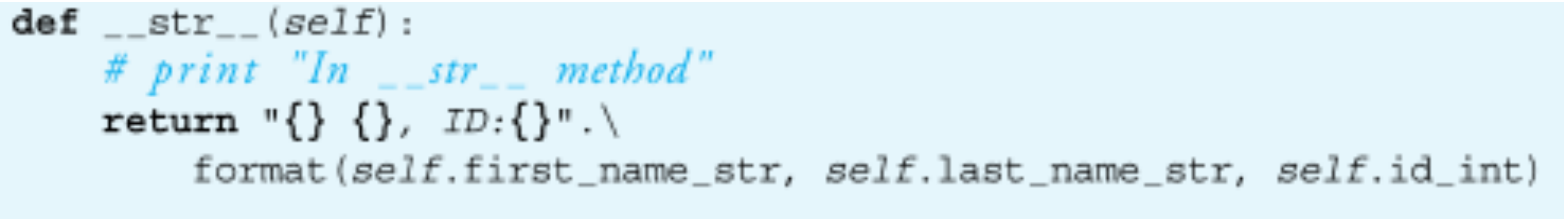
default constructor

- 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

Example: Point Class

- Point_template.py
- Let's tackle an example using Cartesian coordinates: dealing with two points. We want to represent the two-dimensional points on the Cartesian coordinate plane. Each point is completely described by its coordinate pair: (x,y).
- Once we have the instance attribute in place, we can consider what operations we wish to perform on these objects and write methods to take those actions. For illustration, we will choose finding the distance between two points and the sum of two points.
- More specifically, we will implement these methods:
  - Distance
  - sum

Rule 9

Make sure your new class does the right thing

- By “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:
  - 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:
- If the class is well designed, then a <u>user</u> of the class need only use the provided methods to use the class instance.
- The <u>class designer</u> is free to place information in the class that is important to the designer, but not the user, of the class.
- The class designer hides 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

Inheritance

- Inheritance allows the class designer to <u>utilize the design of an existing class to create a new class</u>.
- That is, we can create a new class that specializes an existing class by utilizing the existing class’s attributes, specializing only those attributes that distinguish the new class.
- In this way, classes can share common elements and change only those attributes that distinguish the new class.

Python and Other OOP Languages
Private values

class namespaces are dicts

- the namespaces in every object and module is indeed a namespace
- 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.
- Public means accessible by everyone (developer and user)
- 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.
- Python <u>does</u> not provide support for the designer to indicate attributes that the programmer should not modify directy.
- 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
```

- In the __init__ method, we assign two attributes: name and __attribute.
- By preceding the second with two underscores, we are indicating that __attribute is private and should not be accessed.
- Python changes that attribute name so that, if the provided attribute name (__attribute) is referred outside of the class, an error is thrown.
- However, using the original attribute name within the class, such as in a method of the class (e.g., __str__), is allowed.
- If we do dir(inst1), w see that the name has been changed (mangled) to _NewClass__attribute.
- This change of name does not prevent the programmer from accessing the attribute value. It only provides a layer of obfuscation.
- But, hen an attribute is marked with the double-underscore prefix, it means leave it alone!

More on Classes

The three OOP factors

Remember, we said there were 3 factors that distinguished an Object Oriented Programming langauge:

- encapsulation
- inheritance
- polymorphism

We are still at encapsulation

We said that encapsulation:

- *Hides details* of the implementation so that the program was easier to read and write
- Provides *modularity* that makes an object so that it can be reused in other contexts
- Provides an *interface* (the methods) that are the approved way to deal with the class

One more aspect

A new aspect we should have is *consistency*

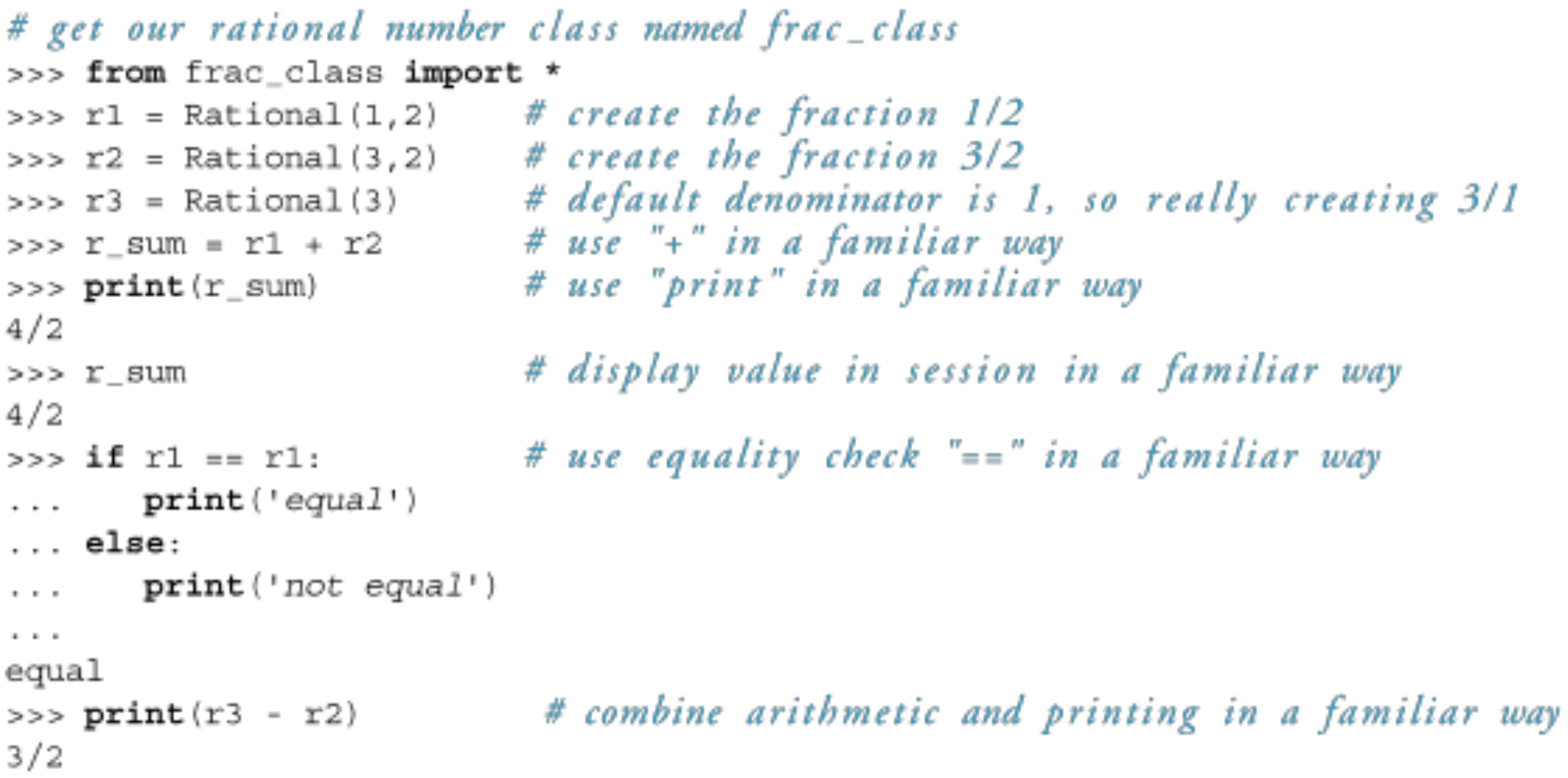
- A new class should be consistent with the rules of the language.
- It should respond to standard methods, i.e., it should behave properly with typical functions (assuming the type allows that kind of call).

An example

Consider a Rational number class. It should respond to:

- construction
- printing
- arithmetic ops (+, -, *, /)
- comparison ops (<, >, <=, >=)

example program



```
# get our rational number class named frac_class
>>> from frac_class import *
>>> r1 = Rational(1,2)     # create the fraction 1/2
>>> r2 = Rational(3,2)     # create the fraction 3/2
>>> r3 = Rational(3)       # default denominator is 1, so really creating 3/1
>>> r_sum = r1 + r2        # use "+" in a familiar way
>>> print(r_sum)           # use "print" in a familiar way
4/2
>>> r_sum                  # display value in session in a familiar way
4/2
>>> if r1 == r1:           # use equality check "==" in a familiar way
...     print('equal')
... else:
...     print('not equal')
...
equal
>>> print(r3 - r2)          # combine arithmetic and printing in a familiar way
3/2
```

just like any other number

- by building the class properly, we can make a new instance of Rational look like any other number syntactically.
- the instance responds to all the normal function calls
- because it is properly encapsulated, it is much easier to use

But how can that work?

Two parts:

- Python can distinguish which operator to use based on <u>types</u>
- Python provides <u>more standard methods </u>that represent the action of standard functions in the language
  - by defining them in our class, Python will call them in the "right way"

More on type

As we have mentioned, a class is essentially a new type

- when we make an instance of a class, we have made an object of a particular type
- 1.36 is a float
- `my_instance = MyClass()`, `my_instance` is a type `MyClass`

Introspection

- Python does not have a type associated with any variable, since each variable is allowed to reference any object
- however, we can query any variable as to what type it presently references
- this is often called ***introspection***. That is, while the program is running we can determine the type a variable references

Python introspection ops

- `type(variable)` returns its type as an object
- `isinstance(variable,type)` returns a boolean indicating if the variable is of that type

```python
def special_sum(a,b):
    ''' sum two ints or convert params to ints
    and add. return 0 if conversion fails '''
    if type(a)==int and type(b)==int:
        result = a + b
    else:
        try:
            result = int(a) + int(b)
        except ValueError:
            result = 0
    return result
```

Operator Overloading

So what does `var1+var2` mean?

The answer:

- it depends
- What it depends on is the type. The + operation has two operands. What are their types?
- Python uses <u>introspection to find the type</u> and then <u>select the correct operator</u>

We've seen this before

What does `var1+var2` do?

- with two strings, we get concatenation
- with two integers, we get addition
- with an integer and a string we get:

```
Traceback (most recent call last):
 File "<pyshell#9>", line 1, in <module>
   1+'a'
TypeError: unsupported operand type(s) for
  +: 'int' and 'str'
```

Operator overloading

- the plus operator is ***overloaded***
- that is, the operator can do/mean different things (have multiple/overloaded meanings) depending on the types involved
- if python does not recognize the operation and that combination of types, you get an error

Python overload ops

- Python provides a set of operators that can be overloaded. You can't overload all the operators, but you can for many
- Like all the special class operations, they use the two underlines before and after.
- They come in three general classes:
  - numeric type operations (+,-,<,>,print etc.)
  - container operations ([ ], iterate,len, etc.)
  - general operations (printing, construction)

Math-like Operators		
Expression	**Method name**	**Description**
x + y	`__add__()`	Addition
x − y	`__sub__()`	Subtraction
x * y	`__mul__()`	Multiplication
x / y	`__div__()`	Division
x == y	`__eq__()`	Equality
x > y	`__gt__()`	Greater than
x >= y	`__ge__()`	Greater than or equal
x < y	`__lt__()`	Less than
x <= y	`__le__()`	Less than or equal
x != y	`__ne__()`	Not equal
Sequence Operators		
len(x)	`__len__()`	Length of the sequence
x in y	`__contains__()`	Does the sequence y contain x?
x[key]	`__getitem__()`	Access element *key* of sequence x
x[key]=y	`__setitem__()`	Set element *key* of sequence x to value y
General Class Operations		
x=myClass()	`__init__()`	Constructor
print (x), str(x)	`__str__()`	Convert to a readable string
	`__repr__()`	Print a Representation of x
	`__del__()`	Finalizer, called when x is garbage collected

TABLE 12.1 Python Special Method Names

```python
class MyClass(object):
        def __init__(self, param1=0):
                ''' constructor, sets attribute value to
                param1, default is 0'''
                print('in constructor')
                self.value = param1

        def __str__(self):
                ''' Convert val attribute to string.'''
                print('in str')
                return 'Val is: {}'.format(str(self.value))

        def __add__(self,param2):
                ''' Perform addition with param2, a MyClass instance.
               Return a new MyClass instance with sum as value attribute'''
                print('in add')
                result = self.value + param2.value
                return MyClass(result)
```

how does v1+v2 map to __add__

```
v1 + v2
```

is turned, by Python, into

```
v1.__add__(v2)
```

- These are ***exactly equivalent expressions***. It means that the first variable calls the __add__ method with the second variable passed as an argument
- v1 is bound to `self`, v2 bound to `param2`

Calling `__str__`

- When does the `__str__` method get called? Whenever a string representation of the instance is required:
  - directly, by saying `str(my_instance)`
  - indirectly, calling `print(my_instance)`

Simple Rational Number class

- a Rational is represented by two integers, the numerator and the denominator
- we can apply many of the numeric operators to Rational

```python
class Rational(object):
    """ Rational with numerator and denominator. Denominator
    parameter defaults to 1"""
    def __init__(self,numer,denom=1):
        print('in constructor')
        self.numer = numer
        self.denom = denom

    def __str__(self):
        """ String representation for printing"""
        print('in str')
        return str(self.numer)+'/'+str(self.denom)

    def __repr__(self):
        """ Used in interpreter. Call __str__ for now """
        print('in repr')
        return self.__str__()
```

__str__ vs __repr__

- __repr__ is what the interpreter will call when you type an instance
  - potentially, the representation of the instance, something you can recreate an instance from.
- __str__ is a conversion of the instance to a string.
  - Often we define `__str__`, have `__repr__` call `__str__`
  - note the call: `self.__str__()`

the `__init__` method

- each instance gets an attribute `numer` and `denom` to represent the numerator and denominator of that instance's values

provide addition

Remember how we add fractions:

- if the denominator is the same, add the numerators
- if not, find a new common denominator that each denominator divides without remainder.
- modify the numerators and add

the lcm and gcd

- the least common multiple (lcm) finds the smallest number that each denominator divides without remainder
- the greatest common divisor (gcd) finds the largest number two numbers can divide into without remainder

LCM in terms of GCD

$$LCM(a,b) = \frac{a * b}{GCD(a,b)}$$

OK, how to find the gcd then?

GCD and Euclid

- One of the earliest algorithms recorded was the GCD by Euclid in his book Elements around 300 B.C.
  - He originally defined it in terms of geometry but the result is the same

The Algorithm

GCD(a,b)

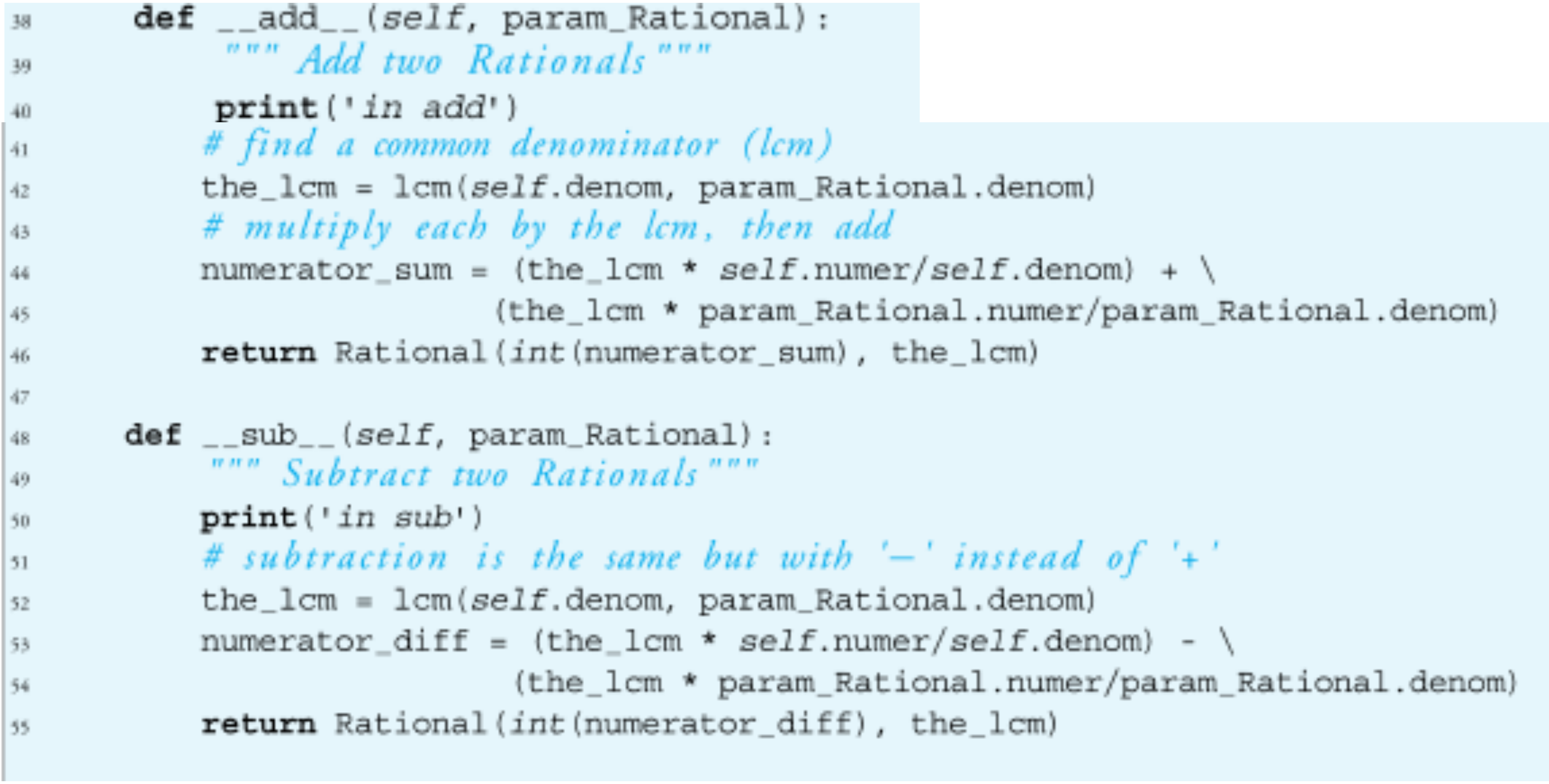
1. If one of the numbers is 0, return the other and halt
2. Otherwise, find the integer remainder of the larger number divided by the smaller number
3. Reapply GCD(a,b) with a $\leftarrow$ smaller and b$\leftarrow$ the remainder from step 2)

```
def gcd(bigger, smaller):
    """Calculate the greatest common divisor of two positive integers."""
    if not bigger > smaller:            # swap if necessary so bigger > smaller
        bigger, smaller = smaller, bigger
    while smaller != 0:                          # 1. if smaller == 0, halt
        remainder = bigger % smaller             # 2. find remainder
        print('calculation, big:{}, small:{}, rem:{}'.\
              format(bigger, smaller, remainder)) # debugging
        bigger, smaller = smaller, remainder  # 3. reapply
    return bigger
```

```

def lcm (a,b):
    """Calculate the lowest common multiple of two positive integers."""
    return (a*b)//gcd(a,b)  # Equation 12.1, // ensures an int is returned
```

Method: Add



```
    def __add__(self, param_Rational):
        """ Add two Rationals"""
        print('in add')
        # find a common denominator (lcm)
        the_lcm = lcm(self.denom, param_Rational.denom)
        # multiply each by the lcm, then add
        numerator_sum = (the_lcm * self.numer/self.denom) + \
                        (the_lcm * param_Rational.numer/param_Rational.denom)
        return Rational(int(numerator_sum), the_lcm)

    def __sub__(self, param_Rational):
        """ Subtract two Rationals"""
        print('in sub')
        # subtraction is the same but with '-' instead of '+'
        the_lcm = lcm(self.denom, param_Rational.denom)
        numerator_diff = (the_lcm * self.numer/self.denom) - \
                         (the_lcm * param_Rational.numer/param_Rational.denom)
        return Rational(int(numerator_diff), the_lcm)
```

Equality

- The equality method is `__eq__`
- It is invoked with the == operator

  ½ == ½ is equivalent to `½.__eq__(½)`
- It should be able to deal with non-reduced fractions:

  ½ == ½ is True

  so is 2/4 == 3/6

```python
def reduce_rational(self):
    """ Return the reduced fractional value as a Rational"""
    print('in reduce')
    # find the gcd and then divide numerator and denominator by gcd
    the_gcd = gcd(self.numer,self.denom)
    return Rational(self.numer//the_gcd, self.denom//the_gcd)

def __eq__(self,param_Rational):
    """ Compare two Rationals for equality, return Boolean"""
    print('in eq')
    # reduce both; then check that numerators and denominators are equal
    reduced_self = self.reduce_rational()
    reduced_param = param_Rational.reduce_rational()
    return reduced_self.numer == reduced_param.numer and\
        reduced_self.denom == reduced_param.denom
```

Fitting in

- What is amazing about the traces of these methods is how many of them are called in service of the overall goal.
- All we did was to provide the basic pieces, and Python orchestrates how they all fit together!

What doesn't work

So r1+r2, but what about

- We said the add we defined would work for two rationals, but what about?

```
r1 + 1   # Rational plus an integer
1 + r1   # commutativity
```

- Neither works right now. How to fix?

r1 + 1

- What's the problem?
  - add expects another rational number as the second argument.
  - Python used to have a coercion operator, but that is deprecated
    - coerce: force conversion to another type
    - deprecate: 'disapproval', an approach that is no longer supported
  - Our constructor would support conversion of an int to a Rational, how/where to do this?

Introspection in __add__

- the add operator is going to have to check the types of the parameter and then decide what should be done
- if the type is an integer, convert it. If it is a Rational, do what we did before. Anything else that is to be allowed needs to be checked

```python
    def __add__(self, param):
        """ Add two Rationals. Allows int as a parameter"""
        print('in add')
        if type(param) == int:  # convert ints to Rationals
            param = Rational(param)
        if type(param) == Rational:
            # find a common denominator (lcm)
            the_lcm = lcm(self.denom, param.denom)
            # multiply each by the lcm, then add
            numerator_sum = (the_lcm * self.numer/self.denom) + \
                (the_lcm * param.numer/param.denom)
            return Rational(int(numerator_sum),the_lcm)
        else:
            print('wrong type')  # problem: some type we cannot handle
            raise(TypeError)
```

what about 1 + r1

- What's the problem
  - mapping is wrong
  - 1 + r1 maps to 1.__add__(r1)
  - no such method for integers (and besides, it would be a real pain to have to add a new method to every type we want to include)
  - user should expect that this should work. Addition is commutative!

radd method

- Python allows the definition of an `__radd__` method: right-operand arithmetic
- The `__radd__` method is called when the `__add__` method fails because of a type mismatch
- `__radd__` reverses the two arguments in the call
- __add__(self,a) defined self+a, while __radd__(self,a) defined a+self.

__radd__ vs __add__

- 1 + r1

  try 1.__add__(r1), failure

  look for an __radd__ if it exists, remap

- 1 + r1

  r1.__radd__(1)

radd

- essentially, all we need `__radd__` to do is remap the parameters.
- after that, it is just add all over again, so we call `__add__` directly
- means we only have to maintain `__add__` if any changes are required

```
def __radd__(self,f):
    return self.__add__(f)
```

Inheritance

Class-Instance relations

- Remember the relationship between a class and its instances
  - a class can have many instances, each made initially from the constructor of the class
  - the methods that an instance can call are initially shared by all instances of a class
  - When referencing a value in an attribute, Python first looks in the instance for the attribute, and, if not found there, it then looks in the class the instance was derived from. In this way, attributes stored in the class are available to every instance derived from the class.

Class-Class relations

- Classes can also have a separate relationship with other classes
- the relationships forms a hierarchy
  - ***hierarchy***: A body of persons or things ranked in grades, orders or classes, one above another

Classes related by a hierarchy

- when we create a class, which is itself another object, we can state how it is related to other classes
- the relationship we can indicate is the class that is 'above' it in the hierarchy
- Every class maintains at least one parent class.

class statement

name of the class above this class in the hierarchy

```
class MyClass (SuperClass):
     pass
```

- The top class in Python is called `object`.
- it is predefined by Python, always exists
- use `object` when you have no superclass

```
class MyClass (object):
    pass


class Child1Class (MyClass):
    pass


class Child2Class (MyClass):
    pass
```

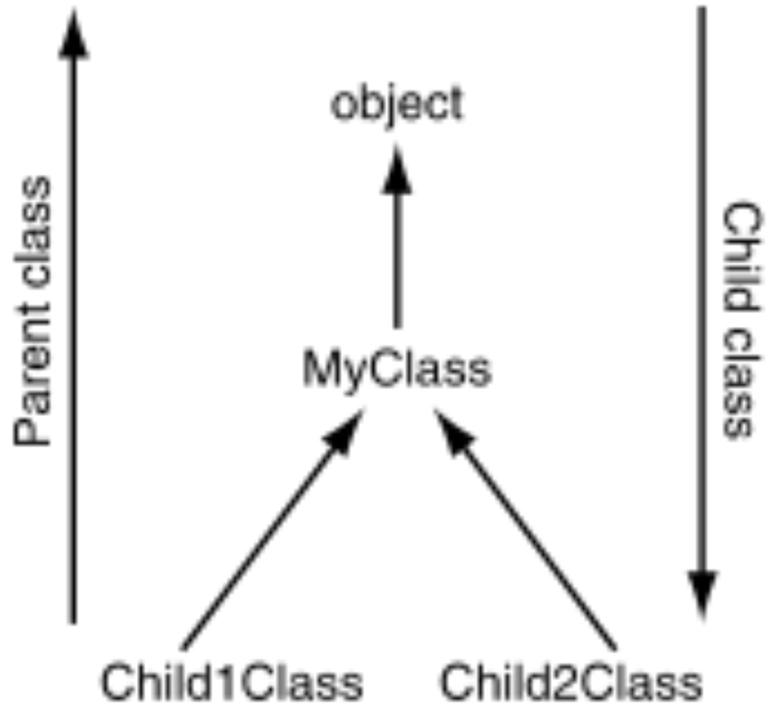
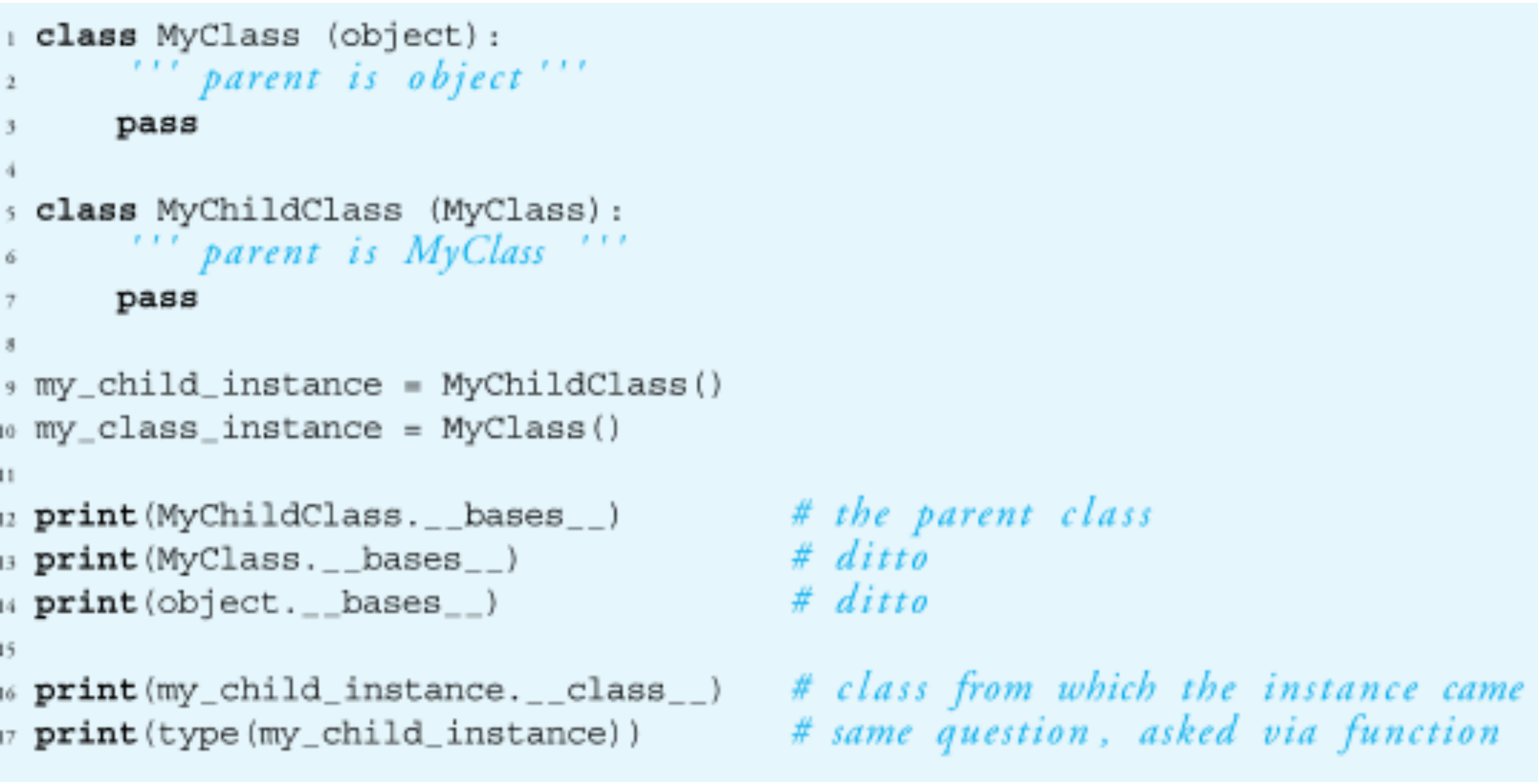


FIGURE 12.1 A simple class hierarchy.

- The is-a relationship is one way, much as the instance-of relationship.
- The instance remember who its class is, but the class does not track its instance.
- A class remember who its parent class is, but the parent class does not track its child classes.



```
class MyClass (object):
    ''' parent is object '''
    pass

class MyChildClass (MyClass):
    ''' parent is MyClass  '''
    pass

my_child_instance = MyChildClass()
my_class_instance = MyClass()

print(MyChildClass.__bases__)             # the parent class
print(MyClass.__bases__)                  # ditto
print(object.__bases__)                   # ditto

print(my_child_instance.__class__)        # class from which the instance came
print(type(my_child_instance))            # same question, asked via function
```

- Each Python class indicates specifically in its class definition who its parent is.
- This relationship is recorded in the __bases__ attribute of each class.

is-a, super and sub class

- the class hierarchy imposes an ***is-a*** relationship between classes
  - MyChildClass ***is-a*** (or is a kind of) MyClass
  - MyClass ***is-a*** (or is a kind of) object
  - object has as a subclass MyClass
  - MyChildClass has as a superclass MyClass
- An example: if you create a Car class, then a Ford class *is a* Car, but a more particular kind of Car. Subsequently, a Mustang class *is a* Ford class, but a more particular kind of Ford.

um, so what?

- the hope of such an arrangement is the saving/re-use of code. If a new class is created as part of an existing class hierarchy, then the new class can <u>reuse</u> existing code from the hierarchy, specializing only those aspects or attributes that are unique to the new class.
- superclass code contains general code that is applicable to many subclasses. By sharing code from the class hierarchy, the coding of classes can be somewhat <u>standardized</u>.
- subclass uses superclass code (via sharing) but specializes code for itself when necessary

Scope for objects, the full story

1. Look in the object for the attribute
2. If not in the object, look to the object's class for the attribute (up the instance-of reation)
3. If not in the object's class, look up the hierarchy of that class for the attribute (up the is-a relation)
4. If you hit object, then the attribute does not exist

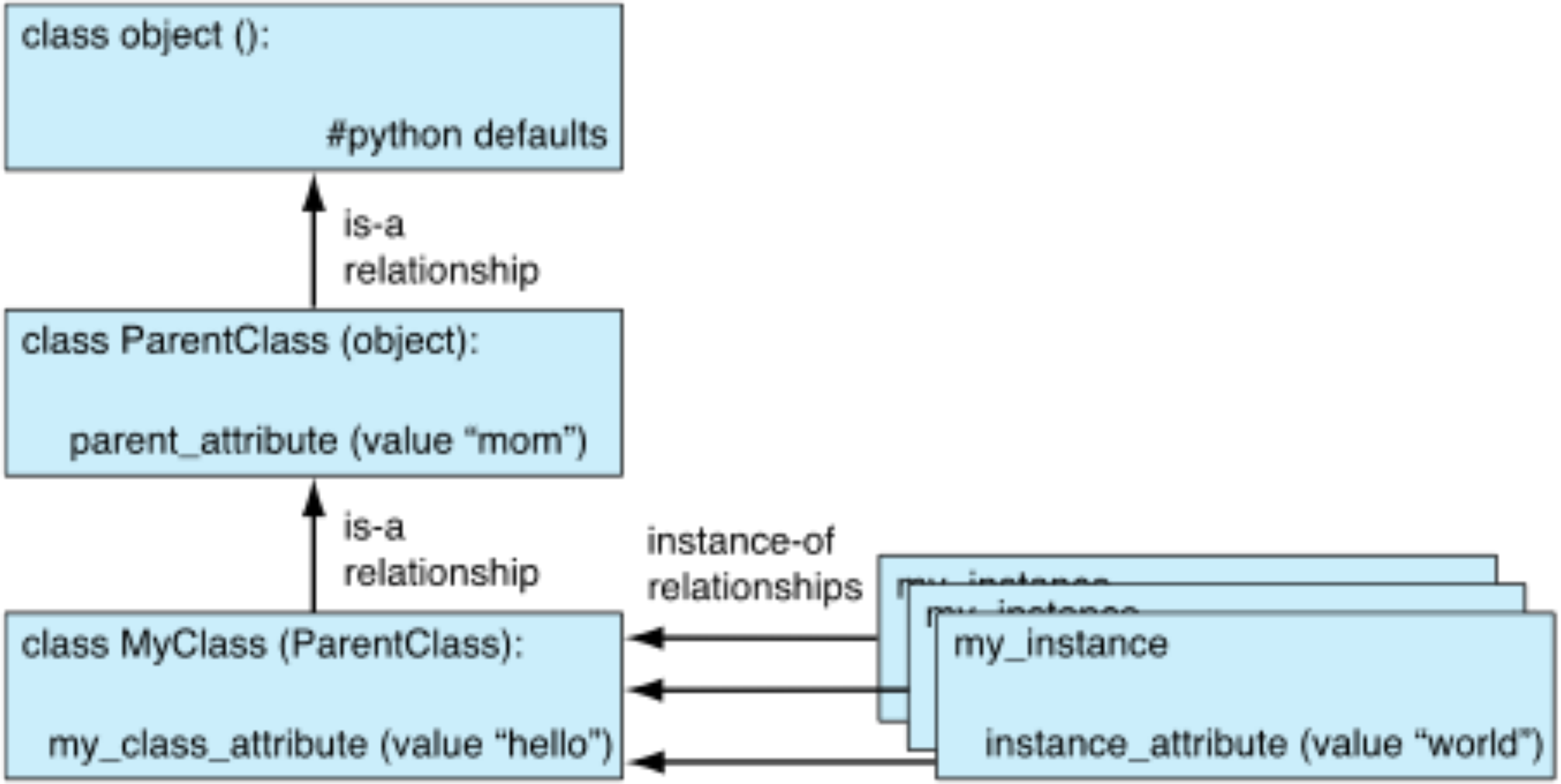

class object ():
#python defaults
is-a relationship
class ParentClass (object):
parent_attribute (value "mom")
is-a relationship
instance-of relationships
class MyClass (ParentClass):
my_class_attribute (value "hello")
my_instance
instance_attribute (value "world")

FIGURE 12.2 The players in the "find the attribute" game.

Inheritance is powerful but also can be complicated

- many powerful aspects of OOP are revealed through uses of inheritance
- However, some of that is a bit detailed and hard to work with. Definitely worth checking out but a bit beyond the class

builtins are objects too

- One nice way, easy way, to use inheritance is to note that all the builtin types are objects also
- thus you can inherit the properties of builtin types then modify how they get used in your subclass
- you can also use any of the types you pull in as modules

specializing a method

- One technical detail. Normal method calls are called ***bound methods***. Bound methods have an instance in front of the method call and automatically pass self

```
my_inst = MyClass()
my_inst.method(arg1,arg2)
```

- `my_inst` is an instance, so the method is bound

unbound methods

it is also possible to call a method without Python binding `self`. In that case, the user has to do it.

- unbound methods are called as part of the class but `self` passed by the user

```
my_inst = MyClass()
MyClass.method(my_inst, arg2, arg3)
```

`self` is passed **explicitly** (`my_inst` here)!

Why???

- Consider an example. We want to specialize a new class as a subclass of list.

```
class MyClass(list):
```

- easy enough, but we want to make sure that we get our new class instances initialized the way they are supposed to, by calling `__init__` of the super class

Why call the super class init?

If we don't explicitly say so, our class may inherit stuff from the super class, but we must make sure we call it in the proper context. For example, our `__init__` would be:

```
def __init__(self):
   list.__init__(self)
# do anything else special to MyClass
```

explicit calls to the super

- we explicitly call the super class constructor using an unbound method (why not a bound method????)
- then, after it completes we can do anything special for our new class
- We **specialize** the new class but inherit most of the work from the super. Very clever!

Gives us a way to organize code

- ***specialization***. A subclass can inherit code from its superclass, but modify anything that is particular to that subclass
- ***over-ride***. change a behavior to be specific to a subclass
- ***reuse-code***. Use code from other classes (without rewriting) to get behavior in our class.

Exercise: Multiple Inheritance

- Multiple inheritance is obtained by listing two or more base classes in parenthesis after the class name.
- Read Python code MultInheritance.py. What is the output of this code?

Class for Scientific Computing

Class Programming

Write a Python class Trapezoidal to evaluate numerical integration using the trapezoidal rule.

$$\int_{-1}^{1} f(x)dx \approx f(-1) + f(1)$$

It is one of the quadrature rules that can be expressed in the general form:

$$\int_{-1}^{1} f(x)dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

Use the class to compute

$$\int_{-1}^{1} x^3 dx$$

Inline lambda functions

- Python offers anonymous inline functions known as lambda function.
- The construction is

“lambda <args>: <expression>”

- It is equivalent to a function with <args> as arguments and <expression> as return value:

```
def somefunc(<args>):
    return <expression>
```

- For example, “lambda x, y, z: 3*x+2*y-z” is a short cut for

```
def somefunc(x, y, z)
    return 3*x + 2*y – z
```

Inline lambda functions

- Lambda function can be used in place where we expect variables.
- Say we have a function taking another function as argument:

```
def fill(a,f)
    n = len(a); dx = 1.0/(n-1)
    for i in range(n):
        x = i * dx
        a[i] = f(x)
```

- A lambda function can be used for the f argument:

```
fill(a, lambda x: <expression>)
```

- Read code Trapezoidal.py.

OOP for Integration

Consider other quadrature rules:

- Simpson's rule

$$\int_{-1}^{1} f(x)dx \approx \frac{1}{3} f(-1) + \frac{4}{3} f(0) + \frac{1}{3} f(1)$$

- Two-point Gauss-Legendre rule

$$\int_{-1}^{1} f(x)dx \approx f(-\frac{1}{\sqrt{3}}) + f(\frac{1}{\sqrt{3}})$$

Many rules of this form can be defined.

Complete the Python program, intergration.py, to implement the quadrature rules.