Light on Python

July 27, 2018

Contents

Chapter 1

Objects

1.1 Introduction

This course is for the adventurous:

- You'll learn Python the way a child would, even if you are an adult. Children are experts in learning. They learn by doing, and pick up words along the way. In this text the same approach is followed. NOT EVERYTHING IS DEFINED OR EVEN EXPLAINED. JUST TRY TO FIND OUT HOW THE EXAMPLE CODE WORKS BY GUESSING AND EXPERIMENTING. The steps taken may seem large and sometimes arbitrary. It's a bit like being dropped into the jungle without a survival course. But don't worry, computer programming isn't nearly as dangerous. And the steps taken in fact follow a carefully planned path. Regularly try to put together something yourself. Play with it. Evolution has selected playing as the preferred way of learning. I will not claim to improve on that.

- You'll be addressed like an adult, even if you are a child. Simple things will be explained simple, but the complexity of complex things will not avoided. The right, professional terminology will be used. If you don't know a word, like "terminology", Google for it. Having a separate child's world populated by comic figures, Santa Claus and storks bringing babies is a recent notion. Before all that, it was quite normal to have twelve year old geniuses. But don't worry, programming can be pure fun, both for children and adults.

- You'll focus upon a very effective way of using Python right from the start. It is called object oriented programming. And you'll learn some functional programming as well. Don't bother what these words mean. It'll become clear underway. Mixing two ways of programming is no greater problem than children being brought up with two or more languages: no problem at all. By the way, those children have markedly healthier brains once they get older. There are also less important things to learn about Python. You learn those gradually if you wish, while using Python. Just stay curious and look things up on the Internet.

I learned to program as a child, my father was programming the first computers in the early 1950's. We climbed through a window into the basement of the office building of his employer, a multinational oil company. Security was no issue back then. Programming turned out to be fun indeed. And it still is, for me!

TIP: Sections marked with *EXTRA * provide additional material if you like to be challenged above the average or already have quite some Python experience. You can become a good and productive Python programmer without ever touching these sections. The most important thing is that you start coding regularly. Try e.g. to write a simulation or a game and build that out gradually, regularly pushing the limits of your Python knowledge a bit further.

1.2 Your first program

Install Python 3.x. The Getting Started topic on www.python.org will tell you how.

IMPORTANT: Python 3.x rather than 2.x is indeed required to run all of the examples correctly.

You will also need an editor. If you're on Windows, Google for Notepad++. If you're on Linux or Apple, you can use Gedit. Then run the following program:

```
1  cities = ['Londen', 'Paris', 'New York', 'Berlin'] # Store 4 strings into a list
2  print ('Class is:', type (cities))                  # Verify that it is indeed a list
3
4  print ('Before sorting:', cities)                   # Print the unsorted list
5  cities.sort ()                                       # Sort the list
6  print ('After sorting: ', cities)                   # Print the sorted list
```
<div align="center">Listing 1.1: prog/sort.py</div>

The pieces of text at the end of each line, starting with #, are comments. Comments don't do anything, they just explain what's happening. *'London'*, *'Paris'*, *'New York'* and *'Berlin'* are strings, pieces of text. You can recognize such pieces of text by the quotes around them. Programmers would say these four objects are instances of class string. To clarify, a particular dog is an instance of class *Dog*. There may be classes for which there are no instances. Class *Dinosaur* is such a class, since there are no (living) dinosaurs left. So a class in itself is merely a description of a certain category of objects.

Line 1 of the previous program is actually shorthand for line 1 of the following program:

```
1  cities = list (('Londen', 'Paris', 'New York', 'Berlin')) # Construct list object from 'tuple' of 4 string objects
2  print ('Class is:', type (cities))                        # Verify that it is indeed a list
3
4  print ('Before sorting:', cities)                         # Print the unsorted list
5  cities.sort ()                                             # Sort the list
6  print ('After sorting: ', cities)                         # Print the sorted list
```
<div align="center">Listing 1.2: prog/sort2.py</div>

So you construct objects of a certain class by using the name of that class, followed by *()*. Inside this *()* there maybe things used in constructing the object. In this case the object is of class *list*, and there's a so called tuple of cities inside the *()*. Since the tuple itself is also enclosed in *()*, you'll have *list ((...))*, as can be seen in the source code. For example *(1, 2, 3)* is a tuple of numbers, and *list ((1, 2, 3))* is a list constructed from this. We could also have constructed this list with the shorthand notation *[1, 2, 3]*, which means exactly the same thing as *list ((1, 2, 3))*. A tuple is an immutable group of objects. So you could never sort a tuple itself. But the list you construct from it is mutable, so you can sort it.

Once it works, try to make small alterations and watch what happens. Actually DO this, it will speed up learning

1.3 Specifying your own classes

Generally, in a computer program you work with many different classes of objects: buttons and lists, images and texts, movies and music tracks, aliens and spaceships, chessboards and pawns.

So, looking at the "real" world: you are an instance of class *HumanBeing*. Your mother is also an instance of class *HumanBeing*. But the object under your table wagging its tail is an instance of class *Dog*. Objects can do things, often with other objects. You're mother and you can walk the dog. And your dog can bark, as dogs do.

Lets create a *Dog* class in Python, and then have some actual objects (dogs) of this class (species):

```
1  class Dog:               # The species is called Dog
2      def bark (self):     # Define that this dog itself can bark
3          print ('Wraff!') # Which means saying "Wraff"
4
5
6  your_dog = Dog ()         # And than lets have an actual dog
7
8  your_dog.bark ()          # And make it bark
```
<div align="center">Listing 1.3: /prog/dog.py</div>

Now lets allow different dogs to bark differently by adding a constructor that puts a particular sound in a particular dog when it's instantiated (born), and then instantiate your neighbours dog as well:

```
1  class Dog:                    # Define the dog species
2      def __init__ (self, sound): # Constructor, named __init__, accepts provided sound
3          self.sound = sound    # Stores accepted sound into self.sound field inside new dog
4
5      def bark (self):          # Define bark method
6          print (self.sound)    # Prints the self.sound field stored inside this dog
7
8  your_dog = Dog ('Wraff')      # Instantiate dog, provide sound "Wraff" to constructor
9  neighbours_dog = Dog ('Wooff') # Instantiate dog, provide sound "Wooff" to constructor
10
11 your_dog.bark ()              # Prints "Wraff"
12 neighbours_dog.bark ()        # Prints "Wooff"
```

Listing 1.4: /prog/neighbours_dog.py

After running this program and again experimenting with small alterations, lets expand it further. You and your mother will walk your dog and the neighbours dog:

```
1  class HumanBeing:             # Define the human species
2      def walk (self, dog):     # The human itself walks the dog
3          print ('\nLets go!')  # \n means start on new line
4          dog.escape ()         # Just lets it escape
5
6  class Dog:                    # Define the dog species
7      def __init__ (self, sound): # Constructor, named __init__, accepts provided sound
8          self.sound = sound    # Stores accepted sound into self.sound field inside new dog
9
10     def bark (self):          # Define bark method
11         print (self.sound)    # It prints the self.sound field stored inside this dog
12
13     def escape (self):        # Define escape method
14         print ('Run to tree') # The dog will run to the nearest tree
15         self.bark ()          # It then calls upon its own bark method
16         self.bark ()          # And yet again
17
18 your_dog = Dog ('Wraff')      # Instantiate dog, provide sound "Wraff" to constructor
19 neighbours_dog = Dog ('Wooff') # Instantiate dog, provide sound "Wooff" to constructor
20
21 you = HumanBeing ()           # Create yourself
22 mother = HumanBeing ()        # Create your mother
23
24 you.walk (your_dog)           # You walk your own dog
25 mother.walk (neighbours_dog)  # your mother walks the neighbours dog
```

Listing 1.5: prog/walking_the_dogs

Run the above program and make sure you understand every step of it. Add some print statements printing numbers, to find out in which order it's executed. Adding such print statements is a simple and effective method to *debug* a program (find out where it goes wrong).

In the last example the *walk* method, defined on line 2, receives two parameters (lumps of data) to do its job: *self* and *dog*. It then calls (activates) the *escape* method of that particular dog: *dog.escape ()*. Lets follow program execution from line 24: *you.walk (your_ dog)*. This results in calling the *walk* method defined on line 2, with parameter *self* referring to object *you* and parameter *dog* referring to object *your_ dog*. The object *you* before the dot in *you.walk (your_ dog)* is passed to the *walk* method as the first parameter, called *self*, and *your_ dog* is passed to the *walk* method as the second parameter, *dog*.

Parameters used in calling a method, like *you* and *your_ dog* in line 24 are called *actual parameters*. Parameters that are used in defining a method, like *self* and *dog* in line 2 are called *formal parameters*. The use of formal parameters is necessary since you cannot predict what the names of the actual parameters will be. In the statement *mother.walk*

(neighbours_ dog) on line 25, different actual parameters, *mother* and *neighbour_ dog*, will be substituted for the same formal parameters, *self* and *dog*. Passing parameters to a method is a general way to transfer information to that method.

1.4   Indentation, capitals and the use of _

As can be seen from the listings, indentation is used to tell Python that something is a part of something else, e.g. that methods are part of a class, or that statements are part of a method. You have to be concise here. Most Python programmers indent with multiples of 4 spaces. For my own non-educational programs I prefer tabs.

Python is case-sensitive: uppercase and lowercase letters are considered distinct. When you specify your own classes, it is common practice to start them with a capital letter and use capitals on word boundaries: *HumanBeing*. For objects, their attributes (which are also objects) and their methods, in Python it is common to start with a lowercase letter and use _ on word boundaries: *bark, your_ dog*.

Constructors, the special methods that are used to initialize objects (give them their start values), are always named _ _*init*_ _ .

There's a recommendation about how to stylize your Python source code, it's called PEP 0008 and its widely followed. But it is strictly Python and I am mostly using a mix of Python and C++. Since many C++ libraries have different naming conventions, I don't usually follow these rules. If you want to learn a style that is consistent over multiple programming languages, use capitals on word boundaries for objects, atributes and methods as well instead of _, but always start them with a lowercase letter. Only class names start with an uppercase letter. By the way *WritingClassNamesLikeThis* or *writingAllOtherNamesLikeThis* is called camel case, while *writing_ all_ other_ names_ like_ this* is called pothole case. You'll find examples of both the camel case and the potholse case style in this course.

Chapter 2

Encapsulation

2.1 Interfaces

All objects of a certain class have the same attributes, but with distinct values, e.g. all objects of class *Dog* have the attribute *self.sound*. And all objects of a certain class have the same methods. For our class *Dog* in the last example, those are the methods *__init__*, *bark* and *escape*. Objects can have dozens or even hundreds of attributes and methods. In line 4 of the previous example, method *walk* of a particular instance of class *HumanBeing,* referred to as *self*, calls method *escape* of a particular instance of class *Dog*, referred to as *dog*.

So in the example *you.walk* calls *your_dog.escape* and *mother.walk* calls *neighbours_dog.escape*. Verify this by reading through the code step by step, and make sure not to proceed until you fully and thoroughly understand this.

In general any object can call any method of any other object. And it also can access any attribute of any other object. So objects are highly dependent upon each other. That may become a problem. Suppose change your program, e.g. by renaming a method. Then all other objects that used to call this method by its old name will not work anymore. And changing a name is just simple. You may also remove formal parameters, change their meaning, or remove a method altogether. In general, in a changing world, you may change your design. As your program grows bigger and bigger, the impact of changing anything becomes disastrous.

To limit the impact of changing a design, standardization is the answer. Suppose we have two subclasses of *HumanBeing*: *NatureLover* and *CouchPotato*. Objects of class *NatureLover* go out with their dogs to enjoy a walk. Objects of class *CouchPotato* just deliberately let the dog escape at the doorstep, that it might walk itself while they're watching their favorite soap. While they both have a *walk* method, walking the dog means something quite different to either of them. A programmer would say that their interface is standard (*walk*), but their implementation is different (calling *dog.follow_me* versus calling *dog.escape*). Let's see this in code:

```
1   class NatureLover:              # Define a type of human being that loves nature
2       def walk (self, dog):       # The NatureLover walks the dog, really
3           print ('\nC\'mon!')     # \n means start on new line, \' means ' inside string
4           dog.follow_me ()        # Just lets it escape
5
6   class CouchPotato:              # Define a type of human being that loves couchhanging
7       def walk (self, dog):       # The CouchPotato walks the dog, well, lets it go
8           print ('\nBugger off!') # \n means start on new line
9           dog.escape ()           # Just lets it escape
10
11  class Dog:                      # Define the dog species
12      def __init__ (self, sound): # Constructor, named __init__, accepts provided sound
13          self.sound = sound      # Stores accepted sound into self.sound field inside new dog
14
15      def _bark (self):           # Define _bark method, not part of interface of dog
16          print (self.sound)      # It prints the self.sound field stored inside this dog
17
```

```
18      def follow_me (self):        # Define escape method
19          print ('Walk behind')    # The dog walks one step behind the boss
20          self._bark ()            # It then calls upon its own _bark method
21          self._bark ()            # And yet again
22
23      def escape (self):           # Define escape method
24          print ('Hang head')      # The dog hangs his head
25          self._bark ()            # It then calls upon its own _bark method
26          self._bark ()            # And yet again
27
28  your_dog = Dog ('Wraff')         # Instantiate dog, provide sound "Wraff" to constructor
29  his_dog = Dog ('Howl')           # Instantiate dog, provide sound "Howl" to constructor
30
31  you = NatureLover ()             # Create yourself
32  your_friend = CouchPotato ()     # Create your friend
33
34  you.walk (your_dog)              # Interface: walk dog, implementation: going out together
35  your_friend.walk (his_dog)       # Interface: walk dog, implementation: sending dog out
```

Listing 2.1: **prog/nature_potato.py**

There's a bit more to this example program. Instances of class *Dog* are meant to be creatable anywhere in the code, in which case constructor *_ _ init _ _* will be called. And their *follow_ me* and *escape* methods are meant to be callable anywhere in the code as well. In other words, the *_ _ init _ _*, *follow_ me* and *escape* methods constitute the interface of class *Dog*, meant for public use. And then there's the *_ bark* method. As you can see it starts with *_* . By starting a method with a single *_*, Python programmers indicate that this method does not belong to the interface of the class, but is only meant for private use. In this case, *_ bark* is only called by methods *follow_ me* and *escape* of the *Dog* class itself. What exactly constitutes private use and what doesn't will be worked out further after explanation of Python's module concept.

It is also possible to prepend a _ to an attribute name, to indicate that this attribute is not part of the interface. But this is rarely done, since many programmers feel that attributes shouldn't be part of the interface anyhow. While there's certainly some sense in that, it is not a general truth. One should always be open to picking the best solution at hand, which sometimes means deviating from textbook wisdom or common practice. Of course following common practice has some advantages of its own, and when working in a team, the best solution may be a standard solution.

In this text, the convention of starting private members with a _ is not stricktly adhered to, as it is a Python-only habit. As such it is once again less practical if Python is combined with C++. Also at the beginning of a project it is often not completely clear what belongs to the interface an what not. You should give yourselve some leeway here. But when you cooperate with multiple developers on a Python-only project, using the _ convention may come in handy to make clear to your colleagues what the interface of a class is, from the source code alone.

TIP: Make your source code self-explanatory. No other documents should be required to understand it. This is because in the end the source code is the only thing that remains up to date through many years of ad hoc changes, company take-overs and job-hopping colleagues. Maybe it shouldn't be so, but trust me, it is.

(Sometimes even the source code of business-critical applications is lost. Ain't no cure against that.)

2.2 Modules

Python programs can be split into multiple source files called modules. Let's do that with the previous example program:

```
1  import bosses
2  import dogs
3
4  your_dog = dogs.Dog ('Wraff')    # Instantiate dog, provide sound "Wraff" to constructor
5  his_dog = dogs.Dog ('Howl')      # Instantiate dog, provide sound "Howl" to constructor
```

```
6
7  you = bosses.NatureLover ()          # Create yourself
8  your_friend = bosses.CouchPotato () # Create your friend
9
10 you.walk (your_dog)                   # Interface: walk dog, implementation: going out together
11 your_friend.walk (his_dog)            # Interface: walk dog, implementation: sending dog out
```

<div align="center">Listing 2.2: prog/dog_walker/dog_walker</div>

```
1  class NatureLover:              # Define a type of human being that loves nature
2      def walk (self, dog):      # The NatureLover walks the dog, really
3          print ('\nC\'mon!')    # \n means start on new line, \' means ' inside string
4          dog.follow_me ()       # Just lets it escape
5
6  class CouchPotato:             # Define a type of human being that loves couchhanging
7      def walk (self, dog):      # The CouchPotato walks the dog, well, lets it go
8          print ('\nBugger off!') # \n means start on new line
9          dog.escape ()          # Just lets it escape
```

<div align="center">Listing 2.3: prog/dog_walker/bosses.py</div>

```
1  class Dog:                     # Define the dog species
2      def __init__ (self, sound): # Constructor, named __init__, accepts provided sound
3          self.sound = sound     # Stores accepted sound into self.sound field inside new dog
4
5      def _bark (self):          # Define _bark method, not part of interface of dog
6          print (self.sound)     # It prints the self.sound field stored inside this dog
7
8      def escape (self):         # Define escape method
9          print ('Hang head')    # The dog hangs his head
10         self._bark ()          # It then calls upon its own _bark method
11         self._bark ()          # And yet again
12
13     def follow_me (self):      # Define escape method
14         print ('Walk behind')  # The dog walks one step behind the boss
15         self._bark ()          # It then calls upon its own _bark method
16         self._bark ()          # And yet again
```

<div align="center">Listing 2.4: prog/dog_walker/dogs.py</div>

As can be seen, program *dog_walker.py* imports modules *bosses.py* and *dogs.py*. By putting these modules in separate files, they could also be used in other programs than *dog_walker*. In order to make this type of reuse practical, it is important that the classes defined in *bosses.py* and *dogs.py* have a standard interface that doesn't change whenever any detail in the *Boss* or *Dog* classes changes. So the use of _ comes in handy here.

2.3   Polymorphism

In the previous example, class *NatureLover* and class *CouchPotato* have the same interface, namely only method *walk*. Since they have the same interface they may be used in similar ways, even though their implementation of the interface is different. Consider the following program:

```
1  import random   # One of Python's many standard modules
2
3  import bosses
4  import dogs
5
6  # Create a list of random bosses
7  humanBeings = []               # Create an emptpy list
8  for index in range (10):       # Repeat the following 10 times, index running from 0 to 9
9      humanBeings.append (       # Append a random HumanBeing to the list by
10         random.choice ((bosses.NatureLover, bosses.CouchPotato)) () # randomly selecting its class
```

```
11      )                                                        # and calling its contructor
12
13  # Let them all walk a new dog with an random sound
14  for humanBeing in humanBeings:   # Repeat the following for every humanBeing in the list
15      humanBeing.walk (            # Call implementation of walk method for that type of humanBeing
16          dogs.Dog (               # Construct a new dog as parameter to the walk method
17              random.choice (      # Pick a random sound
18                  ('Wraff', 'Wooff', 'Howl', 'Kaii', 'Shreek') # fom this tuple of sounds
19              )
20          )
21      )
```

Listing 2.5: prog/dog_walker/poly_walker.py

The *humanBeings* list contains objects of different classes: *NatureLover* and *CouchPotato*. Such a list is called polymorphic which means: "of many shapes". Since objects of class *NatureLover* and objects of class *CouchPotato* have the same interface, in this case only the *walk* method, this is not a problem, we can write *humanBeing.walk*, no matter whether we deal with a *NatureLover* or with a *CouchPotato*. But how they do this walking, the implementation, is different. A *NatureLover* will join the dog, a *CouchPotato* will let it go alone.

So providing a standard interface has more advantages than design flexibility alone. If objects of distinct classes have the same interface, they can easily be used without exactly knowing what particular object class you're dealing with. All elements of the *humanBeing* know how to *walk*. Except they do it differently. Since you don't have to know whether you're dealing with a *NatureLover* or a *CouchPotato* to call its *walk* method, you can store objects of both classes randomly in one object collection, in this case a list, without keeping track of their exact class. It is enough to know they all can *walk*. This careless way of handling different types of objects is called duck typing. If it walks like a duck, swims like a duck, sounds like a duck, let's treat it like a duck. A collection, e.g. a list, containing types of various classes is called a polymorphic object collection.

Objects, encapsulation, standard interfaces and polymorphism are important ingredients in the way of programming that was briefly mentioned in the introduction: object oriented programming. You now know what this means: programming in such a way that you deal with objects that contain attributes and methods. Objects naturally "know" things (attributes) and "can do" things (methods). The alternative would be to keep data and program statements completely separated, a way of working called procedural programming.

Chapter 3

A pinch of functional programming

3.1   List comprehensions

In the introduction the promise was made to teach you some functional programming as well. While this may sound a bit arbitrary and even careless, it is not. The aim of this course is to lead you straight to efficient programming habits, not to merely flood you with assorted facts. The combination of object oriented Programming and functional programming is especially powerful. To show a first glimpse of that power, lets slightly reformulate the previous example, using something called a list comprehension.

```
1   import random   # One of Python's many standard modules
2
3   import bosses
4   import dogs
5
6   # Create a list of random bosses
7   human_beings = [                                   # Start a so called list comprehension
8       random.choice (                                # Pick a random class
9           (bosses.NatureLover, bosses.CouchPotato) # out of this tuple
10      ) ()                                           # and call its constructor to instantiate an object
11      for index in range (10)                       # repeatedly, while letting index run from 0 to 9
12  ]                                                  # End the list comprehension, it will hold 10 objects
13
14  # Let them all walk a new dog with an random sound
15  for human_being in human_beings: # Repeat the following for every human being in the list
16      human_being.walk (           # Call implementation of walk method for that type of human being
17          dogs.Dog (               # Construct a dog as parameter to the walk method
18              random.choice (      # Pick a random sound
19                  ('Wraff', 'Wooff', 'Howl', 'Kaii', 'Shreek') # fom this tuple of sounds
20              )
21          )
22      )
```

Listing 3.1: prog/dog_walker/func_walker.py

While this example resembles the one before, there's a difference. In listing 2.5 you told the computer step by step what to do. In line 7 you first created an empty list, although that is not what you wanted in the end. And then you entered a so called loop, starting at line 8. Cycling through this loop ten times, new *HumanBeing* objects get appended to the list one by one, index running from 0 to 9.

In listing 3.1 you do not first create an empty list. You just specify directly what you want in the end, a list of random objects of class *HumanBeing*, one for each value of index where index running form 0 to 9.

Suppose you want a box with hundred chocolates. You could go to a shop and do the following:

Tell the shopkeeper to give you an empty box

```
While counting from 1 to 100:
        Tell the shopkeeper to put in a chocolate
```

This is the approach taken in listing 2.5. But you could also take a different approach:

```
Tell the shopkeeper to give you a box with 100 chocolates counted out for you.
```

This is the approach taken in listing 3.1.

To tell the shopkeeper chocolate by chocolate how to prepare a box of hundred chocolates is unnatural to most, except for extreme control freaks. But telling a computer step by step what to do IS natural to most programmers. There are a number of disadvantages to the control freak approach:

1. Telling the shopkeeper step by step how to fill the chocolate box keeps you occupied. It would be confusing to meanwhile direct the shopkeeper to fill a bag with cookies, cookie by cookie, because in switching between these tasks, you could easily lose track of the proper counts. A programmer would say you cannot multitask very well with the control freak approach.

2. Even doing one thing at a time, you would still have to remember how many chocolates are already in the box, also if you see your partner kissing your best friend through the shop window. A programmer would say you'd have to keep track of the state of the box. That's error prone, the shopkeeper has other options, he can e.g. measure the total weight of the box, which doesn't require remembering anything.

3. The chocolates are put into the box one by one, a time consuming process. The shopkeeper cannot work in parallel with his assistant, each putting fifty cookies in the box, being ready twice as fast.

In principle the Functional Programming approach is suitable to alleviate this problems. It allows for:

1. Multi-tasking, that is switching between multiple tasks on one processor without confusion, since you only have to specify the end result.

2. Stateless programming, which helps avoiding errors that emerge when at any point program state is not what you assume it to be.

3. Multi-processing, that is performing multiple tasks in parallel on multiple processors.

While standard Python does currently not fully benefit from these advantages, learning this way of programming is a good investment in the future, since having multiple processors in a computer is rapidly becoming the norm. Apart from that, once you get used to things like list comprehensions, they are very handy to work with and result in compact but clear code.

3.2 Transforming all elements of a list

Suppose we fill a list with numbers and from that want to obtain a list with the squares of these numbers. The functional way to do this is:

```
1  even_numbers = [2 * (index + 1) for index in range (10)]        # Create [2, 4, ..., 20]
2  print ('Even numbers:', even_numbers)
3
4  squared_numbers = [number * number for number in even_numbers] # Compute list of squared numbers
5  print ('Squared numbers:', squared_numbers)
```

<div align="center">Listing 3.2: prog/func_square.py</div>

The functional way requires slightly less code than the non-functional way, but that's just a bonus. In the beginning you may prefer the non-functional way, since its explicit steps make it easier to debug. But that will probably shift once you gain experience, since the conciseness of the functional way helps preventing bugs in the first place.

```
1  even_numbers = []
2  for index in range (10):
3      even_numbers.append (2 * (index + 1))
4  print ('Even numbers:', even_numbers)
5
6  squared_numbers = []
7  for even_number in even_numbers:
8      squared_numbers.append (even_number * even_number)
9  print ('Squared numbers:', squared_numbers)
```

<div align="center">Listing 3.3: prog/nonfunc_square.py</div>

3.3   Selecting certain elements from a list

Suppose we have a list with names and from that want to obtain a list with only those names starting with a 'B'. The functional way to do this is:

```
1  all_names = ['Mick', 'Bonny', 'Herbie', 'Bono', 'Ella', 'Ray', 'Barbara']   # Create name list
2  print ('All names:', all_names)
3
4  filtered_names = [name for name in all_names if name [0] == 'B']              # Select names starting with B
5  print ('Filtered names:', filtered_names)
```

<div align="center">Listing 3.4: prog/func_select.py</div>

The non functional way again needs more words:

```
1  all_names = ['Mick', 'Bonny', 'Herbie', 'Bono', 'Ella', 'Ray', 'Barbara']
2  print ('All names:', all_names)
3
4  filtered_names = []
5  for name in all_names:
6      if name [0] == 'B':
7          filtered_names.append (name)
8  print ('Filtered names:', filtered_names)
```

<div align="center">Listing 3.5: prog/nonfunc_select.py</div>

3.4   Computing sum from a list

Suppose we have a list with numbers and from that want to obtain the sum of that numbers. The functional way to do this is:

```
1  even_numbers = [2 * (index + 1) for index in range (10)] # Create [2, 4, 6, ..., 20]
2  print ('Even numbers:', even_numbers)
3
4  total = sum (even_numbers)                                # Compute sum
5  print ('Total:', total)
```

<div align="center">Listing 3.6: prog/func_sum.py</div>

The non functional way is:

```
1  even_numbers = []
2  for index in range (10):
3      even_numbers.append (2 * (index + 1))
4  print ('Even numbers:', even_numbers)
5
6  total = 0
7  for even_number in even_numbers:
```

```
8        total += even_number
9    print ('Total:', total)
```

<div align="center">Listing 3.7: prog/nonfunc_sum.py</div>

3.5   Free functions and lambda expressions

Whereas methods are part of a class, free functions can be defined anywhere. They don't have a self parameter, and are not preceded by an object and a dot, when called.

```
1    def add (x, y):  # Free function, defined outside any class, no self parameter
2        return x + y # It may return a result, but a method could do that also
3
4    def multiply (x, y):
5        return x * y
6
7    sum = add (3, 4)  # Call the first free function
8
9    print ('3 + 4 =', sum)
10   print ('3 * 4 =', multiply (3, 4)) # Call the second free function
```

<div align="center">Listing 3.8: prog/free_functions.py</div>

It is also possible to define free functions that don't have a name. These are called lambda functions, and are written in a shorthand way, as can be seen in the following program:

```
1    functions = [
2        lambda x, y: x + y,     # Shorthand for anonymous add function
3        lambda x, y: x * y      # Shorthand for anonymous multiply function
4    ]
5
6
7    sum = functions [0] (3, 4) # Call the first lambda function
8
9    print ('3 + 4 =', sum)
10   print ('3 * 4 =', functions [1] (3, 4)) # Call the second lambda function
```

<div align="center">Listing 3.9: prog/lambdas.py</div>

The following program makes use of several free functions to compute the area of squares and the volume of cubes from a list of side lengths:

```
1    def power (x, n):   # Define free function, outside any class, no self parameter
2        result = x
3        for i in range (n - 1):     # Note that i runs from 0 to n - 2
4            result *= x             # so this is performed n - 1 times
5        return result
6
7    test = power (2, 8)             # Call free function, no object before the dot
8    print ('test:', test)
9
10   def area (side):               # Define free function, computes area of square
11       return power (side, 2)     # Call power function to do the job
12
13   def volume (side):             # Define free function, computes volume of cube
14       return power (side, 3)     # Call power function to do the job
15
16   def apply (compute, numbers):   # Define free function that applies compute to numbers
17       return [compute (number) for number in numbers] # Return list of computed numbers
18
19   sides = [1, 2, 3]              # List of side lengths
```

```
20  areas = apply (area, sides)      # Let apply compute areas by supplying area function
21  volumes = apply (volume, sides) # Let apply compute volumes by supplying volume function
22
23  print ('sides:', sides)
24  print ('areas:', areas)
25  print ('volumes:', volumes)
```

<div align="center">Listing 3.10: prog/free_functions2.py</div>

Take a good look at the *apply* function. Its first formal parameter, *compute*, is a free function, that will then be applied to each element of the second formal parameter, *numbers*, that is a list. Since the *area* and *volume* functions are only used as actual parameter to *apply*, they can also be anonymous, as is demonstrated in the program below.

```
1   def power (x, n):    # Define free function, outside any class, no self parameter
2       result = x
3       for i in range (n - 1):      # Note that i runs from 0 to n - 2
4           result *= x              # so this is performed n - 1 times
5       return result
6
7   test = power (2, 8)              # Call free function, no object before the dot
8   print ('test:', test)
9
10  def apply (operation, numbers):   # Define free function that applies compute to numbers
11      return [operation (number) for number in numbers] # Return list of computed numbers
12
13  sides = [1, 2, 3]
14
15  areas = apply (lambda side: power (side, 2), sides)   # Define area function and pass it to apply
16  volumes = apply (lambda side: power (side, 3), sides) # Define volume function and pass it to apply
17
18  print ('sides:', sides)
19  print ('areas:', areas)
20  print ('volumes:', volumes)
```

<div align="center">Listing 3.11: prog/lambdas2.py</div>

It is quite possible to give a lambda function a name, like this:

```
1   add = lambda x, y: x + y    # Name add now referes to the lambda function
2   print (add (7, 8))          # and you can call it via that name
```

<div align="center">Listing 3.12: prog/named_lambda.py</div>

Chapter 4

Inheritance

4.1 Implementation inheritance

Classes can inherit methods and attributes from other classes. The class that inherits is called descendant class or derived class. The class that it inherits from is called ancestor class or base class. Look at the following example:

```
 1  class Radio:
 2      def __init__ (self, sound):
 3          self.sound = sound
 4
 5      def play (self):
 6          print ('Saying:', self.sound)
 7          print ()
 8
 9  class Television (Radio):
10      def __init__ (self, sound, picture):
11          Radio.__init__ (self, sound)
12          self.picture = picture
13
14      def play (self):
15          self._show ()
16          Radio.play (self)
17
18      def _show (self):
19          print ('Showing:', self.picture)
20
21  tuner = Radio ('Good evening, dear listeners')
22  carradio = Radio ('Doowopadoodoo doowopadoodoo')
23  television = Television ('Here is the latest news', 'Newsreader')
24
25  print ('TUNER')
26  tuner.play ()
27
28  print ('CARRRADIO')
29  carradio.play ()
30
31  print ('TELEVISION')
32  television.play ()
```

Listing 4.1: prog/radio_vision.py

In line 15 the *play* method of class *Television* calls the *show* method of the same class. In line 16 it calls the *play* method of class *Radio*. Compare 15 to 16. In line 15 *self* is placed before the dot. Since in line 16 the *Radio* class occupies the place before the dot, *self* is passed as first parameter there. The same holds for line 11, where the

constructor of *Television* calls the constructor of *Radio*. Although this class hierarchy is allowed, an experienced designer would not program it like this.

1. A television is not merely some special type of radio with a screen glued on. It has become a totally different device altogether.

2. A radio may have facilities that a television hasn't, e.g. an analog tuning dial. Televisions would inherit that, but it would serve no purpose and just be confusing.

3. It would probably be more flexible to have class *Radio* and class *Television* both inherit from an abstract class: *Microelectronics*. Abstract classes are classes that serve as a general category, but of which there are no objects. The objects themselves are always specialized, so either of class *Radio* or of class *Television*. Abstract base classes are handy to specify an interface without making early choices about how that interface is implemented.

4.2   Interface inheritance

An example of a class hierarchy with an abstract class at the top is given in the following program:

```
 1  import time
 2
 3  class HumanBeing:
 4      def __init__ (self, name):
 5          self.description = name + ' the ' + self.__class__.__name__.lower ()
 6
 7      def walk (self):
 8          self._begin_walk ()
 9          for i in range (5):
10              print (self.description, 'is counting', i + 1)
11          self._end_walk ()
12          print ()
13
14  class NatureLover (HumanBeing):
15      def _begin_walk (self):
16          print (self.description, 'goes to the park')
17
18      def _end_walk (self):
19          print (self.description, 'returns from the park')
20
21
22  class CouchPotato (HumanBeing):
23      def _begin_walk (self):
24          print (self.description, 'lets the dino escape')
25
26      def _end_walk (self):
27          print (self.description, 'catches the dino')
28
29  class OutdoorSleeper (NatureLover, CouchPotato):
30      def _begin_walk (self):
31          NatureLover._begin_walk (self)
32          CouchPotato._begin_walk (self)
33          print (self.description, 'lies on the park bench')
34
35      def _end_walk (self):
36          print (self.description, 'gets up from the park bench')
37          CouchPotato._end_walk (self)
38          NatureLover._end_walk (self)
39
40  for human_being in (NatureLover ('Wilma'), CouchPotato ('Fred'), OutdoorSleeper ('Barney')):
```

```
41      human_being.walk ()
```

<div align="center">Listing 4.2: prog/nature_sleeper.py</div>

Class *HumanBeing* is abstract, since it don't have the methods *begin_walk* and *end_walk*, that are called in *walk* in line 8 and 11. So it's no use creating objects of that class, since they don't know how to *walk*. All other classes inherit the *walk* method, so they don't have to define a *walk* method of their own. Since they all inherit *walk*, they are guaranteed to support the it in their interface. But they define their own specialized implementation of *begin_walk* and *end_walk*. Note that the *begin_walk* and *end_walk* of *OutdoorSleeper* call upon the *begin_walk* and *end_walk* of *NatureLover* and *CouchPotato* to do their job.

Be sure to follow every step of the example program above, since it contains important clues to an object oriented programming style called "Fill in the blanks" programming: Specify as much as you can high up in the class hierarchy (method *walk*), and only fill in specific things (methods *begin_walk* and *end_walk*) in the descendant classes. It is with "Fill in the blanks" programming that true object orientation starts to deliver. While this isn't visible in a small example, "Fill in the blanks" programming makes the source code of your class hierarchy shrink while gaining clarity, a sure sign that you're on the right track. "Fill in the blanks" programming is one place where the DRY principle of programming pays of: Don't Repeat Yourself. If you can specify behaviour in an ancestor class, why specify it over and over again in the descendant classes. If you follow the DRY principle, your code becomes more flexible, because changes in behaviour only have to be made in one single place, avoiding the risk of inconsistent code.

Apart from following the DRY principle, the fact that interface methods defined higher up in the class hierarchy are automatically there in derived classes, is in itself one of the most powerful features of inheritance: Having objects of different subclasses all inherit the same standard interface contributes to design flexibility, since these objects become highly interchangeable, even though their behaviour is different.

As a bonus the size of the code USING these objects also shrinks, since it only has to deal with one type of interface. When switching from procedural to object oriented programming, it is not uncommon to see the source code shrink with a factor five. While briefness never is a goal in itself, it is a very important contribution to clarity: What isn't there doesn't have to be understood. The difference between having to get your head around twenty pages of source code as opposed to a hundred may veryAs an exampl we will be using a game engine library called Pyglet. well be crucial in successfully understanding the work of a colleague, or your own work of several years back, for that matter.

4.3 Inheriting from library classes

In section 4.2 the concept of modules was explained. There are many ready-made modules available for Python. Some are distributed with Python itself. Others are part of so called libraries. A library is a collection of modules that together enable you to make a specific category of programs without coding all the details yourself. For Python there are lots of libraries available to help you build almost any type of computer program. The majority of these libraries are available on https://pypi.python.org/pypi. An important part of the power of Python lies in the fact that so many libraries are available for it, most of them for free. As an example we will be using a library called Pyglet. Pyglet provides basic building blocks, like sprites (small moving objects), text labels and windows to make simulations and games. One way to use these building blocks is to inherit from them. Since the purpose of inheriting in this case is to use their behaviour in our own application, this is an example of implementation inheritance. The following program uses inheritance from Pyglet classes to make some sprites randomly drift apart. Study the source code and experiment with it, making small changes and see what the effect is. Consult the Pyglet documentation where necessary.

```
1   import os
2   import random
3
4   import pyglet
5   from pyglet.gl import *
6
7   class Sprite (pyglet.sprite.Sprite):
8       def __init__ (self, fileName):
9           image = pyglet.image.load (fileName)              # Load it as an image
```

```
10          image.anchor_x = image.width // 2              # Lay its coordinate reference
11          image.anchor_y = image.height // 2             # in its middle
12          pyglet.sprite.Sprite.__init__ (self, image, 0, 0)   # Initialize ancestor
13
14  class Window (pyglet.window.Window):
15      orthoWidth = 1000
16      orthoHeight = 750
17      maxSpeed = 100      # / s
18
19      def __init__ (self):
20          pyglet.window.Window.__init__ (                # Initialize ancestor
21              self, 640, 480, resizable = True, visible = False, caption = 'Drifting sprites'
22          )
23
24          self.on_resize = self.resize                   # Called if window is resized
25          self.on_draw = self.draw                       # Called if window has to be redrawn
26
27          self.set_location (                            # Put window on middle of its screen
28              (self.screen.width - self.width) // 2,
29              (self.screen.height - self.height) // 2
30          )
31
32          self.clear ()                                  # Clear window
33          self.set_visible (True)                        # Show window once it's cleared
34
35          self.sprites = []
36          for fileName in os.listdir (                   # For each file
37              os.path.dirname (os.path.realpath(__file__))   # in the folder of this source file
38          ):
39              if fileName.endswith ('.png'):             #   If its a .png file
40                  self.sprites.append (Sprite (fileName))   #      Append to the sprite list
41
42          pyglet.clock.schedule_interval (               # Install update callback that
43              self.update, 1/20.                         # will be called 60 times per s
44          )
45          pyglet.app.run ()                              # Start pyglet engine
46
47      def resize (self, width, height):                  # When the user resizes the window
48          glViewport (0, 0, width, height)               # Tell openGL window size
49
50          glMatrixMode (GL_PROJECTION)                    # Work with projection matrix
51          glLoadIdentity ()                              # Start with identity matrix
52          glOrtho (                                      # Adapt it to orthographic projection
53              -self.orthoWidth // 2, self.orthoWidth // 2,   # Lay origin in the middle
54              -self.orthoHeight // 2, self.orthoHeight // 2,
55              -1, 1
56          )
57
58          glMatrixMode (GL_MODELVIEW)                     # Work with model matrix
59          glLoadIdentity ()                              # No transforms
60
61      def update (self, deltaT):
62          for sprite in self.sprites:                    # For every sprite in the sprite lst
63              sprite.x += random.uniform (-1, 1) * self.maxSpeed * deltaT # Move it
64              sprite.y += random.uniform (-1, 1) * self.maxSpeed * deltaT
65
66      def draw (self):
67          self.clear ()
68
69          for sprite in self.sprites:                    # For every sprite in the sprite list
70              sprite.draw ()                             # Draw it
71
```

```
72  Window ()
```

Listing 4.3: intro_pyglet/drifting_sprites.py

Chapter 5

Objects and the real world

5.1 Object oriented modeling

One way or another, most computer programs represent something in the real world. Example programs in tutorials are often about administration, the objects representing real world things like companies, departments, employees and contracts. But writing administrative software is just one way to capture reality and put it into a computer. Dynamic modeling of physics, like applied in simulations and games, is another way. An employee would not be modeled by its name, address and salary, but rather by a moving on-screen avatar (stylized image of a person) controlled by a game paddle. Simulations and games are what we'll use as examples in this text. Having objects represent things in the real world, either in an administrative way or by means of simulation is called object oriented modeling, and your eventual computer program is said to be a 'model' of some aspect of the real world ('application domain').

In short, object oriented modeling consists of the following steps:the descendants

1. Analysis: Find out which type of things play a role in the part of the real world that your program is about (the 'application domain') and how they relate to each other. Represent each relevant type of thing by a class. These classes are called 'domain classes'. If a B object denotes a special type of A object, let class B inherit from class A. If a C object refers to a D object, let C have an attribute of class D. Note that attributes merely REFER to an object, they are not the object itself. So two objects may refer to each other, both holding a reference to the other as an attribute. The result of this first step is called a domain model.

2. Design: Try to generalize the concepts in your application domain, in order to come up with a sensible inheritance hierarchy, e.g. looking for common interfaces or common functionality. This will lead to the addition of so called design classes, as opposed to the domain classes that result from step 1. Your domain model has now evolved into a design model.

3. Programming: Elaborate your code to put whole thing to work, in our case in Python. Adjust your class hierarchy as your understanding of the problem at hand grows. Your program will be a working object oriented model of a part or aspect of the real world.

TIP: The steps usually ovelap. It is very efficient to inventorize domain classes and design a class hierarchy using Python syntax right from the start, adding permanent comments to document why you took certain design decisions. Some people like to view the relations between e.g. classes in a graphical way. There exist several tools that generate diagrams from Python source code. Don't go the opposite way: generating source code from diagrams. This only works in the simplest of situations and is too restrictive in the long run. Some people limit the use of the term 'Domain Modeling' to step 1. In my view the resulting computer program itself is the model we're eventually after.

5.2 Pong, the object oriented way

Let's look at the humblest of all computer games: Pong.

1. Analysis: The application domain to be modeled is the real world game of table tennis. Things that play an important role in that application domain are paddles, a ball, a scoreboard and the notion of a game. To play the game, the paddles can be moved. The ball can bounce against the paddles, which changes its direction as dictated by physics. Whenever the ball goes out, the score is adapted. To represent the application domain, we need one object of class *Ball* and two objects of class *Paddle* an object of class *Scoreboard*, and, less obvious since you can not touch or eat it: an object of class *Game*. And we'll have to establish relations between the objects and sometimes between the classes.

2. Design: Each game has attributes (not in the programming sense of the word but in the every day sense). For example the main attributes required for skiing ar skis, sticks and a helmet. In our game the *Paddle* objects, *Ball* object and *Scoreboard* object are all attributes, i.e. they are a specialisation of class *Attribute*. Whereas the scoreboard stays in place, the paddles and ball move around the screen. Such small moving objects are called sprites, so instances of class *Sprite*. Note that *Attribute* and *Sprite* are no domain classes. Rather they are added during design to catch commonalities in the domain classes resulting from step 1. Such classes are called design classes. As we will see, all descendants of class *Attribute* share the same interface, enabling polymorphism.

3. Programming: We need to elaborate those classes to make the program work rather than just sit there. Paddles need to be movable via the keyboard. The ball has to bounce against the paddles and the wall. If the ball goes out,the score has to be adapted. Partially we'll code this functionality ourselves, partially it will be taken from the Pyglet game engine library downloaded from Pypi.

5.3   Step 1, analysis: Drawing up the domain model

In step 1, take stock of the domain classes to obtain the following valid Python program:

```
1  class Paddle:
2          pass     # Placeholder, no code yet
3
4  class Ball:
5          pass
6
7  class Scoreboard:
8          pass
9
10 class Game:
11         pass
```

Listing 5.1: pong/pong1.py

As the second part of step 1, inventorize the relations between objects or classes:

- Each *Paddle* instance needs a *game* attribute that is a reference to game that it's part of. It also needs an index attribute that indicates wether it is the left or the right paddle.

- The *Ball* instance needs a *game* attribute that is a reference to the game that it's part of.

- The *Scoreboard* instance needs a *game* attribute that is a reference to the game that it's part of.

- The *Game* needs a *paddles* attribute that is a list of references to the paddles of the game, a *ball* attribute that is a reference to the ball of the game and a *scoreboard* attribute that is a reference to the scoreboard of the game.

Formulate these relations in Python, to obtain the following code:

```
1  class Paddle:
2      def __init__ (self, game, index):
3          self.game = game     # A paddle knows which game object it's part of
4          self.index = index  # A paddle knows its index, 0 (left) or 1 (right)
5
```

```
 6  class Ball:
 7      def __init__ (self, game):
 8          self.game = game    # A ball knows which game object it's part of
 9
10  class Scoreboard:
11      def __init__ (self, game):
12          self.game = game    # A scoreboard knows which game object it's part of
13
14  class Game:
15      def __init__ (self):
16          self.paddles = [Paddle (self, index) for index in range (2)]    # Pass game as parameter self
17          self.ball = Ball (self)
18          self.scoreboard = Scoreboard (self)
19
20  game = Game ()  # Create game, which will in turn create its paddles, ball and scoreboard
```

Listing 5.2: pong/pong2.py

Take your time to study listing 5.2. Make sure you understand each and every line of it before proceeding. Run it to make sure it is a again a valid Python program, even though it doesn't yet do anything. After every step we take, the intermediate result should be a valid Python program. Check that.

TIP: Use this approach also with your own programs. Make sure you always have a runnable program. If you make changes, keep the last version until the new one runs correctly. It is always easier to find a bug departing from a running version than from a version that does not run at all. Regularly store versions and keep them around forever. I've programmed the larger part of my life, but all the source code will easily fit on one memory stick. A simple, robust way to label versions is by prepending the date and a subnumber, e.g. pong_y15m11d26_2. Of course you can also use a version control system like GitHub but never completely rely on it, use multiple backup strategies in parallel. Loosing hard-fought source code is very frustrating. Store backups outside your computer, preferably in a physically separate location.

5.4   Step 2, design: Turning the domain model into a design model

Continue with step 2, adding design classes to capture commonalities in the domain classes. This is where it gets interesting. There's no single recipe to do this properly. Be prepared to retrace your steps and explore alternative solutions. Raising the bar here pays off manifold once you go to step 3.

In this case the design classes (as opposed to domain classes) *Attribute* and class *Sprite* are added. The term 'attribute' used here has nothing to do with programming. Things you need for a game or sport are called its attributes in everyday language. The term 'sprite' on the other hand, is a programming term. A sprite a small moving object on the screen.

```
 1  class Attribute:      # Attribute in the gaming sense of the word, rather than of an object
 2      def __init__ (self, game):
 3          self.game = game                     # Done in a central place now
 4          self.game.attributes.append (self)  # Insert each attribute into a list held by the game
 5
 6  class Sprite (Attribute):   # Here, a sprite is an recangularly shaped attribute that can move
 7      def __init__ (self, game, width, height):
 8          self.width = width
 9          self.height = height
10          Attribute.__init__ (self, game)        # Call parent constructor to set game attribute
11
12  class Paddle (Sprite):
13      width = 10          # Constants are defined per class, rather than per individual object
14      height = 100        # since they are the same for all objects of that class
15                                  # They are defined BEFORE the __init__, not INSIDE it
16      def __init__ (self, game, index):
17          self.index = index  # Paddle knows its player index, 0 == left, 1 == right
```

```
18              Sprite.__init__ (self, game, self.width, self.height)
19
20  class Ball (Sprite):
21      side = 8
22
23      def __init__ (self, game):
24          Sprite.__init__ (self, game, self.side, self.side)
25
26  class Scoreboard (Attribute):    # The scoreboard doesn't move, so it's an attribute but not a sprite
27          pass
28
29  class Game:
30      def __init__ (self):
31          self.attributes = []     # All attributes will insert themselves into this polymorphic list
32          self.paddles = [Paddle (self, index) for index in range (2)]     # Pass game as parameter self
33          self.ball = Ball (self)
34          self.scoreboard = Scoreboard (self)
35
36  game = Game ()   # Create and run game
```

<div align="center">

Listing 5.3: pong/pong3.py

</div>

While this program starts to really look like something, still it doesn't do anything. To make it work we'll use facilities from the Pyglet library. Pyglet has a Label class that can be used to draw the scoreboard and it has a Sprite class of its own, *pyglet.sprite.Sprite*, to animate our sprites. We'll put a *pygletSprite* attribute of that class into our own *Sprite* class, to add 'sprity' behaviour to it. And we'll put several *Pyglet.text.Label* instances in our *Scoreboard* class, so that it can indeed show scores.

But first we take a high-level look. It may seem that continuously repeating the the following steps would be sufficient:

1. Compute the position of the paddles and the ball from the keyboard input, their previous position, their velocity and the elapsed time.

2. Check for collisions between ball, paddles and walls and adapt the ball and paddle velocity and position to these *collisions*.

But when these three steps are executed sequentially, a problem occurs. Pyglet sprites have an x and a y coordinate. As soon as these coordinates are set, the sprite can be shown at that location. 'Can be', because we have no control over exactly WHEN a sprite will be shown on the screen. The process that actually displays the sprites is asynchroneous. This means that it runs in parallel with our own code, displaying sprites at unexpected moments. So it may very well be that a sprite is drawn after step 1, when the position has not yet been corrected for collisions. This may result in the ball briefly flying right through the paddles. To prevent this, repeat the following steps instead:

1. *predict*: Compute the predicted velocity and position of the paddles and the ball from the keyboard input, their previous position and velocity and the elapsed time, but store these new values in the vX, vY, x and y attributes of our own *Sprite* instances. Do not yet overwrite the x and y attributes of the *pygletSprite* attributes of our *Sprite* instances.

2. *interact*: Check for collisions between ball, paddles and walls and adapt the ball and paddle velocity and position to these collisions, correcting the values of the vX, vY, x and y attributes of our sprites.

3. *commit*: After all corrections have been done, copy x and y from each *Sprite* instance to its *pygletSprite* attribute, to be rendered to the screen by Pyglet.

Before entering the *predict, interact, commit* cycle, it must be possible to *reset* the game attributes: paddles and ball in start position, score 0 - 0. All of this is incorporated in the next version of pong:

```
 1  class Attribute:     # Attribute in the gaming sense of the word, rather than of an object
 2      def __init__ (self, game):
 3          self.game = game                    # Done in a central place now
 4          self.game.attributes.append (self)  # Insert each attribute into a list held by the game
 5
 6          # ========= Standard interface starts here
 7
 8      def reset (self):
 9          pass
10
11      def predict (self):
12          pass
13
14      def interact (self):
15          pass
16
17      def commit (self):
18          pass
19
20          # ========= Standard interface ends here
21
22  class Sprite (Attribute):   # Here, a sprite is an rectangularly shaped attribute that can move
23      def __init__ (self, game, width, height):
24          self.width = width
25          self.height = height
26          Attribute.__init__ (self, game)     # Call parent constructor to set game attribute
27
28  class Paddle (Sprite):
29      width = 10       # Constants are defined per class, rather than per individual object
30      height = 100     # since they are the same for all objects of that class
31                       # They are defined BEFORE the __init__, not INSIDE it
32      def __init__ (self, game, index):
33          self.index = index  # Paddle knows its player index, 0 == left, 1 == right
34          Sprite.__init__ (self, game, self.width, self.height)
35
36  class Ball (Sprite):
37      side = 8
38
39      def __init__ (self, game):
40          Sprite.__init__ (self, game, self.side, self.side)
41
42  class Scoreboard (Attribute):   # The scoreboard doesn't move, so it's an attribute but not a sprite
43      pass
44
45  class Game:
46      def __init__ (self):
47          self.attributes = []    # All attributes will insert themselves into this polymorphic list
48
49          self.paddles = [Paddle (self, index) for index in range (2)]
50          self.ball = Ball (self)
51          self.scoreboard = Scoreboard (self)
52
53          for attribute in self.attributes:
54              attribute.reset ()
55
56      def update (self):                       #   To be called cyclically by game engine
57          for attribute in self.attributes:    #   Compute predicted values
58              attribute.predict ()
59
60          for attribute in self.attributes:    #   Correct values for bouncing and scoring
61              attribute.interact ()
62
```

```
63          for attribute in self.attributes:   #   Commit them to game engine for display
64              attribute.commit ()
65
66  game = Game ()  # Create and run game
```

The methods *reset*, *interact*, *cycle* and *commit* constitute the interface inherited by all subclasses of class *Attribute*. Since all attributes have the same interface functions, these functions can be called in a uniform way by looping over the *attributes* list of the *Game* instance, as can be seen in line 53 - 64 of listing 5.4. Completing step 2, we have now set up the complete program structure without any reference to the Pyglet library.
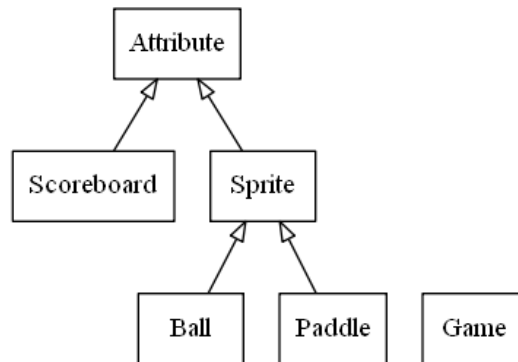

Figure 5.1: Class diagram of pong4.py, showing inheritance

5.5 Step 3, programming: Working out program logic and laying the connection with Pyglet

Now that the overall program structure stands, we will proceed with step 3, programming, by adding specific implementations of the *reset*, *interact*, *cycle* and *commit* functions to the classes derived from *Attribute*, to account for keyboard interaction, motion and collisions As can be seen in the listings, *e.g.* *pyglet.sprite.Sprite* and *Pyglet.text.Label* instances are added, laying the connection with the Pyglet library. You'll find the details about the elements taken from Pyglet in its documentation.

A very important point is that the overall program structure doesn't change much during step 3. If you understood listing 5.4, you are in a good position to gain understanding of listing 5.5, guided by this overall structure. The activity of designing largely boiled down to establishing the overall program structure. After that, filling in the details becomes doable, because everything has its natural place in this structure. The result below is a typical, be it small, Object Oriented application. Study listing 5.5 carefully and experiment with it, making small changes and running the program to see what the effect is.

```
1  import math
2  import random
3  import inspect
4
5  import pyglet
6  from pyglet.gl import *
7
8  orthoWidth = 1000
9  orthoHeight = 750
10 fieldHeight = 650
11
12 class Attribute:     # Attribute in the gaming sense of the word, rather than of an object
13     def __init__ (self, game):
14         self.game = game                    # Attribute knows game it's part of
```

```
15            self.game.attributes.append (self)   # Game knows all its attributes
16            self.install ()                      # Put in place graphical representation of attribute
17            self.reset ()                        # Reset attribute to start position
18
19      def reset (self):        # Restore starting positions or score, then commit to Pyglet
20            self.commit ()      # Nothing to restore for the Attribute base class
21
22      def predict (self):
23            pass
24
25      def interact (self):
26            pass
27
28      def commit (self):
29            pass
30
31  class Sprite (Attribute):    # Here, a sprite is an attribute that can move
32      def __init__ (self, game, width, height):
33            self.width = width
34            self.height = height
35            Attribute.__init__ (self, game)
36
37      def install (self):      # The sprite holds a pygletSprite, that pyglet can display
38            image = pyglet.image.create (
39                self.width,
40                self.height,
41                pyglet.image.SolidColorImagePattern ((255, 255, 255, 255))  # RGBA
42            )
43
44            image.anchor_x = self.width // 2    # Middle of image is reference point
45            image.anchor_y = self.height // 2
46
47            self.pygletSprite = pyglet.sprite.Sprite (image, 0, 0, batch = self.game.batch)
48
49      def reset (self, vX = 0, vY = 0, x = orthoWidth // 2, y = fieldHeight // 2):
50            self.vX = vX          # Speed
51            self.vY = vY
52
53            self.x = x            # Predicted position, can be commit, no bouncing initially
54            self.y = y
55
56            Attribute.reset (self)
57
58      def predict (self):      # Predict position, do not yet commit, bouncing may alter it
59            self.x += self.vX * self.game.deltaT
60            self.y += self.vY * self.game.deltaT
61
62      def commit (self):       # Update pygletSprite for asynch draw
63            self.pygletSprite.x = self.x
64            self.pygletSprite.y = self.y
65
66  class Paddle (Sprite):
67      margin = 30 # Distance of paddles from walls
68      width = 10
69      height = 100
70      speed = 400 # / s
71
72      def __init__ (self, game, index):
73            self.index = index  # Paddle knows its player index, 0 == left, 1 == right
74            Sprite.__init__ (self, game, self.width, self.height)
75
76      def reset (self):        # Put paddle in rest position, dependent on player index
```

```
77          Sprite.reset (
78              self,
79              x = orthoWidth - self.margin if self.index else self.margin,
80              y = fieldHeight // 2
81          )
82
83      def predict (self): # Let paddle react on keys
84          self.vY = 0
85
86          if self.index:  # Right player
87              if self.game.keymap [pyglet.window.key.K]:  # Letter K pressed
88                  self.vY = self.speed
89              elif self.game.keymap [pyglet.window.key.M]:
90                  self.vY = -self.speed
91          else:           # Left player
92              if self.game.keymap [pyglet.window.key.A]:
93                  self.vY = self.speed
94              elif self.game.keymap [pyglet.window.key.Z]:
95                  self.vY = -self.speed
96
97          Sprite.predict (self)   # Do not yet commit, paddle may bounce with walls
98
99      def interact (self):         # Paddles and ball assumed infinitely thin
100          # Paddle touches wall
101          self.y = max (self.height / 2, min (self.y, fieldHeight - self.height / 2))
102
103          # Paddle hits ball
104          if (
105              (self.y - self.height // 2) < self.game.ball.y < (self.y + self.height // 2)
106              and (
107                  (self.index == 0 and self.game.ball.x < self.x) # On or behind left paddle
108                  or
109                  (self.index == 1 and self.game.ball.x > self.x) # On or behind right paddle
110              )
111          ):
112              self.game.ball.x = self.x                 # Ball may have gone too far already
113              self.game.ball.vX = -self.game.ball.vX  # Bounce on paddle
114
115              speedUp = 1 + 0.5 * (1 - abs (self.game.ball.y - self.y) / (self.height // 2)) ** 2
116              self.game.ball.vX *= speedUp              # Speed will increase more if paddle near centre
117              self.game.ball.vY *= speedUp
118
119
120  class Ball (Sprite):
121      side = 8
122      speed = 300 # / s
123
124      def __init__ (self, game):
125          Sprite.__init__ (self, game, self.side, self.side)
126
127      def reset (self):   # Launch according to service direction with random angle offset from horizontal
128          angle =  (
129              self.game.serviceIndex * math.pi     # Service direction
130              +
131              random.choice ((-1, 1)) * random.random () * math.atan (fieldHeight / orthoWidth)
132          )
133
134          Sprite.reset (
135              self,
136              vX = self.speed * math.cos (angle),
137              vY = self.speed * math.sin (angle)
138          )
```

```
139
140      def predict (self):
141          Sprite.predict (self)        # Integrate velocity to position
142
143          if self.x < 0:              # If out on left side
144              self.game.scored (1)    #   Right player scored
145          elif self.x > orthoWidth:
146              self.game.scored (0)
147
148          if self.y > fieldHeight:    # If it hit top wall
149              self.y = fieldHeight     #   It may have gone too far already
150              self.vY = -self.vY       #   Bounce
151          elif self.y < 0:
152              self.y = 0
153              self.vY = -self.vY
154
155  class Scoreboard (Attribute):
156      nameShift = 75
157      scoreShift = 25
158
159      def install (self): # Graphical representation of scoreboard are four labels and a separator line
160          def defineLabel (text, x, y):
161              return pyglet.text.Label (
162                  text,
163                  font_name = 'Arial', font_size = 24,
164                  x = x, y = y,
165                  anchor_x = 'center', anchor_y = 'center',
166                  batch = self.game.batch
167              )
168
169          defineLabel ('Player AZ', 1 * orthoWidth // 4, fieldHeight + self.nameShift)     # Player name
170          defineLabel ('Player KM', 3 * orthoWidth // 4, fieldHeight + self.nameShift)
171
172          self.playerLabels = (
173              defineLabel ('000', 1 * orthoWidth // 4, fieldHeight + self.scoreShift),     # Player score
174              defineLabel ('000', 3 * orthoWidth // 4, fieldHeight + self.scoreShift)
175          )
176
177          self.game.batch.add (2, GL_LINES, None, ('v2i', (0, fieldHeight, orthoWidth, fieldHeight))) # Line
178
179      def increment (self, playerIndex):
180          self.scores [playerIndex] += 1
181
182      def reset (self):
183          self.scores = [0, 0]
184          Attribute.reset (self)  # Only does a commit here
185
186      def commit (self):          # Committing labels is adapting their texts
187          for playerLabel, score in zip (self.playerLabels, self.scores):
188              playerLabel.text = '{}'.format (score)
189
190  class Game:
191      def __init__ (self):
192          self.batch = pyglet.graphics.Batch ()   # Graphical reprentations insert themselves for batch drawing
193
194          self.deltaT = 0                              # Elementary timestep of simulation
195          self.serviceIndex = random.choice ((0, 1))  # Index of player that has initial service
196          self.pause = True                            # Start game in paused state
197
198          self.attributes = []                         # All attributes will insert themselves here
199          self.paddles = [Paddle (self, index) for index in range (2)]    # Pass game as parameter self
200          self.ball = Ball (self)
```

```
201            self.scoreboard = Scoreboard (self)
202
203            self.window = pyglet.window.Window (                          # Main window
204                      640, 480, resizable = True, visible = False, caption = "Pong"
205                  )
206
207            self.keymap = pyglet.window.key.KeyStateHandler ()  # Create keymap
208            self.window.push_handlers (self.keymap)              # Install it as a handler
209
210            self.window.on_draw = self.draw                      # Install draw callback, will be called asynch
211            self.window.on_resize = self.resize                  # Install resize callback, will be called if resized
212
213            self.window.set_location (                           # Middle of the screen that it happens to be on
214                (self.window.screen.width - self.window.width) // 2,
215                (self.window.screen.height - self.window.height) // 2
216            )
217
218            self.window.clear ()
219            self.window.flip ()                                  # Copy drawing buffer to window
220            self.window.set_visible (True)                       # Show window once its contents are OK
221
222            pyglet.clock.schedule_interval (self.update, 1/60.) # Install update callback to be called 60 times per s
223            pyglet.app.run ()                                    # Start pyglet engine
224
225        def update (self, deltaT):                               # Note that update and draw are not synchronized
226            self.deltaT = deltaT                                 # Actual deltaT may vary, depending on processor load
227
228            if self.pause:                                       # If in paused state
229                if self.keymap [pyglet.window.key.SPACE]:        #   If SPACEBAR hit
230                    self.pause = False                           #       Start playing
231                elif self.keymap [pyglet.window.key.ENTER]:      #   Else if ENTER hit
232                    self.scoreboard.reset ()                     #       Reset score
233                elif self.keymap [pyglet.window.key.ESCAPE]:     #   Else if ESC hit
234                    self.exit ()                                 #       End game
235
236            else:                                                # Else, so if in active state
237                for attribute in self.attributes:                #   Compute predicted values
238                    attribute.predict ()
239
240                for attribute in self.attributes:                #   Correct values for bouncing and scoring
241                    attribute.interact ()
242
243                for attribute in self.attributes:                #   Commit them to pyglet for display
244                    attribute.commit ()
245
246        def scored (self, playerIndex):            # Player has scored
247            self.scoreboard.increment (playerIndex) # Increment player's points
248            self.serviceIndex = 1 - playerIndex     # Grant service to the unlucky player
249
250            for paddle in self.paddles:             # Put paddles in rest position
251                paddle.reset ()
252
253            self.ball.reset ()                      # Put ball in rest position
254            self.pause = True                       # Wait for next round
255
256        def draw (self):
257            self.window.clear ()
258            self.batch.draw ()        # All attributes added their graphical representation to the batch
259
260        def resize (self, width, height):
261            glViewport (0, 0, width, height)              # Tell openGL window size
262
```

```
263         glMatrixMode (GL_PROJECTION)                # Work with projection matrix
264         glLoadIdentity ()                          # Start with identity matrix
265         glOrtho (0, orthoWidth, 0, orthoHeight, -1, 1)  # Adapt it to orthographic projection
266
267         glMatrixMode (GL_MODELVIEW)                 # Work with model matrix
268         glLoadIdentity ()                          # No transforms
269
270         return pyglet.event.EVENT_HANDLED          # Block default event handler
271
272  game = Game ()  # Create and run game
```

Listing 5.5: **pong/pong.py**

TIP: There's some more wisdom to be derived from listing 5.5. Note how constants like *orthoWidth* on line 8 and *margin* on line 67 are used. The use of a constant in those cases is justified according to the DRY principle. If two or more numbers are ALWAYS identical, use a named constant for them, so you only have to make changes in one place if their value changes. But the window dimensions on line 203, *640* x *480* pixels, are only used there. Using named constants in this case has no benefits, unless one wants to define all such values in one central place.

Chapter 6

Design patterns

6.1 The solution principles behind your source code

A DESIGN PATTERN IS A SOLUTION PRINCIPLE THAT CAN BE USED OVER AND OVER AGAIN IN DIFFERENT, BUT COMPARABLE, SITUATIONS.

So a design pattern IS NOT a piece of code, but rather the solution principle behind it. Still, Python code can be used to clarify design patterns by example. Part of learning how to design software is to recognize general patterns in your own code and have them at hand as a kind of language independent toolbox, growing with experience. The *predict, interact, commit* solution used in the Pong example is such a design pattern. It can be used in many different games and simulations and in any programming language. One of the reasons to always keep the source code of past projects at hand, is because it contains your personal toolbox of design patterns. I regularly find myself looking up solutions for certain situations that I came up with ten to twenty years ago. It helps me to explicitly notice them and give them a name: *ticked-and-slip protocol*, *event driven evaluation nodes (eden)* and *self inserter*. There are some very well known design patterns, going by names like *observer*, *facade*, *bridge* and *abstract factory*. These patterns are rather general but also rather crude. They all have many variations and refinements, sometimes with different names. The *observer* pattern has a variation called *publisher-subscriber* and another one called *event listener*. It is worth while to look at a few of these widespread design patterns, since they are a useful source of ideas if you have not yet written that much code yourself.

6.2 The Observer pattern

6.2.1 Example situation

We want to build the game of Tic Tac Toe, that's the one where you try to get three noughts or crosses on a row. But we want to have two views of the playing field. The first one is an alphanumerical view, showing a nought as the letter O and a cross as the letter X. The second one is a binary view, shown a nought as the digit 0 and a cross ast the digit 1. The two views have to be kept in sync. A naive strategy would be to let the game logic update both views by writing O's and X'es to the one and 0's and 1's to the other. For this trivial example that would be OK. BUT IN GENERAL IT IS UNDESIRABLE THAT THE GAME LOGIC SHOULD KNOW HOW TO UPDATE A CERTAIN TYPE OF VIEW. The number of possible ways to view the game state is endless, as is the number of instances of each type of view.

6.2.2 Solution principle

Rather the GAME LOGIC SHOULD JUST NOTIFY ALL VIEWS that something has changed. The VIEWS SHOULD THEN KNOW HOW TO UPDATE THEMSELVES, pulling out the relevant information from the game state, and display it any way they think fit, from 0's and 1's on a terminal to shiny balls and cubes in a graphics window, or perhaps boops and beeps from a loudspeaker. Lets elaborate this solution principle in a class diagram:
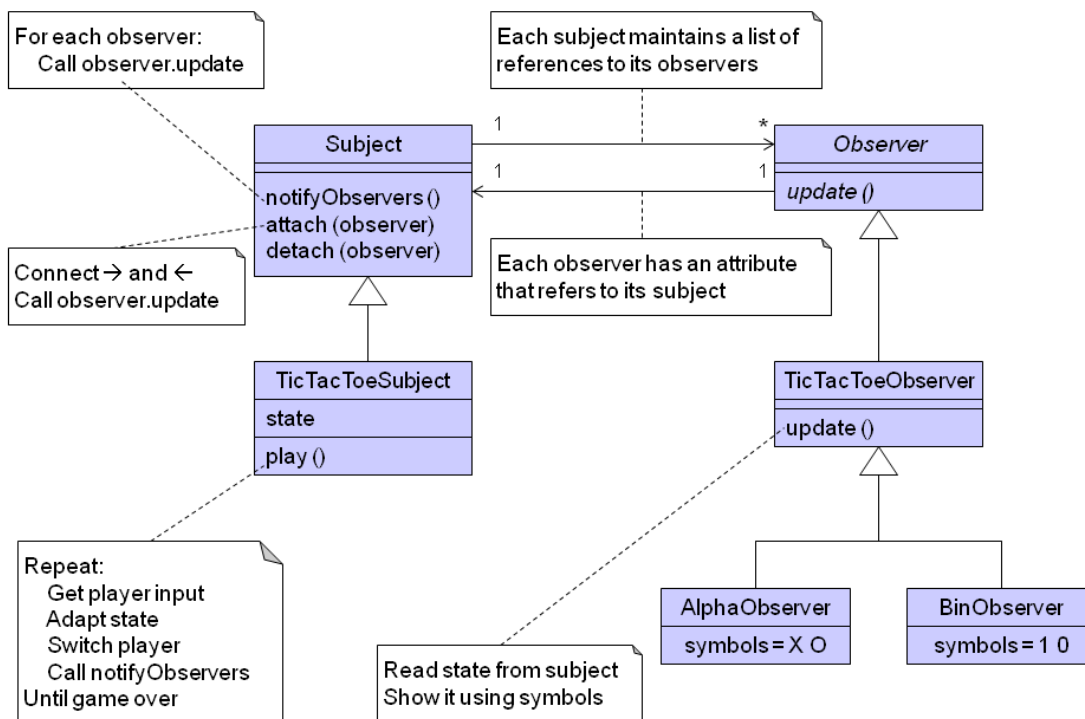
Figure 6.1: Tic Tac Toe Observer example

TicTacToeSubject contains the game logic, altering the game *state* with each move. *TicTacToeSubject* is a specialization of the *Subject* class, embodies one halve of the Observer pattern. Subjects are able to *attach* observers to themselves by placing them into an *observers* list and storing a back reference in *subject* attribute of the *observer* as well. They then can notify their observers of any changes by means of their *notifyObservers* method, that will call the *update* method on each observer.

Class *Observer* embodies the other halve of the Observer pattern. The *Observer* class has an *update* method. It is just there to define the interface, not to do anything useful. Such a method is called abstract, and its name is italicized in the diagram. Having at least one abstract method makes the whole *Observer* class abstract, since no fully functional objects can be instantiated from it. So the name of the *Observer* class is italicized as well.

Class *TicTacToeObserver* inherits from *Observer*. Its *update* method reads the state from the *TicTacToeSubject* and displays it. It isn't called directly on a *TicTacToeObserver*, but on an *AlphaObserver* or a *BinObserver* that inherit this method, so it uses the *symbols* of one of these descendant classes.

6.2.3   Example code

TIP: The class diagram showed in broad lines how the Tic Tac Toe example works. If you use diagrams like that, keep them simple. The right place for details is properly commented source code. Avoid complicated diagramming tools, that subvert freedom of expression by enforcing all kinds of strickt rules. Diagrams are only there to help imagination a bit.

Having understood the class diagram, you're now ready for the real thing: Use the force, read the source!

```
1  class Observer:
2      def update (self):          # Only here to clarify the interface
3          raise Exception ('Abstract method called: Observer.update')
```

```
 4                                  # Google for 'Python' and 'exception'
 5  class Subject:
 6      def __init__ (self):
 7          self.observers = []
 8
 9      def attach (self, observer):
10          self.observers.append (observer)    # Forward link to observers
11          observer.subject = self # Backlink from observer to subject
12          observer.update ()       # Get new observer up to date
13
14      def detach (self, observer):
15          self.observers.remove (observer)
16
17      def notifyObservers (self):
18          for observer in self.observers:
19              observer.update ()
20
21  class TicTacToeObserver (Observer):
22      def update (self):
23          print ('\n'.join ([                # Google for 'Python' and 'join'
24              ' '.join ([                    # and also for 'nested list comprehensions'
25                  self.symbols [value]       # Use common sense and perseverance
26                  for value in row           # to discover what happens here
27              ])                             # Write a small test program to
28              for row in self.subject.state  # experiment with code like this
29          ]), '\n')                          # Try the same without list comprehensions
30
31  class AlphaObserver (TicTacToeObserver):
32      symbols = ('.', 'O', 'X')              # Inherited update will use these symbols
33
34  class BinObserver (TicTacToeObserver):
35      symbols = ('.', '0', '1')              # Inherited update will use these symbols
36
37  class TicTacToeSubject (Subject):
38      def __init__ (self):
39          Subject.__init__ (self)
40
41          self.state = [                     # Initialize with 0's
42              [0 for column in range (3)]    # 0 means empty field, 1 means nought, 2 means cross
43              for row in range (3)           # Nested list comprehensions again
44          ]                                  # Try to reformulate without list comprehensions
45
46      def play (self):
47          even = False                       # The odd player starts, the even player is next
48          while True:
49              print ('X1' if even else 'O0', 'player' )
50              rowKey = input ('Row (q = quit):')  # Variable rowKey will contain a string of characters
51                                                  # rather than an integer number, so e.g. '3' rather than 3
52              if rowKey == 'q':                   # You can't calculate with strings, only with numbers.
53                  break
54
55              columnKey = input ('Column:')
56              self.state [int (rowKey) - 1][int (columnKey) - 1] = 2 if even else 1   # Convert to integers
57              even = not even                # It's the other player's turn now
58              self.notifyObservers ()        # Let the views know something has changed
59
60  ticTacToeSubject = TicTacToeSubject ()      # Create the game
61
62  ticTacToeSubject.attach (AlphaObserver ())  # Attach the observers
63  ticTacToeSubject.attach (BinObserver ())
64
```

```
65  ticTacToeSubject.play ()                          # Start playing
```
<div align="center">Listing 6.1: patterns/observer.py</div>

6.3   The Adapter pattern

6.3.1   Example situation

This example is about a tiny part of a control for an Automated Stacking Crane or ASC. ASC's store and retrieve containers that are kept in stock by even thousands in the so called stacking area of a container terminal. All ASC's together are controlled by the so called Movement Planner. The task of the Movement Planner to plan for efficient storage and retrieval of containers, e.g. keeping containers that have to leave first on top of the stacks or or put interchangeable containers on top of each other. The Movement Planner works with an elementary displacement from A to B called a *move*. The cranes, however, works with so called *get* and *put* orders. To *move* a container from A to B it has to first receive a *get* order, to pick it up at A, and then a *put* order to put it down at B. So there's a mismatch between how the Movement Planner requests a displacement (*move* interface) and how the ASC expects to receive it (*get* and *put* interface). Since there's a mismatch between these two interfaces, what's needed is an adaptor.

6.3.2   Solution principle

Adaptors come in two kinds, the first one being the Object Adapter:
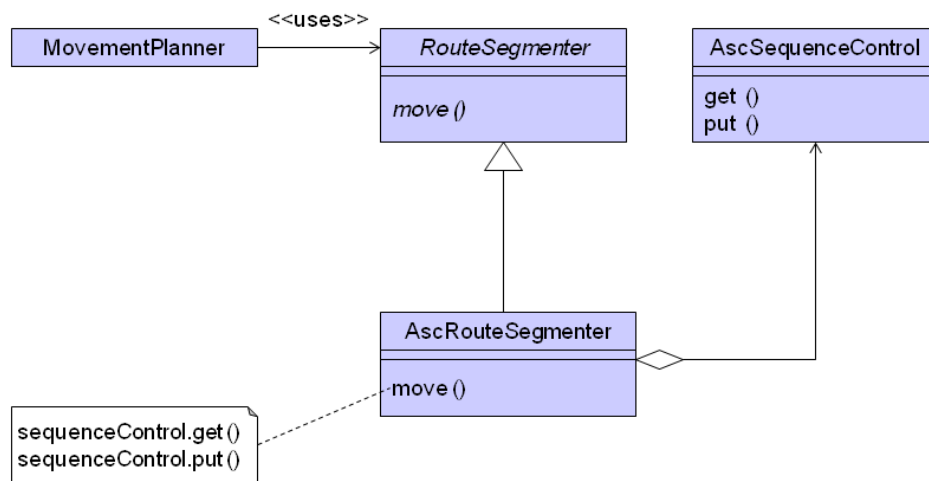
<div align="center">Figure 6.2: ASC Object Adapter example</div>

The interface of class *RouteSegmenter* consists of the *move* method, as required by the *MovementPlanner*. The *AscSequenceControl* has an interface consisting of the *get* and *put* methods. Class *AscRouteSegmenter* inherits its

interface from *RouteSegmenter*. It contains an attribute of type *AscSequenceControl* and it implements its *move* interface method by calling *get* and *put* upon this attribute. The *get* and *put* methods themselves are not added to the interface. Moreover it is possible for an *AscRouteSegmenter* to contain multiple instances of *AscSequenceControl*.

The other kind of adapter is the Class Adapter:
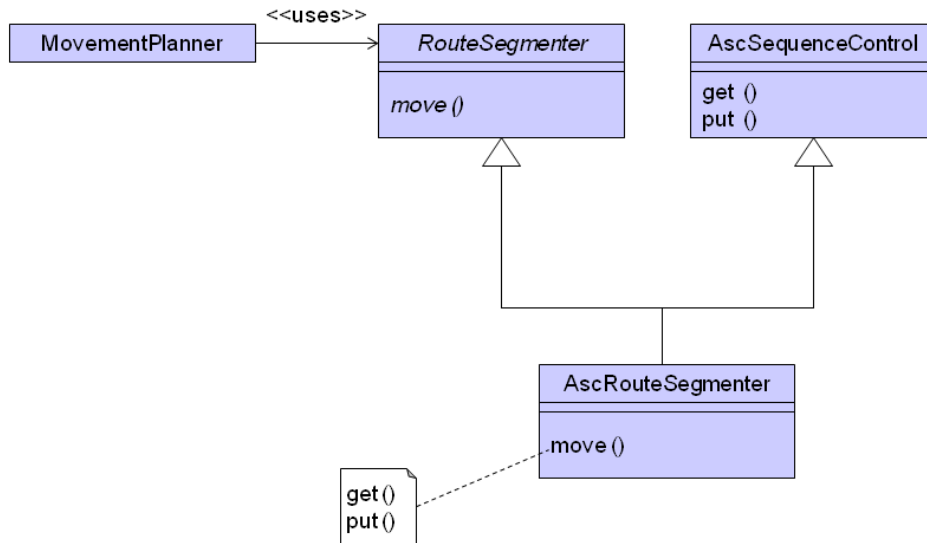
Figure 6.3: ASC Class Adaper example

Class *AscRouteSegmenter* now inherits from both the *RouteSegmenter* interface class and from the *AscSequence-Control* class that contains the implementations of *get* and *put*. Interface method *move* now calls these two inherited methods, not via an attribute but directly. Note that the *AscRouteSegmenter* does NOT CONTAIN an *AscSequence-Control* in that case, it IS an *AscSequenceControl*. This means that its interface contains *move* as well as *get* and *put*. And there can't be multiple instances of *AscSequenceControl* embedded in one *AscRouteSegmenter*.

6.3.3   Example code

The code of the object adaptor version is:

```
1   class MovementPlanner:
2           def __init__ (self, routeSegmenter):
3                   self.routeSegmenter = routeSegmenter
4
5           def batch (self, *locationPairs):          # * means convert parameters to list
6                   print ()
7                   print ('Starting moves')
8                   print ()
9                   for locationPair in locationPairs:
10                          self.routeSegmenter.move (locationPair)
11                  print ('Moves ready')
12
```

```
13  class RouteSegmenter:
14      def move (self, locationPair):
15          raise Exception ('Abstract method called: RouteSegmenter.move')
16
17  class AscSequenceControl:
18      def get (self, location):
19          print ('Picked up container at location:', location)
20
21      def put (self, location):
22          print ('Put down container at location:', location)
23
24  class AscRouteSegmenter (RouteSegmenter):
25      def __init__ (self, ascSequenceControl):
26          self.ascSequenceControl = ascSequenceControl
27
28      def move (self, locationPair):
29          self.ascSequenceControl.get (locationPair [0])
30          self.ascSequenceControl.put (locationPair [1])
31          print ()
32
33  MovementPlanner (AscRouteSegmenter (AscSequenceControl ())) .batch (
34      ('3A3', '2K1'),
35      ('3A2', '2K2'),
36      ('2C1', '9M3'),
37      ('9R4', '1A1'),
38      ('9R3', '1A2')
39  )
```

Listing 6.2: patterns/objectAdapter.py

The code of the Class Adapter is slightly more compact:

```
1   class MovementPlanner:
2       def __init__ (self, routeSegmenter):
3           self.routeSegmenter = routeSegmenter
4
5       def batch (self, *locationPairs):       # * means convert parameters to list
6           print ()
7           print ('Starting moves')
8           print ()
9           for locationPair in locationPairs:
10              self.routeSegmenter.move (locationPair)
11          print ('Moves ready')
12
13  class RouteSegmenter:
14      def move (self, locationPair):
15          raise Exception ('Abstract method called: RouteSegmenter.move')
16
17  class AscSequenceControl:
18      def get (self, location):
19          print ('Picked up container at location:', location)
20
21      def put (self, location):
22          print ('Put down container at location:', location)
23
24  class AscRouteSegmenter (RouteSegmenter, AscSequenceControl):
25      def move (self, locationPair):
26          self.get (locationPair [0])
27          self.put (locationPair [1])
28          print ()
29
30  MovementPlanner (AscRouteSegmenter ()) .batch (
31      ('3A3', '2K1'),
```

```
32          ('3A2', '2K2'),
33          ('2C1', '9M3'),
34          ('9R4', '1A1'),
35          ('9R3', '1A2')
36  )
```

<div align="center">

Listing 6.3: patterns/classAdapter.py

</div>

Nevertheless, in most cases use of the Object Adaptor is to be preferred for two reasons:

1. In general, it is desirable if interfaces are stable and thin. Stable means that they do not have to be changed with every minor change of the design. Standardization only reduces costs if you can rely on standards not constantly changing. Thin means that there are not too many methods or attributes that are part of the interface. Having a lot of methods or attributes in the interface of a class entails that some code elsewhere in the program may be dependent upon the presence of all those methods or attributes. In our class adapter example the *get* and *put* of *AscSequenceControl* are inherited and become part of the interface of *AscRouteSegmenter*. As a consequence of this, the amount of work involved in changing the interface of *AscSequenceControl* (*get* and *put*) could be unnecessarily high, since not only code that uses *AscSequenceControl* directly but also code that uses *AscRouteSegmenter* would have to be changed.

2. The possibility to have multiple instances of *AscSequenceControl* embedded in one *AscRouteSegmenter* contributes to the flexibility of the design.

BACKGROUND: A common misunderstanding is that encapsulation is about hiding attribute data by only accessing them via methods that store or retrieve them. But the main benefit of encapsulation is more general:

<div align="center">

ENCAPSULATION IS ABOUT HIDING DESIGN DECISIONS RATHER THAN ABOUT DATA HIDING.

</div>

Hiding a design decision means that changing this decision has only local impact. This contributes to the flexibility of the design, since the costs of change are limited. Changing an attribute indeed boils down to changing a design decision, but so does changing a method. So striving for thin interfaces is not only about hiding attributes but also about hiding methods. Hiding data may have extra benefits, apart from hiding design decissions. That's what section 6.4 is about.

6.4 The Property pattern

6.4.1 Example situation

Suppose we have a *Circle* class which has the attributes *radius*, *perimeter* and *area* in its interface. It seems that the 'thin interfaces' principle would indicate that e.g. only the *radius* should be present in the interface. But since any real world circle also has a perimeter and an area, no generality is sacrificed by adding them to the interface. And by adding them, code that uses *Circle* could directly use *perimeter* and *area* in addition to *radius*. Storing them as separate attributes would unfortunately mean that they have to be kept in sync explicitly. As an alternative to having the attributes *perimeter* and *area*, we could use methods *getPerimeter* and *getArea* to compute those values on the fly. And we could also add methods *setPerimeter* and *setArea*, to set the *radius* via them. Doing so would would result in a *Circle* interface consisting of *radius*, *getPerimeter*, *setPerimeter*, *getArea* and *setArea*. This unnecesarily exposes the design decision that the radius is actually stored, while the perimeter and the area are computed on the fly. Suppose that after a while, the area of the circle would turn out te be used much more frequent than the radius. Then it would become more attractive to store the area and compute the radius and the perimeter from that. This would lead to a changed interface of *Circle*, consisting of *getRadius*, *setRadius*, *getPerimeter*, *setPerimeter* and *area*, so all the code using *Circle* would have to be adapted. We'd like to avoid that expensive overall changes like that. How to proceed?

6.4.2 Solution principle

One way to go, is to add a getter and setter to the attribute itself as well, right from the start. The interface of *Circle* would then become: *getRadius*, *setRadius*, *getPerimeter*, *setPerimeter getArea*, *setArea*. The decision which

value is actually stored and which other two are computed from it is then hidden. This means that we're free to switch from storing *_radius* to storing *_perimeter* or *_area* at any time. Remember that prepending _ to an attribute or method means that it does not belong to the interface. That is important here, because if people started to directly access *_radius*, storing *_area* instead would still break their code, requiring extra work.

> BACKGROUND: Some of the practical thinking behind Python is revealed by the fact that refraining from direct use of private attributes or methods (the ones that start with a single _) is left to free will.
>
> There may be very good reasons to directly use a private attribute or method afterall, just like there may be very good reasons to break into your own car if you locked in the keys. Computer programs live in the real world, not in an ideal one.

An attribute that is meant to be only accessed via a getter or a setter is called a property. Python supports properties in an elegant way. Rather than having the bulky *getRadius*, *setRadius*, *getPerimeter*, *setPerimeter*, *getArea*, *setArea* interface, that requires the design decision to use a propery to be taken up front, since all external code depends upon this interface, it allows you to maintain the original *radius*, *perimeter*, *area* interface while still using properties.

Without this facility, one might indeed be inclined to use getters and setters for each attribute, just to be prepared for the unknown. With it, one can start out with simple attributes and change them to properties whenever needed, without impacting the rest of the design. It is important to make a clear distinction here: Accessing attributes via getters and setters is called the Property pattern. The fact that Python has transparent way to do so is a bonus, but stricktly spoken not part of the pattern. Since you're learning Python here, we'll shamelessly benefit of that bonus in the example code.

6.4.3   Example code

In the first example, the only thing actually stored in an object of class *Circle* is the private *_radius* attribute. All access to this attribute, including access via the methods of *Circle* itself, is via the associated public *radius* property. Note that the getters and setters start with an _ , since they are never to be called directly by code outside the class. The interface of *Circle* consists of the *radius*, *perimeter* and *area* properties. This interface hides the design decision on what is actually stored.

```
 1  import math
 2
 3  class Circle:
 4          def __init__ (self):
 5                  self._setRadius (0)
 6
 7          def _getRadius (self):
 8                  return self._radius
 9
10          def _setRadius (self, value):
11                  self._radius = value             # Only the radius is actually stored!
12
13          def _getPerimeter (self):
14                  return 2 * math.pi * self.radius                      # Use radius property
15
16          def _setPerimeter (self, value):
17                  self.radius = value / (2 * math.pi)                   # Use radius property
18
19          def _getArea (self):
20                  return math.pi * self.radius * self.radius    # Use radius property
21
22          def _setArea (self, value):
23                  self.radius = math.sqrt (value / math.pi)     # Use radius property
24
25          radius = property (_getRadius, _setRadius)
26          perimeter = property (_getPerimeter, _setPerimeter)
27          area = property (_getArea, _setArea)
```

```
28
29  # Code below, using Circle, does not depend on what is actually stored, _radius or _area
30
31  circle = Circle ()
32
33  circle.radius = 10
34  print ('radius = {}, perimeter = {}, area = {}'. format (circle.radius, circle.perimeter, circle.area))
35
36  circle.area = math.pi ∗ 10000
37  print ('radius = {}, perimeter = {}, area = {}'. format (circle.radius, circle.perimeter, circle.area))
38
39  print ('Attributes:', vars (circle))     # Print all 'real' attributes, so not the properties
```

<div align="center">Listing 6.4: patterns/propertyRadius.py</div>

In the second example, not the _ *radius*, but the _ *area* is stored in a private attribute. The interface of *Circle* still consists of the *radius*, *perimeter* and *area* properties. Since the interface did not change at all, all code using *Circle* doesn't have to change with respect to the previous example. This is a direct benefit from *Circle* having a stable interface, that doesn't change when _ *area* instead of _ *radius* is stored.

```
1   import math
2
3   class Circle:
4           def __init__ (self):
5                   self.area = 0
6
7           def _getRadius (self):
8                   return math.sqrt (self.area / math.pi)   # Use area property
9
10          def _setRadius (self, value):
11                  self.area = math.pi ∗ value ∗ value      # Use area property
12
13          def _getPerimeter (self):
14                  return 2 ∗ math.pi ∗ self.radius                 # Use radius property
15
16          def _setPerimeter (self, value):
17                  self.radius = value / (2 ∗ math.pi)              # Use radius property
18
19          def _getArea (self):
20                  return self._area                # Only the area is actually stored!
21
22          def _setArea (self, value):
23                  self._area = value
24
25          radius = property (_getRadius, _setRadius)
26          perimeter = property (_getPerimeter, _setPerimeter)
27          area = property (_getArea, _setArea)
28
29  # Code below, using Circle, does not depend on what is actually stored, _radius or _area
30
31  circle = Circle ()
32
33  circle.radius = 10
34  print ('radius = {}, perimeter = {}, area = {}'. format (circle.radius, circle.perimeter, circle.area))
35
36  circle.area = math.pi ∗ 10000
37  print ('radius = {}, perimeter = {}, area = {}'. format (circle.radius, circle.perimeter, circle.area))
38
39  print ('Attributes:', vars (circle))     # Print all 'real' attributes, so not the properties
```

<div align="center">Listing 6.5: patterns/propertyArea.py</div>

> TIP: The fact that the member functions of *Circle* also use *radius*, *perimeter* and *area*, rather than calling getters and setters or accessing the private attribute directly, further limits the impact of design changes. One might argue that accessing the underlying attribute (*_radius* or *_area*) directly may result in faster code. This type of so called "peephole optimization" should be avoided if possible, since it violates the DRY principle. This makes design changes harder, including the optimizations that REALLY matter. As a general rule:
>
> > DON'T START OPTIMIZING TOO EARLY, FIRST GIVE PRIORITY TO A CLEAR AND REGULAR DESIGN.
>
> This is not an invitation to be wasteful, but to avoid being "penny wise and pound foolish". If you've defined a standard interface for a class, use it, also in the class itself. O, and yes indeed, there are exceptions to any rule, this one is no exception.

6.5   *EXTRA* The Call Chaining pattern

> TIP: While the call chaining pattern itself isn't too difficult, the example given here is complicated. It uses a lot of special Python facilities and some clever tricks. Depending on your skills you may:
>
> 1. Skip this pattern altogether. Not much is lost if you do.
>
> 2. Superficially skim through section 6.5.1 , study the small code blocks in section 6.5.2 thoroughly, but skip section 6.5.3 completely.
>
> 3. Wrestle through section 6.5.3 as well.

6.5.1   Example situation

The first time I encountered the Call Chaining pattern was in IBM's Visual Age library in 1980. I started to use it myself every now and then. Later on it became part of the Microsoft's LINQ. LINQ makes it possible to select data from a collections of object in a clever way, inspired by SQL. SQL (pronunciation: sequel) or Structured Query Language is a special purpose language to query and update data in a so called relational database, a very popular way to store data into tables on disk.

An example of a SQL statement is:

```
1  SELECT name, address
2  FROM customers
3  WHERE profit > 10,000,000
```

Spent a few minutes reading bout SQL e.g. in Wikipedia. Don't dive to deeply, most of it will be intuitively clear. While SQL is a language in its own right, often the need arises to have SQL statements as part of e.g. a Python program. One way to do that is just to embed them as multi line strings (Google for Python and 'multiline strings'):

```
1  query = '''
2          SELECT name, address
3          FROM customers
4          WHERE profit > 10,000,000
5  '''
6
7  result = database.execute (query)
```

In this code fragment *query* is a string variable containing a piece of SQL. Python doesn't have any idea about the meaning of this piece of SQL, but just passes the string as parameter to method *execute* of a Python object of class *database*. That object just forwards the contents of the SQL string to a special purpose program called the RDBMS (Relational Database Management System). The RDBMS queries the database and returns the selected data to the *execute* function that in turn returns it to *result*. This can be done, since the RDBMS can be controlled from C/C++ and Python in turn can communicate with C/C++ code, a.o by using something called Cython. The

details are not relevant at this point and, on top of that, rather boring. An RDBMS that Python plays nice with is SQLite, but all the others will do as well.

Another way to use SQL from Python is to have special functions in Python like *SELECT*, *FROM* and *WHERE*, to do the querying. The nice thing about this approach is that these functions may be programmed in such a way that they query Python lists rather than a real SQL database. This example gives you an idea of how that might be done in principle. The main thing you can learn from it, is that there's no mystery involved in SQL, or anything else that has to do with computer programming for that matter. So, we are not going to connect to any RDBMS. Rather we'll select data from Python lists. But an interface to an RDBMS could be added. This is exactly what Microsoft did with their LINQ tool. As with LINQ, the language we'll use (or rather, make) is not SQL, but something that looks a lot like it, allowing constructions like:

```
1  result = (
2       FROM (animals) .
3       WHERE (lambda r: r.species == 'human' and r.age > 10) .
4       SELECT (lambda r: (r.gender, r.name, r.age)) .
5       ORDER_BY (lambda r: (r.name, r.age | DESC))
6  )
```

Here the SQL query is not a 'dead' string that doesn't mean anything to Python, it is an alive and kicking chain of Python function calls, each of the functions able to do something useful as part of the query. The secret of this code is in the dots at the end of the lines. It's these dots that give away the Call Chaining pattern.

6.5.2 Solution principle

The solution principle of call chaining is that each call to a function or method returns an object. From this object in turn a method can be called, which returns another object, of which a method can be called, and so on. In the above piece of code the *FROM* function returns an object of class *Table*. Class *Table* has a method *WHERE*, that returns second object, holding a selection of the first *Table* object. This second object is also of class *Table*. Class *Table* also has a method *SELECT*. This method again returns a *Table* instance. Class *Table* also has a method *ORDER_BY*, which is called in the end, returning a string with a sorted version of the contents of the *Table*.

The dots were crucial because they provide the glue between the lines of SQL like code. If *FROM (animals)* returns an object, then F*ROM (animals) .WHERE (...)* is just a method call upon that object. It is the normal dot notation for calling methods that we've used everywhere. So the essence of the pattern is that rather than writing:

```
1  object1 = functionA ()
2  object2 = object1.methodB ()
3  object3 = object2.methodC ()
4  object4 = object3.methodD ()
5  result = object4.methodE ()
```

one writes:

```
1  result = (
2       functionA () .
3       methodA () .
4       methodB () .
5       methodC () .
6       methodD ()
7  )
```

Each method call relies upon the result of the previous call. It is as if the object returned by functionA is handed through a sequence of transformation functions. With database querying each transformation acts as a filter narrowing down the query. But call chaining like this can also be used to e.g. transform graphical objects:

```
1  render (
2       getCube () .
3       stretchX (3) .
4       stretchY (4) .
5       rotateX (math.pi / 2) .
```

```
6              material (shiny) .
7              light (green)
8      )
```

The final stretched, rotated , illuminated object is returned to the *render* function, that puts it on the screen.

6.5.3   Example code

The example code uses a small selection of the many under-the-hood features that make Python such a power tool. They will be explained in a second, heavily commented, version of the source code. But first traverse the bare code criss-cross with a machete and see what you pick up from it without any explanation. Maybe the end is the best place to start, since that's where everything comes together. Don't strive for complete understanding, just get some intuitive sense of what this code roughly does. If some fragments are too hard, skip them. If this leads you to skipping everything, that's completely OK.

```
1   import itertools
2
3   class Accessor:
4          def __init__ (self, table):
5                  self.table = table
6
7          def __getattr__ (self, name):
8                  return self.row [self.table.fieldDict [name]]
9
10         def __call__ (self, row):
11                 self.row = row
12                 return self
13
14  def FROM (*tables):
15         if len (tables) == 1:
16                 return tables [0]
17         else:
18                 result = TABLE (*itertools.chain (*(table.fieldNames for table in tables)))
19                 result.rows = set (
20                         tuple (itertools.chain (*row))
21                         for row in itertools.product (*(table.rows for table in tables))
22                 )
23                 return result
24
25  class Descending:
26         def __ror__ (self, first):
27                 self.field = first
28                 return self
29
30  DESC = Descending ()
31
32  def tuplize (any):
33         return any if type (any) == tuple else (any,)
34
35  class TABLE:
36         def __init__ (self, *fieldNames):
37                 self.fieldNames = fieldNames
38                 self.fieldDict = {fieldName: index for index, fieldName in enumerate (self.fieldNames)}
39                 self.accessor = Accessor (self)
40                 self.rows = set ()
41
42         def __call__ (self, **aliases):
43                 result = TABLE (*(
44                         (aliases [fieldName] if fieldName in aliases else fieldName)
45                         for fieldName in self.fieldNames
```

```
46                          ))
47                          result.rows = self.rows # Only copy reference, since nothing is changed
48                          return result
49
50              def __contains__ (self, fieldOrTuple):
51                          return tuplize (fieldOrTuple) in self.rows
52
53              def INSERT (self, *rows):
54                          self.rows = set (rows)
55
56              def WHERE (self, function):
57                          result = TABLE (*self.fieldNames)
58                          result.rows = {row for row in self.rows if function (self.accessor (row))}
59                          return result
60
61              def SELECT (self, function):
62                          result = TABLE (*tuplize (function (self.accessor (self.fieldNames))))
63                          result.rows = {tuplize (function (self.accessor (row))) for row in self.rows}
64                          return result
65
66              def ORDER_BY (self, function):
67                          aList = list (self.rows)
68
69                          def sortParamPair (descendingOrField, index):
70                                      isDescending = isinstance (descendingOrField, Descending)
71                                      keyGetter = (
72                                                  lambda row: tuplize (function (self.accessor (row))) [index] .field
73                                      ) if isDescending else (
74                                                  lambda row: tuplize (function (self.accessor (row))) [index]
75                                      )
76                                      return (keyGetter, isDescending)
77
78                          sortParamPairs = [
79                                      sortParamPair (descendingOrField, index)
80                                      for index, descendingOrField in enumerate (
81                                                  tuplize (function (self.accessor (self.fieldNames)))
82                                      )
83                          ]
84
85                          aList = list (self.rows)
86                          for keyGetter, isDescending in reversed (sortParamPairs):
87                                      aList.sort (key = keyGetter, reverse = isDescending)
88
89                          return Cursor (self, aList)
90
91  class Cursor:
92              def __init__ (self, table, aList):
93                          self.table = table
94                          self.aList = aList
95                          self.columnWidth = 15
96                          self.fieldFormatString = '{{:{}}}'.format (self.columnWidth)
97
98              def __str__ (self):
99                          return '\n'.join (
100                                     [       ' '.join (
101                                                         [self.fieldFormatString.format (fieldName) for fieldName in self.table.fieldNames]
102                                                 ),
103                                                 ' '.join ([self.columnWidth * '_'] * len (self.table.fieldNames))
104                                     ] +
105                                     [       ' '.join (self.fieldFormatString.format (field) for field in row)
106                                                 for row in self.aList
107                                     ]
```

```
108                     )
109
110  animals = TABLE ('species', 'gender', 'age', 'name', 'length')
111
112  animals.INSERT (
113          ('horse', 'male', 15, 'henry', 1.8),
114          ('human', 'female', 22, 'wilma', 1.7),
115          ('human', 'male', 30, 'john', 1.5),
116          ('human', 'female', 20, 'mary', 1.9),
117          ('human', 'male', 26, 'robin', 1.7),
118          ('human', 'unknown', 25, 'robin', 1.8),
119          ('human', 'female', 27, 'robin', 1.4),
120          ('ape', 'female', 5, 'benji', 1.1)
121  )
122
123  eats = TABLE ('animalSpecies', 'plantSpecies')
124
125  eats.INSERT (
126          ('horse', 'gras'),
127          ('horse', 'oats'),
128          ('human', 'oats'),
129          ('human', 'lettuce'),
130          ('human', 'banana'),
131          ('ape', 'banana')
132  )
133
134  plants = TABLE ('species', 'length')
135
136  plants.INSERT (
137          ('gras', 0.3),
138          ('oats', 0.5),
139          ('banana', 0.2)
140  )
141
142  primateFood = TABLE ('species', 'kind')
143
144  primateFood.INSERT (
145          ('oats', 'wild'),
146          ('banana', 'wild'),
147          ('banana', 'cultivated')
148  )
149
150  adultsCursor = (
151          FROM (animals) .
152          WHERE (lambda r: r.species == 'human' and r.age > 10) .
153          SELECT (lambda r: (r.gender, r.name, r.age)) .
154          ORDER_BY (lambda r: (r.name, r.age | DESC))
155  )
156
157  print ('\n', adultsCursor)
158
159  menu = (
160          FROM (
161                  animals (species = 'aSpecies', length = 'animalLength'),
162                  eats,
163                  plants (species = 'pSpecies', length = 'plantLength')
164          ) .
165          WHERE (lambda r: r.aSpecies == r.animalSpecies and r.plantSpecies == r.pSpecies) .
166          SELECT (lambda r: (r.name, r.animalSpecies, r.animalLength, r.plantSpecies, r.plantLength))
167  )
168
169  menuCursor = \
```

```
170          FROM (menu). \
171          ORDER_BY (lambda r: r.name)
172
173  print ('\n', menuCursor)
174
175  primateCursor = (
176          FROM (menu) .
177          WHERE (lambda r: (r.plantSpecies, 'cultivated') in (
178                  FROM (primateFood) .
179                  SELECT (lambda r: (r.species, r.kind))
180          )) .
181          ORDER_BY (lambda r: r.name)
182  )
183
184  print ('\n', primateCursor)
```

<div align="center">

Listing 6.6: patterns/callChaining.py

</div>

Next thing to look at is an abundantly commented version of the same code. Using comments like this is called Literate Programming. Give it another go, carefully reading the comments. Be prepared to do some detective work.

```
1   import itertools
2   # Module itertools is part of the Python standard library and contains tools for iterators
3   # An iterator resembles a list, but different from a list, an iterator does not hold all of its elements
4   # It produces them on the fly, without storing them all at the same time in memory
5   # Because this sounds a bit alien, just try to follow how iterators are used here
6
7   class Accessor:
8   # Our 'database' will consist of tables, each TABLE holding a unordered set of rows
9   # While these rows are just tuples whose components can be addressed by indices,
10  # they an also be addressed by field name, using a part of the TABLE called the field dictionary
11  # The task of an Accessor is to make this addressing by name instead of by index possible
12  # The Accessor class does this by introducing a kind of virtual attributes, one for each field name
13
14          def __init__ (self, table):
15          # Each Table will create an Accessor for itself and pass itself as parameter to its constructor
16
17                  self.table = table
18                  # At this point the Accessor knowns to which TABLE it belongs
19
20          def __getattr__ (self, name):
21          # Methods between starting and ending with __ are special
22          # In Python accessing anObject.anAttribute results in calling anObject.__getattr__ ('anAttribute')
23          # By overriding (redefining) __getattr__ we make anObject.anAttribute lookup anAttribute in a special way
24
25                  return self.row [self.table.fieldDict [name]]
26                  # The expression self.table.fieldDict [aName] looks up the field index into the field dictionary
27                  # This index is then used to access the field as self.row [aFieldIndex]
28                  # But it looks like Accessor has no row attribute...
29
30          def __call__ (self, row):
31          # Overriding the __call__ special method means that objects of that class can be called as if they were functions
32          # The call anAccessor (aRow) results in the call anAccessor.__call__ (aRow)
33
34                  self.row = row
35                  # This call results in creating a row attribute on the fly
36
37                  return self
38                  # After having created a row, anAccessor returns itself
39                  # Since it has a __getattr__, the call anAccessor (row) .aName results in
40                  # 1. Creating the right row attribute
41                  # 2. Accessing the field with name aName from that row using __getattr__
42
```

```
43  def FROM (*tables):
44  # The *tables means that this function will put all its parameters into a list called tables
45
46          if len (tables) == 1:
47          # The list of tables has length one, only one table is passed in
48
49                  return tables [0]
50                  # Return that one table
51
52          else:
53          # The list contains more than one TABLE
54
55                  result = TABLE (*itertools.chain (*(table.fieldNames for table in tables)))
56                  # Merge all the lists of fieldNames of these tables into
57                  # one long list of fieldNames for the resulting TABLE
58
59                  result.rows = set (tuple (
60                  # Step 3. Convert the resulting list of rows into an unordered set,
61                  #         assign it to the row set of the result TABLE
62
63                          itertools.chain (*row)) for row in
64                          # Step 2. For each row of the carthesian product, merge its subrows
65
66                          itertools.product (*(table.rows for table in tables)
67                          # Step 1. Compute the carthesian product,
68                          #         holding all combinations of subrows from the individual tables
69                  ))
70                  return result
71                  # Return the new, merged TABLE, which is the join of the individual tables (Google for 'SQL full join')
72
73  class Descending:
74  # Objects of this class are used as an indicator that sort order in the ORDER_BY method has to be descending
75          def __ror__ (self, first):
76          # 'reverse or', this gives the | operator a new meaning
77          # If DESC is of class Descending, then DESC.__ror__ (firstObject) can also be written as: firstObject | DESC
78          # If it had been __or__ rather than __ror__, then DESC.__or__ (lastObject) could
79          # have been written as: DESC | lastObject
80
81                  self.field = first
82                  # In the process of evaluating expression firstObject | DESC, DESC.field gets value firstObject
83
84                  return self
85                  # So expression firstObject | DESC will return the object DESC, having an attribute firstObject
86
87  DESC = Descending ()
88
89  def tuplize (any):
90          return any if type (any) == tuple else (any,)
91          # If any is already a tuple, that tuple will be returned
92          # If any is not already a tuple, the tuple (any,), having only one component, will be returned
93          # So a single value will automatically be converted to  tuple with that value as only component
94
95  class TABLE:
96  # Our 'database' consists of tables
97
98          def __init__ (self, *fieldNames):
99          # *fieldNames means that the parameters of the constructor call will be placed in to a list
100                 self.fieldNames = fieldNames
101                 # This list of fieldNames is stored in the TABLE object
102
103                 self.fieldDict = {fieldName: index for index, fieldName in enumerate (self.fieldNames)}
104                 # The field dictionary makes it possible to look up the field index by name
```

```
105                    # So expression self.fieldDict ('aFieldName') gives the index of field 'aFieldName' in each row tuple
106
107                    self.accessor = Accessor (self)
108                    # The accessor object can be coupled to a row by the expression self.accessor (aRow)
109                    # self.accessor.aFieldName then returns the right field of that row as explained with the Accessor class
110
111                    self.rows = set ()
112                    # The rows of a TABLE are unordered, so they are represented by a Python set, not by a list
113
114            def __call__ (self, **aliases):
115            # This function makes it possible to rename the fields of a 'database' TABLE
116            #
117            # aTable (someParameters) is equivalent to aTable.__call__ (someParamters)
118            #
119            # The call aTable (fieldName1 = 'newFieldName1', fieldName2 = 'newFieldName2') uses named parameters
120            # Named parameters are allowed with any function or method call in Python
121            # In this case it will result in  aTable.__call__ (fieldName1 = 'newFieldName1', fieldName2 = 'newFieldName2')
122            #
123            # The **aliases notation means that the values of the named parameters will be stored into the aliases dictionary
124            # The names of the parameters will be the keys to their values in that Python dict
125            # So e.g. aliases ['fieldName1'] will return 'newFieldName1'
126
127                    result = TABLE (*(
128                            (aliases [fieldName] if fieldName in aliases else fieldName)
129                            for fieldName in self.fieldNames
130                    ))
131                    # The resulting table will have the aliases as field names instead of the original ones
132
133                    result.rows = self.rows
134                    # The contents of the new table does not change,
135
136                    return result
137                    # Return that new table, according to the Call Chaining pattern
138
139            def __contains__ (self, fieldOrTuple):
140                    return tuplize (fieldOrTuple) in self.rows
141                    # Looks up a certain tuple of field values
142                    # If it's a single value, tuplize will make it a tuple of 1 component
143
144            def INSERT (self, *rows):
145            # *rows will turn the parameters of a call to INSERT into a list called rows
146                    self.rows = set (rows)
147                    # This list of rows is converted to a set and assigned to the row attribute of TABLE
148
149            def WHERE (self, function):
150                    result = TABLE (*self.fieldNames)
151                    # An empty result TABLE with the same field names as TABLE self is created
152
153                    result.rows = {row for row in self.rows if function (self.accessor (row))}
154                    # Only rows for which function returns True will be inserted into the result TABLE
155
156                    return result
157                    # The result TABLE is returned, once again allowing Call Chaining
158
159            def SELECT (self, function):
160                    result = TABLE (*tuplize (function (self.accessor (self.fieldNames))))
161                    # The field names are treated as just another row
162                    # The accessor turns them in to an object with attributes as explained with the Accessor class
163                    # The attributes in this case will contain their own names,
164                    # since this 'row' contains the field names as fields
165                    # Calling function (accessor (self.fieldNames)) will return a selected field names
166
```

```
167                    result.rows = {tuplize (function (self.accessor (row))) for row in self.rows}
168                    # Calling function (accessor (row)) will return the selected
169                    # field values matching the selected field names
170                    # In this set comprehension, this selection of field values is done for all rows
171                    # A set comprehension is like a list comprehension, but it produces a set instead of a list
172
173                    return result
174                    # Return the resulting TABLE, according to the Call Chaining pattern
175                    # It fill contain only the selected field names and the matching selected fields
176
177          def ORDER_BY (self, function):
178          # The function passed to ORDER_BY can e.g. be lambda r: (r.name, r.age | DESC)
179          # This means: first sort ascending on name, and after that descending on age
180                    aList = list (self.rows)
181                    # Turn the set of rows into a list, since sets cannot be ordered at all
182
183                    def sortParamPair (descendingOrField, index):
184                    # Parameter descendingOrField will e.g. be r.name, or r.age | DESC
185                    # If it's indeed r.age | DESC, it will evaluate to the DESC object, and DESC.field have value age
186                    # Look at the explanation of class Descending to understand this
187                    #
188                    # Parameter index will hold the field index matching the descendingOrField parameer
189                            isDescending = isinstance (descendingOrField, Descending)
190                            # If descendingOrField evaluates to a DESC object,
191                            # sorting on field DESC.field should be done descending
192
193                            keyGetter = (
194                            # keyGetter should be assigned a function that returns the value of the field to sort on
195
196                                    lambda row:
197                            # This lambda is used for a descending sort, it has to unpack the field from DESC
198                                        tuplize (function (
199                                            self.accessor (row)
200                                            # The attributes of the accessor objects are
201                                            # now the field values of this row
202                                            # Some may be wrapped in DESC
203                                        ))
204                                        # At this point we have the a tuple of field values to sort on,
205                                        # some wrapped in DESC
206
207                                        [index]
208                                        # At this point we have the right field from that tuple, as indicated by index
209                                        # but the sort is descending, so it will be wrapped in DESC
210
211                                        .field
212                                        # We unwrap it by returning DESC.field
213
214                                    ) if isDescending else (
215                                    # Use the lambda function above in case a DESC indicator is present,
216                                    # else use the one below
217
218                                    lambda row:
219                            # This lambda is used for an ascending sort, it doesn't have to unpack the field
220                                        tuplize (function (
221                                            self.accessor (row)
222                                            # The attributes of the accessor object are now the
223                                            # field values of this row
224                                        ))
225                                        # At this point we have the a tuple of field values to sort on
226
227                                        [index]
228                                        # At this point we have the right field from that tuple, as indicated by index
```

```
229                              )
230
231                              return (keyGetter, isDescending)
232                              # We return the lambda function that returns the right field,
233                              # and wether or not the sort on that field is descending
234                              # This tuple is called the sort parameter pair
235
236                      sortParamPairs = [
237                              sortParamPair (descendingOrField, index)
238                              for index, descendingOrField in enumerate (
239                                      tuplize (function (
240                                              self.accessor (self.fieldNames)
241                                              # The attributes of the accessor object are now the field names,
242                                              # some of them wrapped in DESC
243                                      ))
244                                      # At this point we have a tuple of field names, some wrapped in DESC
245                              )
246                              # We now pair each field name, some wrapped in DESC, with its index
247                      ]
248
249                      aList = list (self.rows)
250                      for keyGetter, isDescending in reversed (sortParamPairs):
251                      # Going through the keys in the right order
252                              aList.sort (key = keyGetter, reverse = isDescending)
253                              # Sort the list on that keys successively
254
255                      return Cursor (self, aList)
256                      # Return a Cursor for that list
257                      # A Cursor is a ordered list of rows
258                      # It has a multiline string representation featuring a header line and then a line for each row
259
260  class Cursor:
261          def __init__ (self, table, aList):
262                  self.table = table
263                  self.aList = aList
264                  self.columnWidth = 15
265                  self.fieldFormatString = '{{:{}}}'.format (self.columnWidth)
266
267          def __str__ (self):
268                  # The __str__ special function of a class lays down how objects of this
269                  # class are converted to strings.
270                  # Such conversion e.g. takes place of the object is printed
271
272                  return '\n'.join (
273                          # Glue the header and the rows together with linefeeds in between,
274                          # so each will be on its own line
275                          [       ' '.join (
276                                          [self.fieldFormatString.format (fieldName) for fieldName in self.table.fieldNames]
277                                  ),
278                                  # Header line containing field names, glued together with blanks
279
280                                  ' '.join ([self.columnWidth * '_'] * len (self.table.fieldNames))
281                                  # Line of underscores under each field name to separate them from the rows,
282                                  # glued together with blanks
283                          ] +
284                          [       ' '.join (self.fieldFormatString.format (field) for field in row)
285                                  for row in self.aList
286                                  # Many lines of field values glued together with blanks, one line for each row
287                          ]
288                  )
289
290  animals = TABLE ('species', 'gender', 'age', 'name', 'length')
```

```
291  # Create animals table, specifying field names
292
293  animals.INSERT (
294          ('horse', 'male', 15, 'henry', 1.8),
295          ('human', 'female', 22, 'wilma', 1.7),
296          ('human', 'male', 30, 'john', 1.5),
297          ('human', 'female', 20, 'mary', 1.9),
298          ('human', 'male', 26, 'robin', 1.7),
299          ('human', 'unknown', 25, 'robin', 1.8),
300          ('human', 'female', 27, 'robin', 1.4),
301          ('ape', 'female', 5, 'benji', 1.1)
302  )
303  # Insert rows into the animals table
304
305  eats = TABLE ('animalSpecies', 'plantSpecies')
306  # Create table eats, specifying field names
307
308  eats.INSERT (
309          ('horse', 'gras'),
310          ('horse', 'oats'),
311          ('human', 'oats'),
312          ('human', 'lettuce'),
313          ('human', 'banana'),
314          ('ape', 'banana')
315  )
316  # Insert rows into eats table
317
318  plants = TABLE ('species', 'length')
319  # Create plants table, specifying field names
320
321  plants.INSERT (
322          ('gras', 0.3),
323          ('oats', 0.5),
324          ('banana', 0.2)
325  )
326  # Insert rows in to plants table
327
328  primateFood = TABLE ('species', 'kind')
329  # Create primateFood table
330
331  primateFood.INSERT (
332          ('oats', 'wild'),
333          ('banana', 'wild'),
334          ('banana', 'cultivated')
335  )
336  # Insert rows into primateFood table
337
338  adultsCursor = (
339          FROM (animals) .
340          WHERE (lambda r: r.species == 'human' and r.age > 10) .
341          SELECT (lambda r: (r.gender, r.name, r.age)) .
342          ORDER_BY (lambda r: (r.name, r.age | DESC))
343  )
344  # FROM, WHERE, SELECT and ORDER_BY are called one after another,
345  # according to the Call Chaining pattern
346
347  print ('\n', adultsCursor)
348  # The string representation of the resulting Cursor is printed
349
350  menu = (
351          FROM (
352                  animals (species = 'aSpecies', length = 'animalLength'),
```

```
353                eats,
354                plants (species = 'pSpecies', length = 'plantLength')
355          ) .
356          WHERE (lambda r: r.aSpecies == r.animalSpecies and r.plantSpecies == r.pSpecies) .
357          SELECT (lambda r: (r.name, r.animalSpecies, r.animalLength, r.plantSpecies, r.plantLength))
358    )
359    # A full join (Google) is created, some field names of the
360    # animals and plants tables are replaced by aliases
361    # This prevents confusion with identical field names of the eats table
362    # The end product here is not a Cursor but a TABLE, since that is what SELECT returns
363
364    menuCursor = \
365          FROM (menu). \
366          ORDER_BY (lambda r: r.name)
367    # Now we have our Cursor
368
369    print ('\n', menuCursor)
370    # The string representation of the resulting Cursor is printed
371
372    primateCursor = (
373          FROM (menu) .
374          WHERE (lambda r: (r.plantSpecies, 'cultivated') in (
375                FROM (primateFood) .
376                SELECT (lambda r: (r.species, r.kind))
377          )) .
378          ORDER_BY (lambda r: r.name)
379    )
380    # This type of query is called a 'nested query'
381    # It uses information from the primateFood table to select rows from
382    # the menu table that was constructed earlier
383
384    print ('\n', primateCursor)
385    # Print the result of the nested query
```

Listing 6.7: patterns/callChainingLiterate.py

−//−