

Light on Python

November 23, 2015

Contents

1	Introduction	1
2	Objects	2
2.1	Your first program	2
2.2	Specifying your own classes	3
2.3	Indentation, capitals and the use of <code>_</code>	4
3	Encapsulation	5
3.1	Interfaces	5
3.2	Modules	6
3.3	Polymorphism	7
4	A pinch of functional programming	8
4.1	List comprehensions	8
4.2	Transforming all elements of a list	9
4.3	Selecting certain elements from a list	10
4.4	Computing sum from a list	10
4.5	Free functions and lambda expressions	10
5	Inheritance	12
5.1	Implementation inheritance	12
5.2	Interface inheritance	13
5.3	Inheriting from library classes	14

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 be 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 can do so gradually if you wish, while using it. 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 then. Programming turned out to be fun indeed. And it still is, for me!

2 Objects

2.1 Your first program

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

Listing 1: prog/sort.py

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
```

Listing 2: prog/sort2.py

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

2.2 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
```

Listing 3: /prog/dog.py

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 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 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.

2.3 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 sourcecode, 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++, so I don't usually abide by these rules. But in this course I will, for the greater part. If you want to learn a style that is consistent over multiple programming languages, use capitals on word boundaries for objects, attributes and methods as well instead of _, but always start them with a lowercase letter. By the way *WritingClassNamesLikeThis* or *writingAllOtherNamesLikeThis* is called camel case, while *writing_all_other_names_like_this* is called pothole case.

3 Encapsulation

3.1 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, standardisation 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 6: 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.

3.2 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

```

Listing 7: prog/dog_walker/dog_walker

```

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

```

Listing 8: prog/dog_walker/bosses.py

```

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

```

Listing 9: prog/dog_walker/dogs.py

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. To make clear what that interface is, using the `_` prefix is crucial. Anything being prefixed by a single `_`, like the `_bark` method in the above example, does not belong to the interface and is not meant to be accessed outside the module where it is defined. Python does only enforce this partially, it is mainly a convention to be followed voluntarily.

3.3 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 empty 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 constructor
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 10: 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 can all *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. Polymorphic means: of varying shape.

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.

4 A pinch of functional programming

4.1 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, let's 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 11: prog/dog_walker/func_walker.py

While this example resembles the one before, there's a difference. In listing 10 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 11 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 from 0 to 9.

Suppose you want a box with hundred chocolats. 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 10. 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 11.

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 shopwindow. 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 timeconsuming 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.

4.2 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)
```

Listing 12: prog/func_square.py

The non-functional way requires more code than the functional way. Still the beginning you may prefer the non-functional way, since it shows what's happening step by step. But that will probably shift, once you gain experience.

```
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)
```

Listing 13: prog/nonfunc_square.py

4.3 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)
```

Listing 14: prog/func_select.py

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)
```

Listing 15: prog/nonfunc_select.py

4.4 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)
```

Listing 16: prog/func_sum.py

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)
```

Listing 17: prog/nonfunc_sum.py

4.5 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

```

Listing 18: prog/free_functions.py

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

```

Listing 19: prog/lambda.py

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)

```

Listing 20: prog/free_functions2.py

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)

```

Listing 21: prog/lambdas2.py

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

```

Listing 22: prog/named_lambda.py

5 Inheritance

5.1 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 23: 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: *ConsumerElectronicsDevice*. 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.

5.2 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 ()

```

Listing 24: prog/nature_sleeper.py

Class *HumanBeing* is abstract, since it doesn'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 sourcecode 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 off: 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 sourcecode 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 sourcecode as opposed to a hundred may very well be crucial in successfully understanding the work of a colleague, or your own work of several years back, for that matter.

5.3 Inheriting from library classes

In section 5.2 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>.