Introduction to Programming using Python

Programming Course for Biologists at the Pasteur Institute

by Katja Schuerer, Corinne Maufrais, Catherine Letondal, Eric Deveaud, and Marie-Agnes Petit

Introduction to Programming using Python [http://www.python.org/]: Programming Course for Biologists at the Pasteur Institute

by Katja Schuerer, Corinne Maufrais, Catherine Letondal, Eric Deveaud, and Marie-Agnes Petit

Published February, 1 2007

The objective of this course is to teach programming concepts to biologists. It is thus aimed at people who are not professional computer scientists, but who need a better control of computers for their own research. This programming course is part of a course in informatics for biology [http://www.pasteur.fr/formation/infobio/infobio-en.html]. If you are already a programmer, and if you are just looking for an introduction to Python, you can go to this Python course [http://www.pasteur.fr/recherche/unites/sis/formation/python/] (in Bioinformatics).

PDF version of this course [support.pdf]

This course is still under construction. Comments are welcome.

Handouts for practical sessions (still under construction) will be available on request.

Contact: ieb@pasteur.fr

Table of Contents

List of Figures

List of Tables

List of Examples

List of Exercises

Chapter 1. Introduction

1.1. First session

```
Python 2.4.2 (#1, Dec 20 2005, 16:25:40)
[GCC 4.0.0 (Apple Computer, Inc. build 5026)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 1 + 5
6
>>> 2 * 5
10
>>> 1 / 2
0
```

Is it the right answer?

```
>>> float(1 / 2)
0.0
>>> 1 / 2.0
0.5
>>> float(1)/2
0.5
```

```
>>> ’aaa’
’aaa’
>>> len(’aaa’)
3
```

What happened?

```
>>> len(’aaa’) + len(’ttt’)
6
>>> len(’aaa’) + len(’ttt’) + 1
7
>>> ’aaa’ + ’ttt’
’aaattt’
>>> ’aaa’ + 5
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate ’str’ and ’int’ objects
```

Read carefully the error message, and explain it.

How to protect you from this kind of problem?

```
>>> type(1)
<type ’int’>
>>> type(’1’)
```

```
<type ’str’>
>>> type(1.0)
<type ’float’>
```

You can associate a name to a value:

```
>>> a = 3
>>> a
3
```

The interpreter displays the value (3) of the variable (a).

```
>>> myVar = ’one sentence’
>>> myVar
’one sentence’
>>> 1string = ’one string’
  File "<stdin>", line 1
      1string = ’one string’
            ^
SyntaxError: invalid syntax
```

Read carefully the error message, and explain it.

```
>>> myvar
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name ’myvar’ is not defined
```

What appended?

```
>>> a = 2
>>> a
2
>>> a * 5
10
>>> b = a * 5
>>> b
10
>>> a = 1
>>> b
10
```

Why hasn’t b changed?

What is the difference between:

```
>>> b = a * 5
```

and:

```
>>> b = 5
```

?

```
>>> a = 1    in this case a is a number
>>> a + 2
3
>>> a = ’1’   in this case a is a string
>>> a + 1
 Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate ’str’ and ’int’ objects
```

What do you conclude about the type of a variable?

Some magical stuff, that will be explained later:

```
>>> from string import *
```

We can also perform calculus on strings:

```
>>> codon=’atg’
>>> codon * 3
’atgatgatg’
>>> seq1 = ’agcgccttgaattcggcaccaggcaaatctcaaggagaagttccggggagaaggtgaaga’
>>> seq2 = ’cggggagtggggagttgagtcgcaagatgagcgagcggatgtccactatgagcgataata’
```

How do you concatenate seq1 and seq2 in a single string?

```
>>> seq = seq1 + seq2
```

What is the length of the string seq?

```
>>> len(seq)
120
```

Does the string seq contain the ambiguous ’n’ base?

```
>>>  ’n’ in seq
False
```

Does it contain an adenine base?

```
>>> ’a’ in seq
True
```

```
>>> seq[1]
’g’
```

Why?
Because in computer science, strings are numbered from 0 to string length - 1
so the first character is:

```
>>> seq[0]
’a’
```

Display the 12th base.

```
>>> seq[11]
’t’
```

Find the index of the last character.

```
>>> len(seq)
120
```

So, because we know the sequence length, we can display the last character by:

```
>>> seq[119]
’a’
```

But this is not true for all the sequences we will work on.
Find a more generic way to do it.

```
>>> seq[len(seq) - 1]
’a’
```

Python provides a special form to get the characters from the end of a string:

```
>>> seq[-1]
’a’
>>> seq[-2]
’t’
```

Find a way to get the first codon from the sequence

```
>>> seq[0] + seq[1] + seq[2]
’agc’
```

Python provides a form to get ’slices’ from strings:

```
>>> seq[0:3]
’agc’
>>> seq[3:6]
’gcc’
```

How many of each base does this sequence contains?

```
>>> count(seq, ’a’)
35
>>> count(seq, ’c’)
21
>>> count(seq, ’g’)
44
>>> count(seq, ’t’)
12
```

Count the percentage of each base on the sequence.

Example for the adenine representation

```
>>> long = len(seq)
>>> nb_a = count(seq, ’a’)
>>> (nb_a / long) * 100
0
```

What happened? How 35 bases from 120 could be 0 percent?
This is due to the way the numbers are represented inside the computer.

```
>>> float(nb_a) / long * 100
29.166666666666668
```

Now, let us say that you want to find specific pattern on a DNA sequence:

```
>>> dna = """tgaattctatgaatggactgtccccaaagaagtaggacccactaatgcagatcctgga
tccctagctaagatgtattattctgctgtgaattcgatcccactaaagat"""
>>> EcoRI = ’GAATTC’
>>> BamHI = ’GGATCC’
```

Looking at the sequence you will see that EcoRI is present twice and
BamHI just once:

```
tgaattctatgaatggactgtccccaaagaagtaggacccactaatgcagatcctgga
 ~~~~~~                                                ~~~
tccctagctaagatgtattattctgctgtgaattcgatcccactaaaga
~~~                             ~~~~~~
```

```
>>> count(dna, EcoRI)
0
```

Why ??

```
>>> ’atgc’ == ’atgc’
True
>>> ’atgc’ == ’gcta’
False
>>> ’atgc’ == ’ATGC’
False
```

why are ’atgc’ and ’ATGC’ different?

We can change the case of a string:

```
>>> EcoRI = lower(EcoRI)
>>> EcoRI
’gaattc’
>>> count(dna, EcoRI)
2
>>> find(dna, EcoRI)
1
```

```
>>> find(dna, EcoRI, 2)
88
>>> BamHI = lower(BamHI)
>>> count(dna, BamHI)
0

Why ?

Tip: display the sequence:

>>> dna
’tgaattctatgaatggactgtccccaaagaagtaggacccactaatgcagatcctgga\ntccctagctaagatgtattattctgctgtgaattcgatcc


What is this ’\n’ character?

How to remove it?

>>> dna = replace(dna, ’\n’, ”)
>>> dna
’tgaattctatgaatggactgtccccaaagaagtaggacccactaatgcagatcctggatccctagctaagatgtattattctgctgtgaattcgatccca

>>>find(dna, BamHI)
54

Using the mechanisms we have learnt so far, produce the complement of
the dna sequence.
```

1.2. Documentation

1.3. Why Python

The reasons to use Python as a first language to learn programming are manyfold. First, there are studies that show that Python is well designed for beginners [Wang2002] and the language has been explicitely designed by its author to be easier to learn [Rossum99]. Next, it is more and more often used in bioinformatics as a general-purpose programming language, to both build components and applications [Mangalam2002]. Another very important reason is the object-orientation, that is necessary not just for aesthetics but to scale to modern large-scale programming [Booch94][Meyer97]. Finally, a rich library of modules for scripting and network programming are essential for bioinformatics which very often relies on the integration of existing tools.

1.4. Programming Languages

What can computers do for you? Computers can execute tasks very rapidly, but in order to achieve this they need an accurate description of the task. They can handle a greater amount of input data than you can. But they can not design a strategy to solve problems for you. So if you can not figure out the procedure that solve your problem computers cannot help you.

The Computers own language. Computers do not understand any of the natural languages such as English, French or German. Their proper language, also called *machine language*, is only composed of two symbols “0” and “1”, or power “on” - “off”. They have a sort of a dictionary containing all valid words of this language. These words are the basic instructions, such as “add 1 to some number”, “are two values the same” or “copy a byte of memory to another place”. The execution of these basic instructions are encoded by hardware components of the processor.

Programming languages. Programming languages belongs to the group of *formal languages*. Some other examples of *formal languages* are the system of mathematical expressions or the languages chemists use to describe molecules. They have been invented as intermediate abstraction level between humans and computers.

Why do not use natural languages as programming languages? *Programming languages* are design to prevent problems occurring with *natural language*.

Ambiguity Natural languages are full of ambiguities and we need the context of a word in order to choose the appropriate meaning. “minute” for example is used as a unit of time as a noun, but means tiny as adjective: only the context would distinguish the meaning.

Redundancy Natural languages are full of redundancy helping to solve ambiguity problems and to minimize misunderstandings. When you say “We are playing tennis at the moment.”, “at the moment” is not really necessary but underlines that it is happening now.

Literacy Natural languages are full of idioms and metaphors. The most popular in English is probably “It rains cats and dogs.”. Besides, this can be very complicated even if you speak a foreign language very well.

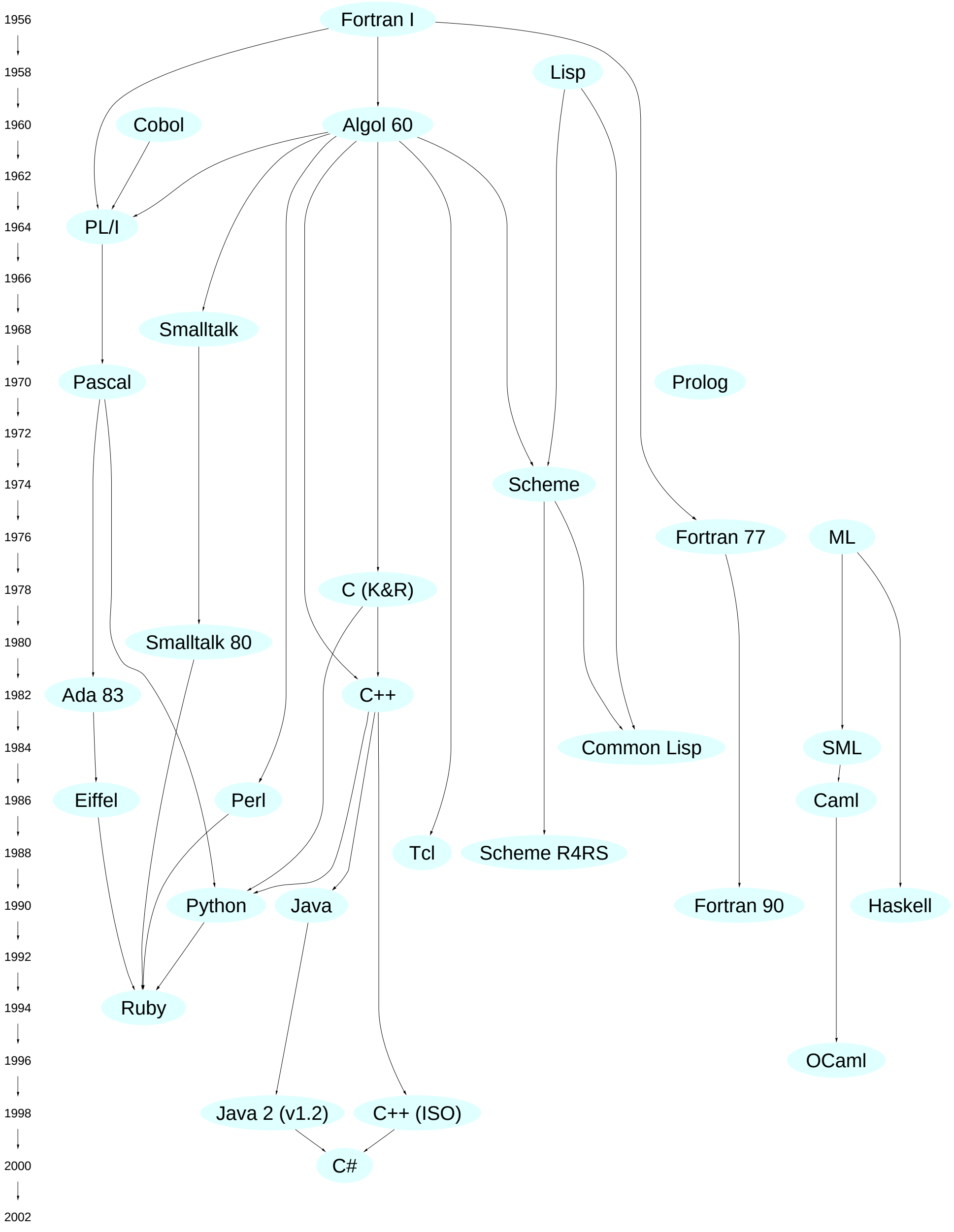
Programming languages are foreign languages for computers. Therefore you need a program that translates your *source code* into the *machine language*. *Programming languages* are voluntarily unambiguous, nearly context free and non-redundant, in order to prevent errors in the translation process.

History of programming languages. It is instructive to try to communicate with a computer in its own language. This let you learn a lot about how processors work. However, in order to do this, you will have to manipulate only 0’s and 1’s. You will need a good memory, but probably you would never try to write a program solving real world problems at this basic level of *machine code*.

Because humans have difficulties to understand, analyze and extract informations of sequences of zeros and ones, they have written a language called *Assembler* that maps the instruction words to synonyms that give an idea of what the instruction does, so for instance **0001** became **add**. *Assembler* increased the legibility of the code, but the instruction set remained basic and depended on the hardware of the computer.

In order to write algorithms for solving more complex problems, there was a need for machine independent *higher level programming languages* with a more elaborated instruction set than the *low level* Assembler. The first ones were *Fortran* and *C* and a lot more have been invented right now. A short history of a subset of programming languages is shown in Figure 1.1.

Figure 1.1. History of programming languages(Source)



Chapter 2. Variables

2.1. Data, values and types of values

In the first session we have explored some basic issues about a DNA sequence. The specific DNA sequence `'atcgat'` was one of our data. For computer scientists this is also a *value*. During the program execution *values* are represented in the memory of the computer. In order to interpret these representations correctly *values* have a *type*.

Type

Types are *sets* of data or *values* sharing some specific properties and their associated operations.

We have modeled the DNA sequence, out of habit, as a *string*.[1] Strings are one of the basic types that Python can handle. In the gc calculation we have seen two other ones: *integers* and *floats*. If you are not sure what sort of data you are working with, you can ask Python about it.

```
>>> type('atcgat')
<type 'str'>
>>> type(1)
<type 'int'>
>>> type('1')
<type 'str'>
```

2.2. Variables or naming values

If you need a *value* more than once or you need the result of a calculation later, you have to give it a name to remember it. Computer scientists also say *binding a value to a name* or *assign a value to a variable*.

Binding

Binding is the process of *naming* a *value*.

Variable

Variables are *names* bound to *values*. You can also say that a *variable* is a *name* that refers to a *value*.

```
>>> EcoRI = 'GAATTC'
```

For Python the model is important because it knows nothing about DNA but it knows a lot about strings.

So the variable `EcoRI` is a name that refers to the string *value* `’GAATTC’`.

The construction used to give names to values is called an *assignment*. Python, as a lot of other programming languages, use the sign = to assign *value* to *variables*. The two sides of the sign = can not be interchanged. The left side has always to be a *variable* and the right side a *value* or a result of a calculation.

Caution

Do not confuse the usage of = in computer science and mathematics. In mathematics, it represents the equality, whereas in Python it is used to give names. So all the following statements are not valid in Python:

```
>>> ’GAATTC’ = EcoRI
SyntaxError: can’t assign to literal
>>> 1 = 2
SyntaxError: can’t assign to literal
```

We will see later how to compare things in Python (Section 11.2).

2.3. Variable and keywords, variable syntax

Python has some conventions for variable names. You can use any letter, the special characters “_” and every number provided you do not start with it. White spaces and signs with special meanings in Python, as “+” and “-” are not allowed.

Important

Python variable names are case-sensitive, so `EcoRI` and `ecoRI` are not the same variable.

```
>>> EcoRI = ’GAATTC’
>>> ecoRI
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name ’ecoRI’ is not defined
>>> ecori
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name ’ecori’ is not defined
>>> EcoRI
’GAATTC’
```

Among the words you can construct with these letters, there are some reserved words for Python and can not be used as variable names. These *keywords* define the language rules and have special meanings in Python. Here is the list of all of them:

```
and       assert    break     class     continue def       del       elif
else      except    exec      finally   for      from      global    if
import    in        is        lambda    not      or        pass      print
raise     return    try       while     yield
```

2.4. Namespaces or representing variables

How does Python find the value referenced by a variable? Python stores *bindings* in a *Namespace*.

Namespace

A *namespace* is a mapping of variable names to their values.

You can also think about a *namespace* as a sort of *dictionary* containing all defined variable names and the corresponding reference to their values.

Reference

A *reference* is a sort of pointer to a location in memory.

Therefore you do not have to know where exactly your value can be found in memory, Python handles this for you via *variables*.

Figure 2.1. Namespace

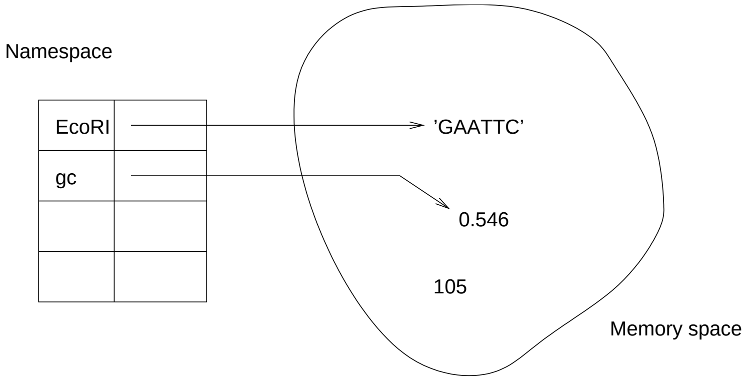


Figure 2.1 shows a representation of some namespace. *Values* which have not been referenced by a *variable*, are not accessible to you, because you can not access the memory space directly. So if a result of a calculation is returned, you can use it directly and forget about it after that. Or you can create a variable holding this value and then access this value via the variable as often as you want.

```
>>> from string import *
>>> cds = """atgagtgaacgtctgagcattaccccgctggggccgtatatcggcgcacaaa
tttcgggtgccgacctgacgcgcccgttaagcgataatcagtttgaacagctttaccatgcggtg
ctgcgccatcaggtggtgtttctacgcgatcaagctattacgccgcagcagcaacgcgcgctggc
ccagcgttttggcgaattgcatattcaccctgtttacccgcatgccgaaggggttgacgagatca
tcgtgctggatacccataacgataatccgccagataacgacaactggcataccgatgtgacattt
attgaaacgccacccgcaggggcgattctggcagctaaagagttaccttcgaccggcggtgatac
gctctggaccagcggtattgcggcctatgaggcgctctctgttcccttccgccagctgctgagtg
ggctgcgtgcggagcatgatttccgtaaatcgttcccggaatacaaataccgcaaaaccgaggag
gaacatcaacgctggcgcgaggcggtcgcgaaaaacccgccgttgctacatccggtggtgcgaac
gcatccggtgagcggtaaacaggcgctgtttgtgaatgaaggctttactacgcgaattgttgatg
tgagcgagaaagagagcgaagccttgttaagtttttttgtttgcccatatcaccaaaccggagttt
caggtgcgctggcgctggcaaccaaatgatattgcgatttgggataaccgcgtgacccagcacta
tgccaatgccgattacctgccacagcgacggataatgcatcgggcgacgatccttggggataaac
cgttttatcgggcggggtaa""".replace("\n","")

>>> float(count(cds, ’G’) + count(cds, ’C’))/ len(cds)
0.54460093896713613
```

Here the result of the gc-calculation is lost.

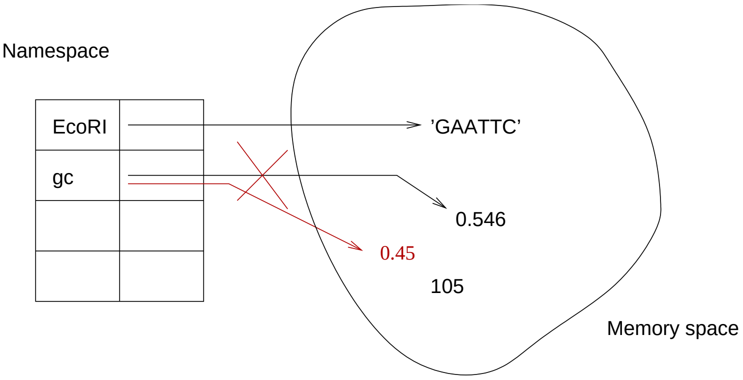
```
>>> gc = float(count(cds, ’G’) + count(cds, ’C’))/ len(cds)
>>> gc
0.54460093896713613
```

In this example you can remember the result of the gc calculation, because it is stored in the variable `gc`.

2.5. Reassignment of variables

It is possible to *reassign* a new value to an already defined variable. This will destroy the reference to its former value and create a new binding to the new value. This is shown in Figure 2.2.

Figure 2.2. Reassigning values to variables



Note

In Python, it is possible to reassign a new value with a different type to a variable. This is called *dynamic typing*, because the type of the variable is assigned dynamically. Note that this is not the case in all programming languages. Sometimes, as in **C**, the type of variables is assigned statically and has to be declared before use. This is some way more secure because types of variables can be checked only by examining the source code, whereas that is not possible if variables are dynamically typed.

Chapter 3. Statements, expressions and functions

3.1. Statements

In our first practical lesson, the first thing we did, was the invocation of the Python interpreter. During the first session we entered *statements* that were read, analyzed and executed by the interpreter.

Statement

Statements are *instructions* or commands that the Python interpreter can execute. Each statement is read by the interpreter, analyzed and then executed.

3.2. Sequences or chaining statements

Program

A *program* is a *sequence* of statements that can by executed by the Python interpreter.

Sequence

Sequencing is a simple *programming feature* that allows to chain instructions that will be executed one by one from top to bottom.

Later we are going to learn more complicated ways to control the flow of a program, such as *branching* and *repetition*.

3.3. Functions

Function

Functions are *named sequences* of statements that execute some task.

We have already used functions, such as:

```
>>> type(’GAATTC’)
<type ’str’>
>>> len(cds)
852
```

For example `len` is a function that calculates the length of things and we asked here for the length of our DNA sequence `cds`.

Function call

Function calls are statements that execute or *call* a function. The Python syntax of *function calls* is the *function name* followed by a comma separated list of *arguments* inclosed into parentheses. Even if a function does not take any argument the parentheses are necessary.

Differences between function calls and variables. As *variable names*, *function names* are stored in a namespace with a reference to their corresponding sequence of statements. When they are called, their name is searched in the namespace and the reference to their sequence of statements is returned. The procedure is the same as for variable names. But unlike them, the following parentheses indicate that the returned value is a sequence of statements that has to be executed. That's why they are even necessary for functions which are called without arguments.

Arguments of functions

Arguments are *values* provided to a function when the function is called. We will see more about them soon.

3.4. Operations

Operations and Operators

Operations are "basic" *functions* with their own syntax.

They have a special *Operator* (a sign or a word) that is the same as a function name. *Unary Operators*, operations which take one argument, are followed by their argument, and *secondary operators* are surrounded by their two arguments.

Here are some examples:

```
>>> 'GTnnAC' + 'GAATTC'
'GTnnACGAATTC'
>>> 'GAATTC' * 3
'GAATTCGAATTCGAATTC'
>>> 'n' in 'GTnnAC'
1
```

This is only a simpler way of writing these functions provided by Python, because humans are in general more familiar with this syntax closely related to the mathematical formal language.

3.5. Composition and Evaluation of Expressions

Composition and Expression

Composition is a way to combine functions. The combination is also called an *Expression*.

We have already used it. Here is the most complex example we have seen so far:

```
>>> float(count(cds, ’G’) + count(cds, ’C’)) / len(cds)
0.54460093896713613
```

What is happening when this *expression* is executed? The first thing to say is that it is a mixed expression of operations and function calls. Let’s start with the function calls. If a function is called with an argument representing a composed expression, this one is executed first and the result value is passed to the calling function. So the `cds` variable is evaluated, which returns the value that it refers to. This value is passed to the `len` function which returns the length of this value. The same happens for the `float` function. The operation `count(cds, ’G’) + count(cds, ’C’)` is evaluated first, and the result is passed as argument to `float`.

Let’s continue with the operations. There is a precedence list, shown in Table 3.1, for all operators, which determines what to execute first if there are no parentheses, otherwise it is the same as for function calls. So, for the operation `count(cds, ’G’) + count(cds, ’C’)` the two `count` functions are executed first on the value of the `cds` variable and “G” and “C” respectively. And the two counts are added. The result value of the addition is then passed as argument to the `float` function followed by the division of the results of the two functions `float` and `len`.

Table 3.1. Order of operator evaluation (highest to lowest)

Operator	Name
`+x, -x, ~x`	Unary operators
`x ** y`	Power (right associative)
`x * y, x / y,x % y`	Multiplication, division, modulo
`x + y, x - y`	Addition, subtraction
`x << y, x >> y`	Bit shifting
`x & y`	Bitwise and
`x | y`	Bitwise or
`x < y, x <= y, x > y, x >= y, x == y, x != y, x <> y, x is y, x is not y, x in s, x not in s<`	Comparison, identity, sequence membership tests
`not x`	Logical negation
`x and y`	Logical and
`lambda args:  expr`	Anonymous function

So, as in mathematics, the innermost function or operation is evaluated first and the result is passed as argument to the enclosing function or operation. It is important to notice that *variables* are evaluated and only their *values* are passed as argument to the function. We will have a closer look at this when we talk about function definitions in Section 8.1.

Exercise 3.1. Composition

Have a look at this example. Can you explain what happens? If you can’t please read this section once again.

```
>>> from string import *
>>> replace(replace(replace(cds, ’A’, ’a’), ’T’, ’A’), ’a’, ’T’)
```

Chapter 4. Communication with outside

4.1. Output

We saw in the previous chapter how to export information outside of the program using the `print` statement. Let's give a little bit more details of its use here.

The `print` statements can be followed by a variable number of values separated by commas. Without a value `print` puts only a newline character on the *standard output*, generally the screen. If values are provided, they are transformed into strings, and then are written in the given order, separated by a space character. The line is terminated by a newline character. You can suppress the final newline character by adding a comma at the end of the list. The following example illustrates all these possibilities:

```
#! /usr/local/bin/python

from string import *

dna = "ATGCAGTGCATAAGTTGAGATTAGAGACCCGACAGTA"

gc = float(count(dna, 'G') + count(dna, 'C'))/ len(dna)

print gc

print "the gc percentage of dna:", dna, "is:", gc
print
print "the gc percentage of dna:", dna
print "   is:", gc
print
print "the gc percentage of dna:", dna,
print "is:", gc
```

producing the following output:

```
caroline:~> python print_gc.2.py
0.432432432432
the gc percentage of dna: ATGCAGTGCATAAGTTGAGATTAGAGACCCGACAGTA is: 0.432432432432

the gc percentage of dna: ATGCAGTGCATAAGTTGAGATTAGAGACCCGACAGTA
   is: 0.432432432432

the gc percentage of dna: ATGCAGTGCATAAGTTGAGATTAGAGACCCGACAGTA is: 0.432432432432
caroline:~>
```

4.2. Formatting strings

Important

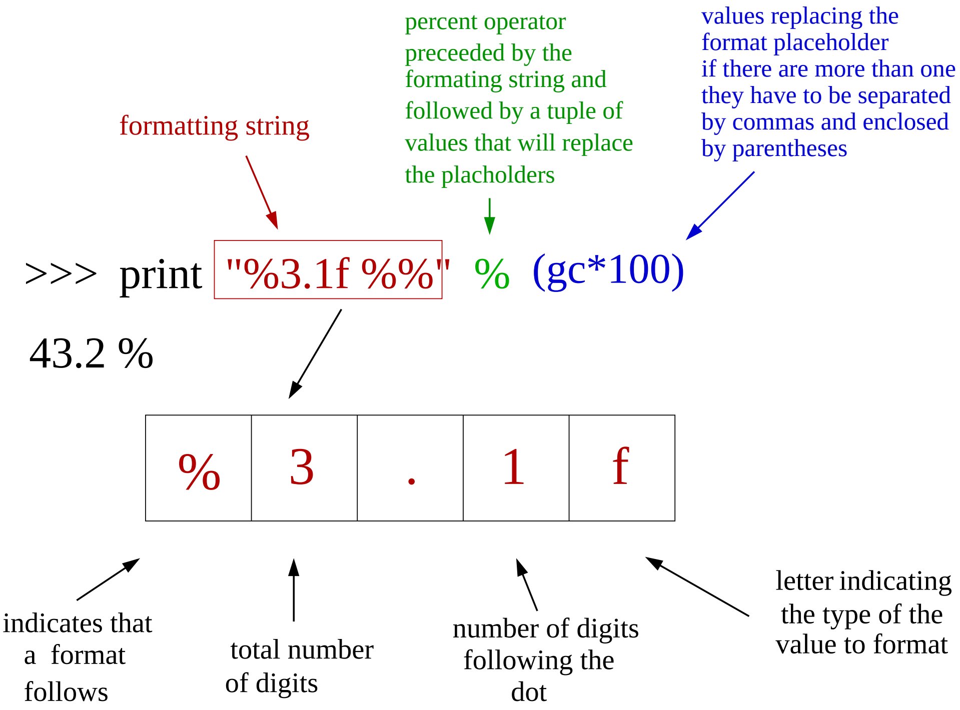
All data printed on the screen have to be character data. But values can have different types. Therefore they have to be transformed into strings beforehand. This transformation is handled by the `print` statement.

It is possible to control this transformation when a specific format is needed. In the examples above, the float value of the gc calculation is written with lots of digits following the dot which are not very significant. The next example shows a more reasonable output:

```
>>> print "%.3f" % gc
0.432
>>> print "%3.1f %%" % (gc*100)
43.2 %
>>> print "the gc percentage of dna: %10s... is: %4.1f %%." % (dna, gc*100)
the gc percentage of dna: ATGCAGTGCA... is: 43.2 %
```

Figure 4.1 shows how to interpret the example above. The `%` (modulo) operator can be used to format strings. It is preceded by the formatting template and followed by a comma separated list of values enclosed in parentheses. These values replace the formatting place holders in the template string. A place holder starts with a % followed by some modifiers and a character indicating the type of the value. There has to be the same number of values and place holders.

Figure 4.1. Interpretation of formatting templates


formatting string
percent operator preceeded by the formating string and followed by a tuple of values that will replace the placholders
values replacing the format placeholder if there are more than one they have to be separated by commas and enclosed by parentheses
>>> print "%3.1f %%" % (gc*100)
43.2 %
%
3
.
1
f
indicates that a format follows
total number of digits
number of digits following the dot
letter indicating the type of the value to format

.

Table 4.1 provides the characters that you can use in the formatting template and Table 4.2 gives the modifiers of the formatting character.

Important

Remember that the type of a formatting result is a string and no more the type of the input value.

```
>>> "%.1f" % (gc*100)
’43.2’
>>> res = "%.1f" % (gc*100)
>>> at = 100 - res
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: unsupported operand type(s) for -: ’int’ and ’str’
>>> res
’43.2’
>>>
```

Table 4.1. String formatting: Conversion characters

Formatting character	Output	Example	Result
`d,i`	decimal or long integer	`"%d" % 10`	’10’
`o,x`	octal/hexadecimal integer	`"%o" % 10`	’12’
`f,e,E`	normal, ’E’ notation of floating point numbers	`"%e" % 10.0`	’1.000000e+01’
`s`	strings or any object that has a `str()` method	`"%s" % [1, 2, 3]`	’[1, 2, 3]’
`r`	string, use the `repr()` function of the object	`"%r" % [1, 2, 3]`	’[1, 2, 3]’
`%`	literal %		

Table 4.2. String formatting: Modifiers

Modifier	Action	Example	Result
name in parentheses	selects the key name in a mapping object	`"%(num)d %(str)s" % { ’num’:1, ’str’:’dna’}`	’1 dna’
`-,+`	left, right alignment	`"%-10s" % "dna"`	’dna_______’
`0`	zero filled string	`"%04i" % 10`	’0010’
number	minimum field width	`"%10s" % "dna"`	’_______dna’
`. number`	precision	`"%4.2f" % 10.1`	’10.10’

4.3. Input

As you can print results on the screen, you can read data from the keyboard which is the standard input device. Python provides the `raw_input` function for that, which is used as follows:

```
>>> nb = raw_input("Enter a number, please:")
Enter a number, please:12
```

The prompt argument is optional and the input has to be terminated by a return.

Important

`raw_input` always returns a string, even if you entered a number. Therefore you have to convert the string input by yourself into whatever you need. Table 4.3 gives an overview of all possible type conversion function.

```
>>> nb
’12’
>>> type(nb)
<type ’str’>
```

```
>>> nb = int(nb)
>>> nb
12
>>> type(nb)
<type ’int’>
```

Notice that a user can enter whatever he wants. So, the input is probably not what you want, and the type conversion can therefore fail. It is careful to test before converting input strings.

```
>>> nb = raw_input("Please enter a number:")
Please enter a number:toto
>>> nb
’toto’
>>> int(nb)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: invalid literal for int(): toto
```

The following function controls the input:

```
def read_number():
    while 1:
        nb = raw_input("Please enter a number:")
        try:
            nbconv = int(nb)
        except:
            print nb, "is not a number."
            continue
        else:
            break
    return nb
```

and produces the following output:

```
>>> read_number()
Please enter a number:toto
toto is not a number.
Please enter a number:12
’12’
```

Table 4.3. Type conversion functions

Function	Description
`int(x [,base])`	converts `x` to an integer
`long(x [,base])`	converts `x` to a long integer

`float(x)`	converts `x` to a floating-point number
`complex(real [,imag])`	creates a complex number
`str(x)`	converts `x` to a string representation
`repr(x)`	converts `x` to an expression string
`eval(str)`	evaluates `str` and returns an object
`tuple(s)`	converts a sequence object to a tuple
`list(s)`	converts a sequence object to a list
`chr(x)`	converts an integer to a character
`unichr(x)`	converts an integer to a Unicode character
`ord(c)`	converts a character to its integer value
`hex(x)`	converts an integer to a hexadecimal string
`oct(x)`	converts an integer to an octal string

Chapter 5. Program execution

5.1. Executing code from a file

Until now we have only worked *interactively* during an *interpreter session*. But each time we leave our session all definitions made are lost, and we have to re-enter them again in the next session of the interpreter whenever we need them. This is not very convenient. To avoid that, you can put your code in a file and then pass the file to the **Python** interpreter. Here is an example:

Example 5.1. Executing code from a file

Take the code for the cds translation as example and put it in a file named `gc.py`:

```
from string import *

cds = """atgagtgaacgtctgagcattaccccgctggggccgtatatcggcgcacaaa
tttcgggtgccgacctgacgcgcccgttaagcgataatcagtttgaacagctttaccatgcggtg
ctgcgccatcaggtggtgtttctacgcgatcaagctattacgccgcagcagcaacgcgcgctggc
ccagcgttttggcgaattgcatattcaccctgtttacccgcatgccgaaggggttgacgagatca
tcgtgctggatacccataacgataatccgccagataacgacaactggcataccgatgtgacattt
attgaaacgccacccgcaggggcgattctggcagctaaagagttaccttcgaccggcggtgatac
gctctggaccagcggtattgcggcctatgaggcgctctctgttcccttccgccagctgctgagtg
ggctgcgtgcggagcatgatttccgtaaatcgttcccggaatacaaataccgcaaaaccgaggag
gaacatcaacgctggcgcgaggcggtcgcgaaaaacccgccgttgctacatccggtggtgcgaac
gcatccggtgagcggtaaacaggcgctgtttgtgaatgaaggctttactacgcgaattgttgatg
tgagcgagaaagagagcgaagccttgttaagtttttttgtttgcccatatcaccaaaccggagttt
caggtgcgctggcgctggcaaccaaatgatattgcgatttgggataaccgcgtgacccagcacta
tgccaatgccgattacctgccacagcgacggataatgcatcgggcgacgatccttggggataaac
cgttttatcgggcggggtaa""".replace("\n","")

gc = float(count(cds, ’g’) + count(cds, ’c’))/ len(cds)

print gc
```

and now pass this file to the interpreter:

```
caroline:~/python_cours> python gc.py
0.54460093896713613
```

Tip

You can name your file as you like. However, there is a convention for files containing python code to have a **py** extension.

You can also make your file executable if you put the following line at the beginning of your file, indicating that this file has to be executed with the **Python** interpreter:

```
#! /usr/local/bin/python
```

(Don’t forget to set the `x` execution bit under UNIX system.) Now you can execute your file:

```
caroline:~/python_cours> ./gc.py
0.54460093896713613
```

This will automatically call the **Python** interpreter and execute all the code in your file.

You can also load the code of a file in a interactive interpreter session with the `-i` option:

```
caroline:~/python_cours> python -i gc.py
0.54460093896713613
>>>
```

This will start the interpreter, execute all the code in your file and than give you a **Python** prompt to continue:

```
>>> cds
’atgagtgaacgtctgagcattaccccgctggggccgtatatcggcgcacaaatttcgggtgccgacctgacgcgcccgttaagcgataatcagtttgaac

>>>cds="atgagtgaacgtctgagcattaccccgctggggccgtatatcggcgcacaaatttcgggtgccgacctgacgcgcccgtt"

>>>cds
’atgagtgaacgtctgagcattaccccgctggggccgtatatcggcgcacaaatttcgggtgccgacctgacgcgcccgtt’
>>>gc
0.54460093896713613
```

Important

It is important to remember that the **Python** interpreter executes code *from top to bottom*, this is also true for code in a file. So, pay attention to *define* things before you *use* them.

Exercise 5.1. Execute code from a file

Take all expressions that we have written so far and put them in a file.

Important

Notice that you have to ask explicitly for printing a result when you execute some code from a file, while an *interactive* interpreter session the result of the execution of a statement is printed automatically. So to view the result of the `translate` function in the code above, the `print` statement is necessary in the file version, whereas during an *interactive* interpreter session we have never written it.

5.2. Interpreter and Compiler

Let's introduce at this point some concepts of *execution* of programs written in *high level programming languages*. As we have already seen, the only language that a computer can understand is the so called *machine language*. These languages are composed of a set of basic operations whose execution is implemented in the hardware of the processor. We have also seen that high level programming languages provide a machine-independent level of abstraction that is higher than the machine language. Therefore, they are more adapted to a human-machine interaction. But this also implies that there is a sort of translator between the high level programming language and the machine languages. There exists two sorts of *translators*:

Interpreter

An **Interpreter** is a *program* that implements or simulates a *virtual machine* using the base set of instructions of a *programming language* as its *machine language*.

You can also think of an **Interpreter** as a *program* that implements a library containing the implementation of the basic instruction set of a *programming language* in *machine language*.

An **Interpreter** reads the statements of a program, analyzes them and then executes them on the virtual machine or calls the corresponding instructions of the library.

Interactive interpreter session

During an *interactive* interpreter session the statements are not only read, analyzed and executed but the result of the *evaluation of an expression* is also *printed*. This is also called a **READ - EVAL - PRINT** loop.

Important

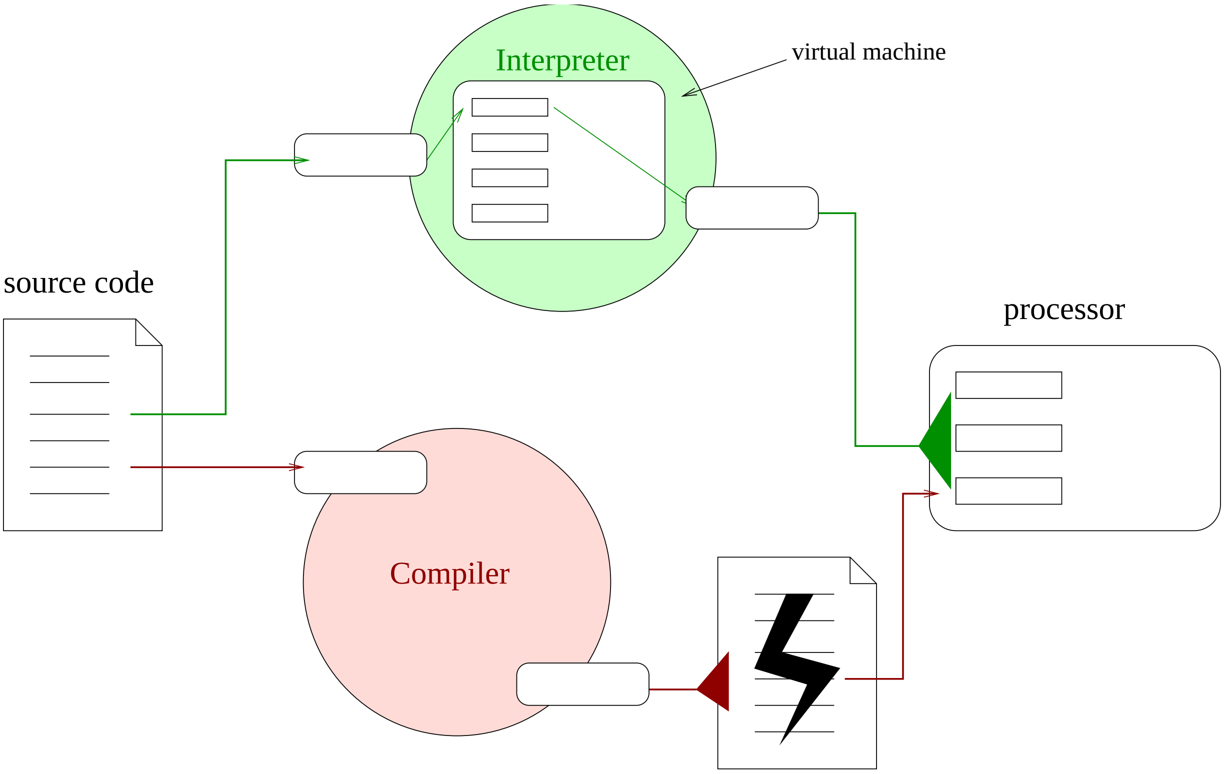
Pay attention, the *READ - EVAL - PRINT* loop is only entered in an *interactive* session. If you ask the *interpreter* to execute code in a file, results of expression evaluations *are not* printed. You have to do this by yourself.

Compiler

A **Compiler** is a *program* that translates code of a *programming language* in *machine code*, also called *object code*. The object code can be executed directly on the machine where it was compiled.

Figure 5.1 compares the usage of interpreters and compilers.

Figure 5.1. Comparison of compiled and interpreted code

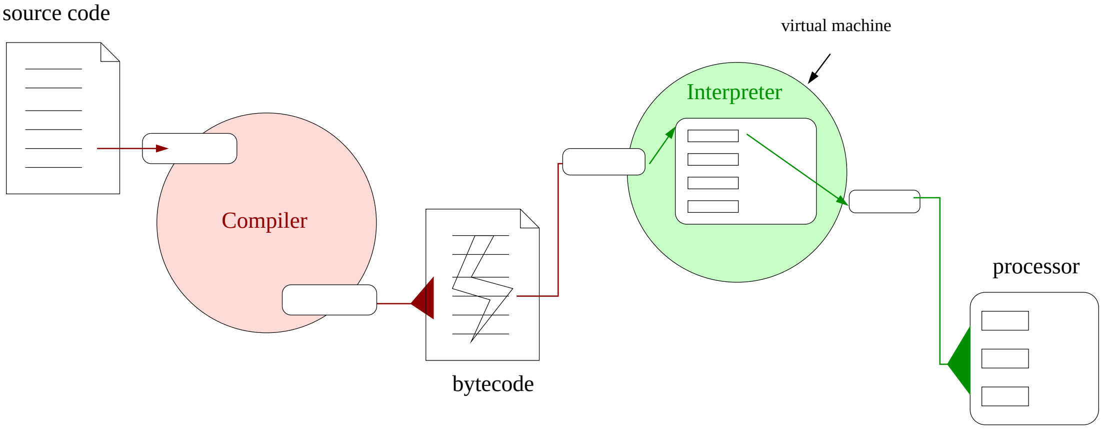


So using a *compiler* separates *translation* and *execution* of a program. In contrast of an *interpreted* program the *source code* is translated only once.

The *object code* is *machine-dependent* meaning that the *compiled* program can only be executed on a machine for which it has been compiled, whereas an *interpreted* program is not *machine-dependent* because the *machine-dependent* part is in the interpreter itself.

Figure 5.2 illustrates another concept of program execution that tries to combine the advantage of more effective execution of compiled code and the advantage of machine-independence of interpreted code. This concept is used by the **JAVA** programming language for example and in a more subtle way by **Python**.

Figure 5.2. Execution of byte compiled code



In this case the *source code* is translated by a *compiler* in a sort of *object code*, also called *byte code* that is then executed by an *interpreter* implementing a *virtual machine* using this *byte code*. The execution of the *byte code* is faster than the interpretation of the *source code*, because the major part of the analysis and verification of the source code is done during the compilation step. But the *byte code* is still *machine-independent* because the machine-dependent part is implemented in the *virtual machine*. We will see later how this concept is used in **Python** (Section 15.1).

Chapter 6. Strings

So far we have seen a lot about strings. Before giving a summary about this data type, let us introduce a new syntax feature.

6.1. Values as objects

We have seen that *strings* have a *value*. But Python *values* are more than that. They are *objects*.

Object

Objects are things that know more than their *values*. In particular, you can ask them to perform specialized tasks that only they can do.

Up to now we have used some special functions handling string data available to us by the up to now *magic* statement `from string import *`. But *strings* themselves know how to execute all of them and even more. Look at this:

```
>>> motif = "gaattc"
>>> motif.upper()
’GAATTC’
>>> motif
’gaattc’
>>> motif.isalpha()
1
>>> motif.count(’n’)
0

>>> motif = ’GAATTC_’
>>> motif + motif
’GAATTC_GAATTC_’
>>> motif * 3
’GAATTC_GAATTC_GAATTC_’
```

At the first glance this looks a little bit strange, but you can read the `.` (dot) operator as: “ask object `motif` to *do* something” as: transform `motif` in an uppercase string (`upper`), ask whether it contains only letters (`isalpha`) or count the number of “n” characters.

Objects as namespaces. How does it work? All objects have their own namespace containing all variable and function names that are defined for that object. As already describeb in Section 2.4 you can see all names defined for an object by using the `dir` function:

```
>>> dir(motif)
[’__add__’, ’__class__’, ’__contains__’, ’__delattr__’, ’__eq__’, ’__ge__’,
’__getattribute__’, ’__getitem__’, ’__getslice__’, ’__gt__’, ’__hash__’,
’__init__’, ’__le__’, ’__len__’, ’__lt__’, ’__mul__’, ’__ne__’, ’__new__’,
’__reduce__’, ’__repr__’, ’__rmul__’, ’__setattr__’, ’__str__’, ’capitalize’,
 ’center’, ’count’, ’decode’, ’encode’, ’endswith’, ’expandtabs’, ’find’,
’index’, ’isalnum’, ’isalpha’, ’isdigit’, ’islower’, ’isspace’, ’istitle’,
’isupper’, ’join’, ’ljust’, ’lower’, ’lstrip’, ’replace’, ’rfind’, ’rindex’,
’rjust’, ’rstrip’, ’split’, ’splitlines’, ’startswith’, ’strip’, ’swapcase’,
’title’, ’translate’, ’upper’]
```

The dot operator is used to access this namespace. It will look up in the namespace of the object for the name following the dot operator.

```
>>> motif.__class__
<type ’str’>
>>> motif.replace(’a’, ’A’)
’gAAttc’
```

Variables and functions defined in object namespaces are called *attributes* and *methods* of the object.

Attribute

An *attribute* is a *variable* defined in a namespace of an object, which can only be accessed via the object himself.

Method

Methods are *functions* defined in a namespace of an *object*.

This is just a little introduction to objects making it possible to use the *object syntax* for the basic types in Python. We will give further explanation into *object-oriented programming* in Chapter 17.

6.2. Working with strings

Strings

Strings are sequences or *ordered collections* of *characters*.

You can write them in Python using quotes, double quotes, triple quotes or triple double quotes. The triple quote notation permits to write strings spanning multiple lines with keeping any line feed.

```
>>> ’ATGCA’
’ATGCA’
>>> "ATGCA"
’ATGCA’
>>> """ATGATA
... AGAGA"""
```

```
’ATGATA\nAGAGA’
```

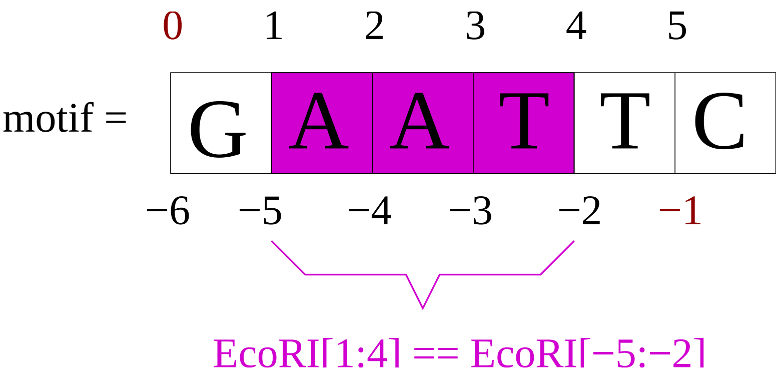
The first thing that we would sum up is how to extract characters or substrings . Characters are accessible using their position in the string which has to be enclosed into brackets following the string. The position number can be positive or negative, which means starting at the beginning or the end of the string. Substrings are extracted by providing an interval of the start and end position separated by a colon. Both positions are optional which means either to start at the beginning of the string or to extract the substring until the end. When accessing characters, it is forbidden to access position that does not exist, whereas during substring extraction, the longest possible string is extracted.

```
>>> motif = ’GAATTC’
>>> motif[0]
’G’
>>> motif[-1]
’C’

>>> motif[0:3]
’GAA’
>>> motif[1:3]
’AA’
>>> motif[:3]
’GAA’
>>> motif[3:]
’TTC’
>>> motif[3:6]
’TTC’
>>> motif[3:-1]
’TT’
>>> motif[-3:-1]
’TT’
>>> motif[:]
’GAATTC’

>>> motif[100]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
>>> motif[3:100]
’TTC’
>>> motif[3:2]
”
```

Caution

Figure 6.1 compares positive and negative indices. Be careful, forward string indices starts always with 0, whereas backward string indices starts with -1.

Figure 6.1. String indices



Caution

It is also important to notice that the character at the end position during a substring extraction is never included in the extracted substring.

Warning

Strings are immutable in Python. This means you can neither change characters or substrings. You have always to create a new copy of the string.

```
>>> motif[1] = 'n'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment

>>> motif[:1] + 'n' + motif[2:]
'GnATTC'
```

A list of all other methods, function and operators and their action on string objects are summarized in Table 6.1 and Table 6.2).

Table 6.1. String methods, operators and builtin functions

Method, Operator, Function	Description
`s + t`	Concatenation
`s * 3`	Repetition
`len(s)`	Returns the length of s
`min(s), max(s)`	Returns the “smallest”, “largest” character of s, depending on their position in the ASCII code

`s.capitalize()`	Capitalize the first character of s
`s.center([width])`	Centers s in a field of length width
`s.count(sub [,start [, end]])`	Counts occurrences of *sub* between *start* and *end*
`s.encode([encoding [, errors]])`	Encode s using *encoding* as code and *error*
`s.expandtabs([tabsize])`	Expands tabs
`s.find(sub [, start [, end]])`	Finds the first occurrence of *sub* between *start* and *end*
`s.index(sub [,start [, end]])`	same as `find` but raise an exception if no occurrence is found
`s.join(words)`	Joins the list of *words* with s as delimiter
`s.ljust(width)`	Left align s in a string of length *width*
`s.lower()`	Returns a lowercase version of s
`s.lstrip()`	Removes all leading whitespace characters of s
`s.replace(old, new [, maxrep])`	Replace maximal *maxrep* versions of substring *old* with substring *new*
`s.rfind(sub [, start [, end]])`	Finds the last occurrence of substring *sub* between *start* and *end*
`s.rindex(sub [,start [, end]])`	Same as `rfind` but raise an exception if *sub* does not exists
`s.rjust(width)`	Right-align s in a string of length *width*
`s.rstrip()`	Removes trailing whitespace characters
`s.split([sep [, maxsplit]]))`	Split s into maximal *maxsplit* words using *sep* as separator (default whitespace)
`s.splitlines([keepends])`	Split s into lines, if *keepends* is 1 keep the trailing newline
`s.strip()`	Removes trailing and leading whitespace characters
`s.swapcase()`	Returns a copy of s with lowercase letters turn into uppercase and vice versa
`s.title()`	Returns a title-case version of s (all words capitalized)
`s.translate(table [, delchars])`	Translate s using translation table *table* and removing characters in string *delchars*
`s.upper()`	Returns an uppercase version of s

Table 6.2. Boolean methods and operators on strings

Method or operator	Description
`s < <=, >=, > t`	Checks if s appears before, before or at the same point, after or at the same point, after than t in an alphabetically sorted dictionary
`s < <= t >=, > r`	Checks if r appears between s and t in an alphabetically sorted dictionary
`s ==, !=, is, not is t`	Checks the identity or difference of s and t
`c in s, c not in s`	Checks if character *c* appears in s
`s.endswith(suffix [,start [,end]])`	Checks if s ends with *suffix*
`s.isalnum()`	Checks whether all characters are alphanumeric

`s.isalpha()`	Checks whether all characters are alphabetic
`s.isdigit()`	Checks whether all characters are digits
`s.islower()`	Checks whether all characters are lowercase
`s.isspace()`	Checks whether all characters are whitespace
`s.istitle()`	Checks whether s is title-case meaning all words are capitalized
`s.isupper()`	
`s.startswith(prefix [, start [, end]]))`	Checks whether s starts with *prefix* between *start* and *end*

Chapter 7. Branching and Decisions

7.1. Conditional execution

Sometimes the continuation of a program depends on a *condition*. We would either execute a part of the program if this condition is fulfilled or adapt the behavior of the program depending on the truth value of the condition.

Branching or conditional statements

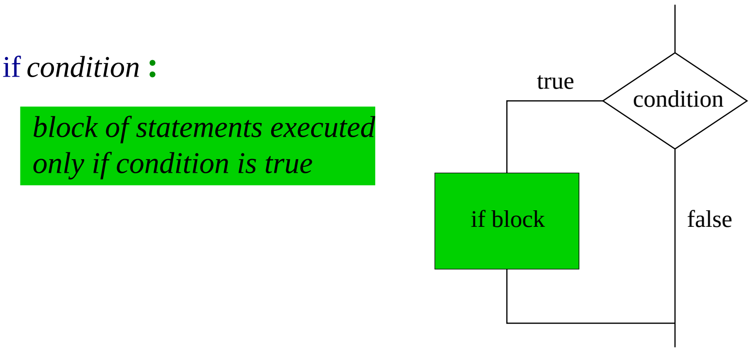
Branching is a feature provided by programming languages that makes it possible to execute a sequence of statements among several possibilities.

The simplest case of a conditional statement can be expressed by the `if` statement.

```
>>> from string import *

>>> seq = 'ATGAnnATG'
>>> if 'n' in seq:
...     print "sequence contains undefined bases"                          ❶
...     nb = count(seq, 'n')
sequence contains undefined bases
```

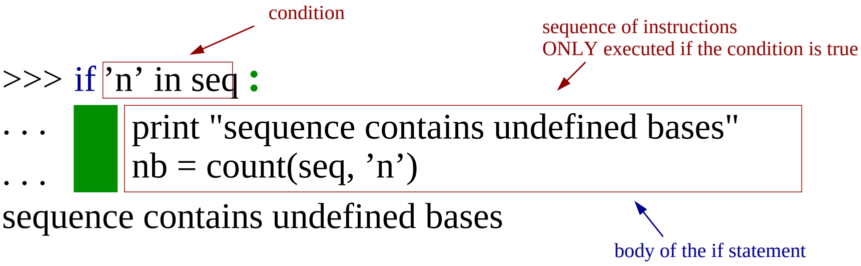
Figure 7.1 shows a general schema of a simple conditional statement.

Figure 7.1. Flow of execution of a simple condition



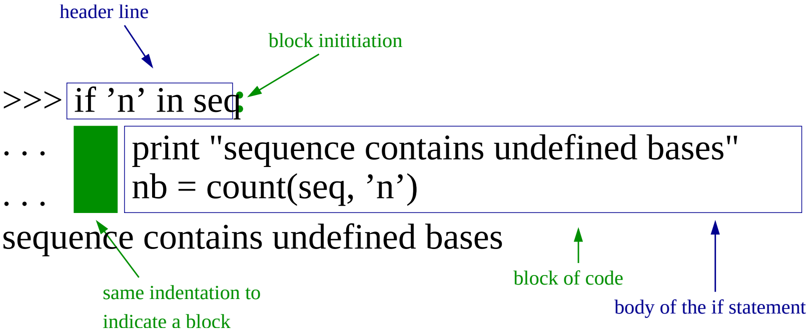
The `if` statement has to be followed by a *condition*. Then a *block* is opened by a colon. This block contains the *sequence* of statements that has to be executed if the *condition* is fulfilled. Figure 7.3 and Figure 7.2 highlight the structural parts of the `if` statement.

Figure 7.2. If statement



```
>>> if 'n' in seq :
...     print "sequence contains undefined bases"
...     nb = count(seq, 'n')
sequence contains undefined bases
```

Figure 7.3. Block structure of the if statement



```
>>> if 'n' in seq:
...     print "sequence contains undefined bases"
...     nb = count(seq, 'n')
sequence contains undefined bases
```

7.2. Conditions and Boolean expressions

The *condition* in the `if` statement has to be a *boolean expression*.

Conditions or Boolean expressions

Conditional or boolean expressions are *expressions* that are either *true* or *false.*

Here are some examples of simple *boolean expressions* or conditions:

```
>>> 1 < 0
False
>>> 1 > 0
True
>>> 1 == 0
False
>>> 'n' in 'ATGCGTAnAGTA'
True
>>> 'A' > 'C'
False
>>> 'AG' < 'AA'
```

```
False
>>> ’AG’ != ’AC’
True
>>> len(’ATGACGA’) >= 10
False
```

In Python the value *true* is represented by `1` and the value *false* by `0`.

Table 7.1 lists all *boolean operators* and their action on strings and numbers.

Table 7.1. Boolean operators

Operator	Action on strings	Action on numbers
`<, <=, >=, >`	alphabetically sorted, lower/lower or equal/greater or equal/greater than	lower/lower or equal/greater or equal/greater than
`==, !=, is, is not`[a]	identity	identity
`in, not in`	membership	-

[a] We will explain the difference between these two identity operators later (Section 11.2).

Important

Do not confuse the assignment sign `=` with the logical operator `==`. The second one is used to compare two things and check their equality whereas the first one is used to bound values to variables. Python does not accept an assignment as a condition but there are other programming languages that use the same syntax for these two statements, but they do not warn when you use `=` instead of `==`.

7.3. Logical operators

The three logical operators `not`, `and` and `or` enable you to compose boolean expressions and by this way to construct more complex conditions. Here are some examples:

```
>>> seq =  ’ATGCnATG’
>>> ’n’ in seq or ’N’ in seq
True
>>> ’A’ in seq and ’C’ in seq
True
>>> ’n’ not in seq
False
>>> len(seq) > 100 and ’n’ not in seq
False
>>> not len(seq) > 100
True
```

Caution

The `or` operation is not an *exclusive* or as it is sometimes used in the current languages. An `or` expression is also true if both subexpressions are true.

```
>>> seq = ’ATGCnATG’
>>> ’A’ in seq or ’C’ in seq
True
```

7.4. Alternative execution

If the condition of a `if` statement is not fulfilled no statement is executed.

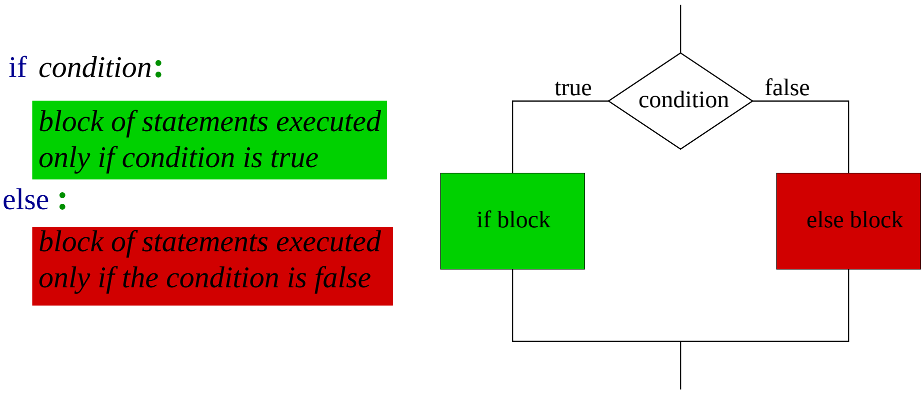
```
>>> seq = ’ATGACGATAG’
>>> if ’n’ in seq:
 print "sequence contains undefined characters"
>>>
```

An *alternative* sequence of statements, that will be executed if the *condition* is not fulfilled, can be specified with the `else` statement.

```
>>> seq = ’ATGACGATAG’
>>> if ’n’ in seq:
...    print "sequence contains undefined bases"
... else:
       print "sequence contains only defined bases"
sequence contains only defined bases
```

Here the `if` and `else` are followed by a *block* containing the statements to execute depending on the truth value of the *condition*. In this case exactly one of them is executed, which is illustrated in Figure 7.4

Figure 7.4. Flow of execution of an alternative condition


if *condition*:
block of statements executed only if condition is true
else :
block of statements executed only if the condition is false
true
condition
false
if block
else block

7.5. Chained conditional execution

In Python, you can specify more than one alternative:

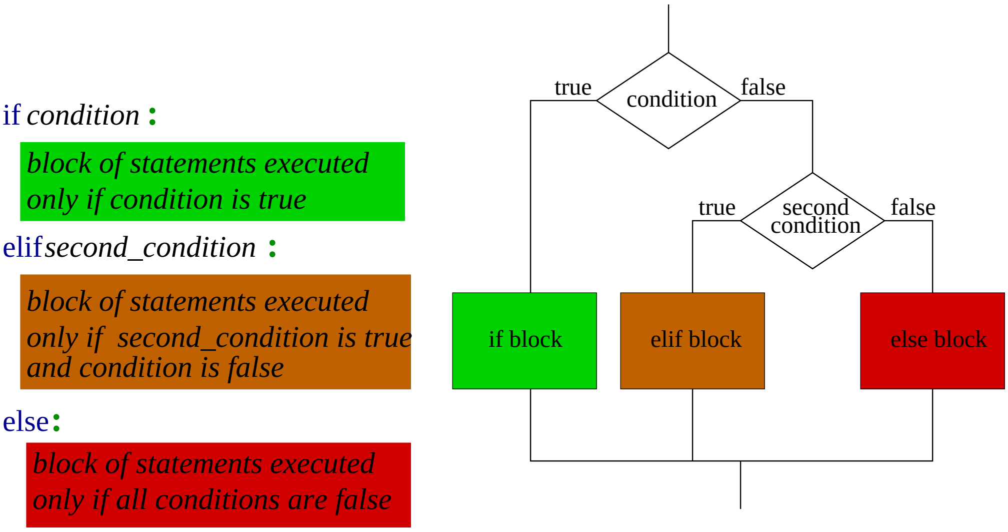
```
>>> seq = ’vATGCAnATG’
>>> base = seq[0]
>>> base
v

>>> if base in ’ATGC’:
...    print "exact nucleotid"
... elif base in ’bdhkmnrsuvwxy’:
...    print "ambiguous nucleotid"
... else:
...    print "not a nucleotid"
...
ambiguous nucleotid
```

The `elif` statement is used to give an alternative condition. What happens when this is executed? The conditions are evaluated from top to bottom. In our example with `base = ’v’` as first condition, the `if` condition `base in ’ATGC’` is false, so the next condition, that of the `elif` statement is evaluated. `base in ’bdhkmnrsuvwxy’` is true, so the block of statements of this clause is executed and `ambiguous nucleotid` is printed. Then the evaluation of the condition is stopped and the flow of execution continues with the statement following the `if-elif-else` construction.

Multiple alternative conditions

Multiple alternative conditions are conditions that are tested from top to bottom. The clause of statements for the first alternative that is evaluated as true is executed. So there is *exactly* one alternative that is executed, even if there are more than one that are true. In this case the clause of the *first* true condition encountered is chosen. Figure 7.5 illustrates this.

Figure 7.5. Multiple alternatives or Chained conditions


if condition :
block of statements executed only if condition is true
elif second_condition :
block of statements executed only if second_condition is true and condition is false
else:
block of statements executed only if all conditions are false
true
condition
false
true
second condition
false
if block
elif block
else block

The `else` statement is optional. But it is more safe to catch the case where all of the given conditions are false.

Exercise 7.1. Chained conditions

The `elif` statement only facilitates the writing and legibility of multiple alternative conditions. How would you write a multiple condition without this statement (Solution 7.1)?

Hint: See the scheme of Figure 7.5.

7.6. Nested conditions

However, construction with multiple alternatives are sometimes not sufficient and you need to nest condition like this:

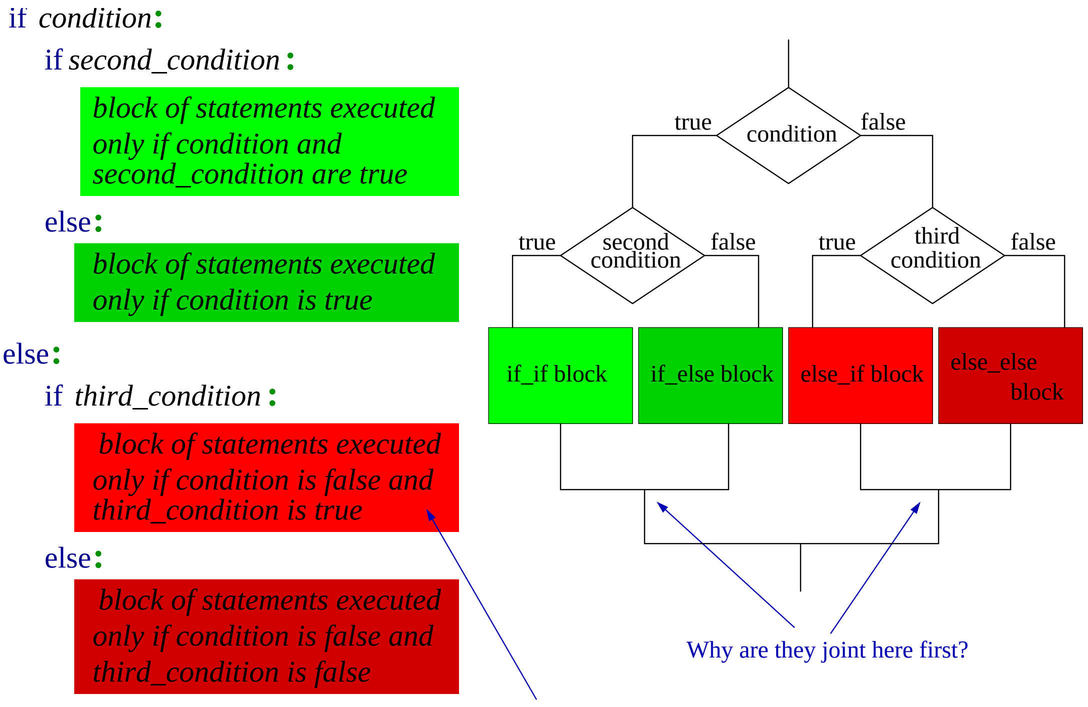
```
>>> primerLen = len(primer)
>>> primerGC  = float(count(primer, ’g’) + count(primer, ’c’))/ primerLen

>>> if primerGC > 50:
...     if primerLen > 20:
...         PCRprogram = 1
...     else:
...         PCRprogram = 2
... else:
...     PCRprogram = 3
```

Exercise 7.2. Nested condition

Why is it impossible to write the above example as chained condition?

Figure 7.6 shows the scheme of *nested conditions*.

Figure 7.6. Nested conditions


if condition:
if second_condition :
block of statements executed only if condition and second_condition are true
else:
block of statements executed only if condition is true
else:
if third_condition :
block of statements executed only if condition is false and third_condition is true
else:
block of statements executed only if condition is false and third_condition is false
true
condition
false
true
second condition
false
true
third condition
false
if_if block
if_else block
else_if block
else_else block

Why are they joint here first?

What is known about the second condition here?

Sometimes you can simplify nested conditions by constructing more complex conditions with *boolean operators*.

```
>>> primerLen = len(primer)
>>> primerGC  = float(count(primer, ’g’) + count(primer, ’c’))/ primerLen

>>> if primerGC > 50:
...     if primerLen > 20:
...         PCRprogram = 1
... else:
...     PCRprogram = 2
```

can be expressed as:

```
>>> if primerGC > 50 and primerLen > 20:
...     PCRprogram = 1
... else:
...     PCRprogram = 2
```

Caution

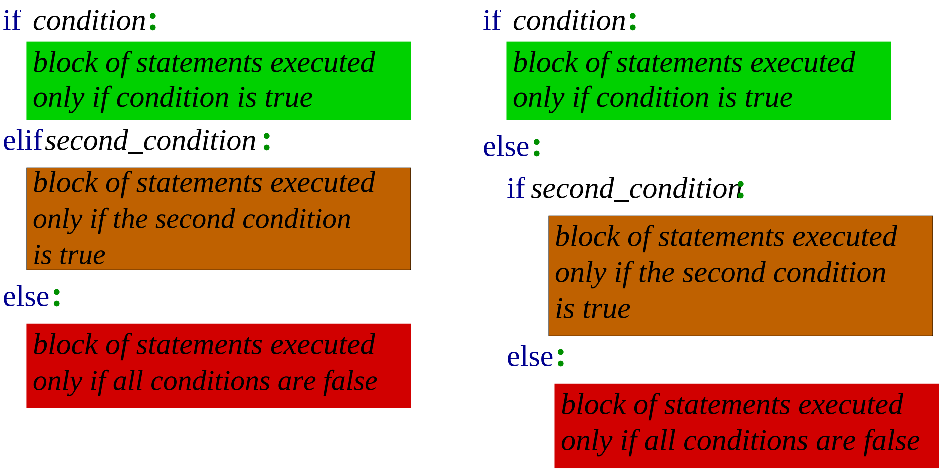
Even if the second version is easier to read, be careful and always check whether the complex condition you have written, is what you really want. Such errors are called semantic error. They can not be detected by the interpreter because the syntax of the program is correct, even if it is not necessarily what you want to compute.

7.7. Solutions

Solution 7.1. Chained conditions

Exercise 7.1

Figure 7.7. Multiple alternatives without elif


if condition:
block of statements executed only if condition is true
elif second_condition :
block of statements executed only if the second condition is true
else:
block of statements executed only if all conditions are false
if condition:
block of statements executed only if condition is true
else:
if second_condition:
block of statements executed only if the second condition is true
else:
block of statements executed only if all conditions are false

Chapter 8. Defining Functions

8.1. Defining Functions

In Section 3.3 we have learnt how to apply or call functions. So let’s remember the example calculating the GC-percentage of a DNA sequence.

```
>>> float(count(cds, ’G’) + count(cds, ’C’)) / len(cds)
```

This calculates the gc percentage of the specific DNA sequence `cds`, but we can use the same formula to calculate the gc percentage of other DNA sequences. The only thing to do is to replace `cds` by the new DNA sequence in the formula. But it is not very convenient to remember the formula and retype it all the time. It would be much easier to type the following instead.

```
>>> gc(’ATGCAT’)
0.33333333333333331
>>> gc(cds)
0.54460093896713613
```

The only thing we have to remember is the name of the new *function* and its use.

Abstraction

The possibility to define such new *function* executing tasks specified by yourself, is an *abstraction* feature, provided by all high level programming languages.

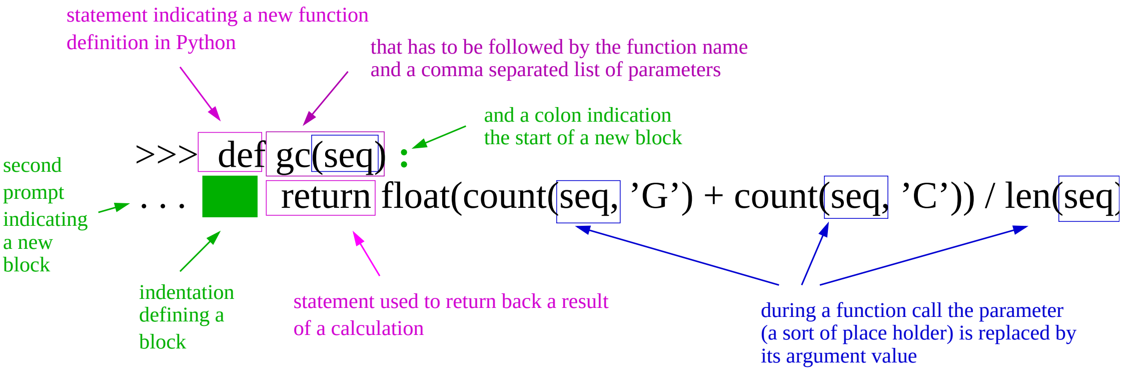
> ### Important
>
> It is also important to notice that functions have to be defined before they are called. You can not use something that is not defined.

Here is the syntax of such a new definition in Python:

```
>>> from string import *

>>> def gc(seq):
...     return float(count(seq, ’G’) + count(seq, ’C’)) / len(seq)
```

Let’s have a closer look at this definition. Figure 8.1 illustrates the structure of a function definition.

Figure 8.1. Function definitions


statement indicating a new function definition in Python
that has to be followed by the function name and a comma separated list of parameters
and a colon indication the start of a new block
second prompt indicating a new block
>>> def gc(seq) :
. . . return float(count(seq, 'G') + count(seq, 'C')) / len(seq)
indentation defining a block
statement used to return back a result of a calculation
during a function call the parameter (a sort of place holder) is replaced by its argument value

`def` and `return` are *basic instructions*.

Basic instruction

Basic instructions are statements that define the language rules and the semantic of Python. They constitute the basic set of instructions of Python. Each *basic instruction* has its own syntax that you have to learn in order to master the programming language.

The `return` basic instruction is used to return the result of a function back, in our example the value of the GC percentage of the specified DNA sequence.

The `def` basic instruction indicates to Python that a function definition follows. It has to be followed by the new function name and a comma separated list of parameter names enclosed into parentheses.

Parameter

Parameters are *variable names*. When the function is called they are bound in the same order to the arguments given.

The body of a function contains the piece of code needed to execute the subtask of the function. In the example above, the body contains only the `return` statement. Here is a more complex example that excludes ambiguous bases from the GC percentage calculation.

```
>>> from string import *

>>> def gc(seq):
...     nbases = count(seq, 'N')
...     gcpercent = float(count(seq, 'G') + count(seq, 'C')) / (len(seq) - nbases)
...     return gcpercent
```

In this example the body of the function contains three instructions (2 assignments and the return statement). The body of a function follows the definition line and is written as an indented *block* initiated by a colon.

Block

Blocks are *structure elements* of a program, that are used to group instructions. They are initiated by a *colon* and are separated from the rest of the code by the same *indentation* step in Python.

In our example, the function body is written as a *block* to separate it from the rest of the program. The Python interpreter used it to detect where the function body starts and ends.

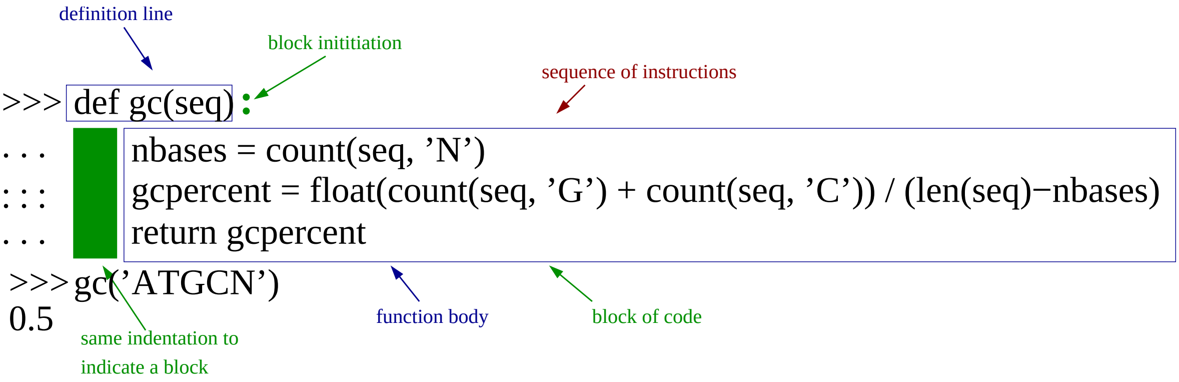
Important

In other programming languages special words or characters are used to indicate the begin and the end of blocks and the indentation is only used to make the code readable for humans. In Python the indentation is used explicitly to do this. So in the above example of a function definition the end of the body is detected when the next line starts without any indentation step. This is also illustrated in Figure 8.2.

Example 8.1. More complex function definition

```
>>> from string import *

>>> def gc(seq):
...    nbases = count(seq, 'N')
...    gcpercent = float(count(seq, 'G') + count(seq, 'C')) /  \
                   (len(seq) - nbases)
...    return gcpercent
...
>>> gc('ATGCN')
0.5
```

Figure 8.2. Blocks and indentation



```
>>> def gc(seq):
...    nbases = count(seq, 'N')
: : :  gcpercent = float(count(seq, 'G') + count(seq, 'C')) / (len(seq)−nbases)
...    return gcpercent
>>> gc('ATGCN')
0.5
```

8.2. Parameters and Arguments or the difference between a function definition and a function call

As we have already said, functions are named sequences of statements that execute a specific task. A *function definition* is the construction that gives a name to this special sequence of statements. Whereas a *function call* is the execution of this sequence of statements. We have also seen that the execution can be parameterized. This means, that there are special variables, the *parameters*, in the function which are bound to values, the *arguments*, during the function call.

Flow of execution. Some precision about the *flow of execution*. If you call a function the statements of the function body are executed. The execution of the current sequence of statements areis interrupted and the *flow of execution* jumps to the sequence of statements named by the function. The function statements are executed and then the *flow of execution* continues with the sequence from where the function was called.

Caution

The statements in the function body are only executed at a function call, so all errors in this piece of code are only reported at the execution.

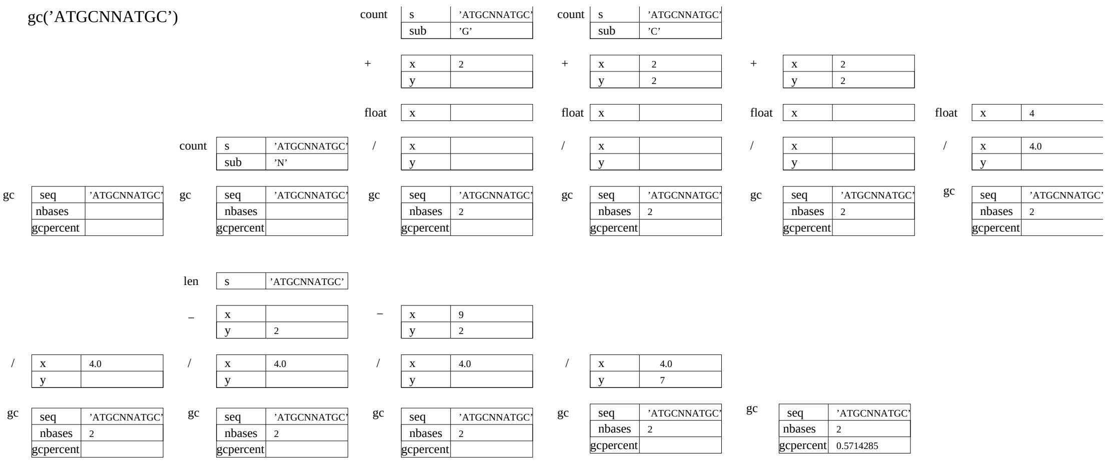
Now if we remember that within a function you can call other functions, the above scheme becomes rapidly disturbing, but Python keeps a track from where a function is called and there is only one statement at a time that is executed. You can visualize this using a *stack diagram* of the function calls.

Stack

A *stack* is an ordered set of things which can be accessed in two ways. You can put things on the top of the stack, and only take back the thing on the top of the stack. A common example is a stack of plates (it is hard to get a plate from within the stack). A stack is sometimes called a *LIFO* an abbreviation of *last-in-first-out*.

So what Python does is to handle a stack of function calls. If a function is called, Python puts the position from where it is called on the stack, executes the function and when the function execution is finished Python gets the position to continue back from the stack. Figure 8.3 shows the stack diagram of the execution of the `gc` function (Example 8.1).

Figure 8.3. Stack diagram of function calls


gc('ATGCNNATGC')
count s 'ATGCNNATGC' sub 'G'
count s 'ATGCNNATGC' sub 'C'
+ x 2 y
+ x 2 y 2
+ x 2 y 2
float x
float x
float x
float x 4
count s 'ATGCNNATGC' sub 'N'
/ x y
/ x y
/ x y
/ x 4.0 y
gc seq 'ATGCNNATGC' nbases gcpercent
gc seq 'ATGCNNATGC' nbases gcpercent
gc seq 'ATGCNNATGC' nbases 2 gcpercent
gc seq 'ATGCNNATGC' nbases 2 gcpercent
gc seq 'ATGCNNATGC' nbases 2 gcpercent
gc seq 'ATGCNNATGC' nbases 2 gcpercent
len s 'ATGCNNATGC'
– x y 2
– x 9 y 2
/ x 4.0 y
/ x 4.0 y
/ x 4.0 y
/ x 4.0 y 7
gc seq 'ATGCNNATGC' nbases 2 gcpercent
gc seq 'ATGCNNATGC' nbases 2 gcpercent
gc seq 'ATGCNNATGC' nbases 2 gcpercent
gc seq 'ATGCNNATGC' nbases 2 gcpercent
gc seq 'ATGCNNATGC' nbases 2 gcpercent 0.5714285

8.3. Functions and namespaces

Even if it is confusing at the beginning, the `dna` variable in the function call is not the same as the `dna` parameter of the `gc` function. During the function call the global variable is evaluated and its value `'ATGACT'` is assigned to the parameter `dna` of the `gc` function.

How does Python distinguish the two variables? It can do so, because they are not in the same namespace. All functions have two associated namespaces, a *local* and a *global* namespace. The *global* namespace is the namespace where the function is defined and the *local* namespace is a namespace created at the function definition where parameters are defined. The local namespace contains also all *local variables*. *Local variables* are variables that are bound to value in the function body. When a function is defined, its source code is analyzed and all used names are sorted in the two namespaces.

If a variable name is searched during a function execution, Python looks in the local namespace for local variables and in the global namespace for global variables. If a global variable is not found there, the builtin namespace is searched. If it is not found there, an error is produced.

builtin namespace

The *builtin namespace* is a *namespace* that contains all predefined function of Python.

You can ask Python which names are defined in the global and the local namespace using the functions `globals()` and `locals()`.

```
 >>> from string import *
>>> def gc(seq):
...     nbases = count(seq, 'N')
...     gcpercent = float(count(seq, 'G') + count(seq, 'C')) / (len(seq) - nbases)
...     print "local namespace:", locals()
...     print "global namespace:", globals()
...     print "names in the current namespace:", dir()
...     return gcpercent
...
>>> seq = ’ATGACGATAGGAGANNTATAGAN’

>>> gc(seq)
local namespace: {’gcpercent’: 0.34999999999999998, ’nbases’: 3,
’seq’: ’ATGACGATAGGAGANNTATAGAN’}
global namespace: {’ascii_lowercase’: ’abcdefghijklmnopqrstuvwxyz’,
 ’upper’: <function upper at 0x1623f0>, ’punctuation’: ’"!#$%&\’()*+,-./:;<=>?@[\\]^_‘{|}~’,

’letters’: ’abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ’,
’seq’: ’ATGACGATAGGAGANNTATAGAN’, ’lstrip’: <function lstrip at 0x160cf0>,
’uppercase’: ’ABCDEFGHIJKLMNOPQRSTUVWXYZ’,
’ascii_letters’: ’abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ’,
’replace’: <function replace at 0x160b90>, ’capwords’: <function capwords at 0x160b10>,

’index_error’: <class exceptions.ValueError at 0x119700>,
’expandtabs’: <function expandtabs at 0x160a50>, ’strip’: <function strip at 0x160cb0>,

’ascii_uppercase’: ’ABCDEFGHIJKLMNOPQRSTUVWXYZ’, ’find’: <function find at 0x160810>,

’gc’: <function gc at 0x160b50>, ’rjust’: <function rjust at  0x160990>,
’ljust’: <function ljust at 0x160950>, ’whitespace’: ’\t\n\x0b\x0c\r ’,
’rindex’: <function rindex at 0x1625a0>, ’capitalize’: <function capitalize at 0x160ad0>,

’atol_error’: <class exceptions.ValueError at 0x119700>, ’octdigits’: ’01234567’,
’lower’: <function lower at 0x1623b0>, ’splitfields’: <function split at 0x160d70>,
’split’: <function split at 0x160d70>, ’rstrip’: <function rstrip at 0x160d30>,
’translate’: <function translate at 0x160a90>,
’__doc__’: None,
’printable’: ’0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&\’()*+,-./:;<=>?@[\

’digits’: ’0123456789’, ’joinfields’: <function join at 0x162520>,
’index’: <function index at 0x162560>,
’__builtins__’: <module ’__builtin__’ (built-in)>,
’swapcase’: <function swapcase at 0x162430>,
’atof_error’: <class exceptions.ValueError at 0x119700>,
’atoi’: <function atoi at 0x1608d0>, ’hexdigits’: ’0123456789abcdefABCDEF’,
’atol’: <function atol at 0x160910>, ’__name__’: ’__main__’,
’atof’: <function atof at 0x160890>, ’count’: <function count at 0x1625e0>,
’lowercase’: ’abcdefghijklmnopqrstuvwxyz’, ’join’: <function join at 0x162520>,
’center’: <function center at 0x1609d0>, ’rfind’: <function rfind at 0x160850>,
’atoi_error’: <class exceptions.ValueError at 0x119700>,
’maketrans’: <built-in function maketrans>, ’zfill’: <function zfill at 0x160a10>}
names in the current namespace: [’gcpercent’, ’nbases’, ’seq’]
0.34999999999999998
```

The dir function shows all the names defined in the current namespace.

```
>>> from string import *

>>> dir()
[’__builtins__’, ’__doc__’, ’__name__’, ’ascii_letters’, ’ascii_lowercase’,
’ascii_uppercase’, ’atof’, ’atof_error’, ’atoi’, ’atoi_error’, ’atol’,
’atol_error’, ’capitalize’, ’capwords’, ’center’, ’count’, ’digits’,
’expandtabs’, ’find’, ’gc’, ’hexdigits’, ’index’, ’index_error’, ’join’,
’joinfields’, ’letters’, ’ljust’, ’lower’, ’lowercase’, ’lstrip’, ’maketrans’,
’octdigits’, ’printable’, ’punctuation’, ’replace’, ’rfind’, ’rindex’, ’rjust’,
’rstrip’, ’seq’, ’split’, ’splitfields’, ’strip’, ’swapcase’, ’translate’,
’upper’, ’uppercase’, ’whitespace’, ’zfill’]
```

8.4. Boolean functions

Whenever a condition is too complex, you can write a *boolean function*.

Boolean function

Boolean functions are *function* that return truth values (in Python either 0 or 1).

It is good style to give them a name that indicates its nature, often the function name starts with is.

Here is an example that tests if a character is a valid amino acid:

```
>>> from string import *

>>> def isAminoAcid(aa):
...     AA = upper(aa)
...     if AA < ’A’ or AA > ’Z’:
...       return 0
...     if AA in ’JOU’:
...       return 0
...     return 1
```

you can also write it as follow:

```
>>> from string import *

>>> def isAminoAcid(aa):
...     AA = upper(aa)
...     if AA < ’A’:
...        ok =  0
...     elif AA > ’Z’:
...        ok = 0
...     elif AA in ’JOU’:
...        ok = 0
...     else:
...        ok = 1
...     return ok
```

or using boolean operators:

Example 8.2. Function to check whether a character is a valid amino acid

```
>>> def isAminoAcid(aa):
...     return (’A’ <= aa <= ’Z’ or ’a’ <= aa <= ’z’) and aa not in ’jouJOU’
```

Using this function makes your code easier to understand because the function name expresses what you want to test:

```
>>> prot = ’ATGAFDWDWDAWDAQW’
>>> oneaa = prot[0]

>>> if isAminoAcid(oneaa):
...    print ’ok’
... else:
       print ’not a valid amino acid’
ok
```

Chapter 9. Collections

9.1. Datatypes for collections

In the proceeding chapter we have seen that *strings* can be interpreted as *collections of characters*. But they are a very restricted sort of collection, you can only put characters in a string and a string is always ordered. But we need often to handle collections of all sorts of objects and sometimes these collections are not even homogeneous, meaning that they may contain objects of different types. Python provides several predefined **data types** that can manage such collections. The two most used structures are called **Lists** and **Dictionaries**. Both can handle collections of different sorts of objects, but what are their differences?

List

Lists are *mutable ordered collections* of objects of different sorts. The objects are accessible using their position in the *ordered collection*.

Dictionary

Dictionaries are *mutable unordered collections* which may contain objects of different sorts. The objects can be accessed using a *key*.

Here are some examples comparing a list version of a collection of enzyme's pattern and a dictionary version of the same collection. Lists are created using a comma separated list of all elements enclosed into brackets, whereas dictionaries are enclosed into braces and contain a comma separated list of key-value pairs, each separated by a colon.

```
>>> patternList = [ 'gaattc', 'ggatcc', 'aagctt' ]
>>> patternList
['gaattc', 'ggatcc', 'aagctt']

>>> patternDict = { 'EcoRI' : 'gaattc', 'BamHI' : 'ggatcc', 'HindIII' : 'aagctt' }
>>> patternDict
{ 'EcoRI' : 'gaattc', 'BamHI' : 'ggatcc', 'HindIII' : 'aagctt' }
```

To access the elements we use the position in the list and the key for the dictionary.

Important

List indices have the same properties as string indices, in particular they start with 0 (remember Figure 6.1).

```
>>> patternList[0]
'gaattc'
>>> patternDict['EcoRI']
```

```
'gaattc'
```

As for strings you can get the number of elements in the collection, as well as the smallest and the greatest element.

```
>>> len(patternList)
3
>>> len(patternDict)
3
```

Lists can be sliced but dictionaries cannot. Remember that dictionaries are unordered collections, so getting a slice does not make any sense.

```
>>> digest = patternList[:1]
>>> digest
['gaattc']
```

You can ask whether an element is in the collection. Dictionaries have two possibilities to perform this.

```
>>> 'gaattc' in patternList
1
>>> patternDict.has_key('HindIII')
1
>>> 'HindIII' in patternDict
1
```

Unlike strings both collections are mutable. This means that you can remove, add or even change their elements.

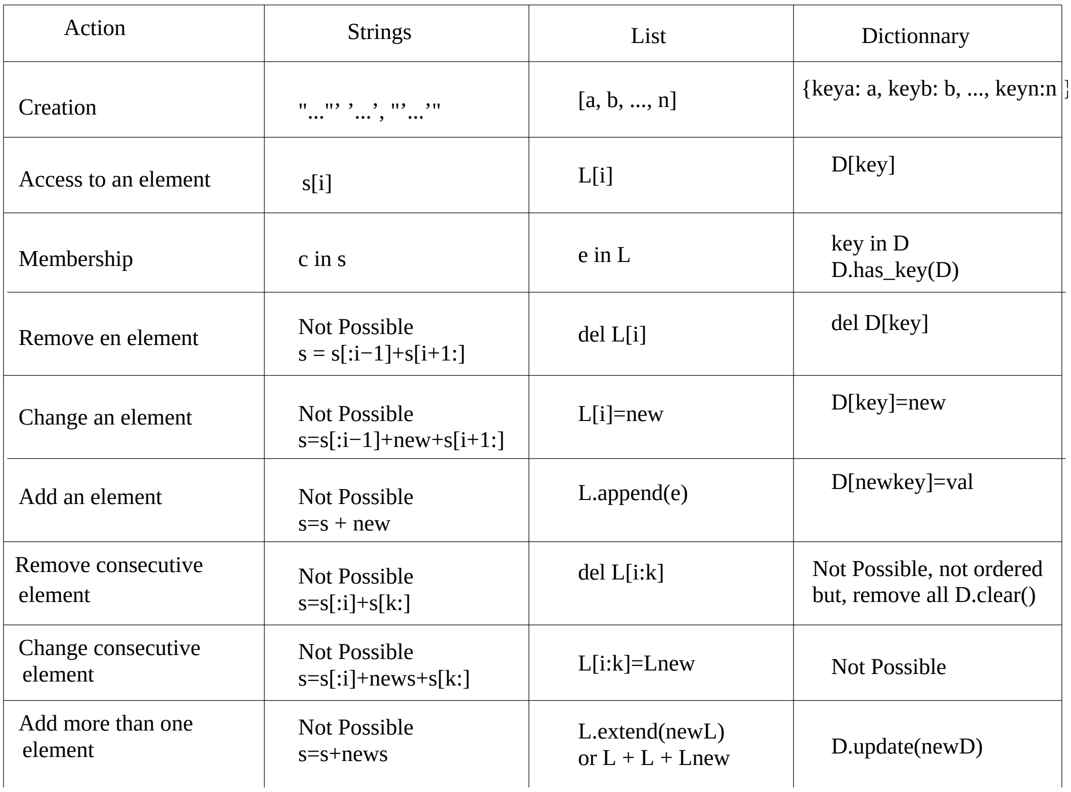
```
>>> del patternList[0]
>>> patternList
['ggatcc', 'aagctt']
>>> patternList[0] = 'gaattc'
>>> patternList
['gaattc', 'aagctt']
>>> patternList.append('ggattc')
>>> patternList
['gaattc', 'aagctt', 'ggattc']
>>> del patternList[:2]
>>> patternList
['ggattc']

>>> del patternDict['EcoRI']
>>> patternDict
{'BamHI': 'ggatcc', 'HindIII': 'aagctt'}
>>> patternDict['EcoRI'] = 'gaattc'
>>> patternDict
{'BamHI': 'ggatcc', 'EcoRI': 'gaattc', 'HindIII': 'aagctt'}
>>> patternDict['BamHI'] =''
>>> patternDict
{’BamHI’: ”, ’EcoRI’: ’gaattc’, ’HindIII’: ’aagctt’}
```

As for strings you cannot access elements that do not exist.

```
>>> patternList[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
>>> patternDict[’ScaI’]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyError: ScaI
```

Figure 9.1 compares different actions on collections for strings, lists and dictionaries.

Figure 9.1. Comparison some collection datatypes



Action	Strings	List	Dictionnary
Creation	"..."’ ’...’, "’...’"	[a, b, ..., n]	{keya: a, keyb: b, ..., keyn:n }
Access to an element	s[i]	L[i]	D[key]
Membership	c in s	e in L	key in D D.has_key(D)
Remove en element	Not Possible s = s[:i−1]+s[i+1:]	del L[i]	del D[key]
Change an element	Not Possible s=s[:i−1]+new+s[i+1:]	L[i]=new	D[key]=new
Add an element	Not Possible s=s + new	L.append(e)	D[newkey]=val
Remove consecutive element	Not Possible s=s[:i]+s[k:]	del L[i:k]	Not Possible, not ordered but, remove all D.clear()
Change consecutive element	Not Possible s=s[:i]+news+s[k:]	L[i:k]=Lnew	Not Possible
Add more than one element	Not Possible s=s+news	L.extend(newL) or L + L + Lnew	D.update(newD)

9.2. Methods, Operators and Functions on Lists

Table 9.1 remembers the action of builtin functions and operators on list objects and Table 9.2 summarizes all methods of list objects.

Table 9.1. Sequence types: Operators and Functions

Operator/Function	Action	Action on Numbers
`[ ... ], ( ... ), " ... "`	creation	
`s + t`	concatenation	addition
`s * n`	repetition [a]	multiplication
`s[i]`	indication	
`s[i:k]`	slice	
`x in s`	membership	
`x not in s`		
`for a in s`	iteration	
`len(s)`	length	
`min(s)`	return smallest element	
`max(s)`	return greatest element	
`s[i] = x`	index assignment	
`s[i:k] = t`	slice assignment	
`del s[i]`	deletion	

[a]

[a] **Important**
shallow copy (see Section 11.2)

Table 9.2. List methods

Method	Operation
`list(s)`	converts any sequence object to a list
`s.append(x)`	append a new element
`s.extend(t)`	concatenation[a]
`s.count(x)`	count occurrences of `x`
`s.index(x)`	find smallest position where `x` occurs in s
`s.insert(i,x)`	insert `x` at position `i`
`s.pop([i])`	removes i-th element and return it
`s.remove(x)`	remove element
`s.reverse()`[b]	reverse
`s.sort([cmp])`[b]	sort according to the `cmp` function

[a] equal to the + operator but inplace
[b] in place operation

Important

It is important to know whether a function or method, that is applied to a *mutable* objects, modifies this object internally or whether it returns a new object containing these modifications. Look at the following example that shows two possibilities to concatenate lists. The + operator creates a new list whereas the method `extend` adds one list to the other:

```
>>> l1 = [ ’EcoRI’, ’BamHI’ ]
>>> l2 = [ ’HindIII’ ]
>>> l1
[’EcoRI’, ’BamHI’]
>>> l2
[’HindIII’]
>>> l1 + l2
[’EcoRI’, ’BamHI’, ’HindIII’]
>>> l1
[’EcoRI’, ’BamHI’]
>>> l2
[’HindIII’]
>>> l1.extend(l2)
>>> l1
[’EcoRI’, ’BamHI’, ’HindIII’]
>>> l2
[’HindIII’]
```

9.3. Methods, Operators and Functions on Dictionaries

Contrary to strings and lists the ordering of the elements in dictionaries does not matter. Elements are accessed by a unique key rather than by an index number.

Important

It is important to notice that *dictionary keys* have to be *unique* and *immutable*.

Table 9.3 gives an overview of the methods and operations on dictionaries.

Table 9.3. Dictionary methods and operations

Method or Operation	Action
`d[key]`	get the value of the entry with key `key` in `d`
`d[key] = val`	set the value of entry with key `key` to `val`
`del d[key]`	delete entry with key `key`
`d.clear()`	removes all entries
`len(d)`	number of items

`d.copy()`	makes a shallow copy[a]
`d.has_key(key)`	returns 1 if `key` exists, 0 otherwise
`d.keys()`	gives a list of all keys
`d.values()`	gives a list of all values
`d.items()`	returns a list of all items as tuples (key,value)
`d.update(new)`	adds all entries of dictionary `new` to `d`
`d.get(key [, otherwise])`	returns value of the entry with key `key` if it exists otherwise returns `otherwise`
`d.setdefaults(key [, val])`	same as `d.get(key)`, but if `key` does not exists sets `d[key]` to `val`
`d.popitem()`	removes a random item and returns it as tuple

[a]

[a] ❗ **Important**

shallow copy (see Section 11.2)

9.4. What data type for which collection

We have seen so far three *collection types* in Python. Which one should you chose in your applications? Because they all have their advantages and disadvantages making some actions easy or difficult to handle, the choice depends on your data and on what you would do with these data. To recognize *strings* is the easiest one, but be aware if you need to change characters, since you will have to construct a new string. So a mutable ordered collection datatype, such as a list, could be more adapted.

For list and dictionaries there are two important issues to consider:

1. Does the ordering of your collection matter?
2. Is it sufficient to access the elements by their position or do you prefer to access the collection with a more complex, sometimes more descriptive, key rather than a number?

If you need an ordered collection you do not have any choice left, because only lists take care of the ordering. However the way of accessing a collection is sometimes not so easy and often worth further consideration.

Chapter 10. Repetitions

10.1. Repetitions

Sometimes you need to do the same thing several times with different data. Whereas humans make often errors during a repetitive procedure, computers execute repetitive tasks very well.

One possibility to write repetitions is to repeat the code as often as needed, but this is neither elegant nor readable, and above all you cannot do this if the number of repetition times is not constant or is not known beforehand.

Just as now, we have seen all that we need to do repetitive tasks.

Exercise 10.1. Repetitions

Try to find out how to perform a repetitive task, only with the statements seen so far. It is not very easy if you have not seen this and we will discuss this later (Chapter 13).

Because repetitions are common programming tasks, high level programming languages provide special structures and statements for this purpose. Python has two of them: `for` and `while`. Let us see the `for` statement first.

10.2. The for loop

Verify protein sequences. Let us start with an example that checks whether all characters of a protein sequence are valid amino acid characters. How could we do this by hand? One possibility is to start at the beginning of the protein sequence and check each character one by one. Here is a more formal description of the algorithm:

```
1. for each character of the protein sequence
    a. if the current character is not an amino acid
        • print the invalid character
    b. end if
2. end for
```

In Python this can be performed with the `for` statement.

The for loop

The `for` loop enables to iterate on an ordered collection of objects and to execute the same sequence of statements for each element.

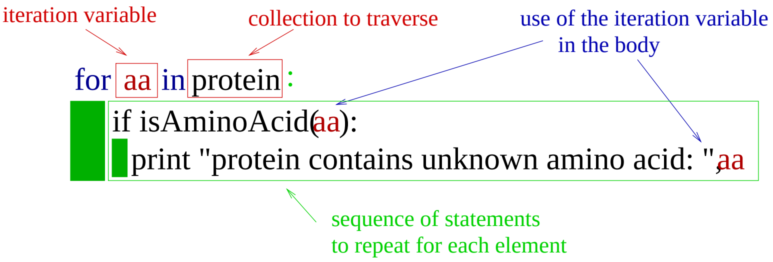
This *sort of iteration* is also known as a *traversal*, because the collection is *traversed* from left to right and a particular task is applied to each element.

So in our example, the collection of elements is a string which we traverse with the `for` loop. We also have to specify what a valid amino acid is, because computers do not know about it. Nevertheless we have already written a function doing this (Example 8.2).

```
>>> protein = "SERLSITPLGPYIGAQIJSGADLTRPLSDNQFEQLYHAVLRHQVVFLRDQAITPQQQRALA"

>>> for aa in protein:
...    if not isAminoAcid(aa):
...        print "protein contains unknown amino acid: ", aa
...
protein contains unknown amino acid:  J
```

As `def` and `if`, the `for` statement is also a statement containing a *block*. The *header line* gives the two essential information for the traversal. It starts with `for` followed by a *variable name* to which each element of the collection is assigned during the execution of the *block* of statements. The second half of the *header* indicates the collection `in` which the elements can be found. As in all functions, operations and expressions, the collection can be specified by a *value*, a *variable* or even a *composed expression*.

Figure 10.1. The for loop



```
for aa in protein:
    if isAminoAcid(aa):
        print "protein contains unknown amino acid: ",aa
```

Getting the position of invalid amino acids. As written above we cannot show the position of the invalid amino acid in the protein sequence, because only the amino acid character is known in the body of the `for` loop. Therefore, we have to iterate over the indices of the sequence. Here is the algorithm:

```
1. for each position in the protein sequence
    a. if the character at this position is not an amino acid
        • print the position and the invalid character
```

   b. **end if**

2. **end for**

What are the possible values of the positions? Remember the string indices, they start at 0 and end at one before the length of the string. Python has a function that gives the possibility to create a list of numbers that is called `range`. Using this function, we can translate the algorithms above into Python:

```
>>> for i in range(len(protein)):
...    if not isAminoAcid(protein[i]):
...        print "protein contains unknown amino acid " , protein[i] , \
...              " at position ", i
```

Note

There is a convention in most programming languages to use `i, k, l` as variable names for the iteration variable.

Translate cds sequences. Let’s take another example. Say that we have a cds sequence and that we would like to know such things as the corresponding amino acid sequence or the codon usage. How could we achieve this? For sure, it looks like a traversal of the cds sequence, but the things we are looking for do not correspond anymore to one nucleotide of the cds sequence but to a codon which is a three letter substring. If we could access the codons one by one, the translation into the amino acid sequence would look like this:

- for each codon in the cds sequence:
    - add the amino acid corresponding to the current codon to the protein sequence

In the body of the loop we need to establish the correspondence between a codon sequence and the amino acid. We know how to do this by hand, by looking up a codon in the codon translation table. A dictionary is the data type that gives us such sort of access to the collection of codons.

```
>>> code = {     ’ttt’: ’F’, ’tct’: ’S’, ’tat’: ’Y’, ’tgt’: ’C’,
...              ’ttc’: ’F’, ’tcc’: ’S’, ’tac’: ’Y’, ’tgc’: ’C’,
...              ’tta’: ’L’, ’tca’: ’S’, ’taa’: ’*’, ’tga’: ’*’,
...              ’ttg’: ’L’, ’tcg’: ’S’, ’tag’: ’*’, ’tgg’: ’W’,
...              ’ctt’: ’L’, ’cct’: ’P’, ’cat’: ’H’, ’cgt’: ’R’,
...              ’ctc’: ’L’, ’ccc’: ’P’, ’cac’: ’H’, ’cgc’: ’R’,
...              ’cta’: ’L’, ’cca’: ’P’, ’caa’: ’Q’, ’cga’: ’R’,
...              ’ctg’: ’L’, ’ccg’: ’P’, ’cag’: ’Q’, ’cgg’: ’R’,
...              ’att’: ’I’, ’act’: ’T’, ’aat’: ’N’, ’agt’: ’S’,
...              ’atc’: ’I’, ’acc’: ’T’, ’aac’: ’N’, ’agc’: ’S’,
...              ’ata’: ’I’, ’aca’: ’T’, ’aaa’: ’K’, ’aga’: ’R’,
...              ’atg’: ’M’, ’acg’: ’T’, ’aag’: ’K’, ’agg’: ’R’,
...              ’gtt’: ’V’, ’gct’: ’A’, ’gat’: ’D’, ’ggt’: ’G’,
...              ’gtc’: ’V’, ’gcc’: ’A’, ’gac’: ’D’, ’ggc’: ’G’,
...              ’gta’: ’V’, ’gca’: ’A’, ’gaa’: ’E’, ’gga’: ’G’,
...              ’gtg’: ’V’, ’gcg’: ’A’, ’gag’: ’E’, ’ggg’: ’G’
...         }
>>> code[’atg’]
’M’
```

Now let us go back to the first part of the problem: getting the codons. If we know where a codon starts, we only have to extract the substring of length 3 starting from this position. Here is the algorithm:

- for each third position in the cds sequence:
    - get the substring of length three starting from this position

Fortunately, the `range` function can take an optional step argument.

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(4,10)
[4, 5, 6, 7, 8, 9]
>>> range(0,10,3)
[0, 3, 6, 9]
```

We can now print the codon list:

```
>>> cds = "atgagtgaacgtctgagcattaccccgctggggccgtatatcggcgcacaataa"

>>> for i in range(0,len(cds),3):
...     print cds[i:i+3],
...
atg
agt
gaa
cgt
ctg
agc
att
acc
ccg
ctg
ggg
ccg
tat
atc
ggc
gca
```

```
caa
taa
taa
```

If we replace the `print` statement with the access to the dictionary of translation, we can translate the codon into the amino acid.

Example 10.1. Translate a cds sequence into its corresponding protein sequence

```
>>> def translate(cds, code):
...     prot = ""
...     for i in range(0,len(cds),3):
...         codon = cds[i:i+3]
...         prot = prot + code[codon]
...     return prot
>>> translate(cds, code)
'MSERLSITPLGPYIGAQ*'
```

What about the computing of the codon usage? We do not need the translation table anymore. But we have to count each codon now. We also need a data structure accessible by codons, although it is not to get information, but to store the result. Here is the algorithm: do not forget that accessing a key which does not exist in a dictionary, is not allowed.

- for each codon in the coding sequence:
    a. if the codon is already in the dictionary of the usage:
        - then add 1 to the count
    b. otherwise:
        - put the codon in the dictionary with count = 1

Here is the corresponding Python code:

```
>>> def count_codons(cds):
...     usage = {}
...     for i in range(0,len(cds),3):
...        codon = cds[i:i+3]
...        if usage.has_key(codon):
...           usage[codon] += 1
...        else:
...           usage[codon] = 1
...     return usage
...
>>> count_codons(cds)
{’acc’: 1, ’atg’: 1, ’atc’: 1, ’gca’: 1, ’agc’: 1, ’ggg’: 1, ’att’: 1, ’ctg’: 2,
 ’taa’: 1, ’ggc’: 1, ’tat’: 1, ’ccg’: 2, ’agt’: 1, ’caa’: 1, ’cgt’: 1, ’gaa’: 1}
```

Exercise 10.2. Write the complete codon usage function

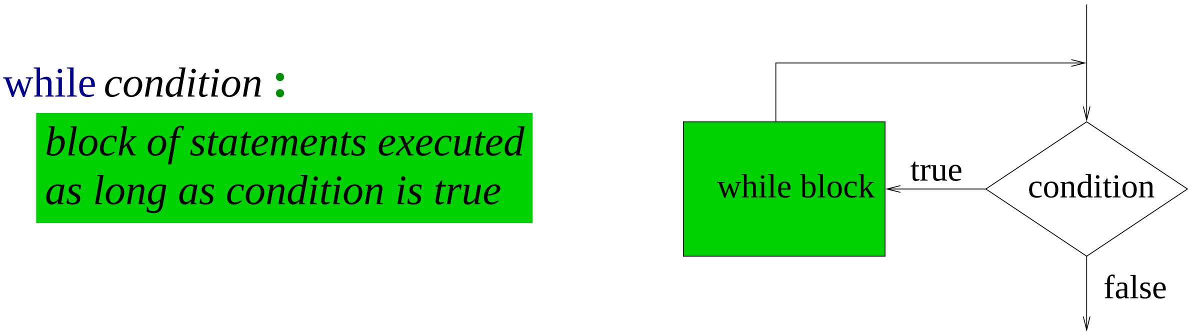
Transform the `count_codons` function into a function getting the real codon usage. Hint: You need to divide each count by the total number of codons.

10.3. The while loop

The while loop

A `while` loop is composed of two parts, a *condition* and a *sequence of statements* to repeat. During the execution, the *sequence of statements* are repeated as long as the *condition* is true. Figure 10.2 illustrates the flow of execution.

Figure 10.2. Flow of execution of a while statement



Before saying more, let us illustrate that with an example. We have already shown how to use the `find` method of strings to find pattern (Section 1.1). Let’s try now to find all occurrences of a specific pattern in a DNA sequence. Here is one possible way to proceed:

Procedure 10.6. Find all occurrences of a pattern in a sequence

INPUT: a DNA sequence *seq* and a pattern *pat*

OUTPUT: a list of positions *matches* containing all start positions of *pat* in *seq*

```
1. matches <- empty list
2. current_match <- position of the first occurrence of pattern pat in sequence seq or -1 if there is not anyone
3. as long as current_match is not -1:
    a. matches <- matches + current_matches
    b. current_match <- position of the next occurrence of pattern pat in sequence seq after current_match or -1 if there is not anyone
4. return matches
```

Example 10.2. First example of a while loop

And here is the implementation in **Python**:

```
def findpos(seq, pat):
   matches = []
   current_match = seq.find(pat)
   while current_match != -1:
       matches.append(current_match)
       current_match =seq.find(pat, current_match+1)
   return matches
```

Let’s follow with a second example that rewrites the `translate` function (Example 10.1) using a `while` loop instead of the `for` traversal.

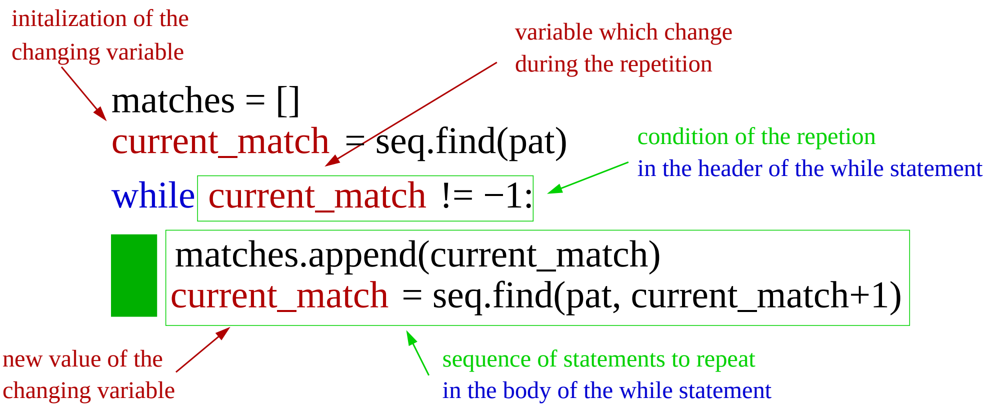
Example 10.3. Translation of a cds sequence using the while statement

This version follows the same procedure as the implementation with the `for` statement.

```
code = {’ttt’: ’F’, ’tct’: ’S’, ’tat’: ’Y’, ’tgt’: ’C’,
        ’ttc’: ’F’, ’tcc’: ’S’, ’tac’: ’Y’, ’tgc’: ’C’,
        ’tta’: ’L’, ’tca’: ’S’, ’taa’: ’*’, ’tga’: ’*’,
        ’ttg’: ’L’, ’tcg’: ’S’, ’tag’: ’*’, ’tgg’: ’W’,
        ’ctt’: ’L’, ’cct’: ’P’, ’cat’: ’H’, ’cgt’: ’R’,
        ’ctc’: ’L’, ’ccc’: ’P’, ’cac’: ’H’, ’cgc’: ’R’,
        ’cta’: ’L’, ’cca’: ’P’, ’caa’: ’Q’, ’cga’: ’R’,
        ’ctg’: ’L’, ’ccg’: ’P’, ’cag’: ’Q’, ’cgg’: ’R’,
        ’att’: ’I’, ’act’: ’T’, ’aat’: ’N’, ’agt’: ’S’,
        ’atc’: ’I’, ’acc’: ’T’, ’aac’: ’N’, ’agc’: ’S’,
        ’ata’: ’I’, ’aca’: ’T’, ’aaa’: ’K’, ’aga’: ’R’,
        ’atg’: ’M’, ’acg’: ’T’, ’aag’: ’K’, ’agg’: ’R’,
        ’gtt’: ’V’, ’gct’: ’A’, ’gat’: ’D’, ’ggt’: ’G’,
        ’gtc’: ’V’, ’gcc’: ’A’, ’gac’: ’D’, ’ggc’: ’G’,
        ’gta’: ’V’, ’gca’: ’A’, ’gaa’: ’E’, ’gga’: ’G’,
        ’gtg’: ’V’, ’gcg’: ’A’, ’gag’: ’E’, ’ggg’: ’G’
       }

def translate_while(cds,code):
    prot = ""
    last_i = len(cds)-3
    i = 0

    while i < last_i:
        codon = cds[i:i+3]
        prot = prot + code[codon]
        i += 3

    return prot
```

In **Python** a `while` loop is written using the `while` statement which is as the `for` and the `if` statements a *block statement*. The header of the *block statement* contains the conditional expression and the body contains the sequence of statements to repeat (Figure 10.3).

Figure 10.3. Structure of the while statement



```
matches = []
current_match = seq.find(pat)
while current_match != −1:
    matches.append(current_match)
    current_match = seq.find(pat, current_match+1)
```

initalization of the
changing variable
variable which change
during the repetition
condition of the repetion
in the header of the while statement
new value of the
changing variable
sequence of statements to repeat
in the body of the while statement

Look at the following example. Do you know what happened during the execution?

```
while 1:
   print "hello"
```

It simply never stops, because 1 is always true and therefore it is neither an algorithm nor a program. But it illustrates an important feature of a `while` loop. The conditional expression has to contain at least one variable whose value changes during the execution of the sequence of statements.

This variable is often a simple iterator, which is increased or decreased until it reaches a limit. One example of such an *iterator* is the variable `i` in Example 10.3. The behavior of the `current_match` variable in Example 10.2 is more complicated, because it represents two situations. Its value is the start position of a pattern occurrence or it is -1 if there are no more occurrences. In this example the second argument of the `find` function ensures that the *next* occurrence is found but no occurrence missed.

Because the iteration variable is used in the condition and the condition is evaluated first when a `while` loop is entered, it has to be defined before entering the loop. This means there are two different affectations of this variable. A first affectation or *initialization* before the loop is entered and a second affectation statement in the repeated sequence of statements that gives it the next value. In general, this is the last statement of the sequence. Even if it is not always the case, and sometimes not necessary, it is a convention to put it at the end if possible because it is easy to find if it has been forgotten.

Warning

Conditions of `while` loops are frequent error sources. They are sometimes difficult to detect because they are semantic errors that produce wrong results on all or sometimes only special input data sets. You have to ensure that the iteration variable changes and that the end condition is always reached.

10.4. Comparison of for and while loops

What are the differences between `while` and `for`? And in which cases it is more appropriated to use one or the other? There are many cases where it is only a question of feeling.

But the `while` loop is the more general one because its loop condition is more flexible and can be more complicated than that of a `for` loop. The condition of a `for` loop is always the same and implicit in the construction. A `for` loop stops if there are no more elements in the collection to treat. For simple traversals or iterations over index ranges it is a good advice to use the `for` statement because it handles the iteration variable for you, so it is more secure than `while` where you have to handle the end of the iteration and the change of the iteration variable by yourself.

The `while` loop can take every boolean expression as condition and permits therefore more complex end conditions. It is also the better choice if the iteration variable does not change evently by the same step or if there are more than one iteration variable. Even if you can handle more than one iteration variable in a `for` statement, the collections from which to choose the values must have the same number of elements.

Exercise 10.3. Rewrite for as while

Imagine that there is no `for` statement in the Python language. How can you rewrite it as `while` statement? Take the following pseudocode as example:

```
for var in collection:
    do something with var
```

The `for` loop is a special case of the

10.5. Range and Xrange objects

We have already introduced the `range` function which generates a list of integers from a start to an end value (the last is not included) with a regular interval. This function is very useful while writing `for` loops because it gives the possibility to iterate rather over the indices of a collection than over their elements. So you have not only access to the value of the element but also its position in the collection.

Sometimes you need to iterate over a great interval of integers. In this case the list generated by the `range` function would be long and therefore take a lot of memory space and time to be constructed. In such cases it is more appropriated to use the `xrange` function which is used in the same way as the `range` function. But rather than to create the list of integers physically in memory, it creates an object that calculates the new value if it is asked for the next element in the list.

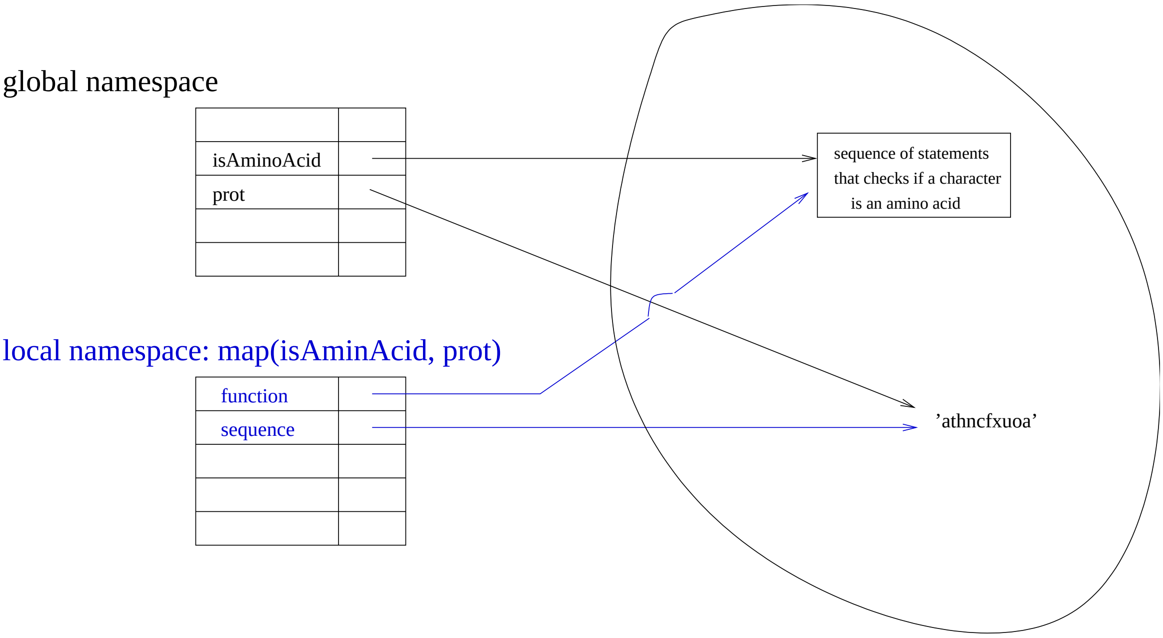
10.6. The map function

We will finish this chapter with the introduction of the function `map` which allows to apply a function to all arguments of an ordered collection. The result is always a list even if the ordered collection is a string. If the function takes more than one argument, as many collections as arguments have to be specified as arguments. `map` takes the first argument from the first collection, the second one from the second collection and so on.

```
>>> map(isAminoAcid, "atgahryuox")
[1, 1, 1, 1, 1, 1, 1, 0, 0, 1]

>>> def add10(n):
...     return n+10
...
>>> map(add10, [0,1,2,3,4,5])
[10, 11, 12, 13, 14, 15]
>>> def add(a,b):
        return a+b
>>> map(add, [1,2,3], [1,2,3])
[2,4,6]
```

Passing functions as arguments. This is the first example where we pass a function as an argument. Even if this looks strange at the beginning, *function names* are references as *variable names* which are bound to the sequences of statements defined in the body of the *function definition*. The sequence of statements can be viewed as an object which has the particularity that it can be called with arguments. Figure 10.4 shows what happens when passing a function as argument. This illustrates that there is no difference between variables and functions as arguments.

Figure 10.4. Passing functions as arguments


global namespace
isAminoAcid
prot
sequence of statements that checks if a character is an amino acid
local namespace: map(isAminAcid, prot)
function
sequence
'athncfxuoa'

Another example that handles function names as references is renaming a function as follows:

```
>>> isAA = isAminoAcid
>>> isAA('a')
1
```

The only difference between function and variable names is that function names are bound to objects that can be called using the function call syntax.

```
>>> isAminoAcid
<function isAminoAcid at 0x111ad0>
>>> isAminoAcid('a')
1
```

Without the parenthesis the object bound to the name `isAminoAcid` is shown by the interpreter, whereas adding the parentheses will call the function object bound to `isAminoAcid`.

Rewriting `map` with `for`. `map` with a function that takes only one argument, can be rewritten as follows..

Rewrite **map**(*func*, *collection*)

```
1. res < empty list
2. for each element in collection:
     • res < res appended with func(element)
3. return res
```

10.7. Solutions

Solution 10.1. Rewrite for as while

Exercise 10.3

```
while collection:
   var = collection[0]
   do something with var
   collection = collection[1:]
```

or using an index

```
i = 0
while i < len(collection):
   var = collection[i]
   do something with var
   i += 1
```

Chapter 11. Nested data structures

11.1. Nested data structures

Composed data structures, such as list, dictionaries and tuples, can be nested in Python. This means that you can use these structures as elements of themselves.

```
>>> l = [1, [2, 3, [4]], [5, 6]]
```

The list above contains 3 elements.

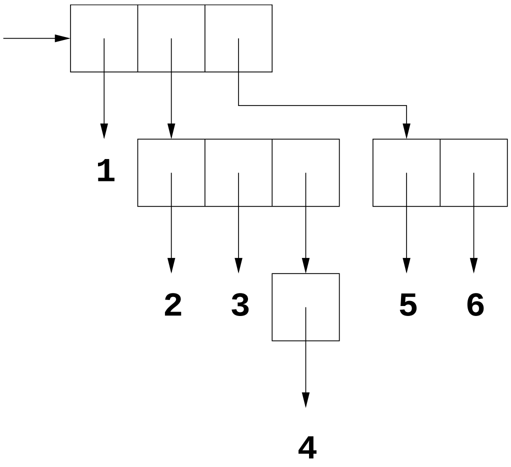
```
>>> len(l)
3
```

The first element is the integer 1, whereas the second and third element are lists themselves.

```
>>> l[0]
1
>>> l[1]
[2, 3, [4]]
>>> l[2]
[5, 6]
```

Figure 11.1 shows a representation of this list. Each element of a list is represented by a box containing a pointer to the value of the element. Like pointers in a namespace representation, these pointers are references. Remember that a *reference* is an address to a memory location (Section 2.4).

Figure 11.1. Representation of nested lists



As far we have said that a list is an ordered collection of some objects, but rather to contain the objects themselves, a list contains *references* to them.

How can we access to the elements of the internal list? Figure 11.1 makes clear there are different levels of the nested structure. Each time we follow a reference we go to the next level of the structure.

To access the value `2` of our example list, we could store the value of the second element in a variable which creates the new reference `l1` to this sublist. And then ask for the first element of `l1`.

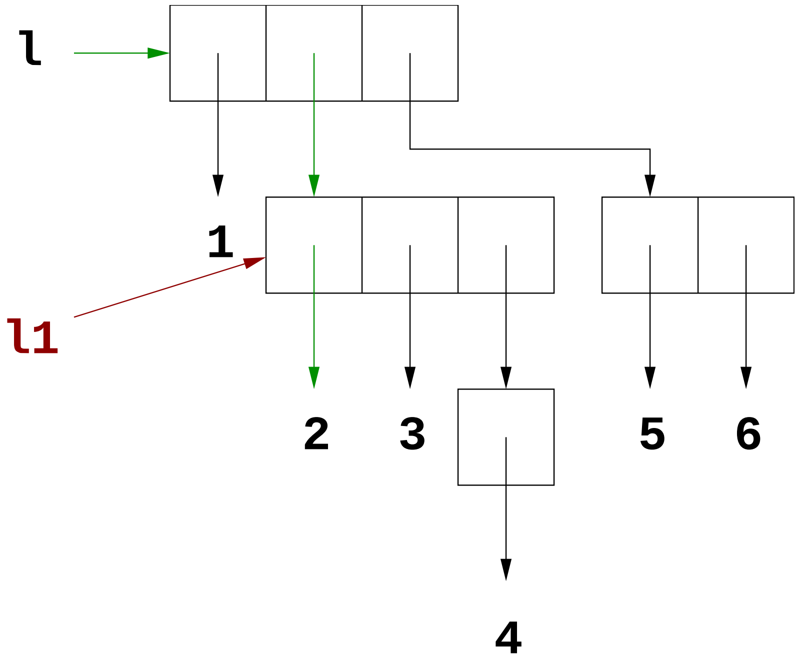
```
>>> l1 = l[1]
>>> l1
[2, 3, [4]]
>>> l1[0]
2
```

But we can also follow the references directly without creating a new variable.

```
>>> l[1][0]
2
```

Figure 11.2 colors the path of the references we have followed to access the value `2`.

Figure 11.2. Accessing elements in nested lists

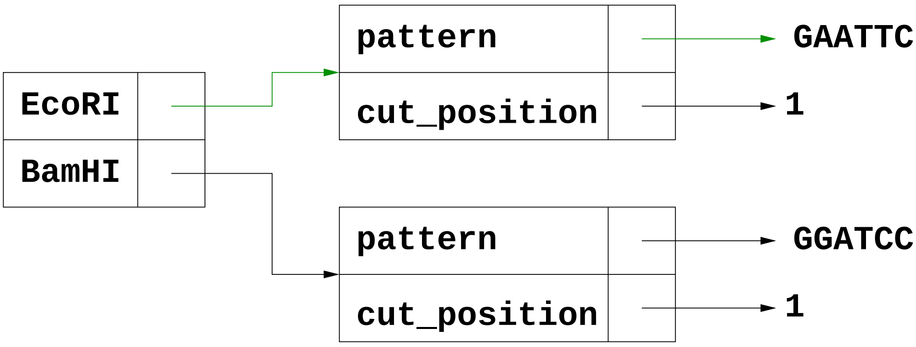


Nested dictionaries. Dictionaries can be nested in the same way as lists. Here is an example of a dictionary that stores restriction enzyme data.

```
>>> enzdict = { ’EcoRI’: {’pattern’: ’GAATTC’, ’cut_position’: ’1’},
      ’BamHI’: {’pattern’: ’GGATCC’, ’cut_position’: ’1’}}
>>> enzdict
{ ’EcoRI’: {’pattern’: ’GAATTC’, ’cut_position’: ’1’},
  ’BamHI’: {’pattern’: ’GGATCC’, ’cut_position’: ’1’}}

>>> enzdict[’EcoRI’][’pattern’]
’GAATTC’
```

Figure 11.3 shows a representation of `enzdict`.

Figure 11.3. Representation of a nested dictionary



Mixed structures. In the same way as above you can construct mixed structures. Here is an example of a restriction enzyme dictionary that stores, in addition to the pattern, all occurrences of the pattern in a sequence.

Example 11.1. A mixed nested datastructure

```
>>>  enzdict = { ’EcoRI’: {’pattern’: ’GAATTC’, ’occ’: [0, 109, 601]},
                 ’BamHI’: {’pattern’: ’GGATCC’, ’occ’: [31, 59]}}
```

Exercise 11.1. Representing complex structures

Try to draw a representation of the the mixed structure in Example 11.1.

11.2. Identity of objects

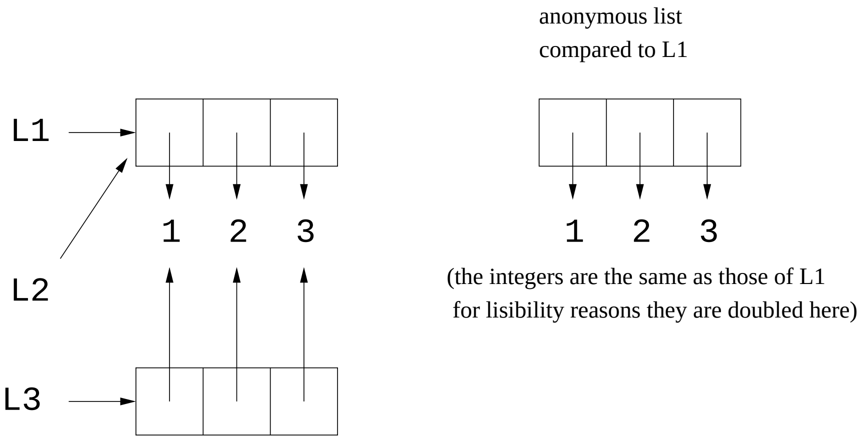
In ??? we have said that there exists the two operators `==` and `is` for comparing objects in Python without explaining their differences.

Both operators compare the identity of objects or values, but the `is` operator looks if their memory locations are the same, whereas the `==` operator rather does a content based comparison. Therefore for simple objects, such as integers, and immutable objects, such as strings, the operators return the same result, but their behavior is different for composed objects as lists and dictionaries. The following listing shows some examples:

```
>>> 10.0 == 10
1
>>> 10.0 is 10
0
>>> 10 is 10
1
>>> 10.0 is 10.0
1

>>> "atg" == "atg"
1
>>> "atg" is "atg"
1
>>> start = "atg"
>>> end = start
>>> start == end
1

>>> L1 = [1, 2, 3]
>>> L1 == [1, 2, 3]
1
>>> L1 is [1, 2, 3]
0
>>> L2 = L1
>>> L1 is L2
1
>>> L3 = L1[:]
>>> L1 is L3
0

>>> [1, [2, 3]] == [1, 2, 3]
0
>>> [1, [2, 3]] == [1, [2, 3]]
1
```

Figure 11.4 shows a representation to illustrate the examples of list comparisons done above.

Figure 11.4. List comparison


anonymous list
compared to L1
L1
L2
L3
1 2 3
1 2 3
(the integers are the same as those of L1
for lisibility reasons they are doubled here)

Identity of objects. In Python all objects have an identifier that is unique for each object. Therefore two objects are the same in the sense of `is` if they have the same identifier. You can ask for the identifier of an object in Python using the `id` function:

```
>>> id(10)
1104200
>>> id(10.0)
1424356

>>> id("atg")
1183232
>>> id(start)
1183232

>>> id(L1)
1177744
>>> id(L2)
1177744
>>> id(L3)
1121328
>>> id([1, 2, 3])
1174048
```

Warning

In Python strings are immutable and handled as the same in the sense of the `is` function if they contain the same sequence of characters, but this is not the case in all programming languages. In C for example, even strings with the same sequence of characters can have different memory locations.

11.3. Copying complex data structures

In the list examples of the above Section 11.2 we have given the example:

```
>>> L3 = L1[:]
```

and shown that `L3` and `L1` are not the same in memory

```
>>> L3 is L1
0
```

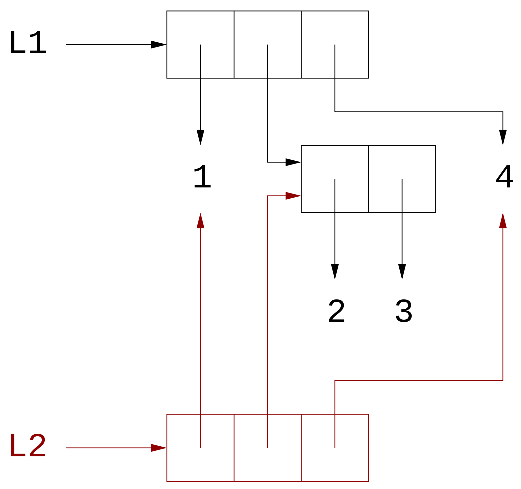
This is due to the fact that slicing a list creates a copy of a list and therefore `L3` is only a copy of `L1` but not the same object in memory.

Let's look at a second example with a nested list structure:

```
>>> L1 = [1, [2, 3], 4]
>>> L1
[1, [2, 3], 4]
>>> L2 = L1[:]
>>> L1 is L2
0
>>> L1[1] is L2[1]
1
```

Figure 11.5 illustrates what happens. In fact slicing creates only a shallow copy of compound objects by creating a new compound object and populating the new object with *references* to the members of the old list.

Figure 11.5. Copying nested structures



11.4. Modifying nested structures

Why is it important to know about the identity of compound object elements? Look at the following:

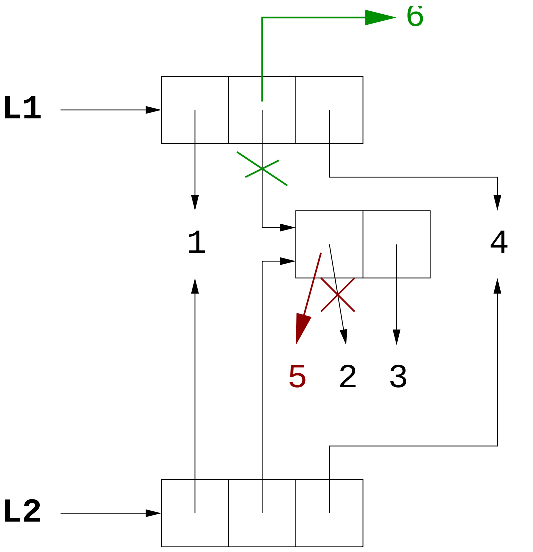
```
>>> L1 = [1, [2, 3], 4]
>>> L2 = L1[:]

>>> L2[1][0] = 5
>>> L2
[1, [5, 3], 4]
>>> L1
[1, [5, 3], 4]
```

This example modifies the first element of the list that is the second element of `L2`. But the second element of `L1` is the same in memory as the second element of `L2`. That is the reason why `L1` is also modified, whereas if we modifie the first level of `L1`, `L2` remains unmodified because that only changes independent references in the two list.

```
>>> L1
[1, [5, 3], 4]
>>> L2
[1, [5, 3], 4]
>>> L1[1] = 6
>>> L1
[1, 6, 4]
>>> L2
[1, [5, 3], 4]
```

Figure 11.6 illustrates in red the first modification and in green the second one.

Figure 11.6. Modifying compound objects



Copying references occurs also when you use variables.

```
>>> a = [1, 2]
>>> b = a
>>> a is b
1
>>> c = [a, 3]
>>> c
[[1, 2], 3]
>>> c[0] is a
1
>>> a[1] = 0
>>> c
[[1, 0], 3]
>>> a
[1, 0]
>>> b
[1, 0]

>>> c[0] = 0
>>> c
[0, 3]
>>> a
[1, 0]
```

Independent copies. It is possible to get an independent copy via the `deepcopy` function defined in the `copy` module.

```
>>> import copy
>>> L1
[1, 2, 4]
>>> L2
[1, [1, 3], 4]
>>> L3 = copy.deepcopy(L2)
>>> L3
[1, [1, 3], 4]
>>> L3[1] is L2[1]
0
```

Chapter 12. Files

During a program execution data are stored in memory. But if the program ends the data are cleared from memory. In this chapter we will explain how programs can communicate data permanently.

In the above sections we have seen how we can communicate *interactively* with programs during their execution. However, if we need to store data permanently, we have to store them in *files*.

Files

Files are entities containing character data which are stored on external devices, such as disks or CD-ROMs.

12.1. Handle files in programs

Files are like *books*. You *open* them to start working, then *read* or *write* in them and you *close* them when you have finished your work. However, you have always to know where you are in the book. As children use their fingers when they start to learn reading, you manipulate a *file pointer* which indicates your current position in the file.

File data are ordered collections, such as lists. But you have to traverse all elements to get to a position, whereas lists can by accessed directly using an index.

Opening. When you open a file in Python using the built-in function `open`, you have to indicate, in addition to its name, whether you want to read from it or write in it. The default is reading.

Working. The *file object* created by the open function has methods to read from the file, to write in it and to move the *file pointer* to another position.

Although Python can handle files containing binary data as well as text data, *file objects* have special functions to work with text files, such as reading line per line.

At this point we like to show two possibilities to handle text file data line per line. The first one uses the loop `while`:

```
infile="<some-file-name">
infh=open(infile)

line=infh.readline()
while line:
    #do something with the line
    line=infh.readline()

infh.close()
```

And the second one the loop `for`.

Example 12.1. Reading from files

```
infile="<some-file-name>"
infh=open(infile)

for line in infh.xreadlines()
    #do something with the line

infh.close()
```

There exists two file methods, `readlines` and `xreadlines`, with the same properties as `range` and `xrange`. The first one reads all the lines of a file and puts them in a list, whereas the second one allows to iterate over all lines without creating the list. The first one is appropriated for small files, but if you are not sure about the number of lines in your file, the second one prevents to overload the memory.

Table 12.1 gives a list of all common used file methods and Table 12.2 explains all possible *modes* to work with files.

Table 12.1. File methods

Method	Action
`read([n])`	reads at most `n` bytes; if no `n` is specified, reads the entire file
`readline([n])`	reads a line of input, if `n` is specified reads at most `n` bytes
`readlines()`	reads all lines and returns them in a list
`xreadlines()`	reads all lines but handles them as a `XRangeType`[a]
`write(s)`	writes strings `s`
`writelines(l)`	writes all strings in list `l` as lines
`close()`	closes the file
`seek(offset [, mode])`	changes to a new file `position=start + offset`. `start` is specified by the mode argument: `mode=0` (default), `start` = start of the file, `mode=1`, `start` = current file position and `mode=2`, `start` = end of the file

[a] See Section 10.5 for more informations

Table 12.2. File modes

Mode	Description
r	read
w	write
a	append

[rwa]b	[reading,writing,append] as binary data (required on Windows)
r+	update+reading (output operations must flush their data before subsequent input operations)
w+	truncate to size zero followed by writing

Closing. Although Python closes all opened files when the interpreter exits, it is a good idea to close them explicitly. Why ???

Important

Pay attention, while `open` is a built in *function* creating a file object, `close` is a *method* of the created file object.

12.2. Reading data from files

Let us work on a concrete example. In Section 10.3, we have written the `findpos` function that finds all occurrences of a pattern in a sequence. In the following example, we will show how we can read restriction site patterns of enzymes described in the ReBase [http://www.rebase.org] database.

ReBase database format. In our example we will use an excerpt of the `gcgenz.dat` file of the restriction enzyme ReBase database. The information for each restriction enzyme is stored in one line of this file. The information about isoschizomers and the header of the file containing explanations have been omitted in the excerpt that we will use. Figure 12.1 shows a part of this file. Each line starts with the enzyme name, followed by the number of bases after which it cuts the restriction site pattern, the sequence of the restriction site, the length of the overhang and some comments about the isoschizomeres and the availability of the enzyme.

Figure 12.1. ReBase file format

```
AarI      11 CACCTGCnnnn’nnnn_   4 !                                    >F 201,386
AatII      5 G_ACGT’C      -4 !  ZraI                              >AEFGIKMNOR 624
AccI       2 GT’mk_AC       2 !  FblI,XmiI                         >ABEGJKMNORSU 288,374,751

Acc65I     1 G’GTAC_C       4 !  KpnI,Asp718I                      >FGINR 505
AciI       1 C’CG_C         2 !                                    >N 497
AclI       2 AA’CG_TT       2 !  Psp1406I                          >IN 140
AfeI       3 AGC’GCT        0 !  Eco47III,Aor51HI,FunI             >IN 15
AflII      1 C’TTAA_G       4 !  BfrI,BspTI,Bst98I,MspCI,Vha464I   >ABJKNO 722
AflIII     1 A’CryG_T       4 !                                    >BGMNS 722
AgeI       1 A’CCGG_T       4 !  AsiAI,BshTI,CspAI,PinAI           >GJNR 739
```

Using the general scheme of reading lines from files shown above (Example 12.1), the following shows one possibility for extracting the restriction site pattern from the line.

Procedure 12.1. read_rebase

INPUT: a *file* in rebase format

OUTPUT: a dictionary *enz_dict* containing all restriction patterns indexed by their name

```
1. enz_dict <- empty dictionary
2. infh <- open file for reading
3. for each line read from infh:
    a. split line in its fields
    b. name <- first field
    c. pat <- third field
    d. clean pat to get a string containing only the sequence recognized by the restriction enzyme
    e. add the cleaned pat to enz_dict with name as key
4. close infh
5. return enz_dict
```

here is one possibility how to translate this procedure into Python:

```
def get_site_only(pat):
    newpat = ""
    for c in pat:
        if c.isalpha():
            newpat +=c
    return newpat

def read_rebase(filename):
    enz_dict={}
    infh= open(filename)
    for line in infh.xreadlines():
        fields = line.split()
        name = fields[0]
        pat = fields[2]
        enz_dict[name] = get_site_only(pat)

    infh.close()
    return enz_dict

print read_rebase("../data/rebase.dat")
```

12.3. Writing in files

Let us continue our restriction site example. Because we have got the enzyme pattern from the ReBase database, we can now process our sequence with all these patterns using the `findpos` function (Example 10.2). There is only one restriction: at the moment the `findpos` function can only find *exact* restriction patterns, so we have to exclude all patterns containing ambiguous bases.

INPUT: a dictionary *enz_dict* containing all restriction site patterns accessible by enzyme name, a sequence *seq* to search for

OUTPUT: list of start position of every occurrence for each pattern in the dictionary.

```
• for each key enzname in enz_dict:
    a. pat <- pattern of enzname in enz_dict
    b. if pat is an exact pattern
        i. occs <- findpos(seq,pat)
        ii. print all occs
```

The `print_matches` in the following listing of functions prints the results of the analysis on the screen.

```
seq = ""

def isexact(pat):
    for c in pat.upper():
        if c not in ’ATGC’:
            return 0
    return 1

def findpos(seq, pat):
    matches = []
    current_match = seq.find(pat)
    while current_match != -1:
        matches.append(current_match)
        current_match =seq.find(pat, current_match+1)
    return matches


def print_matches(enz, matches):
    if matches:
        print "Enzyme %s matches at:" % enz,
        for m in matches:
            print m,
        print
    else:
        print "No match found for enzyme %s." % enz

for enzname in enz_dict.keys():
    pat = enz_dict[enzname]
    if isexact(pat):
        print_matches(enzname, findpos(seq, pat))
```

In order to store the results permanently, we will see now how we can write the information in a file. As for reading, we have to open a file although now in a writing mode, write our results in the file and then close it.

```
def print_matches(enz, matches):
    ofh = open("rebase.res", "w")
    if matches:
        print >>ofh, "Enzyme %s matches at:" % enz,
        for m in matches:
            print >>ofh, m,
        print >>ofh
    else:
        print >>ofh, "No match found for enzyme %s." % enz
    ofh.close()
```

The problem with this `print_matches` function is that it prints only the result of *last* enzyme. Because if we close the file after writing the information, the next time we will open the file for writing the next result, we will overwrite the old result. We have two possibilities to solve this. First, we can open the file to *append* at the end of the file. Or second, we can open the file for writing in the main stream of the program and then pass the file object as argument to `print_matches`, and close the file only when all results have been written. We prefer the second solution.

```
seq = ""

def isexact(pat):
    for c in pat.upper():
        if c not in ’ATGC’:
            return 0
    return 1

def findpos(seq, pat):
    matches = []
    current_match = seq.find(pat)
    while current_match != -1:
        matches.append(current_match)
        current_match =seq.find(pat, current_match+1)
    return matches

def print_matches(ofh, enz, matches):
    if matches:
        print >>ofh, "Enzyme %s matches at:" % enz,
        for m in matches:
            print >>ofh, m,
        print >>ofh
    else:
        print >>ofh, "No match found for enzyme %s." % enz

ofh = open("rebase.res", "w")
for enzname in enz_dict.keys():
    pat = enz_dict[enzname]
    if isexact(pat):
        print_matches(ofh, enzname, findpos(seq, pat))
ofh.close()
```

Although it is possible to use the `write` or `writelines` methods of the file object, we have shown in the above example how to pass a file object to the `print` statement. Which syntax you will use in your own code, is a question of taste. But the code could be difficult to read if you mix them.

12.4. Design problems

Here is the complete program to find all occurrences of restriction sites from a set of enzymes in a sequence. The enzyme data are read from a file and the results are also stored in a file.

```
def findpos(seq, pat):
    matches = []
    current_match = seq.find(pat)
    while current_match != -1:
        matches.append(current_match)
        current_match =seq.find(pat, current_match+1)
    return matches

def isexact(pat):
    for c in pat.upper():
        if c not in ’ATGC’:
            return 0
    return 1

def print_matches(ofh, enz, matches):
    if matches:
        print >>ofh, "Enzyme %s matches at:" % enz,
        for m in matches:
            print >>ofh, m,
        print >>ofh
    else:
        print >>ofh, "No match found for enzyme %s." % enz

def get_site_only(pat):
    newpat = ""
    for c in pat:
        if c.isalpha():
            newpat += c
    return newpat

def read_rebase(filename):
    enz_dict = {}
    infh = open(filename)
    for line in infh.xreadlines():
        fields = line.split()
        name = fields[0]
        pat = fields[2]
        enz_dict[name] = get_site_only(pat)

    infh.close()
    return enz_dict

def findpos(seq, pat):
    matches = []
    current_match = seq.find(pat)
    while current_match != -1:
        matches.append(current_match)
        current_match =seq.find(pat, current_match+1)
    return matches


seq = """atgagtgaacgtctgagcattaccccgctggggccgtatatcggcgcacaaa
tttcgggtgccgacctgacgcgcccgttaagcgataatcagtttgaacagctttaccatgcggtg
ctgcgccatcaggtggtgtttctacgcgatcaagctattacgccgcagcagcaacgcgcgctggc
ccagcgttttggcgaattgcatattcaccctgtttacccgcatgccgaaggggttgacgagatca
tcgtgctggatacccataacgataatccgccagataacgacaactggcataccgatgtgacattt
attgaaacgccacccgcaggggcgattctggcagctaaagagttaccttcgaccggcggtgatac
gctctggaccagcggtattgcggcctatgaggcgctctctgttcccttccgccagctgctgagtg
ggctgcgtgcggagcatgatttccgtaaatcgttcccggaatacaaataccgcaaaaccgaggag
gaacatcaacgctggcgcgaggcggtcgcgaaaaacccgccgttgctacatccggtggtgcgaac
gcatccggtgagcggtaaacaggcgctgtttgtgaatgaaggctttactacgcgaattgttgatg
tgagcgagaaagagagcgaagccttgttaagtttttttgtttgcccatatcaccaaaccggagttt
caggtgcgctggcgctggcaaccaaatgatattgcgatttgggataaccgcgtgacccagcacta
tgccaatgccgattacctgccacagcgacggataatgcatcgggcgacgatccttggggataaac
cgttttatcgggcggggtaa""".replace("\n","").upper()


enzdict = read_rebase("../data/rebase.dat")

ofh = open("rebase.res", "w")
for enzname in enzdict.keys():
    pat = enzdict[enzname]
    if isexact(pat):
        print_matches(ofh, enzname, findpos(seq, pat))
ofh.close()
```

We have written this program step by step by broadening the problem at each step, but we have never looked at the global design of the program. Therefore we have used two strategies while reading and writing. In the reading step all data are read at once, whereas the results are written each time we got them.

In biological problems it is often necessary to handle lots of data and this can lead to problems if you try to handle all of them in memory, as we have done with the enzyme data. Sometimes we need all the data to solve a problem, but often we can treat them each at a time. If the dataset is small, the choice of the strategy does not change a lot, but if you handle a large amount of data, it is worth asking whether you really need it all in memory?

In our restriction problem it is not really necessary to store all the enzyme data in the memory. It is possible to treat enzyme by enzyme with one loop that reads the enzyme data, processes the sequence and writes the result.

Example 12.2. Restriction of a DNA sequence

```
def isexact(pat):
    for c in pat.upper():
        if c not in ’ATGC’:
            return 0
    return 1

def print_matches(ofh, enz, matches):
    if matches:
        print >>ofh, "Enzyme %s matches at:" % enz,
        for m in matches:
            print >>ofh, m,
        print >>ofh
    else:
        print >>ofh, "No match found for enzyme %s." % enz

def get_site_only(pat):
    newpat = ""
    for c in pat:
        if c.isalpha():
            newpat += c
    return newpat

def findpos(seq, pat):
    matches = []
    current_match = seq.find(pat)
    while current_match != -1:
        matches.append(current_match)
        current_match =seq.find(pat, current_match+1)
    return matches


seq = """atgagtgaacgtctgagcattaccccgctggggccgtatatcggcgcacaaa
tttcgggtgccgacctgacgcgcccgttaagcgataatcagtttgaacagctttaccatgcggtg
ctgcgccatcaggtggtgtttctacgcgatcaagctattacgccgcagcagcaacgcgcgctggc
ccagcgttttggcgaattgcatattcaccctgtttacccgcatgccgaaggggttgacgagatca
tcgtgctggatacccataacgataatccgccagataacgacaactggcataccgatgtgacattt
attgaaacgccacccgcaggggcgattctggcagctaaagagttaccttcgaccggcggtgatac
gctctggaccagcggtattgcggcctatgaggcgctctctgttcccttccgccagctgctgagtg
ggctgcgtgcggagcatgatttccgtaaatcgttcccggaatacaaataccgcaaaaccgaggag
gaacatcaacgctggcgcgaggcggtcgcgaaaaacccgccgttgctacatccggtggtgcgaac
gcatccggtgagcggtaaacaggcgctgtttgtgaatgaaggctttactacgcgaattgttgatg
tgagcgagaaagagagcgaagccttgttaagtttttttgtttgcccatatcaccaaaccggagttt
caggtgcgctggcgctggcaaccaaatgatattgcgatttgggataaccgcgtgacccagcacta
```

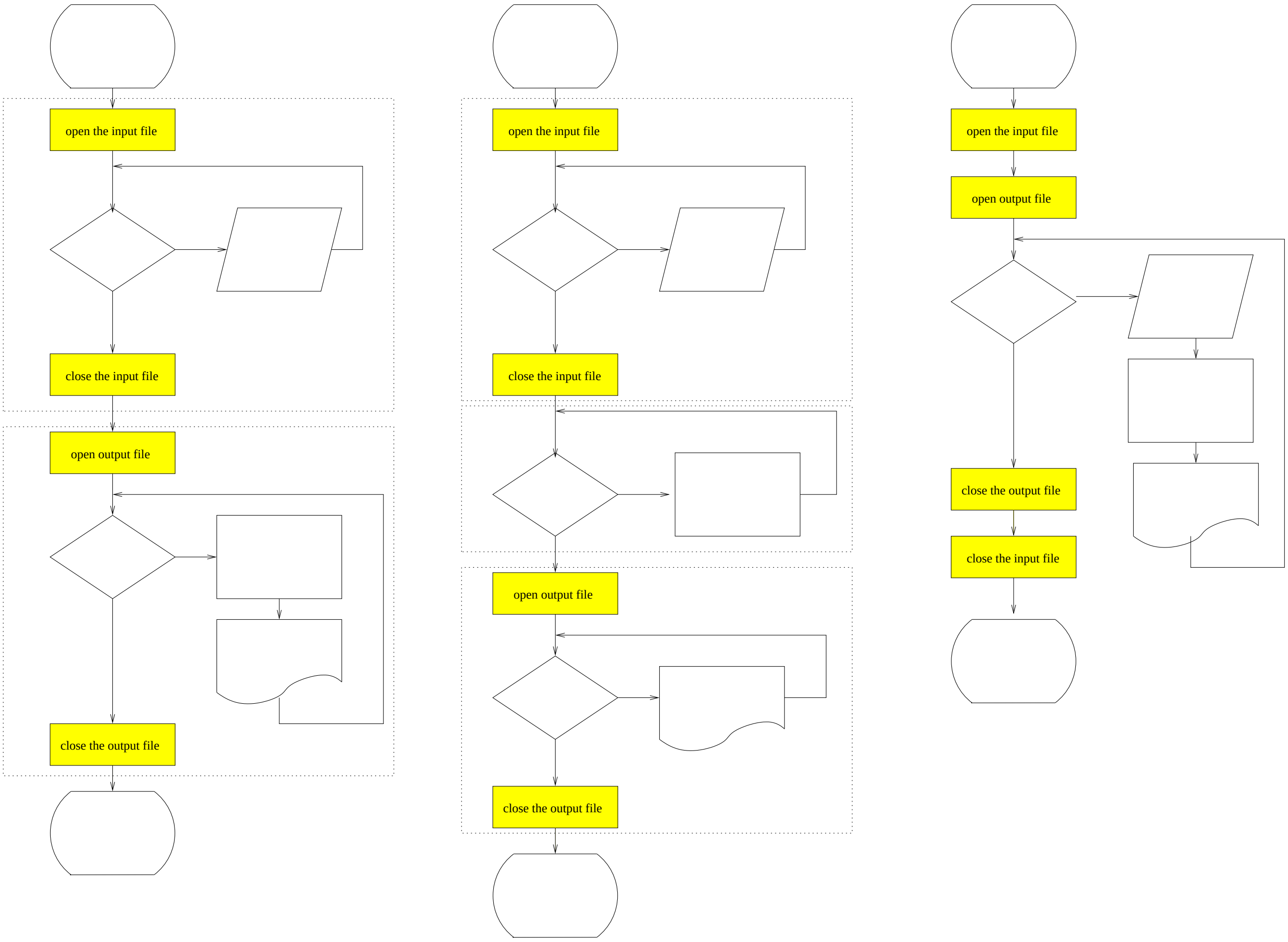
```
tgccaatgccgattacctgccacagcgacggataatgcatcgggcgacgatccttggggataaac
cgttttatcgggcggggtaa""".replace("\n","").upper()


ifh = open("../data/rebase.dat")
ofh = open("rebase.res", "w")

line = ifh.readline()

while line:
    fields = line.split()
    name = fields[0]
    pat = get_site_only(fields[2])

    if isexact(pat):
        print_matches(ofh, name, findpos(seq, pat))
        line = ifh.readline()
    else:
     line = ifh.readline()

ofh.close()
ifh.close()
```

Important

Notice that there are two files opened at the same time during the loop.

Figure 12.2 shows three flowchart comparing our first solution, with the new design and a version that first reads all data in, handles them and writes all results at the end.

Figure 12.2. Flowchart of the processing of the sequence



Exercise 12.1. Multiple sequences for all enzymes

Think about alternative solutions, their advantages and problems if you need to process more than one sequence for all restriction enzymes.

12.5. Documentation strings

The code of the program written in this section (Example 12.2) fills about one page. If you put it away for a while and look on it a week later, it might be difficult to remember all the choices made during the development and it might take a lot of time to understand it. This fact underlines the importance to **document programs**.

For this purpose you can use comments that are ignored by the interpreter. In Python, all lines starting with a `#` character are comments. But Python has a more powerful mechanism to document programs. If the first statement of a function or file is a string, this string is used as documentation string which is available by the **pydoc** command in a shell or the `help` function of the interpreter.

Here is a documented version of our restriction program.

```
#!/usr/bin/python
"""
program that finds all occurences of restriction enzymes in a sequence

at the moment the sequence is contained in the seq variable
  todo: reads the sequence from a field

enzyme data are read from the file: rebase.dat
and results are written to file: restrict.res

restriction: the program finds only occurences of EXACT restriction patterns
"""

def isexact(pat):
    """
checks if a pattern is an exact DNA sequence
all strings containing only the following characters are considered to be
an exact DNA sequence: atgcATGC
    """
    for c in pat.upper():
        if c not in ’ATGC’:
            return 0
    return 1

def print_matches(ofh, enz, matches):
    """
write starting positions of occurences of a restriction site to a file
positions in the sequence starts by 1
    """
    if matches:
        print >>ofh, "Enzyme %s matches at:" % enz,
        for m in matches:
            print >>ofh, m+1,
        print >>ofh
    else:
        print >>ofh, "No match found for enzyme %s." % enz

def get_site_only(pat):
    """
clean a pattern string read from Rebase to obtain only the recognition site
sequence
   """
    newpat = ""
    for c in pat:
        if c.isalpha():
            newpat += c
    return newpat

def findpos(seq, pat):
    """
find all occurences of restriction site pattern in a sequence
RETURN VALUE: a list containing the start positions of all occurences
RESTRICTION: can only process exact patterns, searching with ambiguous
pattern strings would not produce an error
    """
    matches = []
    current_match = seq.find(pat)
    while current_match != -1:
        matches.append(current_match)
        current_match =seq.find(pat, current_match+1)
    return matches

# starting main stream

seq = """atgagtgaacgtctgagcattaccccgctggggccgtatatcggcgcacaaa
tttcgggtgccgacctgacgcgcccgttaagcgataatcagtttgaacagctttaccatgcggtg
ctgcgccatcaggtggtgtttctacgcgatcaagctattacgccgcagcagcaacgcgcgctggc
ccagcgttttggcgaattgcatattcaccctgtttacccgcatgccgaaggggttgacgagatca
tcgtgctggatacccataacgataatccgccagataacgacaactggcataccgatgtgacattt
attgaaacgccacccgcaggggcgattctggcagctaaagagttaccttcgaccggcggtgatac
gctctggaccagcggtattgcggcctatgaggcgctctctgttcccttccgccagctgctgagtg
ggctgcgtgcggagcatgatttccgtaaatcgttcccggaatacaaataccgcaaaaccgaggag
gaacatcaacgctggcgcgaggcggtcgcgaaaaacccgccgttgctacatccggtggtgcgaac
gcatccggtgagcggtaaacaggcgctgtttgtgaatgaaggctttactacgcgaattgttgatg
tgagcgagaaagagagcgaagccttgttaagtttttttgtttgcccatatcaccaaaccggagttt
caggtgcgctggcgctggcaaccaaatgatattgcgatttgggataaccgcgtgacccagcacta
tgccaatgccgattacctgccacagcgacggataatgcatcgggcgacgatccttggggataaac
cgttttatcgggcggggtaa""".replace("\n","").upper()

# open the input and output file
ifh = open("../data/rebase.dat")
ofh = open("rebase.res", "w")

# process enzyme by enzyme
line = ifh.readline()
while line:
    # extract enzyme name and pattern string
    fields = line.split()
    name = fields[0]
    pat = get_site_only(fields[2])

    # find pattern and write the result
    if isexact(pat):
        print_matches(ofh, name, findpos(seq, pat))
    # get the next enzyme
    line = ifh.readline()

# close opened files
ofh.close()
ifh.close()
```

and the result of the documentation formatted with **pydoc**

```
Python Library Documentation: module restrict_documented

NAME
    restrict_documented - Program that find all occurrences of restriction enzymes in a sequence


FILE
FILE
    /home/maufrais/cours_python/code/restrict_documented.py

DESCRIPTION
    At the moment the sequence is contained in the seq variable

    Enzyme data are red from the file: rebase.dat
    and results are written to file: restrict.res

    restriction: the program finds only occurrences of EXACT restriction patterns

FUNCTIONS
    get_site_only(pat)
        clean a pattern string red from Rebase to obtain only the recognition site
        sequence

    isexact(pat)
        checks if a pattern is an exact DNA sequence
        all strings containing only the following characters are considered to be
        an exact DNA sequence: atgcATGC

    print_matches(ofh, enz, matches)
        write starting positions of occurrences of a restriction site to a file
        positions in the sequence starts by 1

    findpos(seq, pat)
        find all occurrences of restriction site pattern in a sequence
        RETURN VALUE: a list containing the start positions of all occurrences
        RESTRICTION: can only process exact patterns, searching with ambiguous
        pattern strings would not produce an error

DATA
    __file__ = './restrict_documented.pyc'
```

```
__name__ = ’restrict_documented’
fields = [’ZraI’, ’3’, "GAC’GTC", ’0’, ’!’, ’AatII’, ’>I’, ’136’]
ifh = <closed file ’../data/rebase.dat’, mode ’r’>
line = ”
name = ’ZraI’
ofh = <closed file ’rebase.res’, mode ’w’>
pat = ’GACGTC’
seq = ’ATGAGTGAACGTCTGAGCATTACCCCGCTGGGGCCGTATATCGGCGC...CATCGGGCGACGA...
```

Chapter 13. Recursive functions

13.1. Recursive functions definitions

In Chapter 10 we have introduced a way to execute repetitive tasks, but we have also mentionned that repetition can be done without the special `for` and `while`loop statements. This chapter will explain this in detail.

Translate a cds sequence into its corresponding protein. Let’s start with the example of the cds translation. In Chapter 10 we have already written functions that solve this task using either `for` (Example 10.1) or `while` (Example 10.3). Let’s remind the pseudocode of the examples:

```
INPUT: a cds sequence cds and a genetic code code

OUTPUT: the translated cds sequence prot

1. prot <- empty string
2. as long as there are still codons to translate:

    a. codon <- get next codon from sequence cds
    b. lookup the corresponding amino acid of codon codon in genetic code code and add it to prot

3. return prot
```

```
1. prot <- empty string
2. for each codon in sequence cds:

    • add the corresponding amino acid of the current codon to sequence prot using genetic code code

3. return prot
```

In both examples the translation is defined as:

- the concatenation of the first codon of the sequence and the translation of the cds sequence without this codon or in more mathematical terms:

- `translation(cds) = code(cds[:3]) + translation(cds[3:])`

This is a *recursive definition* of the translation function.

Recursive functions

A *recursive function* is a *function* that use itself during the calculation procedure.

Recursive definition

A *recursive definition* of a function is a definition that uses the function itself in the definition.

It is important to notice that we need a **terminal condition** otherwise the recursion would never stop. In our case the recursion stops if there are no more codons to translate:

- the protein sequence of the an empty cds sequence is empty
- or in more mathematical terms: `translation("") = ""`

Therefore the pseudocode can be written as follow without using loop structures:

INPUT: a cds sequence *cds* and a genetic code *code*

OUTPUT: the translated sequence *prot*

```
1. if cds is empty:
    a. *** This is the terminal condition ***
    b. return empty string
2. otherwise:
    a. *** This is the recursion ***
    b. codon <- first codon of sequence cds
    c. return the concatenation of the corresponding amino acid of the first codon of sequence cds in genetic
       code code and the translation of the rest of the cds sequence
```

and implemented in Python like that:

```
  code = {’ttt’: ’F’, ’tct’: ’S’, ’tat’: ’Y’, ’tgt’: ’C’,
        ’ttc’: ’F’, ’tcc’: ’S’, ’tac’: ’Y’, ’tgc’: ’C’,
        ’tta’: ’L’, ’tca’: ’S’, ’taa’: ’*’, ’tga’: ’*’,
        ’ttg’: ’L’, ’tcg’: ’S’, ’tag’: ’*’, ’tgg’: ’W’,
        ’ctt’: ’L’, ’cct’: ’P’, ’cat’: ’H’, ’cgt’: ’R’,
        ’ctc’: ’L’, ’ccc’: ’P’, ’cac’: ’H’, ’cgc’: ’R’,
        ’cta’: ’L’, ’cca’: ’P’, ’caa’: ’Q’, ’cga’: ’R’,
        ’ctg’: ’L’, ’ccg’: ’P’, ’cag’: ’Q’, ’cgg’: ’R’,
        ’att’: ’I’, ’act’: ’T’, ’aat’: ’N’, ’agt’: ’S’,
        ’atc’: ’I’, ’acc’: ’T’, ’aac’: ’N’, ’agc’: ’S’,
        ’ata’: ’I’, ’aca’: ’T’, ’aaa’: ’K’, ’aga’: ’R’,
        ’atg’: ’M’, ’acg’: ’T’, ’aag’: ’K’, ’agg’: ’R’,
        ’gtt’: ’V’, ’gct’: ’A’, ’gat’: ’D’, ’ggt’: ’G’,
        ’gtc’: ’V’, ’gcc’: ’A’, ’gac’: ’D’, ’ggc’: ’G’,
        ’gta’: ’V’, ’gca’: ’A’, ’gaa’: ’E’, ’gga’: ’G’,
        ’gtg’: ’V’, ’gcg’: ’A’, ’gag’: ’E’, ’ggg’: ’G’
       }

def rectranslate(cds, code):
    if cds == "":
        return ""
    else:
        codon = cds[:3]
        return  code[codon] + rectranslate(cds[3:], code)

print rectranslate("atgattgctggt", code)
```

13.2. Flow of execution of recursive functions

At first glance recursive functions look a little bit strange because it seems that we use something that we have not defined. But remember that the statements in the body of a function are only executed when the function is called. At the definition step, the function is just added in the current namespace. Therefore it will be defined when we call it in the body.

Writing recursive functions is sometimes difficult, because we doubt that we have defined a function executing the specific task, when we call the it . Therefore it is useful to remember this fact during the development of recursive functions.

Stack diagram of recursive functions. Let’s have a deeper look at the flow of execution of recursive functions. Figure 13.1 shows the stack diagram of function calls.

Figure 13.1. Stack diagram of recursive function calls

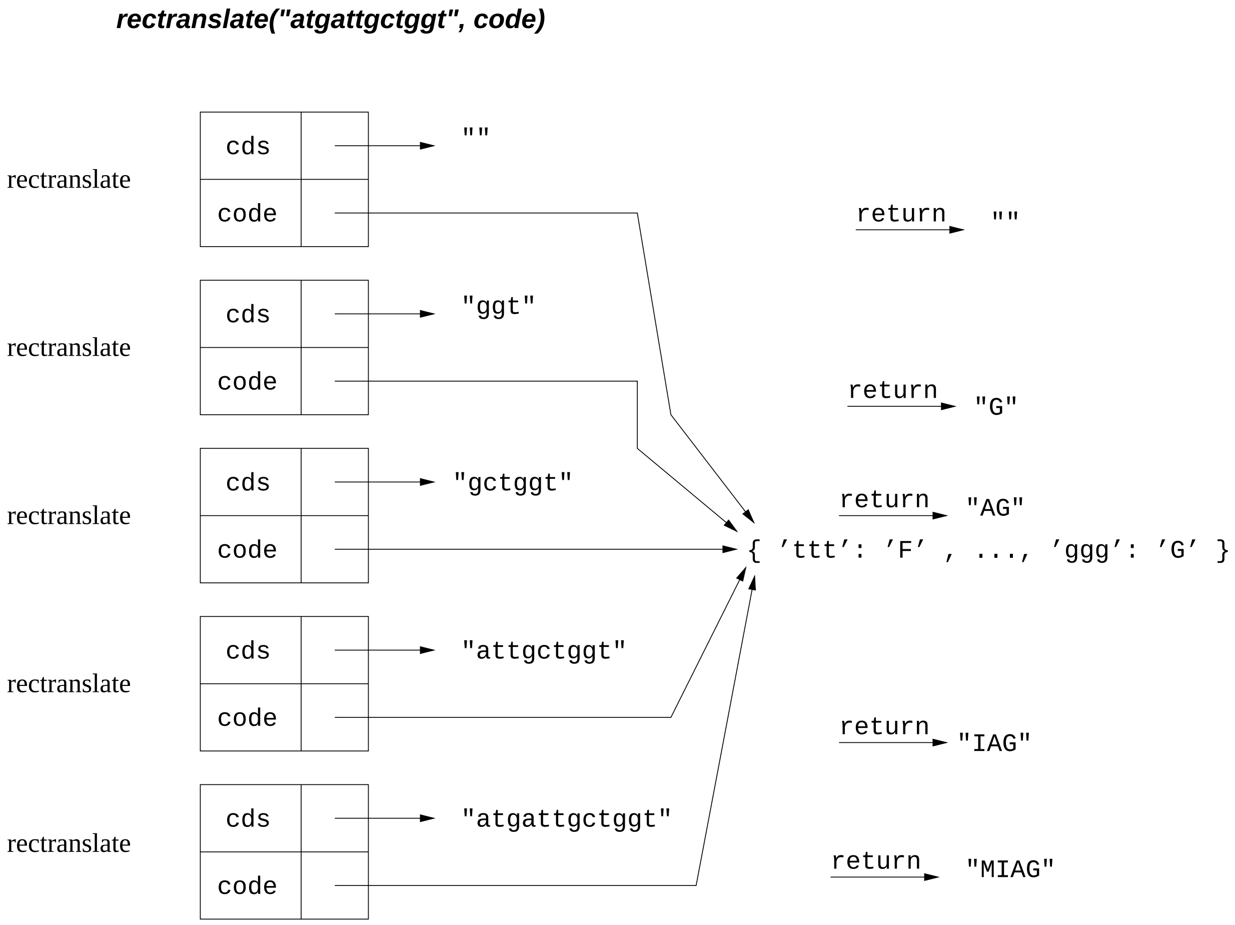


Figure 13.1 shows that the `translate` function is called with a cds sequence of decreasing length and the local function namespaces are piled up until the terminal condition is reached.

```
=> rectranslate("atgattgctggt", code)
===> rectranslate("attgctggt", code)
=====> rectranslate("gctggt", code)
=======> rectranslate("ggt", code)
=========> rectranslate("", code)
=========> return ""
=======> return "G"
=====> return "AG"
===> return "IAG"
=> return "MIAG"
```

Important

Recursive function calls can take a lot of memory space because for each recursive call the local namespace for the given function call has to be stored. It is important to pay attention on this fact because memory space is a limited resource on a computer.

13.3. Recursive data structures

In the Chapter 11, we have worked with examples using nested list. If we examine them a little bit more, we will see that a nested list can be defined recursively.

Nested list

A nested list is an ordered collection of elements that are:

- either items
- or nested lists

The items define the content, whereas the lists define the structure of the collection.

How can we use this? Let’s use phylogenetic trees as examples for such a recursive list structure. Figure 13.2 shows an example of a phylogenetic tree topology that can be implemented in Python as follows:

```
>>> tree = [[[['human', 'chimpanzee'], 'gorilla'], 'orang-utan'], 'gibbon']
```

Figure 13.2 shows the representation of `tree`.

Figure 13.2. A phylogenetic tree topology

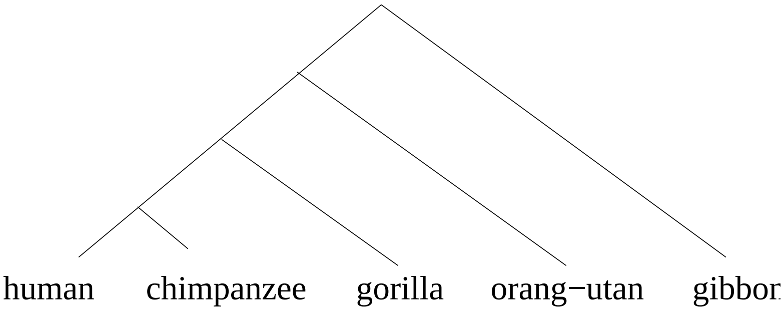
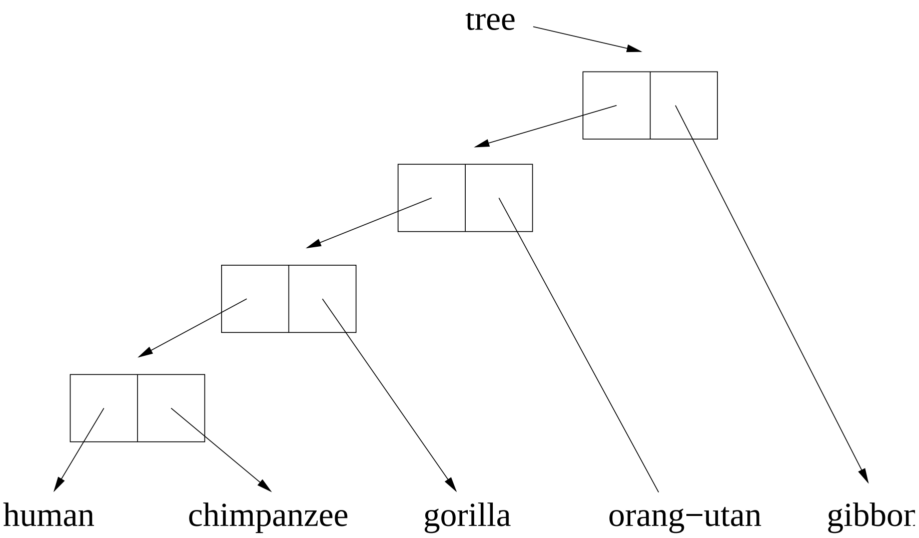


Figure 13.3. Tree representation using a recursive list structure



Working with recursive structures. Assume that we would do something for each branching point in the above tree. A simple traversal using `for` or `while` is no more possible, because both loop structures handle only one level of the list. But with recursive functions that we have introduced in Chapter 13, we can do such things with the following strategy:

Procedure 13.4. tree_traversal(tree)

INPUT: a tree *tree*

1. do what you have to do when you enter the branching point
2. **if** *tree* is a list:

    a. *** recursion step ***
    b. **for each** *element* of *tree*:

        - **tree_traversal**(*element*)

3. **otherwise**:

    - *** end condition of the recursion, here the leaves of the tree ***

4. do what you have to do when you leave the branching point

Doing something when you enter the branching point, is also known as *preorder traversal* of a tree, whereas doing something when leaving a branching point is also called *postorder traversal* of a tree.

We will give two example applications of this strategy for our tree. The first one prints all species of the trees:

```
    tree = [[[[’human’, ’chimpanzee’], ’gorilla’], ’orang-utan’], ’gibbon’]

import types

def print_species(tree):
    if type(tree) is types.ListType:
        for child in tree:
            print_species(child)
    else:
        print tree
```

```
>>> print_species(tree)
human
chimpanzee
gorilla
orang-utan
gibbon
```

and the second one that prints for a tree or a binary tree [2] the two populations that are split at this point.

```
    tree = [[[[’human’, ’chimpanzee’], ’gorilla’], ’orang-utan’], ’gibbon’]

import types

def splits(tree):
    if type(tree) is types.ListType:
        all_leaves = []
        for child in tree:
            child_leaves = splits(child)
            print child_leaves,
            all_leaves += child_leaves
```

A binary tree is a tree with maximal two subtrees for each branching point.

```
        print
        return all_leaves
    else:
        return [ tree ]


def binary_splits(tree):
    if type(tree) is types.ListType:
        left = binary_splits(tree[0])
        right = binary_splits(tree[1])
        print left, right
        return left + right
    else:
        return [ tree ]
```

```
>>> splits(tree)
['human'] ['chimpanzee']
['human', 'chimpanzee'] ['gorilla']
['human', 'chimpanzee', 'gorilla'] ['orang-utan']
['human', 'chimpanzee', 'gorilla', 'orang-utan'] ['gibbon']
['human', 'chimpanzee', 'gorilla', 'orang-utan', 'gibbon']
```

Finally, we will show the code that separates the work to do from the tree traversal itself by functions. The traversal function `tree_traversal` do not change for the task. The user has only to redefine the functions `do_prework`, `do_postwork` and `do_leafwork`.

```
    import types

def do_prework(node):
    print "prework:", node

def do_postwork(node):
    print "postwork:", node

def is_leaf(node):
    return type(node) is not types.ListType

def do_leafwork(leaf):
    print "that is a leaf:", leaf

def tree_traversal (node):
    do_prework(node)

    if not is_leaf(node):
        for child in node:
            tree_traversal(child)
    else:
        do_leafwork(node)

    do_postwork(node)
```

```
>>> tree=[[[[’human’, ’chimpanzee’], ’gorilla’], ’orang-utan’], ’gibbon’]
>>> tree_traversal(tree)
prework: [[[[’human’, ’chimpanzee’], ’gorilla’], ’orang-utan’], ’gibbon’]
prework: [[[’human’, ’chimpanzee’], ’gorilla’], ’orang-utan’]
prework: [[’human’, ’chimpanzee’], ’gorilla’]
prework: [’human’, ’chimpanzee’]
prework: human
that is a leaf: human
postwork: human
prework: chimpanzee
that is a leaf: chimpanzee
postwork: chimpanzee
postwork: [’human’, ’chimpanzee’]
prework: gorilla
that is a leaf: gorilla
postwork: gorilla
postwork: [[’human’, ’chimpanzee’], ’gorilla’]
prework: orang-utan
that is a leaf: orang-utan
postwork: orang-utan
postwork: [[[’human’, ’chimpanzee’], ’gorilla’], ’orang-utan’]
prework: gibbon
that is a leaf: gibbon
postwork: gibbon
postwork: [[[[’human’, ’chimpanzee’], ’gorilla’], ’orang-utan’], ’gibbon’]
```

Chapter 14. Exceptions

14.1. General Mechanism

Exceptions are a mechanism to handle errors during the execution of a program. An exception is *raised* whenever an error occurs:

Example 14.1. Filename error

```
>>> f = open(’my_fil’)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IOError: [Errno 2] No such file or directory: ’my_fil’
```

An exception can be *caught* by the code where the error occurred:

```
try:
   f = open(’my_fil’)
except IOError, e:
   print e
```

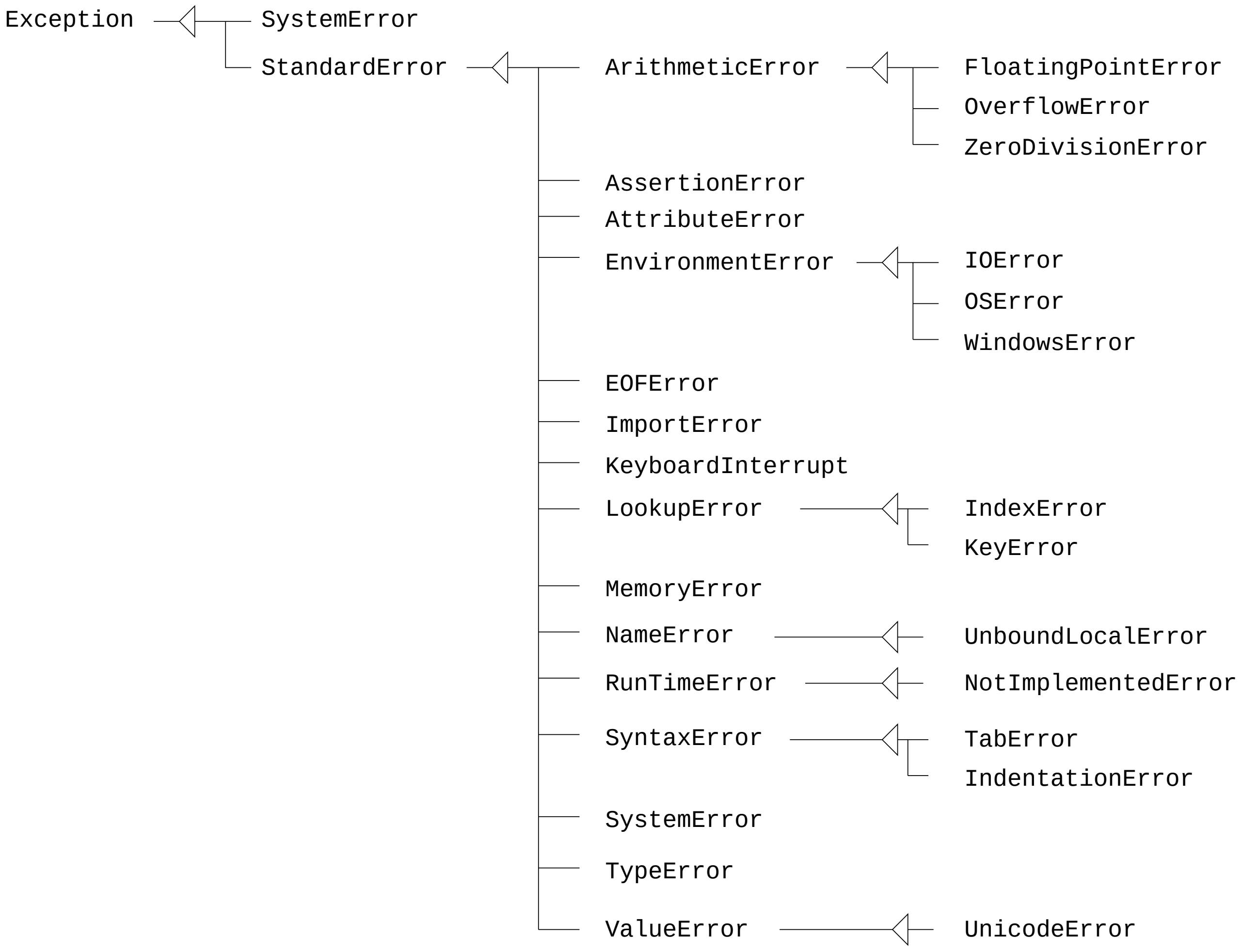
Variable `e` contains the cause of the error:

```
[Errno 2] No such file or directory: ’my_fil’
```

14.2. Python built-in exceptions

Python predefines several exceptions (Figure 14.1).

Figure 14.1. Exceptions class hierarchy



- `AttributeError`: when you attempt to access a non-existing attribute (method or variable) of an object.
- `NameError`: failure to find a global name (module, ...)
- `IndexError`, `KeyError`: occurs when attempting to access either an out-of-range index in a list or a non-existing key in a dictionary
- `TypeError`: passing an inappropiate value to an operation
- `TabError`, `IndentationError`: two kinds of `SyntaxError`

14.3. Raising exceptions

You can also *raise* an exception in your code, if you consider that the program should be interrupted:

```
if something_wrong:
          raise Exception
```

You can associate a message to the `raise` statement:

```
if something_wrong:
          raise Exception, " something went wrong"
```

Example 14.2. Raising an exception in case of a wrong DNA character

```
def check_dna(dna, alphabet='atgc'):
    """ using exceptions """

    for base in dna:
        if base not in alphabet:
            raise ValueError, "%s not in %s" % (base, alphabet)

    return 1
```

14.4. Defining exceptions

Python provides a set of pre-defined exception classes that you can specialize by sub-classing to define specific exceptions for your application (Figure 14.1).

Go to

Since exceptions are defined as classes and by inheritance, you will need some knowledge about classes in order to fully understand this section (see Chapter 17).

Example 14.3. Raising your own exception in case of a wrong DNA character

In the following code, you define an exception `AlphabetError` that can be used when the sequence passed to the function does not correspond to the `alphabet`.

```
class AlphabetError(ValueError):                                          ❶
    pass

def check_dna(dna, alphabet='atgc'):
```

```
""" using exceptions """

for base in dna:
    if base not in alphabet:
        raise AlphabetError, "%s not in %s" % (base, alphabet)

return 1
```

❶ Definition of a new exception in category ValueError: AlphabetError is a class, that is a subclass of class ValueError. The only statement present in class AlphabetError is pass since AlphabetError does not define any new behaviour: it is just a new class name.

Example 14.4. Exceptions defined in Biopython

Some Biopython modules define their own exceptions, such as:

- ParserFailureError (GenBank package)
- BadMatrix (SubsMat package)

Chapter 15. Modules and packages in Python

15.1. Modules

A module is a component providing Python definitions of functions, variables or classes... all corresponding to a given specific thema. All these definitions are contained in a single Python file. Thanks to modules, you can reuse ready-to-use definitions in your own programs. Python also encourages you to build your own modules in a rather simple way.

15.1.1. Using modules

In order to use a module, just use the `import` statement. Let us take an example. Python comes with numerous modules, and a very useful one is the `sys` module (sys stands for "system"): it provides information on the context of the run and the environment of the Python interpreter. For instance, consider the following code:

```
#!/local/bin/python
import sys

print "arguments: ", sys.argv
```

Say that you stored in a modexa.py file, and that you run it like this:

```
./modexa.py 1 a seq.fasta
```

This will produces the following output:

```
arguments:  ['./modexa.py', '1', 'a', 'seq.fasta']
```

Explanation: By using the `argv` variable defined in the `sys` module, you can thus access to the values provided on the command line when launching the program. As shown in this example, the access to this information is made possible by:

- importing the module through the `import` statement, which provides access to the module's definitions
- using the `argv` variable defined in the module by a qualified name: `sys.argv`.

You may also select specific components from the module:

```
       from sys import argv
       print "arguments: ", argv
```

In this case, you only import one definition (the `argv` variable) from the `sys` module. The other definitions are not loaded.

15.1.2. Building modules

You build your own module by creating a Python file. For instance, if the file `ValSeq.py` contains the following code (adapted from the Biopython `ValSeq` module):

Example 15.1. A module

```
# file Valseq.py

valid_sequence_dict = { "P1": "complete protein", \
 "F1": "protein fragment", "DL": "linear DNA", "DC": "circular DNA", \
 "RL": "linear RNA", "RC":"circular RNA", "N3": "transfer RNA", \
 "N1": "other"   }

def find_valid_key(e):
   for key,value in valid_sequence_dict.items():
             if value == e:
                 return key
```

you can use it by loading it:

```
import ValSeq
```

where `ValSeq` is the module name. You can then access to its definitions, which may be variables, functions, classes, etc...:

```
>>> print ValSeq.valid_sequence_dict['RL']
linear RNA
>>> ValSeq.find_valid_key("linear RNA")
       RL
```

15.1.3. Where are the modules?

Modules are mainly stored in files that are searched:

- in your current working directory,
- in `PYTHONHOME`, where Python has been installed,
- in a path, i.e a colon (':') separated list of file paths, stored in the environment variable `PYTHONPATH`. You can check this path through the `sys.path` variable.

Files may be:

- Python files, suffixed by `.py` (when loaded for the first time, compiled version of the file is stored in the corresponding `.pyc` file),
- defined as C extensions,
- built-in modules linked to the Python interpreter.

Exercise 15.1. Locating modules

Sometimes, it is not enough to use **pydoc** or `help`. Looking at the source code can bring a better understanding, *even if you should of course never use undocumented features*.

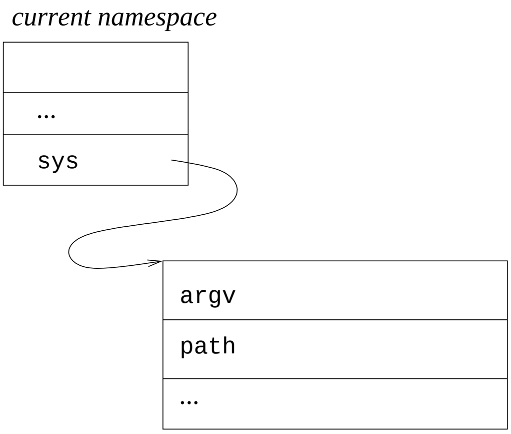
Browse the directory tree `PYTHONHOME`/site-packages/Bio/.

15.1.4. How does it work?

When importing a module, the interpreter creates a new namespace, in which the Python code of the module's file is run. The interpreter also defines a variable (such as `sys`, `ValSeq`, ...) that refers to this new namespace, by which the namespace becomes available to your program (Figure 15.2).

Figure 15.1. Module namespace

```
import sys
```



open

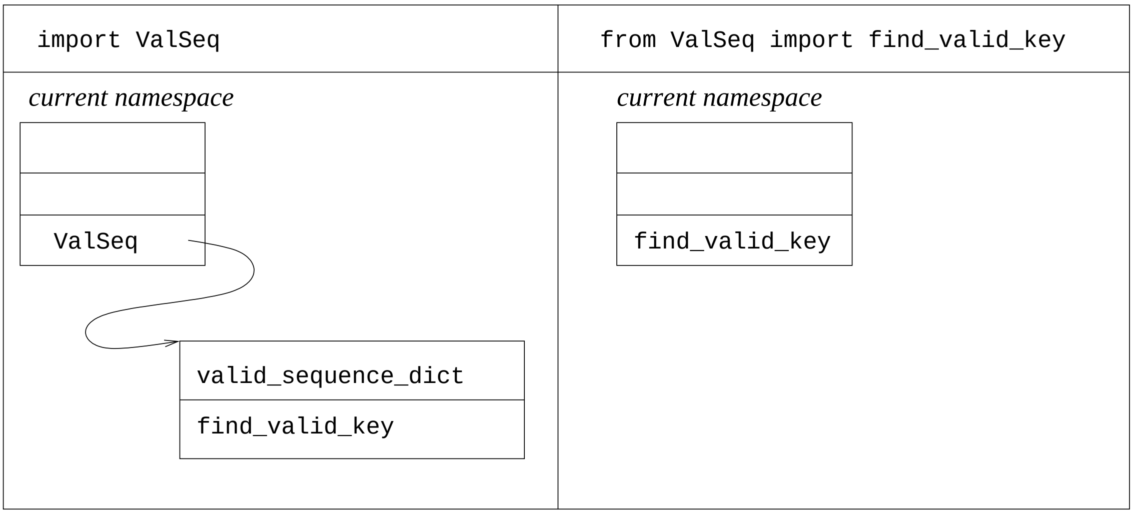
A module is loaded only once, i.e, a second **import** statement will neither re-execute the code inside the module (see Python **reload** statement in the reference guides), nor will it re-create the corresponding namespace.

When selecting specific definitions from a module:

```
>>> from ValSeq import find_valid_key
>>> find_valid_key("linear RNA")
RL
```

the other components stay hidden. As illustrated in Figure 15.2, no new namespace is created, the imported definition is just added in the current name space.

Figure 15.2. Loading specific components



This can causes errors if the definition that is imported needs to access to other definitions of the module, e.g:

```
>>> print valid_sequence_dict[’RL’]
NameError: name ’valid_sequence_dict’ is not defined
>>> print ValSeq.valid_sequence_dict[’RL’]
NameError: name ’ValSeq’ is not defined
```

You can also load "all" the components from a module, which makes them available *directly* into your code:

```
>>> from ValSeq import *
>>> find_valid_key("linear RNA")
```

You probably did this many times in order to use the `string` module’s definitions, right? The result of:

```
>>> from string import *
```

is that all the definitions of the module are copied in your current namespace.

Caution

Be aware of potential names collision: for instance, if you current namespace contains a definition of a variable called, say: `count`, it will be destroyed and overloaded by the `string` module’s definition of the `count` function.

Caution

You can restrict the components being imported by an `import *` statement. The `__all__` variable, also used for packages (Section 15.2), can explicitly list the components that should be directly accessible (see Exercise 15.4).

15.1.5. Running a module from the command line

When the file of a module is run from the command line (instead for being imported):

```
% python ValSeq.py
```

the module does not behaves like a module anymore. It is, instead, run within the default `__main__` module (i.e not the `ValSeq` module):

```
% python -i ValSeq.py
>>> ValSeq.find_valid_key("linear RNA")
NameError: name ’ValSeq’ is not defined
>>> find_valid_key("linear RNA")
’DL’
```

For this reason, the code executed when the module is loaded (e.g: either with import or from the command line) can be made dependent of its current name by testing this name. The current module’s name is stored in a special purpose variable `__name__`:

```
if __name__ == ’__main__’:
   # statements that you want to be executed only when the
   # module is executed from the command line
   # (not when importing the code by an import statement)
          print find_valid_key("linear RNA")
```

15.2. Packages

A package is a set of modules or sub-packages. A package is actually a directory containing either .py files or sub-directories defining other packages.

The dot (.) operator is used to describe a hierarchy of packages and modules. For instance, the module Bio.WWW.ExPASy is located in the file PYTHONHOME/site-packages/Bio/WWW/ExPASy.py. This module belongs to the Bio.WWW package located into the PYTHONHOME/site-packages/Bio/WWW/ directory.

15.2.1. Loading

When loading a package, the __init__.py file is executed. If the __init__.py defines classes, functions, etc... they become available at once, as shown in the following example:

Example 15.2. Using the Bio.Fasta package

```
>>> import Bio.Fasta
>>> handle = open("data/ceru_human.fasta")
>>> it = Bio.Fasta.Iterator(pin, Bio.Fasta.SequenceParser())
>>> seq = it.next()
>>> print seq.seq
>>> it.close()
```

However, loading a package does not automatically load the inner modules. For instance, even though the Bio.Fasta package directory contains the following files:

```
% ls Bio/Fasta
       FastaAlign.py   FastaAlign.pyc  __init__.py     __init__.pyc
```

this does not imply that importing the Bio.Fasta package loads the Bio.Fasta.FastaAlign module:

```
>>> import Bio.Fasta
>>> Bio.Fasta.FastaAlign.parse_file("data/ceru_human.fasta")
AttributeError: 'module' object has no attribute 'FastaAlign'
```

Issuing:

```
>>> from Bio.Fasta import *
```

will however load the Bio.Fasta.FastaAlign, because this module is mentioned in the __all__ attribute in the Bio/Fasta/__init__.py file:

```
    __all__ = [
       'FastaAlign',
    ]
```

Other attributes of interest for packages and modules:

- `__name__`
- `__path__`
- `__file__`

Exercise 15.2. Bio.SwissProt package

Which import statements are necessary to make the following code work?

```
expasy = ExPASy.get_sprot_raw('CERU_HUMAN')
sp = SProt.Iterator(expasy, SProt.RecordParser())
record = sp.next()
print record.keywords
```

???

Exercise 15.3. Using a class from a module

Why does the following code issue an error?

```
from Bio.SubsMat import FreqTable
dict = ...  # whatever
f = FreqTable(dict, 'COUNT')
TypeError: 'module' object is not callable
```

???

Exercise 15.4. Import from Bio.Clustalw

Why does the following code not work?

```
from Bio.Clustalw import *
```

```
     a=ClustalAlignment()
     NameError: name 'ClustalAlignment' is not defined
```

???

15.3. Getting information on available modules and packages

You can use the **help** to get the list of available modules:

```
>>> help("modules")

Please wait a moment while I gather a list of all available modules...
```

To search through the documentation of modules for a specific word, for instance "SProt", also use the **help** command like this:

```
>>> help("modules SProt")

Here is a list of matching modules.  Enter any module name to get more help.

Bio.SwissProt.KeyWList - KeyWList.py
Bio.SwissProt.SProt - SProt.py
Bio.SwissProt (package)
```

The `sys` also contains the dictionary of loaded modules.

Chapter 16. Scripting

16.1. Using the system environment: os and sys modules

There are modules in the Python library that help you to interact with the system.

The `sys` module. The `sys` module provides an interface with the Python interpreter: you can retrieve the `version`, the strings displayed as prompt (by default: ’>>>’ and ’...’), etc... You can find the arguments that were provided on the command line:

```
% python -i prog.py myseq.fasta
>>> import sys
>>> sys.argv
['prog.py', 'myseq.fasta']
```

The file handle for the standard input, output and error are accessible from the `sys` module:

```
>>> sys.stdout.write("a string\n")
a string
>>> sys.stdin.read()
a line
another line
'a line\nanother line\n'                                                    ❶
```

❶ You have to enter a Ctl-D here to end the input.

The `os` module. This module is very helpful to handle files and directories, processus, and also to get environment variables (see `environ` dictionary). One of the most useful component is the `os.path` module, that you use to get informations on files:

```
>>> import os.path
>>> os.path.exists('myseq.fasta')
1
>>> os.path.isfile('myseq.fasta')
1
>>> os.path.isdir('myseq.fasta')
0
>>> os.path.basename('/local/bin/perl')
'perl'
```

Exercise 16.1. Basename of the current working directory

Write the statements to display the `basename` of the current working directory.

The `os.path` module provides a method: `walk` that enables to walk in all the directories from a starting directory and to call a given function on each.

Example 16.1. Walking subdirectories

The following code displays for each directory its name and how many files it contains:

```
>>> def f(arg, dirname, fnames):
...     print dirname, ": ", len(fnames)

>>> os.path.walk('.', f, None)
```

The arguments of function `f` must be: `dirname`, which is the name of the directory, and `fnames` which is a list containing the names of the files and subdirectories in `dirname`. `arg` is a free parameter, that is passed to `walk` (here: None).

Exercise 16.2. Finding files in directories

Find a file of a given name and bigger than a given size in a directory and its sub-directories. Only consider files, not directories.

16.2. Running Programs

You can run external programs from a Python program. There are three major tasks to perform in order to run programs from a script:

- Building the command line.
- Testing for success and checking for potential errors.
- Getting results.

Example 16.2. Running a program (1)

The simplest way to run a program is by using the `system` of the `os` module. The result of the program will be printed on the standard output, which is normally the screen. The return value reports about the success or failure of the execution.

```
import os
cmd="golden swissprot:malk_ecoli"
status = os.system(cmd)
print "Status: ", status
```

Another Python module, `commands`, enables to store the result of the execution in a string:

```
import commands
cmd="golden swissprot:malk_ecoli"
output = commands.getoutput(cmd)
print "Output: ", output
```

To get both result and status:

```
import commands
cmd="golden swissprot:malk_ecoli"
status, output = commands.getstatusoutput(cmd)
print "Output: ", output
print "Status: ", status
```

Example 16.3. Running a program (2)

A more elaborate but lower level interface to run commands is provided by the `popen` function from the `os` module. The following script runs a program that fetches a Swissprot entry given its entry name, and prints it on the screen.

```
import os
import string
cmd="golden swissprot:malk_ecoli"                                    ❶
handle = os.popen(cmd, ’r’)                                          ❷
print string.join(handle.readlines())                                ❸
handle.close()
```

❶ Builds the command line with a program name and the arguments.
❷ Runs the command and stores a handle in the `handle` variable. A handle for a command is the same kind of objects as a file handle: you open it (with the `popen` command, read from it, and close it.
❸ Reads all the lines from the handle, and prints the joint result.

If the program takes time and if you wish to read the result step by step as long as results show up, you can do like this:

```
import os
import sys

cmd="blastall -i " + sys.argv[1] + " -p blastp -d swissprot"
handle = os.popen(cmd, ’r’, 1)
for line in handle:
    print line,
handle.close()
```

What if the entry name does not have a corresponding entry in the database? Let us try the following code:

```
import os
import sys
import string
cmd="golden swissprot:" + sys.argv[1]                              ❶
handle = os.popen(cmd, ’r’)
print string.join(handle.readlines())
status = handle.close()
if status is not None:
    print "An error occured: ", status                             ❷
```

❶ Takes the entry name from the Python command line arguments, by using the `sys` module `argv` variable.

❷ If the provided entry name is invalid, the program returns a non zero value, that is returned by the `close` function.

If you wish to get the complete error message from the program, use the `popen3` function:

```
import os
import sys
cmd="golden swissprot:" + sys.argv[1]
tochild, fromchild, childerror = os.popen3(cmd, ’r’)
err = childerror.readlines()
if len(err) > 0:
     print err
else:
     print fromchild.readlines()
```

In this script, the call returns three objects: one to get results: `fromchild` (*standard output*), one to write to the program - on its *standard input* - (for instance when the program is prompting for a value (see Example 16.4): `tochild`, and one to access to the *standard error* file descriptor of the program.

Example 16.4. Running a program (3)

The next example shows how to run an *interactive* program, both reading the output and writing on the standard input to answer program's questions. The Phylip **dnapars** program, once having read the input file containing an alignment, that is always called `infile`, waits for the user to enter 'y' and 'Return' to proceed. The following script runs dnapars after some file cleaning.

```
import popen2
import os.path

cmd = "dnapars"

if os.path.exists('treefile'):                                    ❶
    os.unlink('treefile')
if os.path.exists('outfile'):
    os.unlink('outfile')

child = popen2.Popen3(cmd)                                        ❷
print "PID: ", child.pid
child.tochild.write("y\n")                                        ❸
child.tochild.close()
child.wait()                                                      ❹
print "".join(child.fromchild.readlines())                        ❺
status = child.fromchild.close()
if status is not None:
    print "status: ", status
```

❶ Removes old **dnapars** output files.
❷ Use of the class `Popen3` that stores the information about the run program. The return value: `child`, is an object representing the "child" processus, that has attributes to get the channels to communicate with the processus: a handle to write to the standard input of the processus (`child.tochild`), and a handle to read its output (`child.fromchild`). See Chapter 17 for more informations on classes.
❸ Answers to program prompt.
❹ This statements helps in cleaning the processus after completion.
❸ Reads results.

16.3. Parsing command line options with getopt

The `getopt` module helps script to parse a complex command line options. Example 16.5 shows an example.

Example 16.5. Getopt example

This example shows a piece of code for handling options in a program called, say, **filteralig**, taking an alignment and filtering some sites according to specific criteria. Options are of the form: '-o1 value1 -o2 value2' where 'o1' and 'o2' are the names of the options and value1, value2 are their argument, which might be optional for some of the parameters. You can use this script this way, for instance:

```
filteralig -t0.7 -f2,3 align_file
```

```
import sys
import getopt

def usage(prog="filtersites"):                                            ❶
    print """
filteralig : filter sites in alignments

 filteralig [-ch] [-t <threshold>] [-f <frames>] [-i <cols>] <alignment>

 -h                 print this message

 -c                 print colum numbers of the original alignment

 -t <threshold>     filter all colums with a conservation above <threshold>
 -f <frames>        filter all codonpositions of frames
                    possible values 1, 2, 3
                    for more than one use syntaxe: ’1,2’
 -i <cols>          filter this columns
                    syntaxe: give a string with the column numbers separated by
 ’,’

 <alignment>        the file has to be in clustalw format

 """

o, a = getopt.getopt(sys.argv[1:], ’ct:f:i:h’)                           ❷
opts = {}
for k,v in o:                                                             ❸
    opts[k] = v
if  opts.has_key(’-h’):                                                   ❹
    usage(); sys.exit(0)
if len(a) < 1:                                                            ❺
    usage(); sys.exit("alignment file missing")
```

❶ A `usage` function is very useful to help the user in case of an error on the command line.

❷ The first parameter for the `getopt` function should be a string containing the actual arguments the script has been called with, not including the script name, available in `sys.argv[0]`.

The second parameter is a string describing the expected options. The options string which is passed to `getopt` is here: ’ct:f:i:h’. This means that the following options are available: c, t, f, i and h. When a ’:’ is added just after, this means that the option expects a value. For instance, the ’-t’ option requires a threshold value. See the `usage`!

The `getopt` function returns tuple, whose first element is a list of (option, value) pairs. The second element is the list of program arguments left after the option list was stripped. Here, a filename for an alignment file is expected.

❸ Storing (option, value) pairs in a dictionary.

❹ If the user has entered a **-h**, help is printed.

❺ Has the user provided a filename ? If so, it is available in `a[0]`.

16.4. Parsing

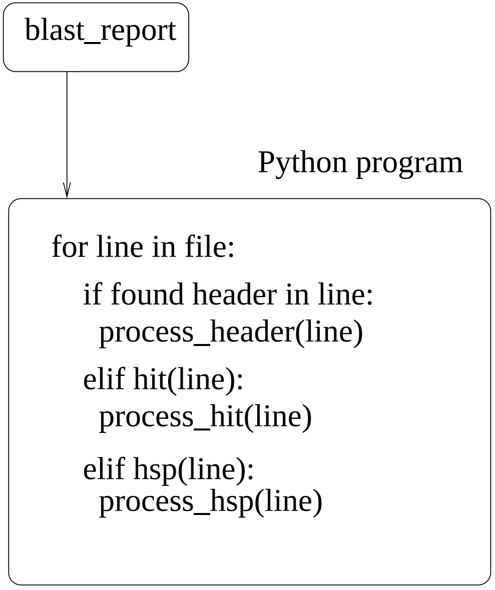
In Bioinformatics, parsing is very important, since it enables to extract informations from data files or to extract results produced by various analysis programs, and to make them available in your programs. For instance, a Blast parser will transform a text output into a list of hits and their alignment, that can be made available as a data structure, such as, for example, Biopython `Bio.Blast.Record` objects, that you can use in a Python program.

The purpose of this section is not to present everything about parsing, but just to introduce some basic notions.

Parsing means analyzing a text and producing structured data in a form that is useful for programs. It can be a list of strings, a set of classes instances, or just a boolean result: this depends on the needs and the parsing system you are using. An important aspect of parsing is the architecture that is used to process the text that you want to analyze.

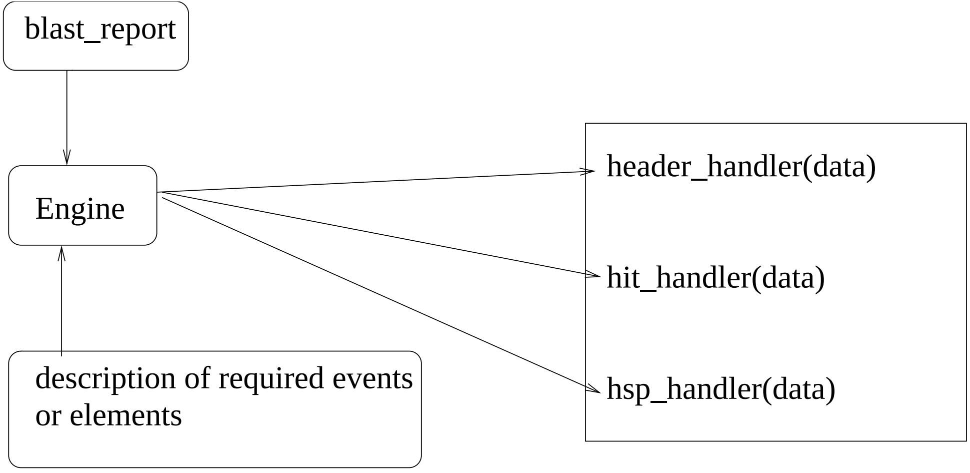
- Parsing can be done by just reading a file or a stream line by line, and by looking for the occurrence(s) of a word, or a pattern. In Figure 16.1, lines are searched for a header pattern, or a hit pattern, and processed accordingly.

Figure 16.1. Manual parsing



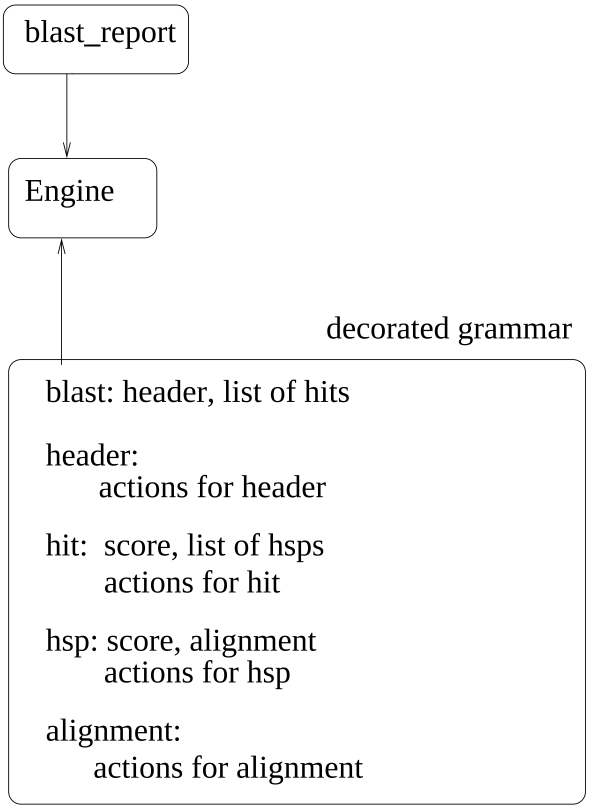
- You can tell an external component which set of words you are interested in, by providing a description of set of words to be found. You then feed this component a document to parse, and wait him to tell you when these words are found as well as sending you the occurrences. This kind of system is said to be *event-driven*, and XML provides tools for such type of parsing.

Figure 16.2. Event-based parsing



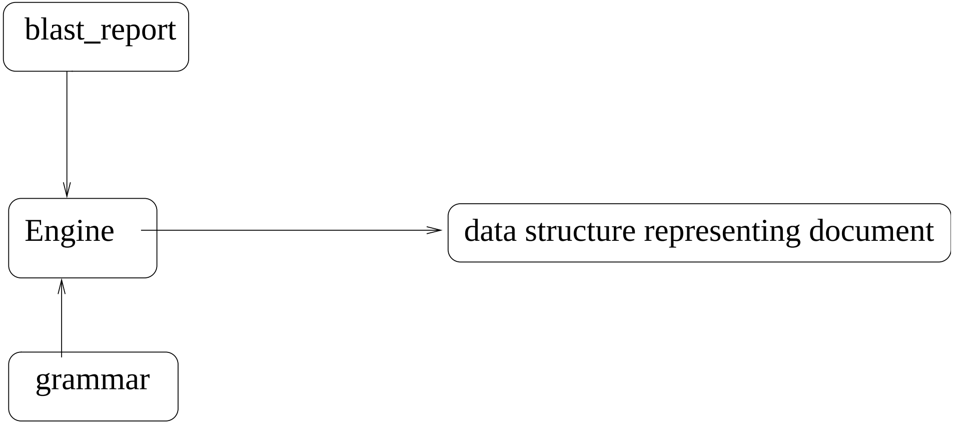
- You can describe the whole document by set of hierarchical subparts, and associate to each sub-part actions to be automatically executed by a parsing engine. Figure 16.3 shows such a system to parse a Blast report. A Blast report is described as a header followed by a list of hits. A hit is described as a score and a list of HSP, and a HSP is described as a score and a list of alignments. You define these subparts by a set of *rules*, sometimes using patterns, in what is usually called a *grammar*, or even a *decorated* grammar, since you decorate each sub-part with an associated action. The `lex` and `yacc` system is such a parsing engine.

Figure 16.3. Parsing: decorated grammar



- You can have a parsing engine process your data according to a grammar, and returns a hierarchical data structure in a form that your program can understand (for instance, in Python, as a set of objects). The XML/DOM engine behaves like this.

Figure 16.4. Parsing result as a hierarchical document



So, in all the cases, there is an engine driving the whole process, be it a simple loop or a specialized component. In this chapter, we will just do some "manual" parsing with patterns that are introduced in Section 16.5, as well as some event-driven parsing will be done as a practical work on abstract frameworks (see Exercise 18.2), and during the Web/XML course.

16.5. Searching for patterns.

16.5.1. Introduction to regular expressions

Regular expression is a term used in computer science to refer to the language that is used to describe patterns to be searched for in a text. The term comes from the computing languages theory where regular expressions are used to denote regular languages. In turn, regular languages are defined as the languages that can be recognized by a finite-state automaton (a topic that will be introduced in the algorithmic course).

The aim of a pattern is to define not only one word to be searched for, but a set of words. This definition is provided in a given language, depending on the system you are working with; the set of corresponding words also depends on the system.

In the Unix shell, for instance:

```
ls s*
```

means list all the files beginning by ’s’, and potentially followed by anything. The command:

```
ls s[ie]n*
```

means list all the files beginning by ’s’, followed by either ’i’ or ’e’, and followed by anything, including nothing. So, in the context of the `ls` function within a Unix shell, the set of words is defined as the files existing on the filesytem (as opposed, for instance, to the files stored on another computer, not being made available through a distributed filesystem).

In the Prosite database, patterns describing protein domains are described, such as:

```
H-C-H-x(3)-H-x(3)-[AG]-[LM]
```

which represent the following set of amino-acid sequences: sequences begining by ’HCH’, followed by 3 positions containing any amino-acid letter, followed by ’H’, followed again by 3 free positions, followed by either ’A’ or ’G’, followed by either ’L’ or ’M’. As you can notice, the language to define patterns is different from the Unix shell, as well as the set of corresponding words. Here, they are just sequences of amino-acids letters.

In the `grep` Unix command, a command to search for patterns in files, although similar to the shell pattern syntax, there is a slight difference. Say that a file contains:

```
science
s
another
```

the following command:

```
grep ’s*’ file
```

will return ... all the lines, because ’s*’ means all the words beginning by 0 or any number of ’s’. In the `grep` command, the set of words is composed of the lines of the file. So the term "set of words" must be understood in a broad sense. It must be understood also that *the set is not actually generated*, of course: it can be infinite! Instead, an operational representation of this set is built, through a finite-state automaton.

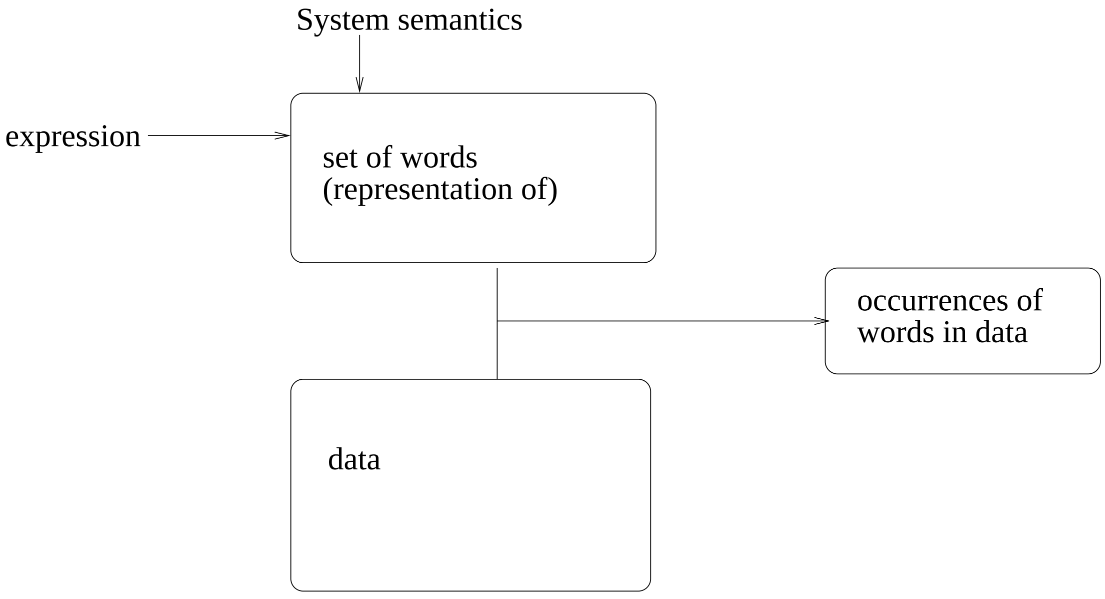
In `SQL` languages, you can also provide patterns:

```
select clone_name from CLONE where clone_id like ’[^A]%[02468]’
```

means that you restrict the query to the clones whose identifier does not begin with a ’A’, is followed by any character N times, and ends with an even number (Sybase).

While the set of words corresponding to a pattern is described by the given expression and the semantics of the system being used, the set of found words, also called *occurrences*, depends on data. So, occurrences are the words from the set of words that were actually found within data. Figure 16.5 summarizes the concepts.

Figure 16.5. Pattern searching



16.5.2. Regular expressions in Python

A detailed presentation of Python regular expressions is available here: Regular Expression HOWTO [http://py-howto.sourceforge.net/regex/regex.html]. To get information about the `re` module, see pydoc, but also the `sre` module (Support for regular expressions), for which `re` is a wrapper.

In Python, regular expressions are handled by a module: `re`:

```
>>> import re
```

Before searching for a pattern, you must first *compile* it (this builds the "set of words", or rather the representation of this set):

```
>>> expression = '[AP]{1,2}D'
>>> pattern = re.compile(expression)
```

`pattern` is a *pattern object*. You then issue a *search*, for instance in the small sequence `seq`, by a request to the pattern object:

```
>>> seq = "RPAD"
>>> match = pattern.search(seq)
```

This establishes the *matching*, or correspondances, between the set of possible words and data. `match` is called a *match object*. To get the occurrences, you can ask the match object for the start and end of the match in the searched text:

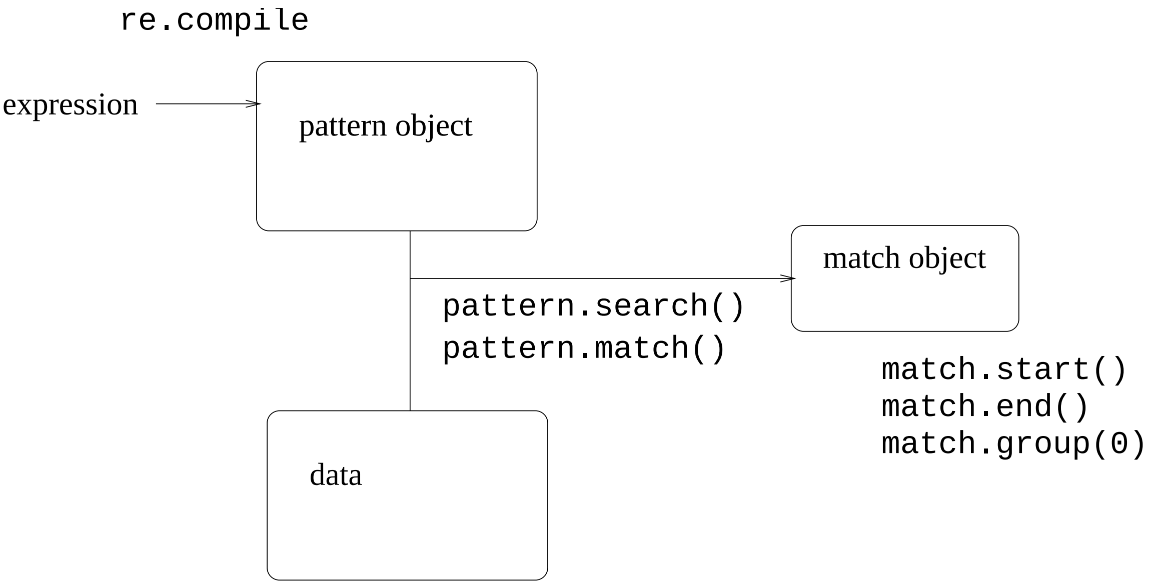
```
>>> print match.start(), match.end(), seq[match.start():match.end()]
1 4 PAD
```

or the `group`:

```
>>> match.group(0)
'PAD'
```

Figure 16.6 summarizes this system.

Figure 16.6. Python regular expressions



Example 16.6. Searching for the occurrence of PS00079 and PS00080 Prosite patterns in the Human Ferroxidase protein

```
import sys
import re
from Bio.SwissProt import SProt

sp = open(sys.argv[1])
iterator = SProt.Iterator(sp, SProt.SequenceParser())
seq = iterator.next().seq
sp.close()

PS00079 = 'G.[FYW].[LIVMFYW].[CST].{8,8}G[LM]...[LIVMFYW]'              ❶
pattern = re.compile(PS00079)                                          ❷
match = pattern.search(seq.tostring())                                 ❸
print PS00079
print match.start(), match.end(), seq[match.start():match.end()]       ❹
```

❶ The regular expression is stored in a string.
❷ The regular expression is compiled in a pattern.
❸ The compiled pattern is searched in the sequence.
❹ The result of the search is printed.

There are several methods to search: `search` and `match`, the difference being that `match` looks for a match *at the beginning of the string*. So, back to the example, the following statement:

```
match = pattern.match(seq.tostring())
```

would return a positive result only if the sequence begins by an occurrence of PS00079.

A convenient feature enables to associate a name to sub-parts of the matched text:

```
import sys
import re
from Bio.SwissProt import SProt

sp = open(sys.argv[1])
iterator = SProt.Iterator(sp, SProt.SequenceParser())
seq = iterator.next().seq
sp.close()

PS00080 = '(?P<copper3>H)CH...H...[AG](?P<copper1>[LM])'                ❶
pattern = re.compile(PS00080)
match = pattern.search(seq.tostring())
print PS00080
print match.start(), match.end(), seq[match.start():match.end()]

print 'copper type 3 binding residue: ', match.group('copper3')         ❷
print 'copper type 1 binding residue: ', match.group('copper1')
```

❶ The regular expression now contains 2 identifiers: `copper1` and `copper3`.
❷ You can print the sub-parts of the result identified by variables: `copper1` and `copper3`.

Shortcuts. The `re` module provides shortcuts to directly search for an expression without compiling the pattern into a pattern object:

```
>>> match = re.search('[AP]{1,2}D', "RPAD")
>>> match.group(0)
'PAD'
```

You can also directly get the occurrences of a pattern object in a string:

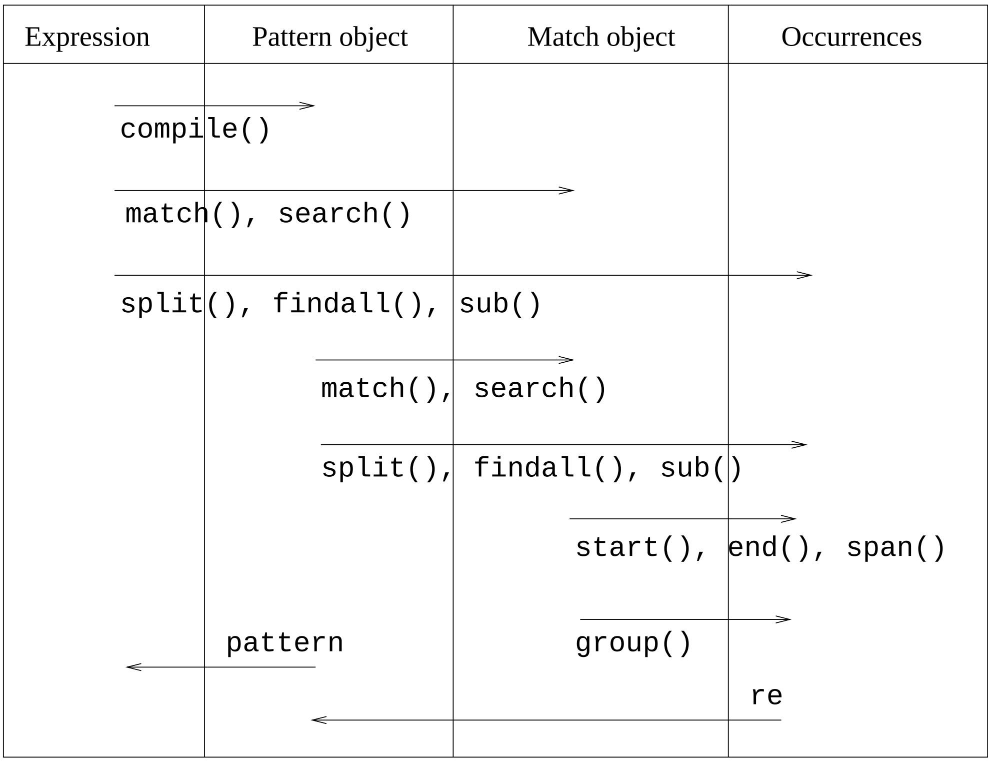
```
>>> pattern = re.compile('[AP]{1,2}D', )
>>> pattern.findall("RPAD")
['PAD']
```

or even the occurences of an expression, without compiling the pattern:

```
>>> re.findall('[AP]{1,2}D', "RPAD")
['PAD']
```

Figure 16.7 summarizes `re` module objects and methods to perform pattern searching.

Figure 16.7. Python regular expressions: classes and methods summary



Search modes.

Text substitutions.

16.5.3. Prosite

This section presents the Prosite classes in Biopython, which have a common interface with the Python pattern module.

16.5.3.1. Prosite Dictionary

Biopython defines several dictionaries to access biological databases. Having a dictionary means that you can fetch an entry by:

```
entry = prosite['PS00079']
```

For this to work, you first need to create the dictionary:

```
prosite = Bio.Prosite.ExPASyDictionary()
```

As you can guess by the name of the module, you actually fetch the Prosite entry on the Web. You could also fetch the Prosite entry from a local database with the golden program (see ???). The entry fetched above is actually a string. In order to have the dictionary return a record, you must rather create it like this:

```
prosite = Bio.Prosite.ExPASyDictionary(parser=Bio.Prosite.RecordParser())
```

16.5.3.2. Prosite patterns

The `Bio.Prosite` package defines a `Pattern` class that enables to create patterns which may be searched for in sequences objects, as in the `re` Python module for regular expressions. The result of a search is a `PrositeMatch`, that behaves in a way similar to a regular expression match.

16.5.4. Searching for patterns and parsing

As a conclusion, let us summarize how pattern searching and parsing interact when analyzing text. Basic text analysis is performed by searching for patterns, that are extracted (or scanned) from a text. Then patterns occurrences may be analyzed (or parsed) as a more general structure. In more complex parsing architectures, as the ones that have been introduced in Section 16.4, it is the *parser* engine which drives the process, by asking a *scanner* to feed him with basic token found in the text. In such systems, regular expressions are defined and used in the scanner.

Chapter 17. Object-oriented programming

17.1. Introduction

This chapter introduces objects, that are a way to organize code and data, and classes, that are a mechanism to describe *new kinds of objects*. We start by the description of what objects, classes and methods are, based on an example (Section 17.2). We then show its class definition (Section 17.3). We then explain how to combine objects in order to build an application (Section 17.4). Finally, technical aspects of classes and objects in Python are presented (Section 17.5).

In the next chapter (Chapter 18), concepts related to object-oriented design will be developped.

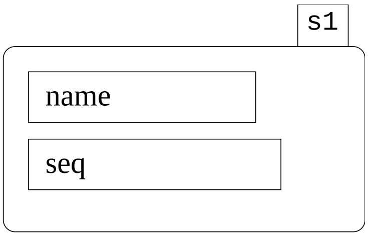
17.2. What are objects and classes? An example

Our example is a class that handles DNA sequences.

17.2.1. Objects description

Let us first start by describing our `DNA` object. Figure 17.1 shows an internal and schematic representation of such an object named `s1`. The object is actually represented in this diagram as a box enclosing *attributes*: the name of the sequence (for instance 'seq1') is stored in a `name` attribute, and the sequence characters (e.g 'aaacaacttcgtaagtata') is stored in a `seq` attribute. Attributes are the same as variables, except that they are associated to an object.

Figure 17.1. A DNA object



17.2.2. Methods

As it is described so far, the `DNA` object is just a kind of record to group data. There are other ways of grouping data in Python: you can either use a list, or a dictionary:

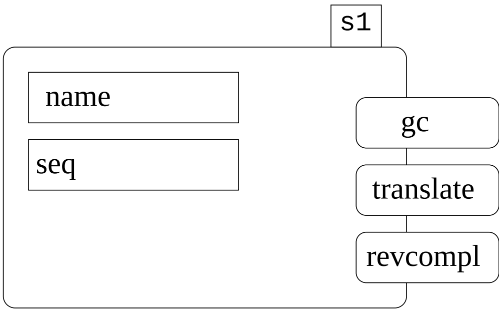
```
>>> seq_record = {'name': 'seq1', 'seq': 'aaacaacttcgtaagtata'}
>>> seq_record['name']
'seq1'
```

but an object is actually more than just that. As shown in Figure 17.2 (and as stated in Section 6.1), an object is indeed more than just a value or a record of values: it can "do" things, and it "knows" what it can do, or. In other words, it has an associated behaviour. This behaviour is described by functions that are associated to it, and these functions are called *methods*. For instance, our `DNA` object can perform operations: it can translate its sequence or it can compute its GC percent by calling the `gc` function:

```
>>> s1.gc()
0.26
```

So, available operations are an important aspect of objects, that we need to add in our representation. In Figure 17.2, methods are represented as "counters", as at the post office.

Figure 17.2. Representation showing object's methods as counters



So, with objects, not only do we have a convenient way to group data that are related, but we also have a way to group data *and their related operations* in the same place.

Object

An object is a construction to group values and operations on these values.

17.2.3. Classes

Now, how do you define objects, such as our `DNA` object, with their attributes and methods?

In object-oriented programming, you define objects by defining a class for them.

Class

A class is an *object maker*: it contains all the statements needed to create an object, its attributes, as well as the statements to describe the operations that the object will be able to perform.

The term *class* can be somewhat misleading, for it also refers to something belonging to a *classification*. In object-oriented programming, a class can indeed also be a member of a set of classes organized in a hierarchy. This aspect will be introduced later (see Section 18.4), through the concept of inheritance. But for now, we will use the term *class* as just a maker of objects, or an object *factory*.

Having a class defined, you can create as many objects as needed. These objects are called *instances* of this class. All the instances created with a given class will have the same structure and behaviour. They will only differ regarding their state, i.e regarding the value of their attributes. Namely, names and sequences will not be the same for all sequences objects that will be created.

Instance

Instances are objects created with a class.

The behaviour of the instances, i.e, the operations that are available for them either to modify them or to ask them for services, is described by functions associated to the class. These functions are called *methods*.

Method

A method is a *function* defined for a class, specifying the behaviour of the class instances.

> ### Important
>
> Classes and instances are not the same: the class is the maker, whereas the instance is the object being made, according to the model defined by the class.

17.2.4. Creating objects

How are we going to create `DNA` objects? You have actually already created objects throughout this course: strings, lists, etc... However, most of the time the objects you have manipulated were not directly created by your own code, but rather by other components: for instance the “[]” operator creates a list, the “""” operator creates a string, the `open` function creates a file handle, etc...

But the actual direct syntax to create objects or to *instantiate* a class, i.e to create an instance of class, is by calling a function with the same name as the class. So, in order to create an instance of the `DNA` class we have just defined, i.e an object which handles a DNA sequence, you do:

```
>>> s1 = DNA()
```

After the `DNA()` function is called, the instance of the class `DNA` is referred to by the `s1` variable, just like with any Python value. You could also provide more information at instantiation. Let’s say that we can create the `name` and `seq` attributes, we can do :

```
>>> s2 = DNA('seq2', 'acaagatgccattgtcccccggcctcctgctgctgctgctctccggggcca')
```

This time, the new `DNA` instance has been created with values for `name` and `seq` attributes. You can also use keyword arguments (especially if you are no longer sure in what order they should be entered) :

```
>>> s2 = DNA(name=’seq2’, seq=’acaagatgccattgtcccccggcctcctgctgctgctgctctccggggcca’)
```

So, this creates the object and makes it available from the `s2` variable. You can now use this object to call `DNA` class methods, by using the dot operator:

```
>>> s2.gc()
0.66
```

The interpretor can find which class `s2` belongs to, and calls the `gc` method that is defined for the `DNA` class. You can also actually access to object’s attributes by using the dot operator:

```
>>> s2.name
’seq2’
>>> s2.seq
’acaagatgccattgtcccccggcctcctgctgctgctgctctccggggcca’
```

17.3. Defining classes in Python

So, how do we actually define classes? For instance, let us describe the `DNA` class. The program listed in Example 17.1 defines a `DNA` class as defined in Figure 17.2. The definition of the class is composed of two main parts: a header, providing the name of the class, and a body, that is composed of a list of definitions, mainly method definitions, but sometimes also assignments (see Section 17.5.1).

Example 17.1. DNA, a class for DNA sequences

```
                                                                      ❶
class DNA:

    def __init__(self, name=None, seq=None):                         ❷
        self.name = name
        self.seq = lower(seq)                                         ❸

    def gc(self):
        count_c = self.seq.count(’c’)                                 ❹
        count_g = self.seq.count(’g’)
        return float(count_c + count_g) / len(self.seq)

    def revcompl(self):
        revseq = ”
        for c in self.seq:
            revseq = c + revseq                                           ❺

        revcompseq = ”
        for base in revseq:
            if base == ’a’:
                revcompseq += ’t’
            elif base == ’t’:
                revcompseq += ’a’
            elif base == ’c’:
                revcompseq += ’g’
            elif base == ’g’:
                revcompseq += ’c’

        return revcompseq

    def translate(self, frame=0):                                         ❻
        """
        frame: 0, 1, 2, -1, -2, -3
        """
        if frame < 0 :
            seq = self.revcompl()
            frame = abs(frame) - 1
        else:
            seq = self.seq

        if frame > 2:
            return ”

        protseq = ”

        for i in range(frame,len(seq) - 2,3):
            codon = seq[i:i+3]
            protseq += Standard_Genetic_Code[codon]                       ❼

        return protseq
```

❶ This statement declares and creates `DNA` as a class.
❷ The `__init__` method is automatically called at instance creation (see below).
❸ Initialization of instances attributes (`name` and `seq`).
❹ This method defines how to compute the GC percent of the sequence.
❺ This method defines how to compute the reverse complement of the sequence.
❻ This method defines how to translate the DNA sequence into a protein sequence.
❼ The `Standard_Genetic_Code` dictionary is defined elsewhere.

The `self` parameter represents the object itself. You could of course use any other word like `carrot` or `ego`, but this would not help the reading of your code by others... So `self` is present in the Class and methods definitions each time the reference to the object instance is needed.

Let us first look at one of these methods definitions, the `gc` method, which computes the GC percent of the sequence:

```
   def gc(self):
     count_c = self.seq.count('c')
     count_g = self.seq.count('g')
     return float(count_c + count_g) / len(self.seq)
```

Method definitions follow exactly the same pattern as standard function definitions, except that they must have declared a first parameter (here: `self`) *for referring to the instance*. Indeed, an instance identificator, actually a reference to the instance, is required in order for the statements within methods to access to the current instance attributes: here, the access to the `seq` attribute is needed in order perform the count. In fact, Python *automatically passes the instance reference as the first argument of a method*. Hence, it is associated to the first parameter which is the `self` parameter. You don't need to specify the argument for the reference to the instance. This is automatically done by Python. In fact, calling:

```
>>> s2.gc()
0.66
```

is equivalent to:

```
>>> DNA.gc(s2)
0.66
```

The interpretor can find which class `s2` belongs to, and handles the passing of the instance reference argument automatically.

How does the method computes its result? For this, it needs to access to the character sequence of the DNA object. This is done by using the `seq` attribute, that was defined at instantiation (i.e by the `__init__` method, see below). Within the method, the attribute is available through the object by the dot operator: `self.seq`. This shows that the object attributes are always available, at least as long as the object itself still exists. Attributes are thus accessible from all the methods of the class. They are a way to share data among methods. The method also use local variables: `count_c` and `count_g` to compute intermediate results. These variables have a local scope, restricted to the method's namespace, exactly like local variables that are defined in functions.

Let us now look at the `translate` method.

```
  def translate(self, frame=0):
     """
     frame: 0, 1, 2, -1, -2, -3
     """
     if frame < 0 :
         seq = self.revcompl()
         frame = abs(frame) - 1
     else:
         seq = self.seq

     if frame > 2:
         return ”

     protseq = ”
     nb_codons = len(seq)/3

     for i in range(frame,len(seq) - 2,3):
         codon = seq[i:i+3]
         protseq += Standard_Genetic_Code[codon]

     return protseq
```

To call this method, you must provide the frame:

```
>>> s2.translate(0)
TRCHCPPASCCCCSPGP
```

Indeed, this method has declared a `frame` parameter. It thus takes two arguments: `self` and `frame`. As for the `gc` method, the first parameter does not have to be specified at calling time, only the remaining ones.

Let us now finally look at the `__init__` method.

```
    def __init__(self, name=None, seq=None):
        self.name = name
        self.seq = lower(seq)
```

This is a special method, which, when defined, is called at class instantiation, e.g when you run the following statement:

```
>>> s2 = DNA(name=’seq2’, seq=’acaagatgccattgtcccccggcctcctgctgctgctgctctccggggcca’)
```

the `__init__` method defined for the `DNA` class is in fact called with 3 arguments. As for the other methods, the `self` argument is automatically provided by Python as a reference to the newly created instance. You don’t have to provide an `__init__` method, but it is usually the good place to put initialization statements. Initial values for attributes can be passed as arguments and associated to attributes. A good practice is to assign them default values, such as None. You can also notice that the `seq` attribute is initialized with the `lower` string function: the other methods will thus not have to check for this in order to perform their computation.

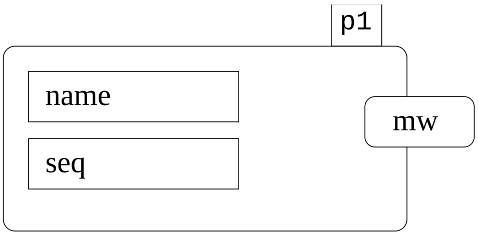
17.4. Combining objects

A program is generally not built on a single object, but rather on a combination of several objects that interact together. In the `DNA` example, we can have several other kinds of objects: protein sequences, motifs, ... The protein object could be either created from an initial sequence of protein amino-acids, or computed by the

DNA translate method. Protein objects could have specific methods for protein sequences, such as hydrophobicity or molecular weight. These objects will be instances of the class Protein.

First of all, let us look at how we could represent a simplified Protein object. Figure Figure 17.3 shows an object having one method: mw, and 2 attributes: name and seq.

Figure 17.3. A Protein object.



The definition of the Protein class follows:

```
class Protein:

    weight = {"A":71.08,"C":103.14 ,"D":115.09 ,"E":129.12 ,"F":147.18 ,"G":57.06 ,"H":137.15 ,"I":11


    default_prosite_file = ’prosite.dat’

    def __init__(self, name=None, seq=None):
        self.name = name
        self.seq = upper(seq)

    def mw(self):
        molW = 0
        for aa in self.seq:
            molW += Protein.weight[aa]

        #add water at the end of protein
        molW += 18.02
        #convert in Kda
        molW = molW / 1000

        return molW
```

You can notice that the class starts by the definition of weight and default_prosite_file variables. As we will see later, this *class variable* is available to all instances of the class.

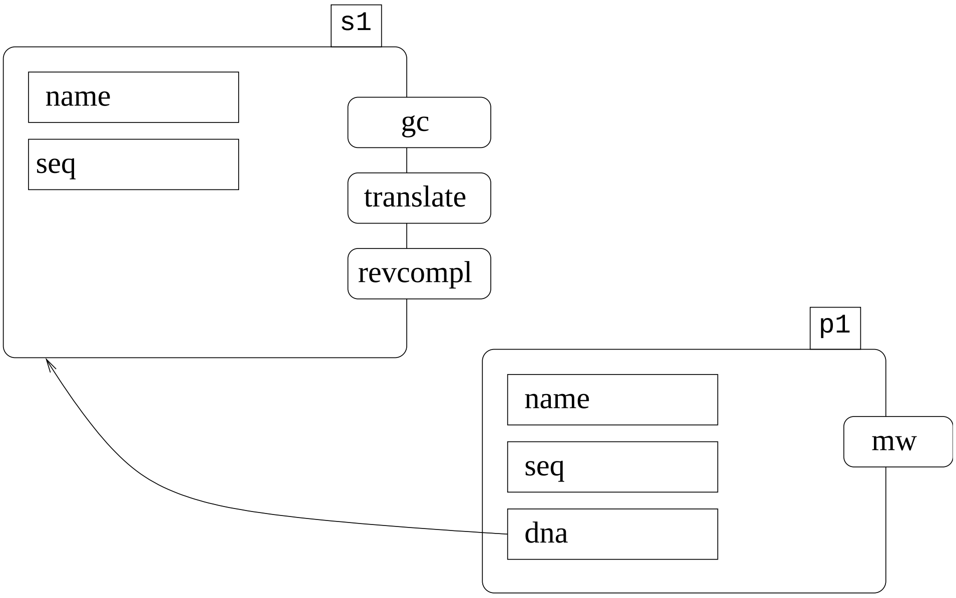
Now, the DNA objects knows how to be translated, right? So it would be more clever for the DNA class translate method to return a Protein object... The new definition of the translate method is:

```
     def translate(self, frame=0):
        """
        frame: 0, 1, 2, -1, -2, -3
        """
        if frame < 0 :
            seq = self.revcompl()
            frame = abs(frame) - 1
        else:
            seq = self.seq

        if frame > 2:
            return ”

        protseq = ”
        nb_codons = len(seq)/3

        for i in range(frame,len(seq) - 2,3):
            codon = seq[i:i+3]
            protseq += Standard_Genetic_Code[codon]

        new_protein = Protein(name=self.name + " translation", seq=protseq)

 return new_protein
```

Look at the returned value: it is now a `Protein` object. The argument for the `seq` parameter of the `Protein` class's `__init__` method is the value of the newly computed `protseq`, and the argument for the `name` is constructed from the `Protein` object's name.

In the `Protein` object, we might also be interested in keeping the reference to the initial `DNA` object. This can help to analyze the protein sequence later. Figure 17.4 shows the `DNA` and `Protein` objects, and the link between them. Now, the `Protein` object has 3 attributes: `name`, `seq` and `dna`.

Figure 17.4. Protein and DNA objects.



The `__init__` method of the `Protein` class is now:

```
    def __init__(self, name=None, seq=None, dna=None):
        self.name = name
        self.seq = upper(seq)
        self.dna = dna
```

The final code for `Protein` object instantiation in the `translate` method is now:

```
        new_protein = Protein(name=self.name + " translation",  seq=protseq, dna=self)

        return new_protein
```

Look at the value provided for the `dna` parameter of the `Protein` `__init__` method. It is a reference to the `DNA` object, i.e: `self`. Notice that none of the parameters is mandatory (except `self` of course). In particular, the `dna` parameter does not have to be provided when the `Protein` is directly created from a file, as opposed to translated from a `DNA` object.

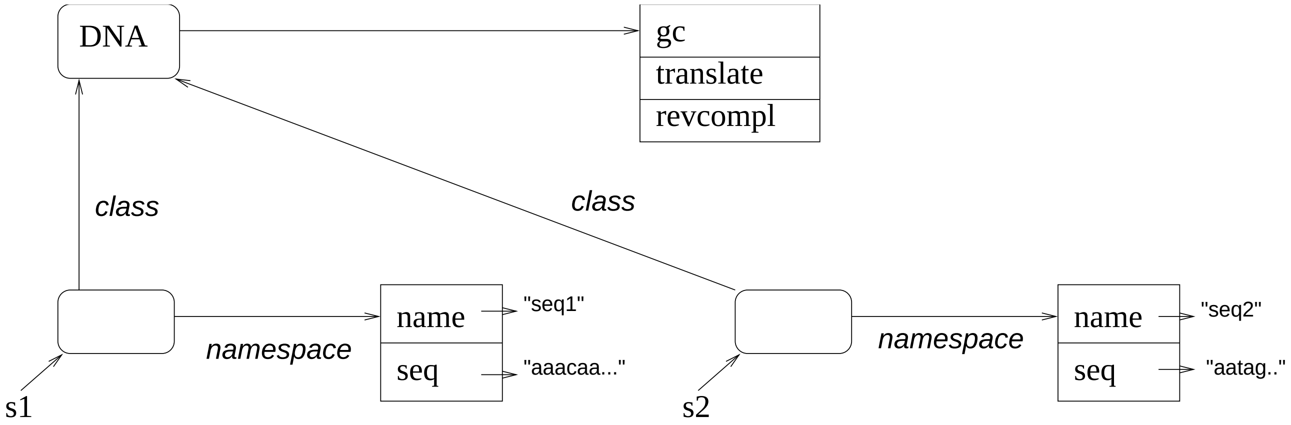
17.5. Classes and objects in Python: technical aspects

The aim of this section is to clarify technical aspects of classes and objects in Python.

17.5.1. Namespaces

Classes and instances have their own namespaces, that is accessible with the dot ('.') operator. As illustrated by Figure 17.5, these namespaces are implemented by dictionaries, one for each instance, and one for the class (see also [Martelli2002]).

Figure 17.5. Classes and instances namespaces.



Instances attributes. As we have learnt, a class may define attributes for its instances. For example, attributes of `s1`, such as the `name`, are directly available through the dot operator:

```
>>> s1.name
'seq1'
```

The dictionary for the instance attributes is also accessible by its `__dict__` variable, or the `vars()` function:

```
>>> s1.__dict__
{'seq: 'aaacaacttcgtaagtata', 'name': 'seq1'}
>>> vars(s1)
{'seq': 'aaacaacttcgtaagtata', 'name': 'seq1'}
```

The `dir()` command lists more attributes:

```
>>> dir(s1)
['__doc__', '__init__', '__module__', 'gc', 'translate', 'name', 'seq']
```

because it is not limited to the dictionary of the instance. It actually also displays its class attributes, and recursively the attributes of its class base classes (see Section 18.4). You can add attributes to an instance that were not defined by the class, such as the `annotation` in the following:

```
>>> s1.annotation = 'an annotation'
>>> s1.__dict__
{’seq: ’aaacaacttcgtaagtata’, ’name’: ’seq1’, annotation: ’an annotation’}
```

Adding attributes on-the-fly is not something that is available in many object-oriented programming languages! Be aware that this type of programming should be used carefully, since by doing this, you start to have instances that have different behaviour, at least if you consider that the list of attributes defines a behaviour. This is not the same as having a different *state* by having different values for the same attribute. But this matter is probably a topic of discussion.

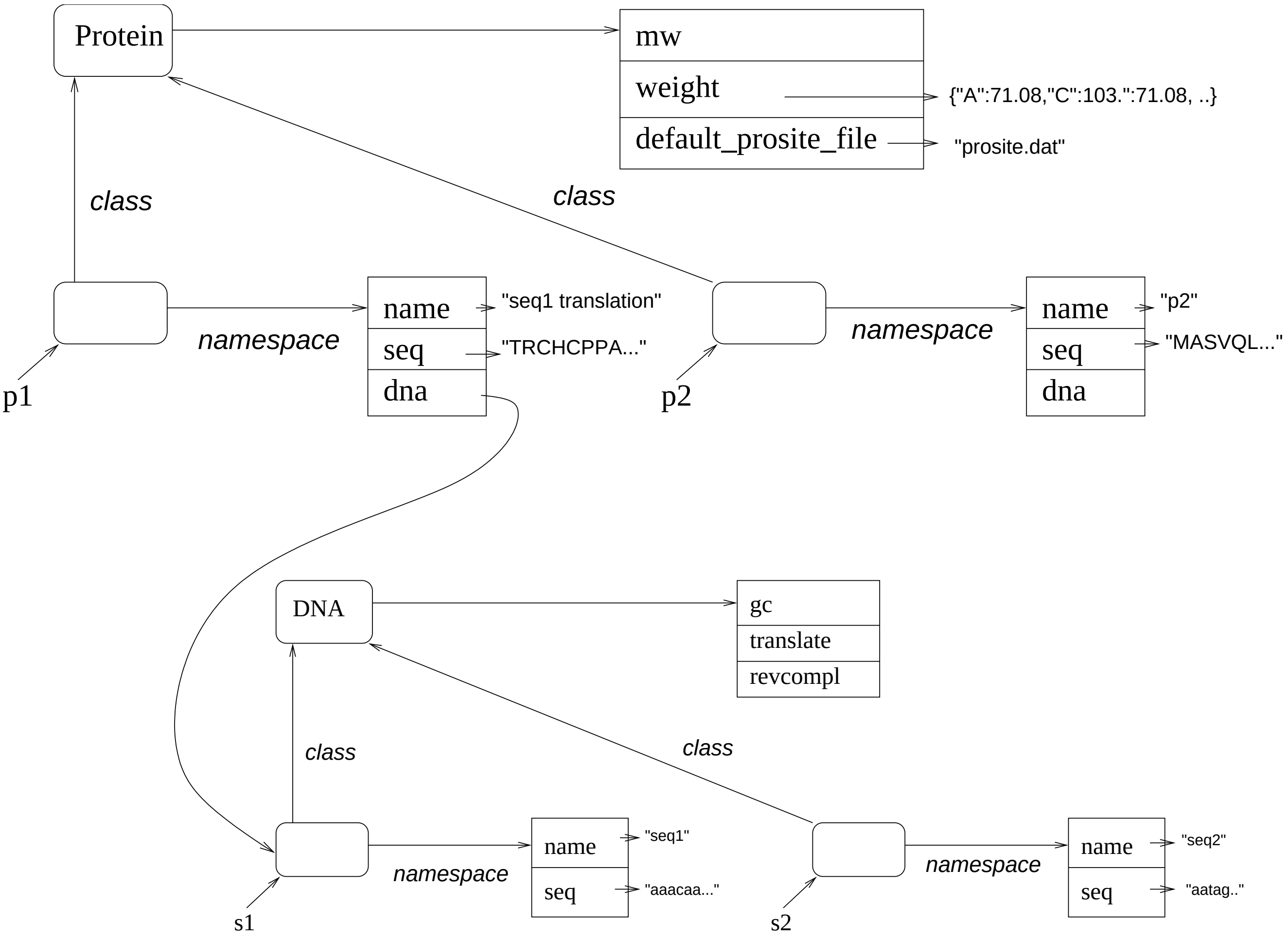
Class attributes. It is also possible to define attributes *at the class level*. These attributes will be shared by all the instances (Figure 17.6). You define such attributes in the class body part, usually at the top, for legibility:

```
class Protein:
  ...
  weight = {"A":71.08,"C":103.14 ,"D":115.09 ,"E":129.12 ,"F":147.18 ,"G":57.06 ,"H":1

  default_prosite_file = ’prosite.dat’
  ...
```

To access this attribute, you use the dot notation:

```
>>> Protein.default_prosite_file
’prosite.dat’
>>> Protein.weight
{"A":71.08,"C":103.14 ,"D":115.09 ,"E":129.12 ,"F":147.18 ,"G":57.06 ,"H":137.15 ,"I":113.17 ,"K":128
```

Figure 17.6. Class attributes in class dictionary



You can also access to this attribute through an instance:

```
>>> p1.default_prosite_file
'prosite.dat'
```

You *cannot* change the attribute through the instance, though:

```
>>> p1.default_prosite_file = 'myfile.dat'                     ❶
>>> Protein.default_prosite_file
'prosite.dat'
```

❶ This just creates a new `default_prosite_file` attribute for the `p1` instance, which masks the class attribute, by the way.

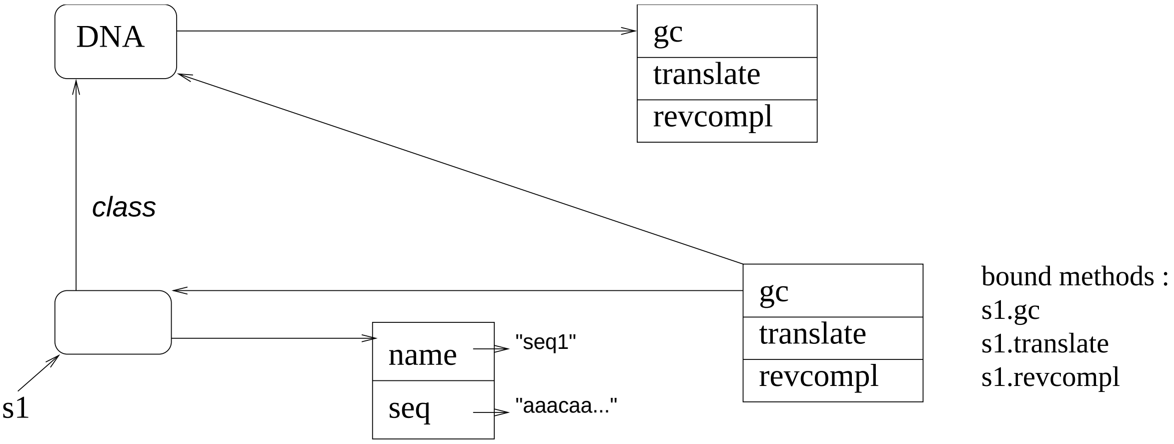
The class attributes are displayed by the `pydoc` command, as opposed to the instance attributes (see Section 17.5.5).

Class methods are referenced in the class dictionary: but what is their *value* actually? As shown in Figure 17.6, the class dictionary entries for methods are pointing to standard Python functions. When accessing to a method through an instance name, such as in `p1.mw`, there is an intermediate data structure, that itself points to the class and the instance. Objects in this data structure are called *bound methods*:

```
>>> s1.gc
<bound method DNA.gc> of <__main__.DNA instance at 0x4016a56c>
```

They are said to be bound, because they are bound to a particular instance, and know about it. This is really important, for it is the only way to know what to do, and on which object to operate.

Figure 17.7. Classes methods and bound methods



17.5.2. Objects lifespan

Once it is created, an object’s lifespan depends on the fact that there are references on it. Namely, as opposed to variables present within functions, an object can still exist after exiting the function or method where it has been created, as long as there is a valid reference to it, as shown in the following example:

```
class C1: pass

class C2:
    def show(self):
        print "I am an instance of class ", self.__class__

def create_C2_ref_in(p):
    p.c2 = C2()                                             ❶
```

```
c1 = C1()
create_C2_ref_in(c1)                                                    ❷
c1.c2.show()                                                            ❸
```

❶ This function creates an instance of class `C2` and stores its reference in an attribute of `p`, an instance of class `C1`.
❷ Creation of the `C2` instance by calling `create_C2_ref_in`
❸ This statement displays: "I am an instance of class __main__.C2"

As you can observe, the `C2` instance exists after exiting the `create_C2_ref_in` function. It will exist as long as its reference remains in the `c1.c2` attribute. If you issue:

```
c1.c2 = None
```

There will be no reference left to our `C2` instance, and it will be automatically deleted. The same would happen if you would issue an additional call to the `create_C2_ref_in` function:

```
create_C2_ref_in(c1)
```

it would overwrite the preceeding reference to the former `C2` instance, and delete it. You can check this by asking the `c1.c2` reference for its identity:

```
id(c1.c2)
```

Of course, another way to delete an object is to use the `del` function:

```
del c1.c2
```

17.5.3. Objects equality

Instances equality cannot be tested by the `==` operator.If we come back to our DNA class:

```
>>> a = DNA('seq1', 'acaagatgccattgtc')
>>> b = DNA('seq1', 'acaagatgccattgtc')
>>> a.__dict__
{'name': 'seq1', 'seq': 'acaagatgccattgtc'}
>>> b.__dict__
{'name': 'seq1', 'seq': 'acaagatgccattgtc'}
>>> a == b
False
>>> a.__dict__ == b.__dict__
```

```
True
```

This means that the equality operator must be defined by the programmer. We will see the `__eq__` special method later in Section 18.3.3.

Instances identity means that two objects are in fact *the same* object, or more exactly, that two variables refer to the same object.

```
>>> a = DNA('seq1', 'acaagatgccattgtc')
>>> b = a
>>> b == a
True
>>> b is a
True
```

As for all Python objects, identity implies equality.

17.5.4. Classes and types

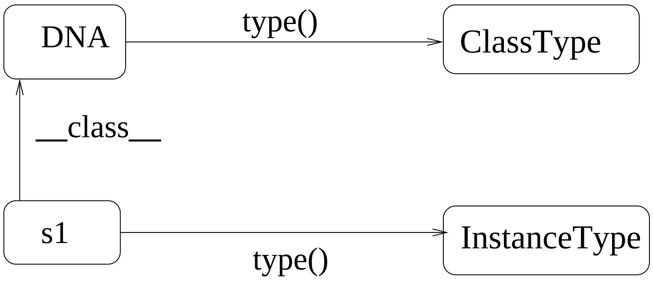
Types in Python include integer,floating-point numbers, strings, lists,dictionaries, etc... Basically, types and classes are very similar. There is a general difference between them, however, lying in the fact that there are literals for built-in types, such as:

```
34
'a nice string'
0.006
[7, 'a', 45]
```

whereas there is no literal for a class. The reason for this difference between types and classes is that you can define a predicate for recognizing expressions of a type [Wegner89], while, with class, you cannot, you can only define collections of objects after a template.

As shown in Figure 17.8, the Python `type()` can be used to know whether a variable is a class or an instance. It will very basically answer `ClassType` or `InstanceType`, as defined in module `types`, but it will not tell you which class an instance belongs to.

Figure 17.8. Types of classes and objects.



17.5.5. Getting information on classes and instances

It is important to know what classes are available, what methods are defined for a class, and what arguments can be passed to them. First, classes are generally defined in modules, and the modules you want to use should have some documentation to explain how to use them. Then, you have the **pydoc** command that lists the methods of the class, and describes their parameters. The following command displays information on the `DNA` class, provided it is in the `sequence.py` file:

```
   pydoc sequence.DNA
```

See also the embedding module, which might bring additional documentation about related components. This may be important when several classes work together, as is described in Section 17.4.

When you consult the documentation of a class with the **pydoc** command, you get most of the time a strange list of method names, such as `__str__` or `__getitem__`. These methods are special methods to redefine operators, and will be explained in the next chapter on object-oriented design (Section 18.3.3).

Caution: the defined instances attributes will not be listed by pydoc, since they belong to the instances rather than to the class. That is why they should be described in the documentation string of the class. If they are not, which sometimes happens..., run the Python interpretor and create an instance, then ask for its dictionary or use the **dir()** command:

```
>>> s1 = DNA()
>>> dir(s1)
['__doc__', '__init__', '__module__', 'gc', 'revcompl', 'translate', 'name', 'seq']
```

Information on instances. There are some mechanisms to know about the class of a given instance. You can use the special attribute `__class__`:

```
>>> s1 = DNA()
>>> s1.__class__
class __main__.DNA at 0x81d1d64>
```

You can even use this information to create other instances:

```
>>> s2=s1.__class__()
>>> s2
__main__.DNA instance at 0x8194ca4>
```

This can be useful if you need to create an object of the same class of another source object, without knowing the class of the source object. You can also ask whether an object belongs to a given class:

```
>>> isinstance(s1,DNA)
```

```
True
```

As mentionned above, the Python `type()` will not provide the class of an instance, but just: `InstanceType` (see Figure 17.8).

Chapter 18. Object-oriented design

18.1. Introduction

We first describe general software engineering guidelines to design components (Section 18.2.5). We next move on to the general question about why and how to design classes, and more specifically, how to design new *abstract data types* (Section 18.3). We will then re-discuss the issue of flexibility (Section 18.5) and describe inheritance (Section 18.4). Last, we will introduce the design patterns catalog (Section 18.6), which are well-known design solutions to standard problems in object-oriented programming.

18.2. Components

18.2.1. Software quality factors

The topics introduced in this section address some of the issues of software quality, and how Python can help on this matter.

Before entering into details, let us just summarize some important concepts (you can find a good and more exhaustive description in [Meyer97]). There is no absolute quality in software: depending on the context, scale, scope and goals of the program being developped, you might very well either write on-the-fly short pieces of code to solve a temporary problem, or spend a significant effort to have your application follow an industrial design process. So, rather than only a global so-called standard that should be applied for each program you write, there are a few *quality factors* to be evaluated according to the actual needs of the project. Among these factors, one usually distinguish between internal and external factors. *External* quality factors are the ones that corresponds to visible requirements, directly important for the user of the software, such as **validity**, **robustness** or **efficiency**. *Internal* quality factors are properties of the code itself, such as legibility or modularity. In fact, internal factors often indirectly help to get external quality. Some factors, such as **reusability**, **extensibility** and **compatibility**, that we will study more thoroughly here, belong to external quality factors in the sense that they can be required by the users of a software library or the programmers of a shared source code; they are also important internal quality factors in the sense that they make the internal quality of the source code better. The aim of this chapter is mainly to describe these factors, as well as internal quality factors.

18.2.2. Large scale programming

Theoretically, in order to get a program that performs a given task and solves the problem you have specified, a basic set of instructions such as: branching, repetitions, expressions and data structures can be sufficient. Now, the programs that you produce can become a problem by themselves, for several reasons:

- They can become very large, resulting in thousand lines of code where it is becoming difficult to make even a slight change (*extensibility* problem).

- When an application is developped within a team, it is important for different people to be able to share the code and combine parts developped by different people (*compatibility*) ; having big and complex source files can become a problem.

- During your programmer's life, or within a team, you will very often have to re-use the same kind of instructions set: searching for an item in a collection, organizing a hierarchical data structure, converting data formats, ...; moreover, such typical blocks of code have certainly already been done elsewhere (*reusability*). Generally, source code not well designed for re-use can thus be a problem.

So, depending on the context of the project, there are some issues which are just related to the *management of source code* by humans, as opposed to the specification of the task to perform. And if you think about it, you probably tend to use variable names that are relevant to your data and problem, aren't you? So, why? This is probably not for the computer, but, of course, rather for the human reader. So, in order to handle source structure and management issues, several conceptual and technical solutions have been designed in modern programming languages. This is the topic of this chapter.

Let us say that we have to develop source code for a big application and that we want this source code to be spread and shared among several team members, to be easy to maintain and evolve (extensible), and to be useful outside of the project for other people (reusable). What are the properties a source code should have for these purpose?

- it should be divided in small manageable chunks

- these chunks should be logically divided

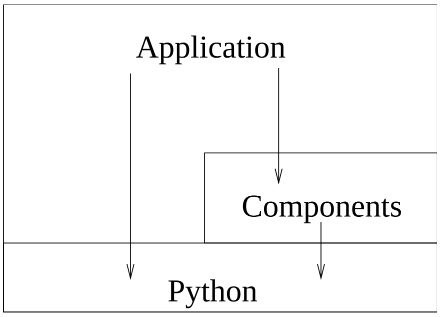
- they should be easy to understand and use

- they should be as independant as possible: you should not have to use chunk A each time you need to use chunk B

- they should have a clear *interface* defining what they can do

The most important concept to obtain these properties is called *modularity*, or how to build modular software *components*. The general idea related to software components is that an application program can be built by combining small logical building blocks. In this approach, as shown in figure Figure 18.1, building blocks form a kind of high-level language.

Figure 18.1. Components as a language



18.2.3. Modularity

The simplest form of modularity is actually something that you already know: writing a function to encapsulate a block of statements within a logical unit, with some form of generalization, or abstraction, through the definition of some parameters. But there are more general and elaborated forms of components, namely: modules and packages.

So, what is modularity? As developped in [Meyer97], modularity is again not a general single property, but is rather described by a few principles:

- **A few interfaces**: a component must communicate with as few other components as possible. The graph of dependencies between components should be rather loosely coupled.
- **Small interfaces**: whenever two components communicate, there should be as few communication as possible between them.
- **Explicit interfaces**: interfaces should be explicit. Indirect coupling, in particular through shared variables, should be made explicitly public.
- **Information hiding**: information in a component should generally remain private, except for elements explicitly belonging to the interface. This means that it should not be necessary to use non public attributes elements of a component in order to use it. In languages such as Python, as we will see later, it is technically difficult to hide a component's attributes. So, some care must be taken in order to *document* public and private attributes.

- **Syntactic units**: Components must correspond to syntactic units of the language. In Python, this means that components should correspond to known elements such as modules, packages, classes, or functions that you use in Python statements:

  ```
  import dna
  from Bio.Seq import Seq
  ```

  `dna`, `Bio`, `Bio.Seq` and `Seq` are syntactic units, not only files, directories or block of statements. In fact, Python really helps in defining components: almost everything that you define in a module is a syntactic unit.

You can view this approach as though not only the user of the application would be taken into account, but also the programmer, as the user of an intermediate level product. This is why there is a need for *interfaces design* also at the component level.

18.2.4. Methodology

These properties may be easier to obtain by choosing an appropriate design methodogy. A design methodoly should indeed:

- help in defining components by successive *decomposition*;
- help in defining components that are easy to *combine*;
- help in designing *self-understandable* components: a programmer should be able to understand how to use a component by looking only at this component;
- help in defining *extensible* components; the more independant components are, the more they are easy to evolve; for instance, components sharing an internal data structure representation are difficult to change, because you have to modify all of them whenever you change the data structure.

18.2.5. Reusability

Programming is by definition a very repetitive task, and programmers have dreamed a lot of being able to pick off-the-shelves general purpose components, relieving them from this burden of programming the same code again and again. However, this objective has, by far, not really been reached so far. There are several both non technical and technical reasons for this. Non-technical reasons encompass organisational and psychological obstacles: although this has probably been reduced by the wide access to the Web, being aware of existing software, taking the time to learn it, and accepting to use something you don't have built yourself are common difficulties in reusing components. On the technical side, there are some conditions for modules to be reusable.

1. *Flexibility*: One of the main difficulty for making reusable components lies in the fact that, while having the impression that you are again programming the same stereotyped code, one does not really repeat exactly the same code. There are indeed slight variations that typically concern to following aspects (for instance, in a standard table lookup):

- types: the exact data type being used may vary: the table might contain integers, strings, ...
- data structures and algorithms may vary: the table might be implemented with an array, a dictionary, a binary search tree, ... ; the comparison function in the sort procedure may also vary according to the type of the items.

So, as you can understand from these remarks, the more *flexible* the component is, the more reusable it is. Flexibility can be partly obtained by modularity, as long as modules are well designed. However, in order to get real flexibility, other techniques are required, such as genericity, polymorphism, or inheritance, that are described in Section 18.5.

2. *Independancy towards the internal representation*: by providing a interface that does not imply any specific internal data structure, the module can be used more safely. The client program will be able to use the same interface, even if the internal representation is modified.
3. *Group of related objects*: it is easier to use components when all objects that should be used together (the data structures, data and algorithms) are actually grouped in the same component.
4. *Common features*: common features or similar templates among different modules should be made shareable, thus making the whole set of modules more consistent.

18.3. Abstract Data Types

18.3.1. Definition

What is an abstract data type? Well, a data type is what we have just presented in chapter Chapter 17: in Python, as in many object-oriented language, it is a class. So, why *abstract*? One of the main objectives in component building, as we have seen in section Section 18.2, is to provide components that a programmer (you, your colleagues, or any programmer downloading your code) can be confident in. In order to get this programmer, who is a *client* of your code, confident into your class, you have to make it as stable as possible, and the best method to get a stable class is to define it at a level where *no implementation decision is visible*. In other words, defining a class should consist in defining a data type providing some abstract *services*, whose *interface* are clearly defined in term of parameters and return value and potential side effects. The advantages of this approach are the following:

- The implementor of the class can change the implementation for maintenance, bug fixes or optimization reasons, without disturbing the client code.
- The data type is defined as a set of high-level services with a *semantic contract* defining both the output that is provided and the input that is required from the client. This actually correspond to the general definition of a type in programming: a type is defined as a set of possible values and a *set of operations* available on these values.

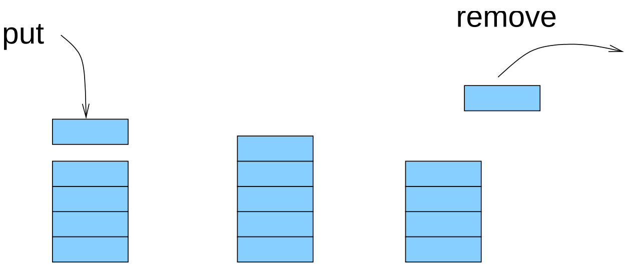
Among the methods defined in the interface of a data type, there are methods that build or change the state of the corresponding object, which are called *constructors* and *modificators*, and there are methods to access the object, which are called *accessors*.

Example 18.1. A Stack

Our example is a stack. It is directly inspired from [Meyer97], chapter 6.

Stacks serve to pile up objects of any kind, and take them out according to the rule "last in, first out" (LIFO). Imagine a pile of plates at home, in your cupboard, the safest way to take a plate out is to take the last one put in (Figure 17.8). Such structures are omni-present in programing, they serve to move along graphs, to compile programs, etc...

Figure 18.2. A stack



Stacks can be implemented in different ways: you can use a list, and either add the new item at the beginning or at the end of the list.

Conceiving an abstract data type (ADT) for the stack consists in describing the functions needed making *abstraction* of the implementation choice that will be made at the end (and that may change). Indeed in all cases, the basic services are the same, and can be given common names, that do not "betray" how the list is used "inside" of the code. The set of services is listed in Table 18.1.

Table 18.1. Stack class interface

Name	Input	Output	Description
put	a stack and an item	a stack	places the item on top of the stack
remove	a stack	a stack	if stack is not empty, removes the last item
item	a stack	an item	if stack is not empty, returns the item on top of the pile, without removing it
empty	a stack	a Boolean	tells whether the stack is empty
make		a stack	creates a new stack

The description provided by Table 18.1 should suffice for the client to use the class. What is more important here, is that the client does not have to know anything about the internal representation of the stack.

In this ADT *accessors* are: `item` and `empty`, *constructor* is `make` , and *modificators* are: `put` and `remove`.

The next step,once this description of the ADT is made for the client, consists in *specifying* the ADT, that is, describing in formal or mathematical language what the functions are doing. This involves four steps:

- Types: this corresponds in Python to the classes that are used for the ADT. In our example there is one class, Stack.
- Functions: That is their names and the input and output they have, as shown in Table 18.1, first three columns.
- Axioms: The rules to which the functions obey, and that are sufficient to describe them. For the stack, these axioms are, for any stack s and element e:

  - item(put(s, e))=e
  - remove(put(s, e))=s
  - empty(make) is True
  - empty(put(s, e) is False

- Pre-conditions: The functions we need will not operate under all conditions. For example, you cannot remove an element from an empty stack. These pre-conditions can be enumerated as follows for the stack:

  - remove(s) requires : not empty (s)
  - item(s) requires not empty(s)

Axioms and pre-conditions formulate more precisely what is said inthe fourth column of Table 18.1

18.3.2. Information hiding

It is a good design practice to protect an instance local data. Why is that? The first reason is that it makes the interface of the class more easy to grasp. The client just knows what needs to be known in order for the component to be used properly. A second reason is that class data should rather be handled by the class methods: there might be some coherence to be maintained among different attributes, that a method can check. Another important reason is to avoid the risk of collisions, that could happen between the class and the potential base classes, as we will explain in Section 18.4, devoted to inheritance.

The problem is that there is no real mechanism in Python to prevent a client code from accessing to an instance attributes, as there are in other languages (such as Java or C++), where you can declare *private* and *public* attributes. Python provides just the following: if you name a variable with a '__' prefix and no '_' suffix, such as:

```
__list = []
```

within a class named, e.g: `Stack`, Python automatically changes it to: `_Stack__list`. You can still access it, but you are aware that you are probably doing something odd, and it can help to avoid name collisions.

Another way to distinguish *public* and *private* attributes and methods is to prefix the private ones with a single '_' character. This does not provoke any automatic addition by Python, but it is warning the reader that this part of the code is private. Another style matter is the naming of classes, where it is a good idea to capitalize the first letter of class names, to distinguish them from variable or function names. Be aware finally that some methods in Python are framed by double ’__’ on both side, they are called *special methods* and described in the next p aragraph.

The lesson to be learnt is that attributes you want to be accessible should be mentionned explicitely, and documented as such.

Now that the ADT is specified, that we know which attributes and methods are public or private, We can turn to the implementation part of our example. As shown in Example 18.2, we have decided to use a Python list, and to add new elements to the end of this list.

Example 18.2. Stack class

```
class Stack:
    """
    A class to handle stacks
    """

    def __init__(self):
        self._list=[]

    def put(self, elem):
        """
        places an item on top of the stack
        """
        self._list.append(elem)

    def remove(self):
        """
        if not empty, removes the last item placed on the stack
        else does not do anything
        """
        if self._list:
            self._list.pop()

    def item(self):
        """
        if not empty, returns the item on top of the stack
        without removing it
        else does not do anything
        """
        if self._list:
            return self._list[-1]

    def empty(self):
        """
        tells if the stack is empty
        """
        if len(self._list)==0:
            return True
        return False
```

18.3.3. Using special methods within classes

The usual Python operators, such as '+', the '[ ]' operator to access an item or a slice, the assignment '=' operator, the equality '==' operator, but also 'print' etc..., are defined with *special methods* that have different definitions for different types. Indeed adding two integers is not the same as adding two strings. It is very convenient to redefine these special methods inside the classes created with Python. The very basic *special methods* encountered in most classes is the one allowing to use 'print'.

Such *special methods* have obligate names, and are framed by '__', such as __str__ for 'print'. A set a very precious pages in the book 'Python. Essential reference' concerns all the tables (3.7 to 3.10) listing these obligate names. Here below in Table 18.2, we will list a small number of examples.

Table 18.2. Some of the special methods to redefine Python operators

Operator	Special method	Parameters	Description
[]	__getitem__	self, k	access to item or slice specified by `k`
==	__eq__	self, other	test for equality between 2 objects
.	__getattr__	self, name	access to a *non existing* attribute named `name`
. and =	__setattr__	self, name, value	modification of an attribute
len()	__len__	self	length computation
print	__str__	self	string form for instance value
del	__del__	self	deletion of an instance

Example 18.3. Defining operators for the DNA class

Remember what we said in Section 17.5.3: different objects, even with the same state, i.e with the same attributes values, are considered as not equal by Python. There is a way however to specify how the '==' operator should behave for your class. The following code redefines the '==' operator as returning True whenever the `name` and `seq` attributes are equal.

```
class DNA:

    ...

    def __eq__(self, other):
        if self.name == other.name and self.seq == other.seq:
           return True
        return False

>>> s1 = DNA(name='seq2', seq='acaagatgccattgtcccccggcctcctgctgctgctgctctccggggcca')

>>> s2 = DNA(name=s1.name, seq=s1.seq)
>>> s1 == s2
True
>>> s1 is s2
False
```

Another important operator that could be useful for sequences is the '[]' operator. for strings, this operator enables to access a character at a specific position. This operator is defined by the `__getitem__` special method.

```
class DNA:

    ...

    def __getitem__(self, i):
        return self.seq[i]

>>> print s1[3]
'a'
```

Exercise 18.1. Operators for the DNA class

Which additional operators could you define for `DNA` class? Implement 1 or 2 of them.

18.4. Inheritance: sharing code among classes

18.4.1. Introduction

What we have seen so far, *object-based* programming, consists in designing programs with objects, that are built with classes. In most object-oriented programming languages, you also have a mechanism that enables classes to share code. The idea is very simple: whenever there are some commonalities between classes, why having them repeat the same code, thus leading to maintenance complexity and potential inconsistency? So the principle of this mechanism is to define some classes as being the same as other ones, with some differences.

Example 18.4. Inheritance example (1): sequences

So far, we have design two different classes for different types of sequences: indeed, having all the methods defined only for protein sequences also available for DNA sequences can be considered a problem. At the same time, some functionalities are common to both types of sequences: cleaning the sequence, producing a Fasta format to name a few. So we keep our two classes to deal with DNA and protein sequences, and we add a new class: `Sequence`, which will be a common class to deal with general sequence functions. In order to specify that a `DNA` (or a `Protein`) *is a* `Sequence` in Python is:

```
class DNA(Sequence):
```

The `DNA` class is called a *subclass*, and the `Sequence` is called a *superclass* or a *base class*. Following the `class` statement, there are *only* the definitions specific to the `DNA` or `Protein` classes: you do not need to re-specify the other ones. The following code shows how to specify the `Sequence` superclass.

```
class Sequence:

    def __init__(self, name=None, seq=None):
        self.name = name
        self.seq = seq
        self.clean()

    def clean(self):
        seq = ""
        for c in self.seq:
           if c in self.alphabet:
              seq += c
        self.seq = seq

    def __str__(self):
        return ">"+self.name+"\n"+self.seq

    def __getitem__(self, i):
        return self.seq[i]

    def getname(self):
        return self.seq

    def getseq(self):
        return self.seq



class DNA(Sequence):

      alphabet = "atcg"

      def gc(self):
          """GC percent"""
          ...

      def revcompl(self):
          """reverse complement"""
          ...

      def translate(self):
          """translation into a protein"""
          ...



class Protein(Sequence):

    weight = {"A":71.08,"C":103.14 ,"D":115.09 ,"E":129.12 ,"F":147.18 ,"G":57.06 ,"H":137.15 ,"I":11


    alphabet = weight.keys()

    def mw(self):
        molW = 0
        for aa in self.seq:
            molW += self.weight[aa]

        molW += 18.02
        molW = molW / 1000

        return molW
```

Look at how we use these definitions:

```
>>> s = DNA("s1", "attgccctt")
>>> s.gc()
66.66
>>> print s
>s1
attgccctt
```

The `__str__` method that is called when issuing a **print** statement is not defined for the `DNA` class. So python looks up one level further, finds a definition at the `Sequence` class definition level and calls it. Provided with the `self` reference to the object, exactly the same way as for `DNA` class methods, it can access to the attributes in order to produce a printable string of the object.

The `clean` method is a common method, except that it uses a specific alphabet for each type of sequence. This is why each subclass must define an `alphabet` class attribute. When referring to `self.alphabet`, python looks up in the object's namespace first, then one level up in the class namespace and finds it.

Now look carefuly at the `__init__` method: it is now defined in the superclass. There are indeed common things to do for both `DNA` and `Protein` classes: attributes initialization, sequence cleaning, ... So it is enough to define the `__init__` method at the superclass level: whenever the `__init__` method, or any method is not defined at the subclass level, it is the method at the superclass level which is called (see below Section 18.4.1.2). But, remember, the DNA sequence is supposed to be in lowercase, whereas the protein sequence is in uppercase.

How can we do that? The idea is to define an `__init__` method at the subclass level that calls the `__init__` method of the superclass, and then proceed to actions specific to the `DNA` or the `Protein` sequences.

```
class Sequence:

    def __init__(self, name=None, seq=None):
        self.name = name
        self.seq = seq
        self.clean()

...

class DNA(Sequence):

      def __init__(self, name=None, seq=None):
        Sequence.__init__(self, name, lower(seq))

...

class Protein(Sequence):

    def __init__(self, name=None, seq=None, dna=None):
        Sequence.__init__(self, name, upper(seq))
        self.dna = dna  # when called by DNA.translate()
...
```

18.4.1.1. Overloading

Let us decide that a generic `mw` method can be generalized and put at the `Sequence` class level. This way, a potential `RNA` subclas of `Sequence` will have a pre-defined `mw` method, common to all the molecular sequences. We do not need any `mw` method at the `Protein` level anymore. But, since the computation of the molecular weight differs slightly for DNA sequences, we define a specific `mw` method in the `DNA` class.

```
class Sequence:

    def mw(self):
        molW = 0
        for c in self.seq:
            molW += self.weight[]

        return molW

...

class DNA(Sequence):
    def mw(self):
        """
        Computes the molecular weight of the 2 strands of a DNA sequence.
        """
        molW = 0
        for c in self.seq:
            molW += self.weight[]

        for c in self.revcompl():
            molW += self.weight[]

        return molW

...
```

When a method is redefined (or *overriden*) in the subclass, which is the case of the `mw` method, it is said that the method is *overloaded*. This means that, according to the actual class of a sequence instance, which can be either `DNA` or `Protein`, the method actually called can vary.

Overloading

Overloading a method is to redefine at a subclass level a method that exists in upper classes of a class hierarchy.

Another term that is used in object-oriented programming is the term: "polymorphism". "Polymorphism" litteraly means: several forms. In other words, the name `mw` has several meanings, depending on the context of the call.

Polymorphism

A method is said to be polymorphic if it has several definitions at different levels of a class hierarchy.

The term "polymorphism" is also used about operators. The ’+’ operator, for instance, is polymorphic in the sense that it refers to different operations when used with strings and integers.

18.4.1.2. How does it work? Dynamic binding.

As we said above, by declaring `DNA` as a subclass of `Sequence`, you do not need to re-specify in `DNA` all the methods that are already defined in `Sequence`. The way it works is that Python looks up methods starting from the actual class of the object, for instance `DNa` and, if not found, by looking up further in `Sequence` super class for the needed method or attribute. So, when calling:

```
>>> my_dna.gc()
```

python looks up for the `gc` method (Figure 18.3). Python finds it in the current class. When looking up for other methods, such as the `__str__` method, which is not defined in the `DNA` class, Python follows the graph of base classes (Figure 18.4). Here, we only have one: `Sequence`, where `__str__` is defined.

Figure 18.3. Dynamic binding (1)

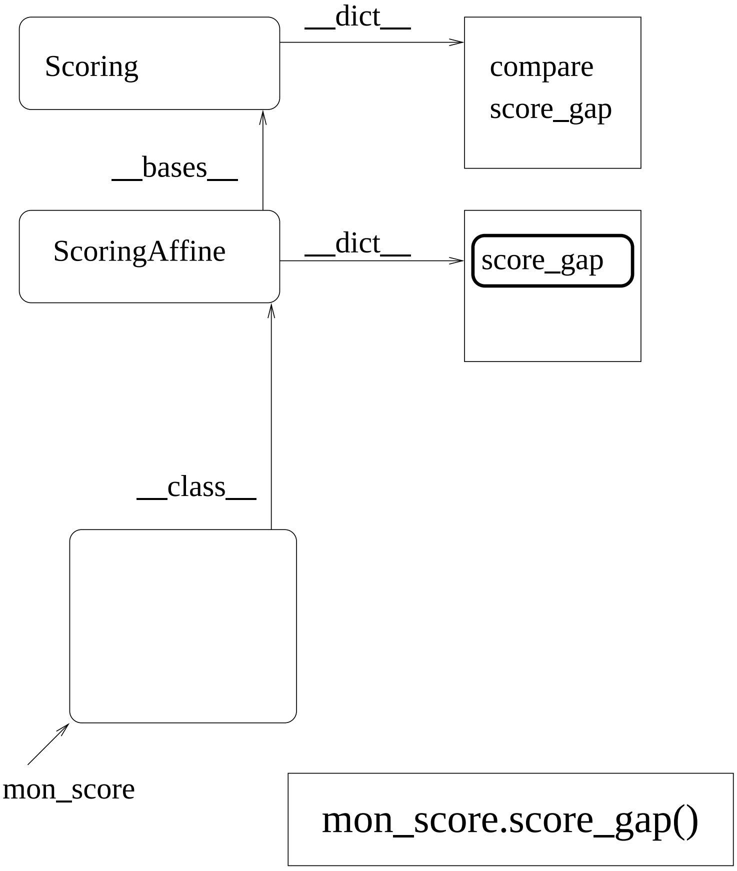
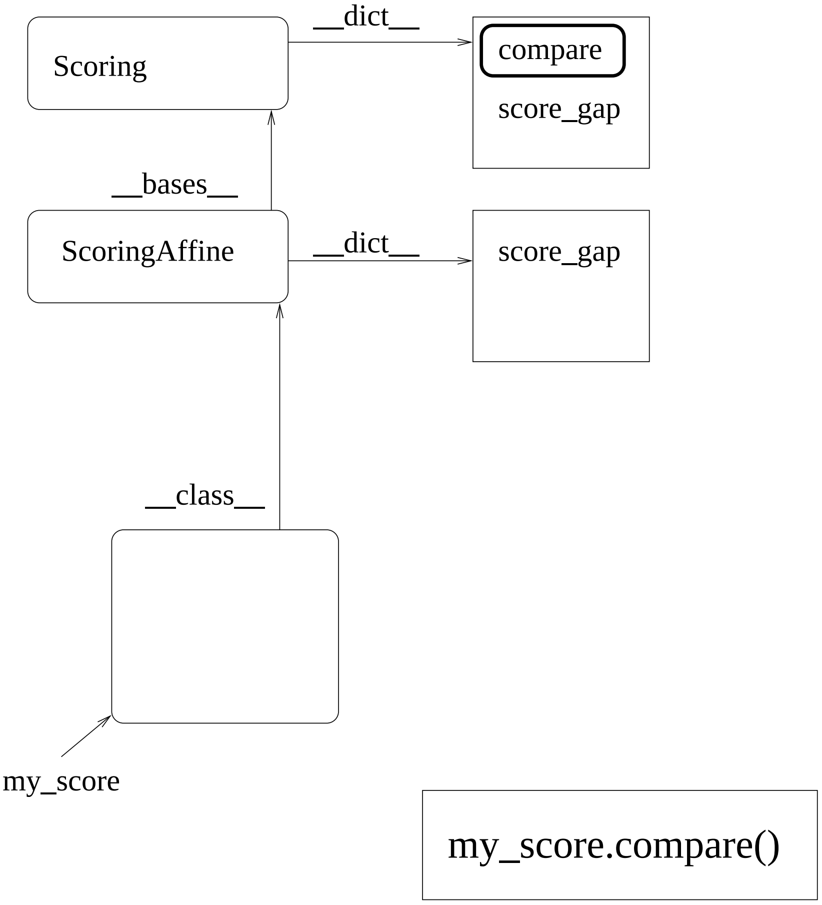


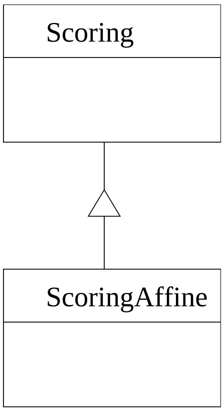
Figure 18.4. Dynamic binding (2)



Forms of inheritance. There are two forms of inheritance: *extension* and *specialisation*. In other words, inheritance can be used to extend a base class adding functionalities, or to specialise a base class. In the case of the `DNA` class, it is rather an extension. Sometimes, the term *subclass* is criticized because, in the case of extension, you actually provide *more* services, rather than a *subset*. The term subclass fits better to the case of specialization, where the subclass addresses a specific subset of the base class potential objects (for instance, dogs are a subset of mammals).

UML diagrams. Classes relationships (as well as instances relationships, not represented here) can be represented by so-called UML diagrams, as illustrated in Figure 18.5.

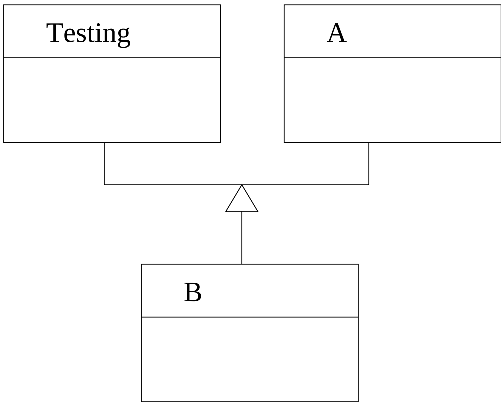
Figure 18.5. UML diagram for inheritance



18.4.1.3. Multiple inheritance

In Python, as in several programming language, you can have a class inherit from several base classes. Normally, this happens when you need to mix very different functionalities. For instance, you want to wrap your class with a class that provides tools for testing (Figure 18.6), or services for persistence. Inheriting from classes that come from the same hierarchy can be tricky, in particular if methods are defined everywhere: you will have to know how the interpretor choose a path in the classes graph. But this case is more like a theoretical limit case, that should not happen in a well designed program.

Figure 18.6. Multiple Inheritance



18.4.2. Discussion

Benefits of inheritance.

- Regarding *flexibility*, the inheritance mechanism provides a convenient way to define variants at the method level: with inheritance indeed, the *methods become parameters*, since you can redefine any of them. So you get more than just the possibility of changing a parameter value.
- Regarding *reusability*, inheritance is very efficient, since the objective is to reuse a code which is already defined. Components designed with the idea of reuse in mind very often have a clean inheritance hierarchy in order to provide a way for programmers to adapt the component to their own need.
- Inheritance also provides an elegant *extensibility* mechanism, by definition. It lets you extend a class without changing the code of this class, or the code of the module containing the class.

Combining class or combining objects? Using inheritance is not mandatory. The main risk of using it too much is to get a complex set of classes having a lot of dependencies and that are difficult to understand. There are generally two possibilities to get things done in object-oriented programming:

- *Inheritance:* you combine classes in order to get a "rich" class providing a set of services coming from each combined class.
- *Composition:* you combine objects from different classes.

The use of composition instead of inheritance is also illustrated by the design patterns from [Gamma95], that are introduced in Section 18.6.

Problem with inheritance for extension. When using inheritance to extend a base class, you might want to have a method in the subclass not just overloading the method in the base class, but as a complement. In this case, one usually first calls the base class, and then performs the complementary stuff. In Python, you have to know explicitely the name of the superclass to perform this call (see for instance method `__init__`). Be aware, that this can become rather tricky sometimes, for you have to design a real protocol describing the sequence of calls that have to be done among classes, when not only one method is involved.

When using inheritance or composition: summary. The question of choosing between inheritance and composition to combine objects A and B results in deciding whether A *is-a* B or whether A *has-a* B. Unfortunately, it is not always possible to decide about this, only on the basis of the nature of A and B. There a few guidelines, though (see [Harrison97], chapter 2).

- The main advantage of inheritance over composition is that method binding, e.g lookup and actual method call, is done automatically: you do not have to perform the method lookup and call, whereas, when combining objects, you have to know which one has the appropriate method. For instance, a `Protein` instance may *have a* `dna` if created by `translate`.
- Use composition when you catch yourself making exceptions to the *is-a* rule and redefining several inherited methods or willing to protect access to the base class attributes and/or methods. In such a case, the advantage described in the previous item of having automatic attribute and method access becomes a problem.

- Use composition when the relationships between objects are dynamic. For instance, a different way to design sequence tools, such as `clean`, `__str__`, etc... could be to design a `SeqTools` class:

  ```
  class SeqTools:

       def __init__(self, seqobject):
           self.seqobject = seqobject

       del clean(self):
           seq = ""
           for c in self.seqobject.seq:
                if c in self.seqobject.alphabet:
                   seq = seq + c
           self.seqobject.seq = seq

  ...
  ```

  The `DNA` or `Protein` instances could then have a reference to a `SeqTools` instance, created at instantiation:

  ```
  class DNA:

        def __init__(self, name=None, seq=None):
          self.toolbox = build_seqtools(self)
          self.name = name
          self.seq = seq
          self.toolbox.clean()

  def build_toolbox(seqobject):
       return SeqTools(seqobject)
  ```

  The main advantage is flexibility: you could more easily redefine which sequence toolbox to use, provided the public interface is the same, than with inheritance. Indeed, inheritance relationships are fixed at definition time:

  ```
  class DNA(Sequence):
  ...
  ```

- Use composition when a single object must have more than one part of a given type.

- Use composition to avoid deep inheritance hierarchies.

- If you can’t decide between inheritance and composition, take composition.

- Do not use inheritance when you get too many combined method calls between base class and sub-classes, which can happen when using inheritance for extension.

- Use inheritance when you want to build an abstract framework, which purpose is to be specialized for various uses (see Exercise 18.2). A good example is the parsing framework in Biopython, that lets you create as many new parsers as needed. Inheritance also provides a good mechanism to design several layers of abstraction, that define interfaces that programmers must follow when developping their components. Bioperl modules, although not developped in a true object-oriented language, are a good example of this.

Exercise 18.2. Example of an abstract framework: Enzyme parser

Design a parser for the Enzyme database, using Biopython parsing framework. See ??? for more details.

18.5. Flexibility

18.5.1. Summary of mechanisms for flexibility in Python

Earlier, in Section 18.2.5, we have seen that one of the powerful properties a software piece can have is to be reusable. We also concluded that the more flexible it is, the more general and thus reusable it will be. Let us now summarize some mechanisms in object-oriented programming that help in achieving more flexibility.

- *Genericity*: genericity (as available in the ADA programming language), is a technique to define parameters for a module, exactly as you define parameters for a function, thus making the module more general. For instance, you can define the type of an element of a Table module, or the function to move to the next item, as being generic. There is no specific syntax in Python to define generic components, but in a way, it is not really necessary because Python is dynamically typed, and can handle functions as parameters for other functions.

- *Inheritance*: as we have just seen in Section 18.4 this describes the ability, in object-oriented programming, to derive a class from another, either to extend it or to specialize it.

- *Overloading*, which refers to the possibility for an operator or a function to behave differently according to the actual data types of their operands or arguments. This feature does not directly address the problem of flexibility: it is rather an elegant syntactic mean not to have different operators or names to perform similar tasks on different objects or sets of objects. In this sense, it is actually a form of polymorphism. In object-oriented programming, overloading moreover exactly means being able to redefine a method in a derived class, thus enabling polymorphism of instances: given an instance, it is possible that you do not know exactly to which class in a hierarchy it belongs (e.g `Scoring` or `ScoringAffine`). In other programming languages, such as Java, there is another kind of overloading: it is possible, *within the same class*, to have several definitions of the same method, but with different *signatures*, i.e a different set of parameters. This does not exist in Python, and as described below (see), you have to handle it manually.

- *Polymorphism*: refers to the possibility for something to be of several forms, e.g, for an object to be of any type. In Python, for instance, lists are polymorphic, since you can put items of any type in a list. Overloading is one aspect of polymorphism, polymorphism of methods. Polymorphism brings a lot of flexibility, just only because you do not have to define as many classes as potential data type for it, as you would have to do in statically typed languages. In this sense, Python helps in designing more simple classes frameworks.

Some of these techniques, mainly inheritance, are available in Python, other, such as genericity and overloading, are not, or in a restricted way.

18.5.2. Manual overloading

If you want a method to behave differently according to the parameters, you must either create subclasses or write the code to analyze the parameters yourself. In Example 18.5, there is an example a method dealing with different types of parameters, that are manually handled.

Example 18.5. Curve class: manual overloading

```
def __getitem__(self, key):
    """
    Extended access to the curve.
    c[3]
    c[’toto’]
    c[’x<100’]
    c[’x==y’]
    """
    _x = []
    _y = []
    _result = None
    if type(key) is ListType:                                         ❶
        _x = key
        _x.sort()
        _y = self.y(_x)
        _result = Curve(zip(_x, _y))
    elif type(key) is SliceType:                                      ❷
        _x = self._range_x(key.start, key.stop)
        _y = apply(self.y, _x)
        if _x and _y:
            _result = Curve(zip(_x, _y))
    elif self._curve.has_key(key):                                    ❸
        _result = self._curve[key]
    elif type(key) is StringType:                                     ❹
        _x = self._tag_x(key)
        if len(_x) > 0:                                               ❺
            _x.sort()
            _y = self.y(_x)
            _result = Curve(zip(_x, _y))
        else:                                                         ❻
            _result = self._getexpr(key)
    if _result is not None:
        return _result
```

❶ Recognizes the argument as a list.

❷ Recognizes the argument as a slice.
❸ Recognizes the argument as an index.
❹ Recognizes the argument as a string, that is going to be further analyzed as a tag name or an expression.
❺ Recognizes the argument as a tag name.
❻ Recognizes the argument as an expression, such as 'x< y' or 'y>10'.

Python is not able to do this analysis automatically, because it is dynamically typed. You do not have any mean to specify the *type* of the parameter. In a language providing static typing and full method overloading, i.e also within the same class, you could have several `__getitem__` definitions that would look like:

```
def __getitem__(self, key: List):
     ...
def __getitem__(self, key: Slice):
     ...
def __getitem__(self, key: String):
     ...
```

As you can notice, this is of course not valid Python code, since there is no possibility to define the type of a parameter in Python.

18.6. Object-oriented design patterns

Presenting Python classes and various related mechanisms without any description of their common uses would be like defining the `for` and `while` statements without any description of the typical uses of these constructions, and how they adapt to various programming needs. It is the purpose of this section to present common uses of classes and objects, the so-called *design patterns*, that have been used for a long time by programmers since object-oriented programming was born.

Computer science is full of design patterns, and there is no exception for object-oriented programming. A catalog of object-oriented has been published by [Gamma95], and [Christopher2002] has a chapter dedicated to a few of them, with examples implemented in Python. Design patterns are not programs: they are widely used *design* choices to build programs. As such, they form a kind of conceptual catalog that you are encouraged to *reuse* to build your application. Not only are they useful in order not to reinvent the wheel and to save your time, but also because they provide a *comprehension framework* for programmers who intend to reuse your code and who need to understand it. This part does not aim at an exhaustive presentation of object-oriented design patterns, which are very well described elsewhere. Its main purpose is to give a taste of it, with examples in bioinformatics, and to introduce the main related ideas.

There are three categories of object-oriented patterns:

- *Creational* patterns: patterns that can be used to *create* objects.
- *Structural* patterns: patterns that can be used to *combine* objects and classes in order to build structured objects.
- *Behavioral* patterns: patterns that can be used to build a computation and to control data flows.

Creational patterns. There are two main categories of creational patterns: those for creating objects without having to know the class name, that you could call "abstract object makers" (*abstract factory* and *factory method*), and those to ensure a certain property regarding object creation, such as prohibiting more than one instance for a class (*singleton*), building a set of instances from different classes in a consistent way (*builder*), or creating an instance with a specific state (*prototype*).

- *Abstract factory:* an abstract factory is an object maker, where, instead of specifying a class name, you specify the kind of object you want. For instance, say that you want to create an agent to run analysis programs, you can ask a factory to do it for you:

  ```
  clustalw = AnalysisFactory.program('clustalw')
  result = clustalw.run(seqfile = 'myseqs')
  print result.alig
  ```

  The `clustalw` object is an instance of, say, the `AnalysisAgent.Clustalw` class, but you do not have to know about it at creation time. The only thing you know is the name of the program you want ('clustalw'), and the factory will do the rest for you.

- *Factory method:* a factory method is very similar to an abstract factory: just, instead of being a class, it is a method.

  - For instance, you can create a sequence object (`Bio.Seq.Seq` in Biopython) by asking the `get_seq_by_num` method of an alignment object (`Bio.Align.Generic.Alignment`):

    ```
    first_seq = align.get_seq_by_num(0)
    ```

    The method which creates this instance of `Bio.Seq.Seq` is a factory method. The difference with a factory class is also that the factory method is often more than an object maker: it sometimes incorporates much more knowledge about the way to create the object than a factory would.

  - A more simple factory method would be a `new` method defined in a class to create new instances of the same class:

    ```
    my_scoring = scoring.new()
    ```

    In this case, notice that in order to create `my_scoring`, you really do not have to know the actual class of `scoring`: the only thing you know is that you will get *the same one*, even if there is a whole hierarchy of different classes of scoring.

  - Another example could be a factory method in the `Plot` class:

    ```
    p = Plot()
    p.new_curve([...])
    ```

    that would create an instance of the `Curve` class and draw it as soon as created.

- *Singleton:* ensures that you cannot create more than one instance. For example, if you can define a class to contain operations and data for genetic code: you need only one instance of this class to perform the task. Actually, this pattern would not be implemented with a class in Python, but rather with a module, at least if you can define it statically (a dynamic singleton could not be a module, for a module has to be a file):

  ```
  >>> import genetic_code
  >>> genetic_code.aa('TTT')
  F
  ```

- *Prototype:* this pattern also lets you create a new object without knowing its class, but here, the new target object is created with the same state as the source object:

  ```
  another_seq = seq.copy()
  ```

  The interest is that you do not get an "empty" object here, but an object identical to `seq`. You can thus play with `another_seq`, change its attributes, etc... without breaking the original object.

- *Builder:* you sometimes need to create a complex object composed of several parts. This is the role of the builder.

  - For instance, a builder is needed to build the whole set of nodes and leafs of a tree.

  - Or you could design a builder to create both `Curve` and `Plot` instances in a coherent way as parts of a `GraphicalCurves` complex object:

    ```
    gc = GraphicalCurves(file='my_curves')
    ```

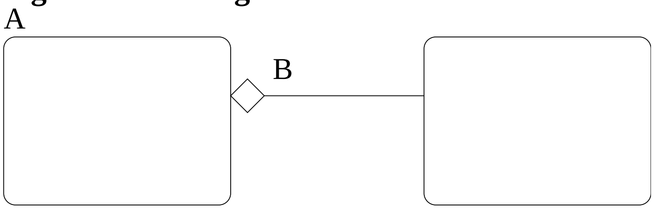
    For instance, `my_curves` might contain description of set of curves to display in the same plot or in different plots.

  - The Blast parser in Biopython simultaneously instantiates several classes that are all component parts of of hit: `Description`, `Alignment` and `Header`.

Structural patterns. Structural patterns address issues regarding how to combine and structure objects. For this reason, several structural patterns provide alternative solutions to design problems that would else involve inheritance relationships between classes.

- *Decorator, proxy, adapter:* these patterns all enable to combine two (or more) components, as shown in Figure 18.7. There is one component, A, "in front" of another one, B. A is the visible object a client will see. The role of A is either to extend or restrict B, or help in using B. So, this pattern is similar to subclassing, except that, where a sub-class *inherits* a method from a base class, the decorator *delegates* to its decoratee when it does not have the required method. The advantage is flexibility (see Section 18.4.2): you can combine several of these components in any order at run time without having to create a big and complex hierarchy of subclasses.

  Figure 18.7. Delegation

  Generally, the Python code of the A class looks like:

```
class A:
        def __init__(self, b):
            """storing of the decoratee b (b is an instance of class B)"""
            self.b = b

        def __getattr__(self, name):
            """
               methods/attributes A does not know about are delegated
               to b
            """
            return getattr(self.b, name)
```

  At use time, an instance of class A is created by providing a b instance.

```
b = B()
a = A(b)
print a.f()
```

  Everything that class A cannot perform is forwarded to b (providing that class B knows about it).

  - The *decorator* enables to add functionalities to another object. Example 18.6 shows a very simple decorator that prints a sequence in uppercase.

Example 18.6. An uppercase sequence class

```
import string

class UpSeq:

    def __init__(self, seq):
        self.seq = seq

    def __str__(self):
        return string.upper(self.seq)

    def __getattr__(self,name):
        return getattr(self.seq,name)
```

The way to use it is for instance:

```
>>> s=UpSeq(DNA(name=’name1’, seq=’atcgctgtc’))
>>> print s
ATCGCTGTC
>>> s[0:3]
’atc’
>>> len(s)
9
```

Exercise 18.3. An analyzed sequence class

How would you design sequence classes where all sequence analyses of type `f` would be available as:

```
seq.f()
```

without actually defining all the possible `f` methods within the sequence class?

- The *proxy* rather handles the access to an object. There are several kinds of proxy:
    - *protection proxy*: to protect the access to an object.

Exercise 18.4. A partially editable sequence

How would you design a sequence class where the only allowed editions would be:

```
del seq[i]
```

on a gap character, and:

```
seq[i] = ’A’
```

on a ’N’ character?

- *virtual proxy*: to physically fetch data only when needed. Database dictionaries in Biopython work this way:

  ```
  prosite = Bio.Prosite.ExPASyDictionary()
  entry = prosite[’PS00079’]
  ```

  Data are fetched only when an access to an entry is actually requested.

- *remote proxy*: to simulate a local access for a remotely activated procedure.

- The *adapter* (or *wrapper*) helps in connecting two components that have been developped independantly and that have a different interface. For instance, the `Pise` package transforms Unix programs interfaces in standardized interfaces, either Web interfaces, or API. For instance, the **golden** Unix command has the following interface:

  ```
  bash> golden embl:MMVASP
  ```

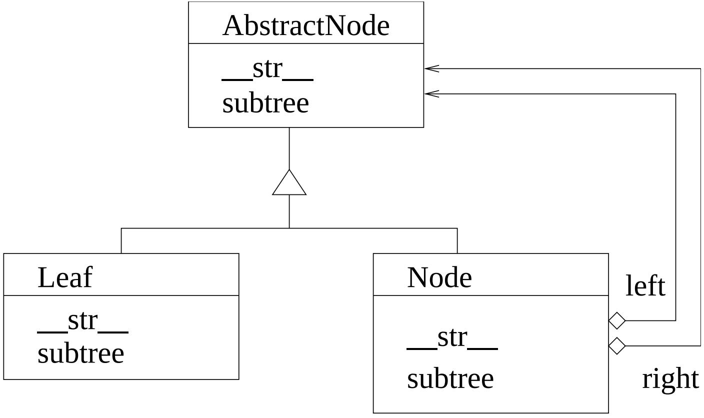
  But the `Pise` wrapper enables to run it and get the result by a Python program, having an interface defined in the Python language:

  ```
  factory = PiseFactory()
  golden = factory.program("golden",db="embl",query="MMVASP")
  job = golden.run()
  print job->content()
  ```

- *Composite*: this pattern is often used to handle complex composite recursive structures. Example 18.7 shows a set of classes for a tree structure, illustrated in Figure 18.8. The main idea of the composite design pattern is to provide an *uniform interface* to instances from different classes in the same hierarchy, where instances are all components of the same composite complex object. In Example 18.7, you have two types of nodes: Node and Leaf, but you want a similar interface for them, that is at least defined by a common base class, AbstractNode, with two operations: print subtree. These operations should be callable on any node instance, without knowing its actual sub-class.

```
>>> t1 = Node ( Leaf ( ’A’, 0.71399),
         Node ( Node ( Leaf(’B’, -0.00804),
                       Leaf(’C’, 0.07470),
                0.15685),
                Leaf (’D’, -0.04732)
              )
         )
>>> print t1
(A: 0.71399, ((B: -0.00804, C: 0.0747), D: -0.04732))
>>> t2 = t1.right.subtree()
>>> print t2
((B: -0.00804, C: 0.0747), D: -0.04732)
>>> t3 = t1.left.subtree()
>>> print t3
’A’: 0.71399
```

Figure 18.8. A composite tree



Example 18.7. A composite tree

❶

```
class AbstractNode:

    def __str__(self):
        pass
    def subtree(self):
        pass
                                                                               ❷
class Node(AbstractNode):
    def __init__(self, left=None, right=None, length=None):
        self.left=left
        self.right=right
        self.length=length

    def __str__(self):
        return "(" + self.left.__str__() + ", " + self.right.__str__() + ")"

    def subtree(self):
        return Node(self.left, self.right)

class Leaf(AbstractNode):                                                      ❸
    def __init__(self, name, length=None):
        self.name = name
        self.length=length
        self.left = None
        self.right = None

    def __str__(self):
        return self.name + ": " + str(self.length)

    def subtree(self):
        return Leaf(self.name, self.length)
```

❶ Abstract class `AbstractNode`, base class for both `Node` and `Leaf`
❷ Internal nodes are instances of `Node` class.
❸ Leafs are instances of `Leaf` class.

Behavioral patterns. Patterns of this category are very useful in sequence analysis, where you often have to combine algorithms and to analyze complex data structure in a flexible way.

- *Template*: this pattern consists in separating the invariant part of an algorithm from the variant part. In a sorting procedure, you can generally separate the function which compares items from the main body of the algorithm. The template method, in this case, is the `sort()` method, whereas the `compare()` can be defined by each subclass depending on its implementation or data types. In the dynamic programming align function, the template methods can be the `begin` `end` and `inner` methods. The `Scoring` method may vary depending on the preferred scoring scheme, as defined in subclasses of the `Scoring` class.

- *Strategy*: it is the object-oriented equivalent of passing a function as an argument. The `Scoring` is a strategy. **Variations on methods.** So, you don't necessarily need inheritance to have a function be a parameter. For instance, the following function enables you to provide your own functions for `score_gap` and `compare`:

  ```
      def align(matrice, begin, inner, end, score_gap, compare):
      ...
  ```

  So, do we need a subclass for this or even to have a `Scoring` class at all? The answer is that with class and inheritance, your methods are all pooled together in a "packet", but there is some additional burden on your side, since you have to define a subclass and instantiate it. On the other hand, passing a function as a parameter has a limit: you can't change the default values for parameters, such as the default value for a gap initiation or extension in our example.

- *Iterator*: an iterator is an object that let you browse a sequence of items from the beginning to the end. Generally, it provides:

  - a method to start iteration
  - a method to get the next item
  - a method to test for the end of the iteration

  Usually, one distinguishes *internal* versus *external* iterators. An external iterator is an iterator which enables to do a `for` or a `while` loop on a range of values that are returned by the iterator:

  ```
  for e in l.elements():
       f(e)
  ```

  or:

  ```
  i = l.iterator()
  e = i.next()
  while e:
     f(e)
     e = i.next()
  ```

  In the above examples, you control the loop. On the other hand, an internal iterator just lets you define a function or a method (say, `f`) to apply to all elements:

  ```
  l.iterate(f)
  ```

  In the Biopython package, files and databases are generally available through an iterator.

  ```
  handle = open(...)                                                  ❶
  ```

```
iter = Bio.Fasta.Iterator(handle)                                              ❷
seq = iter.next()                                                              ❸
while seq:
  print seq.name
  print seq.seq                                                                ❹
  seq = iter.next()
handle.close()
```

❶ Starting the iterator.
❷ Getting the next element.
❸ Testing for the end of the iteration.
❹ Getting the next element.

- *Visitor*: this pattern is useful to specify a function that will be applied on each item of a collection. The Python `map` function provides a way to use visitors, such as the `f` function, which visits each item of the `l` list in turn:

  ```
  >>> def f(x):
   return x + 1
  >>> l=[0,1,2]
  >>> map(f,l)
  [1, 2, 3]
  ```

  The `map` is an example of an internal iterator (with the `f` function as a visitor).

- *Observer* The *observer* pattern provides a framework to maintain a consistent distributed state between loosely coupled components. One agent, the observer, is in charge of maintaining a list of *subscribers*, e.g components that have subscribed to be informed about changes in a given state. Whenever a change occurs in a state, the observer has to inform each subscriber about it.
  A well-known example is the Model-View-Controller [Krasner88] framework. The view components, the ones who actually display data, subscribe to "edit events" in order to be able to refresh and redisplay them whenever a change occurs.

Bibliography

Python Books

[Beazley2001] *Python*. Essential Reference. 2. David M. Beazley. 0-7357-1091-0. New Riders. 2001.

[Christopher2002] *Python Programming Patterns*. Thomas W Christopher. 0-13-040956-1. Prentice Hall. 2002.

[Deitel2002] *Python*. How to program. H. M. Deitel, P. J. Deitel, J. P. Liperi, and B. A. Wiederman. 0-13-092361-3. Prentice Hall. 2002.

[Downey2002] *How to Think Like a Computer Scientist*. Learining with Python. Allen Downey, Jeffrey Elkner, and Chris Meyers. 0-97167775-0-6. Green Tea Press. 2002. www.thinkpython.com [http://www.thinkpython.com].

[Grayson2000] *Python and Tkinter Programming*. John E. Grayson. 1-884777-81-3. Manning Publications Co.. 2000.

[LutzAscher2001] *Learning Python*. Mark Lutz and David Ascher. 1-56592-464-9. O'Reilly. 2001.

[Lutz2001] *Programming Python*. Mark Lutz. 0-596-00085-5. O'Reilly. 2001.

[Martelli2002] *Python Cookbook*. Alex Martelli and David Ascher. 0-596-00167-3. O'Reilly. 2002.

Other Books

[Meyer97] *Object Oriented software Construction*. Bertrand Meyer. 0-13-629155-4. Prentice Hall Professional Technical Reference. 1997.

[Booch94] *Object-Oriented Analysis and Design with Applications*. Grady Booch. Addison-Wesley Object Technology Series. 1994.

[Gamma95] *Design Patterns, Elements of Reusable Object-Oriented Software*. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Addison Wesley. 1995.

[Harrison97] *Tcl/Tk Tools*. Mark Harrison. O'Reilly. 1997.

Articles

[Wegner89] *BYTE Magazine*. “Learning the language”. Peter Wegner. 245-253. march, 1989.

[Mangalam2002] *The Bio* toolkits - a brief overview*. “Briefings in Bioinformatics”. Mangalam Harry. 296-302. 2002.

[Rossum99] *Computer Programming for Everybody*. Guido van Rossum. 1999.

[Wang2000] *A Qualitative Analysis of the Usability of Perl, Python, and Tcl*. *Proceedings of The Tenth International Python Conference*. Lingyun Wang and Phil Pfeiffer. 2002.

[Krasner88] *Journal of Object-Oriented Programming (JOOP)*. 1. 3. “A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80”. G Krasner and S Pope. 26-49. 1988.