Python course in Bioinformatics

by Katja Schuerer and Catherine Letondal

The picture above represents the 3D structure of the Human Ferroxidase [http://srs.ebi.ac.uk/srs6bin/cgilib/wgetz?-id+4SU6q1lomZ3+-e+[SWALL:`CERU_HUMAN'] protein, that we use in some of the exercises in this course.

This course is designed for biologists who already have some programming knowledge, in other languages such as perl or C. For this reason, while presenting a substantial introduction to the Python language, it does not constitute an introduction to programming itself (as [Tis2001] or our course in informatics for biology [http://www.pasteur.fr/formation/infobio/infobio-en.html], with an online programming course [http://www.pasteur.fr/formation/infobio/python/] in Python). What distinguishes this course from general Python introductory courses, is however the important focus on biological examples that are used throughout the course, as well as the suggested exercises drawn from the field of biology. The second half of the course describes the Biopython (http://www.biopython.org/) set of modules. This course can be considered a complement to the Biopython tutorial, and what’s more often refers to it, by bringing practical exercises using these components.

Contact: help@pasteur.fr [mailto:help@pasteur.fr]
Comments are welcome.

PDF version of this course [support.pdf]
Table of Contents

1. General introduction .................................................................................. 1
   1.1. Running Python .................................................................................. 1
   1.2. Documentation .................................................................................. 2
      1.2.1. General informations ................................................................. 2
      1.2.2. Getting information .................................................................... 2
      1.2.3. Making documentation .............................................................. 4
   1.3. Working environment .......................................................................... 4
      1.3.1. Emacs ......................................................................................... 4
2. Introduction to basic types in Python .......................................................... 7
   2.1. Strings ............................................................................................... 7
   2.2. Lists .................................................................................................. 10
   2.3. Tuples ............................................................................................... 13
   2.4. Sequence types: Summary .................................................................. 15
      2.4.1. Lists and Tuples ....................................................................... 16
      2.4.2. Xrange types .......................................................................... 16
      2.4.3. Strings and Unicode strings ...................................................... 16
      2.4.4. Buffers ...................................................................................... 17
   2.5. Dictionaries ....................................................................................... 17
   2.6. Numbers ........................................................................................... 20
   2.7. Type conversions .............................................................................. 21
   2.8. Files .................................................................................................. 23
      2.8.1. The print statement .................................................................... 25
3. Syntax rules ............................................................................................... 27
   3.1. Indentation ......................................................................................... 27
      3.1.1. Line structure .......................................................................... 27
      3.1.2. Block structure ........................................................................ 27
   3.2. Special objects ................................................................................... 28
4. Variables and namespaces ......................................................................... 31
   4.1. Variables ........................................................................................... 31
      4.1.1. Multiple assignments ................................................................ 32
   4.2. Assignments, references and copies of objects .................................... 32
   4.3. Namespaces ...................................................................................... 34
      4.3.1. Accessing namespaces ............................................................... 35
5. Control flow ............................................................................................. 39
   5.1. Conditionals ..................................................................................... 39
   5.2. Loops ............................................................................................... 40
      5.2.1. while ....................................................................................... 40
      5.2.2. for ......................................................................................... 41
      5.2.3. More about loops ..................................................................... 41
6. Functions ................................................................................................. 45
   6.1. Some definitions ............................................................................... 45
   6.2. Operators ......................................................................................... 46
      6.2.1. Order of evaluation .................................................................. 46
# Table of Contents

1. **Introduction**  
   1.1. Getting started with Python  
   1.2. Installing Python

2. **Python Number crunching**  
   2.1. Basic types of numbers  
   2.2. Basic operations with numbers

3. **Python Data types**  
   3.1. Strings  
   3.2. Lists  
   3.3. Tuples  
   3.4. Dictionaries  
   3.5. Sets  
   3.6. Numbers  
   3.7. Special variables  
   3.8. Functions  
   3.9. Iterators  
   3.10. Indexing

4. **Control Flow and Loops**  
   4.1. Control flow  
   4.2. Loops  
   4.3. Exceptions

5. **Functions**  
   5.1. Basic functions  
   5.2. Advanced functions  
   5.3. Decorators

6. **Classes and Objects**  
   6.1. Basic class definition  
   6.2. Defining functions  
   6.3. Defining constructors  
   6.4. Defining methods  
   6.5. Inheritance  
   6.6. Variable number of parameters  
   6.7. Default values of parameters  
   6.8. Passing arguments to parameters

7. **Modules and packages**  
   7.1. Modules  
   7.2. Packages

8. **Exceptions**  
   8.1. General Mechanism  
   8.2. Python built-in exceptions  
   8.3. Raising exceptions  
   8.4. Defining exceptions  
   8.5. Handling exceptions

9. **Operands and operators**  
   9.1. Operators  
   9.2. Expressions

10. **Biopython: Introduction**  
    10.1. Introduction  
    10.2. Getting information on a class  
    10.3. Creating instances  
    10.4. Using classes

11. **Biopython: Working with sequences**  
    11.1. Introduction  
    11.2. Documentation  
    11.3. Bio.Seq and Bio.SeqRecord modules  
    11.4. Bio.SwissProt.SProt and Bio.WWW.ExPASy

12. **Biopython: Working with databases**  
    12.1. Introduction  
    12.2. Getting information on a class  
    12.3. Creating instances  
    12.4. Using classes  
    12.5. Using databases

13. **Biopython: Running bioinformatics programs**  
    13.1. Introduction  
    13.2. Running programs  
    13.3. Running other programs

14. **Biopython: Advanced topics**  
    14.1. Introduction  
    14.2. Advanced topics

15. **Glossary**

---

6.2.2. Object comparisons .................................................. 47  
6.2.3. (dot) operator .......................................................... 47  
6.2.4. String formatting .................................................... 47  
6.3. Defining functions ...................................................... 48  
6.4. Passing arguments to parameters .................................... 49  
6.4.1. Reference arguments ................................................. 49  
6.4.2. Passing arguments by keywords .................................... 50  
6.5. Default values of parameters ........................................ 51  
6.6. Variable number of parameters ...................................... 52  
7. Functional programming or more about lists .......................... 57  
8. Exceptions ........................................................................... 59  
8.1. General Mechanism ....................................................... 59  
8.2. Python built-in exceptions ............................................. 59  
8.3. Raising exceptions ....................................................... 60  
8.4. Defining exceptions ....................................................... 61  
9. Modules and packages ....................................................... 63  
9.1. Modules ........................................................................... 63  
9.1.1. Where are the modules? ................................................ 63  
9.1.2. Loading ....................................................................... 64  
9.2. Packages ........................................................................... 66  
9.2.1. Loading ....................................................................... 67  
10. Classes: Using classes ...................................................... 71  
10.1. Creating instances ....................................................... 71  
10.2. Getting information on a class ....................................... 72  
11. Biopython: Introduction ................................................... 75  
11.1. Introduction .................................................................... 75  
11.2. Documentation ................................................................ 75  
11.3. Bio.Seq and Bio.SeqRecord modules ................................ 76  
11.3.1. Using Seq class ......................................................... 77  
11.3.2. Sequences reading and writing ..................................... 77  
11.3.3. Bio classes for sequences .......................................... 78  
11.4. Bio.SwissProt.SProt and Bio.WWW.ExPASy ......................... 83  
11.4.1. Reading entries ......................................................... 83  
11.4.2. Regular expressions in Python ..................................... 84  
11.4.3. Prosite ...................................................................... 86  
11.5. Bio.GenBank .................................................................. 88  
11.5.1. Reading entries .......................................................... 88  
11.6. Running Blast and Clustalw ............................................ 89  
11.6.1. Blast ...................................................................... 89  
11.6.2. Clustalw .................................................................. 91  
11.6.3. Running other bioinformatics programs under Pise .......... 94  
12. Classes: Defining a new class ............................................. 95  
12.1. Basic class definition .................................................... 95  
12.2. Defining operators for classes ....................................... 97  
12.3. Inheritance ................................................................. 100  
12.4. Classes variables .......................................................... 101
13. Biopython, continued ......................................................... 103
  13.1. Parsers ......................................................................... 103
    13.1.1. Introduction .......................................................... 103
    13.1.2. Exercises: building parsing classes for Enzyme .......... 106
    13.1.3. Iterator .............................................................. 107
    13.1.4. Exercises: building parsing classes for Enzyme (cont) .. 108
    13.1.5. Dictionary .......................................................... 109
    13.1.6. Using the parsers classes ....................................... 109
    13.1.7. Building parsing classes for phylogenetic trees .......... 109
  13.2. Practical: studying disulfid bonds in Human Ferroxidase 3D structure and alignments ... 111
    13.2.1. Working with PDB ............................................... ??
    13.2.2. Study of disulfid bonds ......................................... ??
14. Graphics in Python ................................................................. 117
  14.1. Tutorials ........................................................................ ??
  14.2. Software ........................................................................ ??
  14.3. Summary of examples and exercises with some graphics in this course .................... ??
A. Solutions ................................................................................ ??
  A.1. Introduction to basic types in Python .................................. 121
  A.2. Control Flow ................................................................... ??
  A.3. Functions ......................................................................... ??
  A.4. Modules and packages ..................................................... ??
  A.5. Biopython: Introduction .................................................. ??
    A.5.1. Bio.Seq package ....................................................... ??
    A.5.2. Bio.SwissProt.SProt and Bio.WWW.ExPASy ................... ??
    A.5.3. GenBank .................................................................. ??
    A.5.4. Blast .................................................................... ??
    A.5.5. Clustalw .................................................................. ??
  A.6. Classes ............................................................................ ??
  A.7. Biopython, continued ...................................................... ??
    A.7.1. Enzyme .................................................................... ??
    A.7.2. Building parsing classes for phylogenetic trees .......... ??
    A.7.3. PDB ................................................................... ??
B. Bibliography .............................................................................. ??
List of Figures

2.1. Diagram of some built-in types .................................................. 21
4.1. Assignment by referencing ....................................................... 33
4.2. Reference copy ........................................................................... 33
6.1. Referencing Arguments ................................................................. 49
8.1. Exceptions class hierarchy ......................................................... 59
9.1. Loading specific components ....................................................... 65
11.1. Overview of the Biopython course ............................................. 76
11.2. Seq, SeqRecord and SeqFeatures modules and classes hierarchies ........................................ 78
11.3. SeqRecord links to other classes ............................................... 80
13.1. Parsers class hierarchy ............................................................... 104
A.1. Plotting codons frequencies ....................................................... 132
A.2. Cys conserved positions ............................................................. 146
A.3. Biopython Alphabet class hierarchy .......................................... 151
# List of Tables

2.1. Built-in sequence types ......................................................... 15
2.2. Sequence types: Operators and Functions .................................. 15
2.3. Operations on mutable sequence types ...................................... 16
2.4. List methods ........................................................................ 16
2.5. Dictionary methods and operations ......................................... 18
2.6. Number built-in types .......................................................... 20
2.7. Type conversion functions ...................................................... 22
2.8. File methods ........................................................................ 23
2.9. File modes ........................................................................... 23
6.1. Order of operator evaluation (highest to lowest) ......................... 46
6.2. String formatting: Conversion characters .................................. 47
6.3. String formatting: Modifiers .................................................... 47
List of Examples

2.1. Introduction to strings ................................................................. 7
2.2. slices ...................................................................................... 7
2.3. Find substrings ....................................................................... 9
2.4. Introduction of lists .................................................................. 10
2.5. Functions returning a list .......................................................... 11
2.6. Generate all possible digests with two enzymes ....................... 12
2.7. Distance of two points in space ................................................. 14
2.8. Introduction to dictionaries ......................................................... 17
2.9. Protein 3-Letter-Code to 1-Letter-Code ..................................... 19
2.10. Calculation with complex numbers ........................................... 20
2.11. Reading Fasta ........................................................................ 24
2.12. Print statement ....................................................................... 26
3.1. None and pass ........................................................................ 28
4.1. Local variable definition ............................................................ 31
4.2. Global statement .................................................................... 31
4.3. ............................................................................................... 32
4.4. Assignment by referencing ....................................................... 32
4.5. Copy composed objects ............................................................. 33
4.6. Independent copy .................................................................... 34
4.7. Function execution namespaces ................................................ 35
5.1. Test the character of a DNA base .............................................. 39
5.2. More complex tests .................................................................. 39
5.3. Find all occurrences of a restriction site .................................... 40
5.4. Remove whitespace characters from a string ......................... 41
5.5. Find a unique occurrence of a restriction site ............................. 42
5.6. Find all possible start codons in a cds ...................................... 42
6.1. Differences between functions and procedures ....................... 45
6.2. Defining functions .................................................................. 48
6.3. Remove enzymes with ambiguous restriction patterns ................ 49
6.4. Passing arguments by keywords .............................................. 50
6.5. Default values of parameters .................................................... 51
6.6. Variable number of parameters ............................................... 52
6.7. Optional arguments as keywords .............................................. 53
8.1. Filename error ......................................................................... 59
8.2. Raising an exception in case of a wrong DNA character ........... 61
8.3. Raising your own exception in case of a wrong DNA character .... 61
8.4. Exceptions defined in Biopython .............................................. 62
9.1. A module ................................................................................ 63
9.2. Loading a module's components .............................................. 64
9.3. Using the Bio.Fasta package ..................................................... 67
11.1. Building Seq sequences from strings ....................................... 77
11.2. Reading a FASTA sequence with the Bio.Fasta package .......... 77
11.3. Reading a FASTA sequence with the Bio.Seqio.FASTA module ... 78
List of Exercises

2.1. GC content .......................................................... 7
2.2. DNA complement ..................................................... 7
2.3. Restriction site occurrences as a list ................................ 12
2.4. Restriction digest .................................................... 12
2.5. Get the codon list from a DNA sequence ......................... 13
2.6. Reverse Complement of DNA ....................................... 13
2.7. String methods ...................................................... 17
2.8. Translate a DNA sequence .......................................... 19
2.9. Operators ............................................................ 20
2.10. Write a sequence in fasta format ............................... 26
2.11. Header function .................................................... 26
5.1. Count ambiguous bases ............................................ 41
5.2. Check DNA alphabet ............................................... 42
6.1. DNA complement function ......................................... 48
6.2. Variable number of arguments ..................................... 53
9.1. Loading and using modules ......................................... 63
9.2. Creating a module for DNA utilities ............................... 64
9.3. Locating modules .................................................... 64
9.4. Locating components in modules .................................. 66
9.5. Bio.Seq module ...................................................... 66
9.7. Using a class from a module ....................................... 68
9.8. Import from Bio.Clustalw .......................................... 68
11.1. Length of a Seq sequence ......................................... 77
11.2. GC content of a Seq sequence .................................... 77
11.3. Write a sequence in FASTA format ............................. 78
11.4. Code reading: Bio.sequiltis ....................................... 78
11.5. Random mutation of a sequence .................................. 81
11.6. Random mutation of a sequence: count codons frequency .... 82
11.7. Random mutation of a sequence: plot codons frequency .......... 83
11.8. Code reading: connecting with ExPASy and parsing SwissProt records 83
11.9. SwissProt to FASTA ............................................... 83
11.10. Fetch an entry from a local SwissProt database ................ 84
11.11. Enzymes referenced in a SwissProt entry ...................... 86
11.12. Print the pattern of a Prosite entry ............................ 87
11.13. Display the Prosite references of a SwissProt protein .......... 87
11.14. Search for occurrences of a protein PROSITE patterns in the sequence 87
11.15. Extracting the complete CDS from a GenBank entry ............ 89
11.16. Local Blast, run and display results ........................... 89
11.17. Remote Blast, run and save results ............................ 90
11.18. Remote Blast, parse results ..................................... 90
11.19. Local PSI-Blast ................................................... 90
11.20. Search Prosite patterns with PHI-blast ........................ 90
Chapter 1. General introduction

1.1. Running Python

There are several ways to run Python code:

1. from the interpreter:

```python
>>> dna = 'gcatgacgttattacgactctgtcacgccgcggtgcgactgaggcgtggcgtctgctggg'
>>> dna
'gcatgacgttattacgactctgtcacgccgcggtgcgactgaggcgtggcgtctgctggg'
```

2. from a file:
   If file `mydna.py` contains:

   ```bash
   #!/local/bin/python
   dna = 'gcatgacgttattacgactctgtcacgccgcggtgcgactgaggcgtggcgtctgctggg'
   print dna
   ```
   it can be executed from the command line:

   ```bash
caroline:~> python mydna.py
gcatgacgttattacgactctgtcacgccgcggtgcgactgaggcgtggcgtctgctggg
   ```
   or using the `#!` notation:

   ```bash
caroline:~> ./mydna.py
gcatgacgttattacgactctgtcacgccgcggtgcgactgaggcgtggcgtctgctggg
   ```

   It is also possible to execute files during an interactive interpreter session:

   ```bash
caroline:~> python
   Python 2.2.1c1 (#1, Mar 27 2002, 13:20:02)
   [GCC 2.95.4 (Debian prerelease)] on linux2
   Type "help", "copyright", "credits" or "license" for more information.
   >>> execfile('mydna.py')
gcatgacgttattacgactctgtcacgccgcggtgcgactgaggcgtggcgtctgctggg
   ```

   or to load a file from the command line before entering Python in interactive mode (-i):
caroline:~> python -i mydna.py
gcatgacgttattacgactctgtcacgccgcggtgcgactgaggcgtggcgtctgctggg
>>>  
this is very convenient when your Python file contains definitions (functions, classes,...) that you want to test interactively.

3. from other programs embedding the Python interpreter:

```c
#include <Python.h>

int main(int argc, char** argv) {

    Py_Initialize();
    PyRun_SimpleString("dna = 'atgagag' + 'tagagga' ");
    PyRun_SimpleString("print 'Dna is:', dna ");

    return 0;
}
```

1.2. Documentation

1.2.1. General informations

General informations about Python and BioPython can be found:

1.2.2. Getting information

There are several ways to obtain documentation within the Python environment:

- from the command line using the **pydoc** command
- by the **help()** function during an interactive interpreter session

The **pydoc** command and the **help()** function provided with a string argument search the PYTHONPATH for an object of this name. But the **help()** function can also be applied directly on an object.

```python
>>> def ambiguous_dna_alphabet():
...     " returns a string containing all ambiguous dna bases "
...     return "bdhkmnrsuvwxy"
...>
>>> help('ambiguous_dna_alphabet')
no Python documentation found for 'ambiguous_dna_alphabet'

>>> help(ambiguous_dna_alphabet)
Help on function ambiguous_dna_alphabet in module __main__:

ambiguous_dna_alphabet()
    returns a string containing all ambiguous dna bases

ambigous_dna_alphabet is not defined in a module on the PYTHONPATH.
```

- by the function **dir(obj)** which displays the names defined in the local namespace (see Section 4.3.1) of the object **obj**. If no argument is specified **dir** shows the definitions of the current module.

```python
>>> dir()
['__builtins__', '__doc__', '__name__']

>>> dna = 'atgacgatagacataga'
>>> dir(dna)
['__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', 'rstrfind', '
split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', '
'translate', 'upper']
```
1.2.3. Making documentation

If the first statement of a module, class or function is a string, it is used as the documentation which can be accessed by the \_\_doc\_ attribute of the object. The \_\_doc\_ attribute contains the raw documentation string whereas the help() function prints it in a human readable format.

>>> ambiguous_dna_alphabet.__doc__
' returns a string containing all ambiguous dna bases '

>>> help(ambiguous_dna_alphabet)
Help on function ambiguous_dna_alphabet in module __main__:
ambiguous_dna_alphabet()
    returns a string containing all ambiguous dna bases

If a string is enclosed by triple quotes or triple double-quotes it can span several lines and the line-feed characters are retained in the string.

1.3. Working environment

1.3.1. Emacs

Python provides an editing mode for emacs, which will be automatically loaded if the following lines are present in the .emacs file.

```
(autoload 'python-mode "python-mode" "Python editing mode." t)
(setq auto-mode-alist
    (cons '\(\"\.*py\"\) . python-mode) auto-mode-alist)
(setq interpreter-mode-alist
    (cons '\("python\" . python-mode
            interpreter-mode-alist))
```

Within this emacs mode, from the "Python" menu, you can start an interactive interpreter session or (re)execute the python buffer, functions and classes definitions.
Chapter 1. General introduction

⚠️ Important

The python-mode is very useful because it resolves indentation problems occurring if tab and space characters are mixed (see Section 3.1.2).

⚠️ Caution

You can copy-paste a block of correct indented code into an interactive interpreter session. But take care, that the block does not contain empty lines.
Chapter 1. General introduction
Chapter 2. Introduction to basic types in Python

2.1. Strings

We are going to start the introduction to strings with some examples of DNA manipulations. Execute the following lines in the Python interpreter and look at the results:

Example 2.1. Introduction to strings

```python
>>> dna = 'gcatgacgttattacgacctctgtcagcgccggtgcgtgcgactgaggcgcgtgcgctg'  
>>> dna
'gcatgacgttattacgacctctgtcagcgccggtgcgtgcgactgaggcgcgtgcgctg'  
>>> dnasuite = 'cctttacttcgcctccgccctgcattccgttcctggcctcg'  
>>> dna = dna + dnasuite  
>>> dna
'gcatgacgttattacgacctctgtcagcgccggtgcgtgcgactgaggcgcgtgcgctgcctttacttcgcctccgccctgcattccgttcctggcctcg'  
>>> from string import *  
>>> len(dna)
103  
>>> 'n' in dna
0  
>>> count(dna, 'a')
10  
>>> replace(dna, 'a', 'A')
'gcAtgAcgttAttAcgActtctgtcAcgcccggtgcgActgAggcgtggtgcgactgtgcgtggtgcgcctttAgtcgcctccgcctgcAttcggttctgcgcctg'  
❶ This will be explained later (Section 9.1).
```
Exercise 2.1. GC content

Calculate the GC percent of dna. (Solution A.1)

Exercise 2.2. DNA complement

Calculate the complement of dna (Solution A.2).

Go to

See Section 6.2 and work on Section 6.3 before you continue here.

The following syntax enables the access of subparts of strings:

Example 2.2. slices

```python
>>> EcoRI = 'gaattc'
>>> EcoRI[0]
'g'
>>> EcoRI[-1]
'c'

>>> EcoRI[1:3]
'aa'
>>> EcoRI[3:]
'ttc'
>>> EcoRI[:]
'gaattc'
>>> EcoRI[:1]
'gaatt'

Caution

If one of the start or end specification of a slice is out of range it is ignored. The result is empty if both are
out of range or incompatible with each other.

Negative indices access strings from the end.
Caution

Positive numbering starts with 0 but negative numbering with -1.

The next example searches for non ambiguous restriction sites:

Example 2.3. Find substrings

```python
>>> dna = """"ttcacctatgaatggactgtcccccaagaagttaggacccacactacgagatctcttggtgtctagtaagatgttctctgtggatcttttagaatcagaggtgtatctccagaagatgtattacttcgtgctgagctgtgattattcccctcaagtttcggtctctcgagtctgattcaatgtgctgctattgtagtctgataacagctgtgctgcta"""

>>> EcoRI = 'gaattc'
>>> BamHI = 'ggatcc'
>>> HindIII = 'aagctt'
>>> find (dna, EcoRI)
186
>>> index (dna, EcoRI)
186

>>> find (dna, HindIII)
-1
>>> index (dna, HindIII)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    File "/usr/local/lib/python2.2/string.py", line 141, in index
      return s.index(*args)
  ValueError: substring not found in string.index

>>> find (dna, EcoRI, 187)
9
```
If no match is found find returns -1 whereas index produce an error (For more explanations on exceptions see Chapter 8).

How to find all sites for EcoRI?

Go to

Work on the exercises in Section 5.2 to answer this question.

2.2. Lists

Lists are arbitrary collections of objects that can be nested. They are created by enclosing the comma separated items in square brackets. As strings they can be indexed and sliced, but as opposite to strings, it is also possible to modify them.

Example 2.4. Introduction of lists

```python
>>> EcoRI = 'gaattc'
>>> BamHI = 'ggatcc'
>>> HindIII = 'aagctt'

>>> renz = [ EcoRI, BamHI, HindIII ]

>>> renz
['gaattc', 'ggatcc', 'aagctt']

>>> tree = [ 'Bovine', [ 'Gibbon', [ 'Orang', [ 'Gorilla', [ 'Chimp', 'Human' ]]]], 'Mouse' ]

>>> tree
['Bovine', ['Gibbon', ['Orang', ['Gorilla', ['Chimp', 'Human']]]], 'Mouse']

>>> digest = [ renz[0], renz[1] ]

>>> digest
['gaattc', 'ggatcc']


>>> digest
['gaattc', 'aagctt']

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

>>> del digest[1]
```
Chapter 2. Introduction to basic types in Python

>>> digest
['gaattc']

>>> digest = digest + renz[1:3]

>>> digest
['gaattc', 'ggatcc', 'aagctt']

>>> digest.append(EcoRI)

>>> digest
['gaattc', 'ggatcc', 'aagctt', 'gaattc']

>>> digest.pop()
'gaattc'

>>> digest
['gaattc', 'ggatcc', 'aagctt']

>>> digest.insert(2, 'ttcgaa')

>>> digest
['gaattc', 'ttcgaa', 'ggatcc', 'aagctt']

>>> digest.reverse()

>>> digest
['aagctt', 'ttcgaa', 'ggatcc', 'gaattc']

1. list creation
2. replace an element or a slice
3. deletion of an element
4. concatenation of two lists via the + operator.

Caution

This merges the two list whereas the method append() includes its argument in the list.
5. insertion of an element

Example 2.5. Functions returning a list

>>> range(3)
[0, 1, 2]

>>> range(10,20,2)
[10, 12, 14, 16, 18]

>>> range(5,2,-1)
[5, 4, 3]

>>> aas = "ALA TYR TRP SER GLY".split()

>>> aas
['ALA', 'TYR', 'TRP', 'SER', 'GLY']
Chapter 2. Introduction to basic types in Python

>>> " ".join(aas)
'ALA TYR TRP SER GLY'

>>> l = list('atgatgcaccacgtacga')
['a', 't', 'g', 'a', 't', 'g', 'c', 'c', 'c', 'a', 'c', 'g',
 't', 'a', 'c', 'g', 'a']

The next example generates all possibilities of digests using two enzymes from a list of enzymes. It is more complex and use a nested list and the range function introduced above (Example 2.5).

**Example 2.6. Generate all possible digests with two enzymes**

def all_2_digests(enzymes):
    """ generate all possible digests with 2 enzymes ""

    digests = []
    for i in range(len(enzymes)):
        for k in range(i+1, len(enzymes)):
            digests.append( [enzymes[i], enzymes[k]] )
    return digests

    If the first statement of a function definition is a string, this string is used as documentation (see Section 1.2.3).

>>> all_2_digests(['EcoRI', 'HindIII', 'BamHI'])
[['EcoRI', 'HindIII'], ['EcoRI', 'BamHI'], ['HindIII', 'BamHI']]
Chapter 2. Introduction to basic types in Python

Exercise 2.3. Restriction site occurrences as a list
Transform Example 5.3 that finds restriction sites to return a list containing all restriction site occurrences instead of printing them (Solution A.3).

Exercise 2.4. Restriction digest
Write a function that returns the length of all restriction fragments of a DNA sequence and that takes a list of restriction enzymes and the DNA sequence. (Solution A.4)

Tip
For each enzyme you need two informations, the restriction pattern and the position where the enzyme cuts its pattern. You can model an enzyme as a list containing this two informations, for example:

\[
\text{EcoRI} = [\text{'gaattc'}, 1]
\]

Tip
If you to do something with list, try to find out if there is a method of list objects that even implements your task. You can use the \text{dir} function to get all methods of a list object (Section 1.2.2).

Exercise 2.5. Get the codon list from a DNA sequence
Write a function that returns the list of codons for a DNA sequence and a given frame (Solution A.5).

Exercise 2.6. Reverse Complement of DNA
Write a function returning the reverse complement of a DNA. Remember Exercise 2.2 that calculates the complement of DNA. (Solution A.6)

Go to
Before you continue see Section 4.2 to get a deeper inside in variable assignments and read also Section 6.4 that explain how arguments can be passed to the function parameters.

2.3. Tuples
Tuples are like lists but they cannot be modified. Items have to be enclosed by parentheses instead of square brackets to create a tuple instead of a list. In general all that can be done using tuples can be done with lists, but sometimes it is more secure to prevent internal changes.

An appropriate use of tuples in a biological example could be the 3D-coordinates of an atom in a structure. The example calculates distances between atoms in protein structures. Atoms are represented as tuples of their coordinates x,y,z in space.

**Example 2.7. Distance of two points in space**

```python
from math import *

def distance(atom1, atom2):
    dx = atom1[0] - atom2[0]
    return sqrt(dx*dx + dy*dy + dz*dz)

>>> atom1 = (1.5, 2.0, 5.1)
>>> atom1
(1.5, 2.0, 5.0999999999999996)
>>> atom2 = (1.4, 4.6, 6.1)
>>> distance(atom1, atom2)
2.7874719729532704

but:

>>> atom1[0] = 1.0
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn’t support item assignment
```

**Caution**

When you create a tuple with only one value, a comma has to follow the value. This is necessary to make difference with parentheses that group expression. Look at the following example:

```python
>>> renz = ('EcoRI')
>>> renz
'EcoRI'
```
>>> renz = ('EcoRI',)
>>> renz
('EcoRI',)

Note

Tuples are used internally to pass arguments to the string format operator % (Section 6.2.4) and to pass a variable number of arguments to a function (Section 6.6).

Go to

Follow the last links in the note above to learn how you can pass a variable list of arguments to a function. You can also look at Section 4.1.1 which describes a special syntax of assignments using tuples.

2.4. Sequence types: Summary

Sequences hold ordered sets of objects. In the first three sections of this chapter we have introduced strings, lists and tuples. Table 2.1 completes the list of built-in sequences types of Python. Table 2.2 gives a list of operators and functions which can be applied to all sequence types. Table 2.3 gives the additional manipulation possibilities of mutable sequence types.

Table 2.1. Built-in sequence types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Elements</th>
<th>Mutable</th>
</tr>
</thead>
<tbody>
<tr>
<td>StringType</td>
<td>Character string</td>
<td>Characters only</td>
<td>no</td>
</tr>
<tr>
<td>UnicodeType</td>
<td>Unicode character string</td>
<td>Unicode characters only</td>
<td>no</td>
</tr>
<tr>
<td>ListType</td>
<td>List</td>
<td>Arbitrary objects</td>
<td>yes</td>
</tr>
<tr>
<td>TupleType</td>
<td>Immutable List</td>
<td>Arbitrary objects</td>
<td>no</td>
</tr>
<tr>
<td>XRangeType</td>
<td>return by xrange()</td>
<td>Integers</td>
<td>no</td>
</tr>
<tr>
<td>BufferType</td>
<td>Buffer, return by buffer()</td>
<td>arbitrary objects of one type</td>
<td>yes/no</td>
</tr>
</tbody>
</table>

Table 2.2. Sequence types: Operators and Functions

<table>
<thead>
<tr>
<th>Operator/Function</th>
<th>Action</th>
<th>Action on Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[ ... ]</code>, <code>( ... )</code>, <code>&quot;</code></td>
<td>creation</td>
<td></td>
</tr>
<tr>
<td><code>...&quot;</code></td>
<td>concatenation</td>
<td>addition</td>
</tr>
<tr>
<td><code>s + t</code></td>
<td>repetition</td>
<td>multiplication</td>
</tr>
<tr>
<td><code>s * n</code></td>
<td>indexation</td>
<td></td>
</tr>
<tr>
<td><code>s[i]</code></td>
<td>slice</td>
<td></td>
</tr>
<tr>
<td><code>s[i:k]</code></td>
<td>membership</td>
<td></td>
</tr>
<tr>
<td><code>x in s</code></td>
<td>iteration</td>
<td></td>
</tr>
<tr>
<td><code>x not in s</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>for a in s</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>len(s)</code></td>
<td></td>
<td>length</td>
</tr>
</tbody>
</table>
Chapter 2. Introduction to basic types in Python

\[ \text{min}(s) \quad \text{return smallest element} \]
\[ \text{max}(s) \quad \text{return greatest element} \]

Important

shallow copy (see Example 4.5)

Table 2.3. Operations on mutable sequence types

<table>
<thead>
<tr>
<th>Operator/Function</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>[i] = x</td>
<td>index assignment</td>
</tr>
<tr>
<td>[i:k] = t</td>
<td>slice assignment</td>
</tr>
<tr>
<td>del [i]</td>
<td>deletion</td>
</tr>
</tbody>
</table>

2.4.1. Lists and Tuples

Lists and tuples are collections of objects. They can hold different sort of object and they can be nested to organise the objects. The main difference between them is that list can be modified whereas tuples can not. Table 2.4 contains a summary list of list and tuple methods.

Table 2.4. List methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>[] (s)</td>
<td>converts any sequence object to a list</td>
</tr>
<tr>
<td>s.append(x)</td>
<td>append a new element</td>
</tr>
<tr>
<td>s.extend(t)</td>
<td>concatenation</td>
</tr>
<tr>
<td>s.count(x)</td>
<td>count occurrences of (x)</td>
</tr>
<tr>
<td>s.index(x)</td>
<td>find smallest position where (x) occurs in (s)</td>
</tr>
<tr>
<td>s.pop(i)</td>
<td>removes (i)-th element and return it</td>
</tr>
<tr>
<td>s.remove(x)</td>
<td>remove element</td>
</tr>
<tr>
<td>s.reverse()</td>
<td>reverse</td>
</tr>
<tr>
<td>s.sort([cmp])</td>
<td>sort according to the (cmp) function</td>
</tr>
</tbody>
</table>

\(^a\)equal to the + operator

\(^b\)in place operation

2.4.2. Xrange types

The range([start[, end[, stride]]]) function creates a list of integers from optional start to end with the optional stride (see Example 2.5 for an example). The xrange([start[, end[, stride]]) function, rather than creating a list containing all values, returns an immutable sequence object that calculates the value when needed. This saves memory for long sequences.

Xrange objects has only the method tolist() that returns a list containing all values.
Chapter 2. Introduction to basic types in Python

2.4.3. Strings and Unicode strings

Strings and Unicode strings are immutable collections of characters. They can be inclosed by quotes, double-quotes and triple-(double)-quotes. In double-quoted strings special characters are expanded and triple-quoted strings can span multiple lines. The line-feed character is retained in the last case.

```python
>>> mydoc="""This is a doc string, 
... spanning 2 lines."""
>>> mydoc
'This is a doc string, 
spanning 2 lines.'
```

Exercise 2.7. String methods

Find all methods of a string object.

They have a special the operator `%` (modulo) to format them. (remember Section 6.2.4).

2.4.4. Buffers

Buffers are sequence interfaces to a memory region that treats each byte as a 8-bit character. They can be created by the `buffer(obj [, offset] [, size])` function and share the same memory as the underlying object `obj`. This is an type for advanced use, so we will not say more about them.

2.5. Dictionaries

Dictionaries are collections of objects that are accessed by a key. They are created using a comma separated list of key-value pairs separated by colon enclosed in braces. Example 2.8 shows some examples of dictionary manipulation and Table 2.5 provides an overview of dictionary methods.

Example 2.8. Introduction to dictionaries

```python
... "ARG" : "R", "LYS" : "K", "MET" : "M", "CYS" : "C", 
... "TYR" : "Y", "THR" : "T", "PRO" : "P", "SER" : "S", 
... "TRP" : "W", "ASP" : "D", "GLU" : "E", "ASN" : "N", 
... "GLN" : "Q", "PHE" : "F", "HIS" : "H", "VAL" : "V"}
```

```python
>>> code['VAL']
'V'
>>> code.has_key('NNN')
0
>>> code.keys()
['CYS', 'ILE', 'SER', 'GLN', 'LYS', 'ASN', 'PRO', 'THR', 'PHE', 'ALA', 'HIS', 'GLY', 'ASP', 'LEU', 'ARG', 'TRP', 'VAL', 'GLU', 'TYR', 'MET']
```
Chapter 2. Introduction to basic types in Python

>>> code.values()

>>> code.items()

>>> del code['CYS']

>>> del code['MET']

>>> code

>>> code.update({'CYS':'C', 'MET':'M', '?':'?'})

>>> code

>>> one2three = {}

>>> for key,val in code.items():
... one2three[val] = key
... One2three={val: key}

>>> one2three
{'A': 'ALA', 'C': 'CYS', 'E': 'GLU', 'D': 'ASP', 'G': 'GLY', 'F': 'PHE', 'I': 'ILE', 'H': 'HIS', 'K': 'LYS', 'M': 'MET', 'L': 'LEU', 'N': 'ASN', 'Q': 'GLN', 'P': 'PRO', 'S': 'SER', 'R': 'ARG', 'T': 'THR', 'W': 'TRP', 'V': 'VAL', 'Y': 'TYR', '?': '?'}

<table>
<thead>
<tr>
<th>Method or Operation</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>d[key]</td>
<td>get the value of the entry with key key in d</td>
</tr>
<tr>
<td>d[key] = val</td>
<td>set the value of entry with key key to val</td>
</tr>
<tr>
<td>del d[key]</td>
<td>delete entry with key key</td>
</tr>
<tr>
<td>d.clear()</td>
<td>removes all entries</td>
</tr>
<tr>
<td>len(d)</td>
<td>number of items</td>
</tr>
<tr>
<td>d.copy()</td>
<td>makes a shallow copy of d</td>
</tr>
<tr>
<td>d.has_key(key)</td>
<td>returns 1 if key exists, 0 otherwise</td>
</tr>
<tr>
<td>d.keys()</td>
<td>gives a list of all keys</td>
</tr>
<tr>
<td>d.values()</td>
<td>gives a list of all values</td>
</tr>
<tr>
<td>d.items()</td>
<td>returns a list of all items as tuples (key,value)</td>
</tr>
<tr>
<td>d.update(new)</td>
<td>adds all entries of dictionary new to d</td>
</tr>
<tr>
<td>d.get(key [, otherwise])</td>
<td>returns value of the entry with key key if it exists otherwise returns otherwise</td>
</tr>
</tbody>
</table>

Table 2.5. Dictionary methods and operations

18
Chapter 2. Introduction to basic types in Python

```python
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
```

• Important

shallow copy (see Example 4.5)

Example 2.9. Protein 3-Letter-Code to 1-Letter-Code

```python
def three2one(prot, sep=None):
    """ translate a protein sequence from 3 to 1 letter code
    sep - separator if not one of the whitespace characters
    """
    code = {'GLY': 'G', 'ALA': 'A', 'LEU': 'L', 'ILE': 'I',
            'ARG': 'R', 'LYS': 'K', 'MET': 'M', 'CYS': 'C',
            'TYR': 'Y', 'THR': 'T', 'PRO': 'P', 'SER': 'S',
            'TRP': 'W', 'ASP': 'D', 'GLU': 'E', 'ASN': 'N',
            'GLN': 'Q', 'PHE': 'F', 'HIS': 'H', 'VAL': 'V'}

    newprot = ''
    for aa in prot.split(sep):
        newprot += code.get(aa, '?')

    return newprot
```

This is an example of a default argument of a functional parameter.

It can be run as follow:

```none
>>> prot ="""GLN ALA GLN ILE THR GLY ARG PRO GLU TRP ILE TRP LEU
... ALA LEU GLY THR ALA LEU MET GLY LEU GLY THR LEU TYR
... PHE LEU VAL LYS GLY MET GLY VAL SER ASP PRO ASP ALA
... LYS LYS PHE TYR ALA ILE THR THR LEU VAL PRO ALA ILE"""
>>> three2one(prot)
'QAQITGRFEGIWLALGATLMSGTGLTYFLVKGVMVDPDKFYAIITLVPAl'
```
Go to

See Section 6.5 to learn more about default values of functional parameters.

Exercise 2.8. Translate a DNA sequence

Write a function that takes a cDNA sequence and a genetic code and that returns the translated protein sequence.
(Solution A.7)

Note

Local namespaces of objects, that contains their method and attribute definitions, are implemented as dictionaries (Section 4.3.1). Another internal use of dictionaries is the possibility to pass a variable list of parameters using keywords (Example 6.7).

Go to

Remember how to pass a variable number of arguments to a function (Section 6.6) and look how to do the same using keywords (Example 6.7).

2.6. Numbers

This section provides a short introduction to numbers in Python. Table 2.6 shows all built-in number types of Python and Example 2.10 shows an example of complex numbers which haves a built-in type in Python. Arithmetics in Python can be done as expected from pocket calculators.

Table 2.6. Number built-in types

<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>integers</td>
<td>10</td>
</tr>
<tr>
<td>long integers (“unlimited size”)</td>
<td>1000000000000000000L</td>
</tr>
<tr>
<td>floating point numbers (64-bit precision)</td>
<td>10.1</td>
</tr>
<tr>
<td>complex numbers</td>
<td>3+4j</td>
</tr>
</tbody>
</table>

Example 2.10. Calculation with complex numbers

```python
>>> (3+4j)
(3+4j)
>>> (3+4j) + (4+2j)
(7+6j)
>>> (3+4j).real
3.0
>>> (3+4j).imag
```

20
4.0

Exercise 2.9. Operators

Compare the behaviour of some operator application (see the operator list in Table 6.1) to numbers, strings and lists. Test at minimum:

• a + b
• a * b
• a % b

2.7. Type conversions
It is sometimes necessary to convert variables from one type into another. For example, if you need to change some of the characters of a string, you will have to transform the string in a mutable list. Likewise, see Solution A.1 where it was necessary to convert integers into floating point numbers. Table 2.7 provides the list of all possible type conversions.

**Table 2.7. Type conversion functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int(x [,base])</code></td>
<td>converts $x$ to an integer</td>
</tr>
<tr>
<td><code>long(x [,base])</code></td>
<td>converts $x$ to a long integer</td>
</tr>
<tr>
<td><code>float(x)</code></td>
<td>converts $x$ to a floating-point number</td>
</tr>
<tr>
<td><code>complex(real [,imag])</code></td>
<td>creates a complex number</td>
</tr>
<tr>
<td><code>str(x)</code></td>
<td>converts $x$ to a string representation</td>
</tr>
</tbody>
</table>
Chapter 2. Introduction to basic types in Python

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

Go to
Read Section 4.3 to get a deeper inside into Python namespaces.

2.8. Files

The open(<filename>, [<mode>]) function opens a file with the specified access rights (see Table 2.9) and returns a FileType object. Table 2.8 list some of the methods available for FileType objects.

Table 2.8. 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
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

*aSee Section 2.4.2 for more informations

Table 2.9. File modes
Mode | Description
---|---
r | read
w | write
a | append
[rwa]b | [reading,writing,append] as binary data (required on Windows)
Chapter 2. Introduction to basic types in Python

Example 2.11. Reading Fasta

This example shows how to read sequence entries from a fasta file (data/seqs.fasta). You first have the format independent main loop of the program that reads the file sequence by sequence. The command line has to be replaced by instructions that do what should be done.

```python
f = open("seq.fasta")
entry = get_fasta(f)
while entry:
    # ... do what you have to do
    entry = get_fasta(f)

f.close()
```

1. Open the sequence file.
2. Loop over the file reading entry by entry and doing what you want to do.
3. Close the file.

The second part shows the code of the function `get_fasta` that reads one sequence from a fasta file.

Reading fasta files is not as simple as reading files in other sequence formats, because there is no explicit end of a sequence entry. You have to read the start of the following entry to know that your sequence is finished. The following shows two possibilities to handle this problem while reading the file line per line:

The first solution stores the line read too far:

```python
$header = None
def get_fasta(fh):
    """ read a fasta entry from a file handle """
    global _header
    if _header:
        header, _header = _header, None
    else:
        header = fh.readline()
        # end of file detection
        if not header:
            return header
        if header[0] != '>':
            return None
```
seq = ""
line = fh.readline()
while line and line[0] != ’>’:
    seq += line[:-1]
    line = fh.readline()
_header = line

return header[1:-1], seq

By default all variables are defined in the local namespace. Before looking at the second solution of the
problem, read Section 4.1 for how to differentiate between local and global variables.

The second possibility seeks the current file position to the start of the new entry, before returning the sequence.
So all but the first header lines are read twice:

def get_fasta(fh):
    """ read a fasta entry from a file handle """
    header = fh.readline()
    # eof detection
    if not header:
        return header
    # no fasta format
    if header[0] != ’>’:
        return None
    seq = ""
    line = fh.readline()
    while line:
        if line[0] == ’>’:
            # go back to the start of the header line
            fh.seek(-len(line), 1)
            break
        seq += line[:-1]
    line = fh.readline()

    return header[1:-1], seq

2.8.1. The print statement

All FileType objects have a write method to write strings to them. But sometimes the print statement can
be more conveniently used.
By default `print` writes the given string to the standard output and adds a line-feed. If a comma separated list of strings is given, then all strings will be joined by a single whitespace before printing. The addition of a trailing comma prevents the line-feed, but in this case a final whitespace is added.

**Example 2.12. Print statement**

```python
>>> renz = ['gaattc', 'ggatcc', 'aagctt']
>>> print renz
['gaattc', 'ggatcc', 'aagctt']
>>> print renz[0]
gaattc

>>> print "EcoRI pattern:", renz[0]
EcoRI pattern: gaattc
>>> print "EcoRI pattern: %s" % renz[0]
EcoRI pattern: gaattc

>>> for enz in renz:
...     print enz,
...

>>> log = open("log", "a")
>>> print >>log, "Handle restriction site:", renz[0]
>>> log.close()
```

The default destination can be redirected using the special `>>file` operator where `file` is the destination `FileType` object.

**Exercise 2.10. Write a sequence in fasta format**

Write a function that takes a `file` object (such as the one opened by the `open` function), a sequence, its ID and description as arguments, and write the sequence to the file (Solution A.8).

**Tip**

It is better to exclude the `open` and `close` functions to be able to write more than one sequence to a file.

**Exercise 2.11. Header function**

Write a function that takes the header line of a fasta entry and that returns the ID and description of the sequence (Solution A.9).
Chapter 3. Syntax rules

3.1. Indentation

3.1.1. Line structure

In Python you normally have one instruction per line. Long instructions can span several lines using the line-continuation character “\". Some instructions, as triple quoted strings, list, tuple and dictionary constructors or statements grouped by parentheses do not need a line-continuation character. It is possible to write several statements on the same line, provided they are separated by semi-colons.

```python
>>> dna = dna + \\
... 'aaagagagat'
>>> dna
'ataaaaaaaaagtatgcgggcgggagaaagagagat'

>>> primers = ['aaaata', 'ggttgt']
>>> primers
['aaaata', 'ggttgt']

>>> dna += 'aaataggat'; primers += ['tttgtta']
>>> dna
'ataaaaaaaaagtatgcgggcgcgggcgggagaaagagagataaataggattttatataaaaaaaaagtatgcgggcgcgggcgggagaaagagagataaataggattttat'

3.1.2. Block structure

Blocks of code, as function bodies, loops or conditions, are identified by indentation. The indentation length of the first statement of a block is arbitrary, but all instructions of a block have to be indented the same.

⚠️ Caution

Do not mix tab and space characters. The indentation length is not the length you see in the buffer, but equal to the number of separation characters.

The python-mode of emacs deals with this issue: if you use tab characters, emacs will replace them by space characters.
A block of code is initiated by a colon character followed by the indented instructions of the block. A one line block can also be given one the same line as the colon character.

```python
>>> dna = 'ataaaaaaaagtatgcgggcgcgggcgcg'
>>> primer = 'tgctcgctc'
>>> if dna.find(primer):
...     'found'
... else:
...     'not found'
...     'found'

>>> if dna.find(primer): 'found'
... else: 'not found'
...     'found'

>>> if dna.find(primer):
...     found = 1
...     'found'
...     'found'

but:

```python
>>> if dna.find(primer):
...     found = 1
...     'found'
...     File "<string>", line 3
...     'found'
...     SyntaxError: invalid syntax
```

### 3.2. Special objects

None is the empty or null object. It is always false and has its own type, the NoneType

Statements such as: if, while and def require a block of code containing at least one instruction. If there is nothing to do in the block, just use the pass statement.

**Example 3.1. None and pass**

```python
>>> found = None

>>> if found:
...     pass
```
>>> if found:
... else:
  File "<string>", line 2
else:
  ^
IndentationError: expected an indented block

Return to the function definition section (Section 6.3).
Chapter 3. Syntax rules
Chapter 4. Variables and namespaces

4.1. Variables

Variables have a type but are never declared in Python. They are instantiated when they are assigned for the first time. By default, variables are defined in the local namespace, or have to be declared explicitly as global variables, using the `global` statement.

⚠️ Caution

The first assignment of a value stands for the variable declaration. If a value is assigned to a variable in a function body, the variable will be local, *even if there is a global variable with the same name, and this global variable has been used before the assignment.*

**Example 4.1. Local variable definition**

```python
>>> enz = []
>>> def add_enz(*new):
...    enz = enz + list(new)
...  
>>> add_enz('EcoRI')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 2, in add_enz
UnboundLocalError: local variable 'enz' referenced before assignment
```

This rule does not apply in the case of method calls. In the following example, the variable `enz` is only *used*, not *assigned*, even if `enz` is actually modified internally.

```python
>>> def add_enz(*new):
...    enz.extend(list(new))
>>> add_enz('EcoRI')
>>> enz
['EcoRI']
```

The `global` statement has to be used to declare `enz` as a global variable.

**Example 4.2. Global statement**

```python
>>> def add_enz(*new):
...    global enz
    ...    enz = enz + list(new)
    ...
```
>>> add_enz('BamHI', 'HindIII')
>>> enz
[ 'EcoRI', 'BamHI', 'HindIII']

4.1.1. Multiple assignments

The following example shows how to assign several variables in a single statement.

Example 4.3.

>>> (EcoRI, BamHI) = ('gaattc', 'ggatcc')
>>> EcoRI
'gaattc'
>>> BamHI
'ggatcc'

you can also omit the parentheses:

>>> EcoRI, BamHI = 'gaattc', 'ggatcc'

This is a convenient way to return multiple values from a function.

4.2. Assignments, references and copies of objects

Assignment $a = b$ creates a new reference to the content of $b$ and saves it in $a$. This means that $a$ and $b$ refer to the same object. If $b$ is a mutable object and one of his items is modified, $a$ will also change. Figure 4.1 illustrates Example 4.4 given below.
Example 4.4. Assignment by referencing

```python
>>> digest = ['EcoRI', 'HindIII']
>>> digest2 = digest
>>> digest2
['EcoRI', 'HindIII']
>>> digest2[1] = 'BamHI'
>>> digest2
['EcoRI', 'BamHI']
>>> digest
['EcoRI', 'BamHI']
```

Figure 4.1. Assignment by referencing

The same strategy is used for the copy of composed objects. A target object is created and populated by new references to the items of the source object. Figure 4.2 illustrates what happens in Example 4.5.

Example 4.5. Copy composed objects

```python
>>> firstserie = all_2_digests(['EcoRI', 'HindIII', 'BamHI'])
>>> firstserie
[['EcoRI', 'HindIII'], ['EcoRI', 'BamHI'], ['HindIII', 'BamHI']]
>>> newserie = firstserie[1:]
>>> newserie
[['EcoRI', 'BamHI'], ['HindIII', 'BamHI']]
>>> newserie[1][0]='SarI'
>>> newserie
[['EcoRI', 'BamHI'], ['SarI', 'BamHI']]
>>> firstserie
[['EcoRI', 'HindIII'], ['EcoRI', 'BamHI'], ['SarI', 'BamHI']]
```
4.3. Namespaces

Example 4.6. Independent copy

```python
>>> firstserie = all_2_digests(['EcoRI', 'HindIII', 'BamHI'])
>>> firstserie
[['EcoRI', 'HindIII'], ['EcoRI', 'BamHI'], ['HindIII', 'BamHI']]

>>> import copy
>>> newserie = copy.deepcopy(firstserie)[1:]
>>> newserie
[['EcoRI', 'BamHI'], ['HindIII', 'BamHI']]

>>> newserie[1][0]='SarI'
>>> newserie
[['EcoRI', 'BamHI'], ['SarI', 'BamHI']]

>>> firstserie
[['EcoRI', 'HindIII'], ['EcoRI', 'BamHI'], ['HindIII', 'BamHI']]```
Chapter 4. Variables and namespaces

There are three different namespaces in Python: a local namespace, a module namespace and a global namespace. The latter contains all built-in functions. The module namespace contains all the function definitions and variables of a module. It can be accessed using the . (dot) operator. A local environment is created at function calls. It includes all the parameters and local variables of the function. Function definitions can be nested, and nested functions have their own local namespace.

**Example 4.7. Function execution namespaces**

```python
>>> enz = []

>>> def add_enz(*new):
...     def verif():
...         print "enz: ", enz
...         print "new: ", new
...     verif()
...     enz.extend(list(new))

>>> add_enz('EcoRI')
enz: []
new: ('EcoRI',)

>>> enz
[ 'EcoRI' ]
```

**Caution**

This behaviour only exists in Python version 2.2. Previous versions have only one function execution namespace. In this case, the `new` variable in Example 4.7 is not accessible within the `verif` function.

4.3.1. Accessing namespaces

Variable names are resolved by searching the namespaces in the following order: local namespaces (function execution namespaces potentially nested), current module namespace and global namespace containing built-in definitions.

When object methods or attributes are addressed using the . (dot) operator, namespaces searching is different. Each object has its own local namespace implemented as a dictionary named `__dict__`. This dictionary is searched for the name following the . (dot) operator. If it is not found, the local namespace of its class, accessible via the `__class__` attribute, is searched for. If it is not found there, a lookup on the parent classes is performed. Since modules are objects, accessing the namespace of a module use the same mechanism.

```python
>>> enz = ['EcoRI']

>>> enz.__dict__
```

35
Traceback (most recent call last):
  File "<stdin>" , line 1, in ?
AttributeError: 'list' object has no attribute '__dict__'
>>> enz.__class__.__dict__
<dict-proxy object at 0x815776c>

>>> print enz.__class__.__dict__
{'sort': <method 'sort' of 'list' objects>,
 'ne': <slot wrapper '__ne__' of 'list' objects>,
 'reverse': <method 'reverse' of 'list' objects>,
 '__getslice__': <slot wrapper '__getslice__' of 'list' objects>,
 'insert': <method 'insert' of 'list' objects>,
 '__len__': <slot wrapper '__len__' of 'list' objects>,
 '__getattribute__': <slot wrapper '__getattribute__' of 'list' objects>,
 'remove': <method 'remove' of 'list' objects>,
 'append': <method 'append' of 'list' objects>,
 '__setitem__': <slot wrapper '__setitem__' of 'list' objects>,
 'pop': <method 'pop' of 'list' objects>,
 '__add__': <slot wrapper '__add__' of 'list' objects>,
 '__gt__': <slot wrapper '__gt__' of 'list' objects>,
 '__rmul__': <slot wrapper '__rmul__' of 'list' objects>,
 '__lt__': <slot wrapper '__lt__' of 'list' objects>,
 '__eq__': <slot wrapper '__eq__' of 'list' objects>,
 '__init__': <slot wrapper '__init__' of 'list' objects>,
 '__mul__': <slot wrapper '__mul__' of 'list' objects>,
 'extend': <method 'extend' of 'list' objects>,
 '__delitem__': <slot wrapper '__delitem__' of 'list' objects>,
 '__delattr__': <slot wrapper '__delattr__' of 'list' objects>,
 '__getitem__': <slot wrapper '__getitem__' of 'list' objects>,
 'contains': <slot wrapper '__contains__' of 'list' objects>,
 'index': <method 'index' of 'list' objects>,
 '__setslice__': <slot wrapper '__setslice__' of 'list' objects>,
 'count': <method 'count' of 'list' objects>,
 '__iadd__': <slot wrapper '__iadd__' of 'list objects>,
 '__le__': <slot wrapper '__le__' of 'list objects>,
 '__repr__': <slot wrapper '__repr__' of 'list objects>,
 '__hash__': <slot wrapper '__hash__' of 'list objects>,
 '__new__': <built-in method __new__ of type object at 0x80f1aa0>,
 '__doc__': "list() -> new list
list(sequence) -> new list
initialized from sequence's items",
'ge': <slot wrapper 'ge' of 'list objects>,
'__mul__': <slot wrapper '__mul__' of 'list objects>}

>>> dir (enz)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
 '__delr__': '__setslice__', '__doc__', '__eq__', '__ge__', '__getattribute__',
 '__getitem__', '__getslice__', '__gt__', '__hash__', '__iadd__',
 '__imul__', '__init__', '__le__', '__len__', '__lt__', '__mul__',
 '__ne__', '__new__', '__reduce__', '__repr__', '__rmul__', '__setattr__',
 '__setitem__', '__setslice__', '__str__', 'append', 'count', 'extend',

36
Chapter 4. Variables and namespaces

'index', 'insert', 'pop', 'remove', 'reverse', 'sort']

Go back

Return to Section 2.7.
Chapter 4. Variables and namespaces
Chapter 5. Control flow

5.1. Conditionals

The `if` statement and the optional `else` and `elif` statements perform tests.

Example 5.1. Test the character of a DNA base

```python
>>> base = "e"
>>> if base in "atgc":
...     "exact"
...     ...

>>> if base in "atgc":
...     "exact"
...     elif base in "bdhkmnrsuvwxy":
...     "ambiguous"
...     else:
...     "unknown"
...     ...
'unknown'
```

More complex tests can be written with the `and`, `or` and `not` operators.

Example 5.2. More complex tests

```python
>>> base in 'atgc'
0
>>> base not in 'atgc'
1
>>> not base in 'atgc'
1

>>> base.isalpha() ❶
1

>>> base.isalpha() and base in 'atgc'
0
>>> base.isalpha() or base.isspace() ❷
1

>>> not None
1
>>> not 0
1
>>> not "
```
Here we ask for the `isalpha` method of the string object `base` (see Section 6.2.3). The object `None` is the special “empty” object. It is always false. Some expressions that are false. A logical expression returns 0 if it is `false` and the value of the last evaluation otherwise.

The components of the logical expression are evaluated until the value of the entire expression is known. Here the expression `1/0` is not executed because 1 is `true` and so the entire expression is `true`.

5.2. Loops

The two statements `while` and `for` are used to write loops in Python.

5.2.1. `while`

The `while` construct executes a block of code while a condition is true.

Example 5.3. Find all occurrences of a restriction site

```python
from string import *

def restrict(dna, enz):
    "print all start positions of a restriction site"
    site = find (dna, enz)
    while site != -1:
        print "restriction site \%s at position \%d\" \ (enz, site)
        site = find (dna, enz, site + 1)
```
Chapter 5. Control flow

>>> restrict(dna, EcoRI)
restriction site gaattc at position 188
restriction site gaattc at position 886
restriction site gaattc at position 1326

5.2.2. for

The loop construct for iterates over all members of a sequence.

⚠️ Caution

This is equivalent to the foreach statement in some other programming languages. It is not the same as the for statement in most other programming languages.

Example 5.4. Remove whitespace characters from a string

>>> from string import *
>>> whitespace = '	
\0\0\r' >>> dna = "aaattcctgaga\n gccctg\ngaaatcc\ngtttct\nctg\n''
>>> for s in whitespace:
...    dna = replace(dna, s, '')
... >>> dna
'aaattcctgaga gccctg\ngaatcc
gtttct\ntg' Exercise 5.1. Count ambiguous bases

Write a function returning the number of ambiguous bases in a DNA sequence (Solution A.10).
5.2.3. More about loops

Python provides the following advanced features while executing a loop:

- to quit a loop before the end condition is true by using `break`
- to go directly to the next iteration step by using `continue`
- to execute code only if the loop was not interrupted with `break` by using the `else` statement following the `while` clause

⚠️ Caution

The `else` statement is also executed if the loop is not entered.

Example 5.5. Find a unique occurrence of a restriction site

```python
def restrict_uni(dna, enz):
    """ find unique restriction sites """
    found = None
    site = dna.find(enz)
    while site != -1:
        if found:
            break
        found = site
        site = dna.find(enz, found+1)
    else:
        if found is not None:
            return found ❶

❶ The test ensures that a restriction site occurrence at position 0 is also true.
```
Exercise 5.2. Check DNA alphabet

Write a loop to verify all bases in a DNA sequence. (Solution A.11).

Example 5.6. Find all possible start codons in a cds

def find_starts (cds):
    """ find start codons in a cds """
    start = -1
    while 1:
        start = cds.find("atg", start+1)

        if start == -1:
            break

        if start % 3:
            continue

        print "possible start codon at position %d" % start

The `continue` statement is used to skip all `atg` codons that are out of frame.

Go back

Return at the end of Section 2.1.
Chapter 6. Functions

6.1. Some definitions

Function
A function is a piece of code that performs a specific sub-task. It takes arguments that are passed to parameters (special place holders to customise the task) and returns a result.

Operator
An operator is a function that takes one or two arguments and that is invoked by the following syntax: \( \text{arg1 op arg2} \).

Note
Operators are defined by special methods in Python:

```python
>>> "atgacta" + "atgataga"
"atgactaatgataga"
```

Procedure
The terms "function" and "procedure" are often used as if they would be interchangeable. However, the role of a procedure is not to return a value, but to perform an action, such as printing something on the terminal or modifying data (i.e., something which is sometimes called "doing side-effects" in functional programming parlance).

Strictly speaking, the definition of a function is the same as the mathematical definition: given the same arguments, the result will be identical, whereas the behaviour of a procedure can vary, even if the task is invoked with the same arguments.

In Python, as in most programming languages, there is no difference in function and procedure definitions or calls. But if no return value is specified or if the return value is empty, then the empty object `None` is returned. It is important to know if the called function returns a result.
Example 6.1. Differences between functions and procedures

```python
>>> enznames = [ 'EcoRI', 'BamHI', 'HindIII' ]
```

```python
>>> enznames.index('BamHI')
1
```

```python
>>> enznames.reverse()
```

```python
>>> enznames
['HindIII', 'BamHI', 'EcoRI']
```

The `reverse()` method executes an inversion of the list `enzname`. It does it **inplace**, and does not construct a new list.

Method

A method is a function or procedure that is associated with an object. It executes a task an object can be asked for. In Python it is called via the `. (dot) operator.

```python
>>> dna='atgctcgctgc'
```

```python
>>> dna.upper()
'ATGCTCGCTGC'
```

6.2. Operators

6.2.1. Order of evaluation

Table 6.1 provides the precedence of Python operators. They are listed from the highest to the lowest priority. Operators listed on the same row have equal priority.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(..), [..], {..}, </code>'..`</td>
<td>Constructors</td>
</tr>
<tr>
<td>s[i], s[i:j], s.attr, f(...)</td>
<td>Indexing, slicing and function calls</td>
</tr>
<tr>
<td><code>+x, -x, ~x</code></td>
<td>Unary operators</td>
</tr>
<tr>
<td><code>x ** y</code></td>
<td>Power (right associative)</td>
</tr>
<tr>
<td><code>x * y, x / y, x % y</code></td>
<td>Multiplication, division, modulo</td>
</tr>
<tr>
<td><code>x + y, x - y</code></td>
<td>Addition, subtraction</td>
</tr>
<tr>
<td><code>x &lt;&lt; y, x &gt;&gt; y</code></td>
<td>Bit shifting</td>
</tr>
<tr>
<td><code>x &amp; y</code></td>
<td>Bitwise and</td>
</tr>
<tr>
<td>`x</td>
<td>y`</td>
</tr>
</tbody>
</table>

46
Chapter 6. Functions

\[ \begin{align*}
&x < y, x <= y, x > y, x >= y, x == y, \\
&x != y, x <> y, x is y, x is not y, x \\
in s, x not in s< \\
not x & \quad \text{Logical negation} \\
x and y & \quad \text{Logical and} \\
lambda args: expr & \quad \text{Anonymous function}
\end{align*} \]

6.2.2. Object comparisons

The == operator tests the equality of objects, whereas the is operator tests their identity. Two objects are identical if they refer to the same place in memory. For numbers and strings there is no difference in the result. List and tuples are equal if all their members are equal and dictionaries are equal if they have the same set of keys and the value of each key is also equal.

6.2.3. . (dot) operator

Everything in Python is an object, and the base types are implemented as classes. The . (dot) operator is used to ask an object to do something, or more formally to access its attributes and methods.

6.2.4. String formatting

The % (modulo) operator applied to strings formats them. Table 6.2 provides the characters that you can use in the formatting template and Table 6.3 gives the modifiers of the formatting character.

**Table 6.2. String formatting: Conversion characters**

<table>
<thead>
<tr>
<th>Formatting character</th>
<th>Output</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>d, i</td>
<td>decimal or long integer</td>
<td>&quot;%d&quot; % 10</td>
<td>'10'</td>
</tr>
<tr>
<td>o, x</td>
<td>octal/hexadecimal integer</td>
<td>&quot;%o&quot; % 10</td>
<td>'12'</td>
</tr>
<tr>
<td>f, e, E</td>
<td>normal, 'E' notation of floating point numbers</td>
<td>&quot;%e&quot; % 10.0</td>
<td>'1.000000e+01'</td>
</tr>
<tr>
<td>s</td>
<td>strings or any object that has a str() method</td>
<td>&quot;%s&quot; % [1, 2, 3]</td>
<td>'[1, 2, 3]'</td>
</tr>
<tr>
<td>r</td>
<td>string, use the repr() function of the object</td>
<td>&quot;%r&quot; % [1, 2, 3]</td>
<td>'[1, 2, 3]'</td>
</tr>
<tr>
<td>%</td>
<td>literal %</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 6.3. String formatting: Modifiers**

<table>
<thead>
<tr>
<th>Modifier name in parentheses</th>
<th>Action</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>-,-, +</td>
<td>left, right alignment</td>
<td>&quot;%-10s&quot; % &quot;dna&quot;</td>
<td>'dna_______'</td>
</tr>
</tbody>
</table>
6.3. Defining functions

Functions are defined with the `def` statement followed by the name of the function, and the parameter list in parentheses. The result of the calculation is returned by the `return` statement.

**Example 6.2. Defining functions**

The following example transforms Exercise 2.1, that calculates the GC percentage of a DNA sequence, into a function:

```python
>>> def gc(dna):
...     return (count(dna, 'c') + count(dna, 'g')) / float(len(dna)) * 100.0
...     
>>> gc('atgtaatgatat')
16.666666666666664
>>> gc(dna) ❷
64.077669902912632
```

❶ The Python interpreter displays two different kinds of prompts. The first `>>>` is the normal one. The second `...` indicates the continuation of a block.

⚠️ **Caution**

Although the name of the argument (`dna`) is the same as the name of the parameter, their values are not the same.
Go to

Read also Section 3.1.2 to learn more about Python syntax. You might need to read Section 5.1 as well to understand the examples given in the syntax section.

Exercise 6.1. DNA complement function

Write a function to calculate the complement of a DNA sequence. (Solution A.12)

Go back

Return to Section 2.1 to carry on with the introduction to strings.

6.4. Passing arguments to parameters

6.4.1. Reference arguments

When a function is invoked, a reference to the value of the argument is passed to the parameter.

Example 6.3. Remove enzymes with ambiguous restriction patterns

The following function removes all restriction enzyme patterns that contains ambiguous bases from a list.

```python
def remove_ambiguous_renz(Lenz):
    """ remove enzymes with ambiguous restriction patterns """
    for i in range(len(Lenz)):
        if not check_dna(Lenz[i]):
            del Lenz[i]
```

Figure 6.1 illustrates what happens when `remove_ambiguous_renz()` is invoked as follow:

```python
>>> renz = ['gaattc', 'ggatcc', 'aagctt', 'ggannntcc']
>>> remove_ambiguous_renz(renz)
>>> renz
['gaattc', 'ggatcc', 'aagctt']
```
6.4.2. Passing arguments by keywords

When a function is invoked with a tuple of arguments, they will be associated to parameters according to their position in the tuple. But it is also possible to pass arguments by keywords. This means that the arguments are assigned to parameters by explicitly naming them.

Example 6.4. Passing arguments by keywords

The following function constructs the command line for the blast program:

```python
def blast2(query, program, database):
    return "blastall -p %s -d %s -i %s" % (program, database, query)
```

The arguments can be passed by position:

```bash
>>> blast2("seq.fasta", "blastp", "swissprot")
'blastall -p blastp -d swissprot -i seq.fasta'
```

or by explicit naming:

```bash
>>> blast2(program='blastp', database='swissprot', query='seq.fasta')
'blastall -p blastp -d swissprot -i seq.fasta'
```

One advantage is that you do not have to know in what order parameters are declared in the function.

It is possible to mix the two mechanisms:
Chapter 6. Functions

>>> blast2("seq.fasta", program='blastp', database='swissprot')
'blastall -p blastp -d swissprot -i seq.fasta'

But arguments passed by position must be provided first:

>>> blast2("seq.fasta", program='blastp', 'swissprot')
File "<string>", line 1
  blast2("seq.fasta", program='blastp', 'swissprot')
    ^
SyntaxError: invalid syntax

6.5. Default values of parameters

Default values of parameters can be defined in the function definition.

Example 6.5. Default values of parameters

To use “blastp” and “swissprot” as default values for program and database parameters, the blast2() function can be redefined as follow:

```python
def blast2(query, program='blastp', database='swissprot'):
    return "blastall -p %s -d %s -i %s" % (program, database, query)
```

So, you can now call it this way:

```python
>>> blast2('seq.fasta')
'blastall -p blastp -d swissprot -i seq.fasta'
>>> blast2('seq.fasta', 'blastp', 'swissprot')
'blastall -p blastp -d swissprot -i seq.fasta'
>>> blast2('seq.fasta', database='nrprot')
'blastall -p blastp -d nrprot -i seq.fasta'
```
Default values are referenced when the function is defined.

⚠ **Caution**

Be careful if you pass mutable objects as default values. The content of the default value can be modified after function definition if there is also a global reference to it.

Redefinition of `blast2()` when `params` is defined as:

```python
params = { 'e': 1.0,
           'm': 8,
           'F': 'S 10 1.0 1.5' }
```

```python
def blast2(query, program='blastp', database='swissprot', params=params):
    command = "blastall -p %s -d %s -i %s" % (program, database, query)
    if params:
        for para, value in params.items():
            command += " -%s '%s'" % (para, value)
    return command
```

creates the following behaviour:

```python
>>> blast2('seq.fasta')
"blastall -p blastp -d swissprot -i seq.fasta -m '8' -e '1.0' -F 'S 10 1.0 1.5'"

>>> params['q']=-6

>>> blast2('seq.fasta')
"blastall -p blastp -d swissprot -i seq.fasta -q '-6' -m '8' -e '1.0' -F 'S 10 1.0 1.5'"
```

The default behaviour of the `blast2` function has been changed.

It’s risky to keep global references to default values: when using global variables, rather make a deep copy of the object (see Example 4.6).

### 6.6. Variable number of parameters

A function can take additional optional arguments by prefixing the last parameter with an * (asterix). Optional arguments are then available in the tuple referenced by this parameter.

**Example 6.6. Variable number of parameters**

```python
def multi_blast2 (query, program, database, *more_queries):
```
Chapter 6. Functions

for q in (query,) + more_queries:
    print blast2(q, program, database)

>>> multi_blast2('seq.fasta', 'blastp', 'database', 'seq2.fasta')
blastall -p blastp -d database -i seq.fasta
blastall -p blastp -d database -i seq2.fasta

❶ (query,) is a tuple of one element. The comma is necessary because (query) is the syntax to indicate precedence.

Exercise 6.2. Variable number of arguments

Transform Example 2.6 such that it can be applied as follow: (Solution A.13)

>>> all_2_digests('EcoRI', 'HindIII', 'BamHI')
[['EcoRI', 'HindIII'], ['EcoRI', 'BamHI'], ['HindIII', 'BamHI']]

instead of:

>>> all_2_digests(['EcoRI', 'HindIII', 'BamHI'])
[['EcoRI', 'HindIII'], ['EcoRI', 'BamHI'], ['HindIII', 'BamHI']]

Go back

Return to the end of the introduction to tuples (Section 2.3).
Optional variables can also be passed as keywords, if the last parameter is preceded by **. In this case, the optional variables are placed in a dictionary.

Example 6.7. Optional arguments as keywords

def blast2(query, program='blastp', database='swissprot', **params):
    command = "blastall -p %s -d %s -i %s" % (program, database, query)
    if params:
        for para, value in params.items():
            command += " -%s '%s'" % (para, value)
    return command

blast2() can now be invoked by:
Chapter 6. Functions

```python
>>> blast2('seq.fasta')
'blastall -p blastp -d swissprot -i seq.fasta'

>>> blast2('seq.fasta', m=8, e=1.0, F='S 10 1.0 1.5')
"blastall -p blastp -d swissprot -i seq.fasta -m '8' -e '1.0' -F 'S 10 1.0 1.5'"

You can also directly pass a dictionary as argument:

def blast2(query, program='blastp', database='swissprot', params=None):
    command = "blastall -p %s -d %s -i %s" % (program, database, query)
    if params:
        for para, value in params.items():
            command += " -%s '%s'" % (para, value)
    return command

Now pass the dictionary:

```python
>>> params = { 'e': 1.0, ...
...             'm': 8, ...
...             'F': 'S 10 1.0 1.5' }

>>> blast2('seq.fasta', params=params)
"blastall -p blastp -d swissprot -i seq.fasta -q '-6' -m '8' -e '1.0' -F 'S 10 1.0 1.5'"
```

As for required arguments, you can mix positional and keyword based assignment for optional arguments.

```python
def multi_blast2(query, *more_queries, **params):
    database = params.get('database', 'swissprot')
    program = params.get('program', 'blastp')
    for q in (query,) + more_queries:
        print blast2(q, program, database, params)

Invoked as:

```python
>>> multi_blast2 ('seq.fasta', 'seq2.fasta', m=8, e=1.0, F="")
"blastall -p blastp -d swissprot -i seq.fasta -m '8' -e '1.0' -F '"S 10 1.0 1.5"'
"blastall -p blastp -d swissprot -i seq2.fasta -m '8' -e '1.0' -F '"S 10 1.0 1.5""
```

54
Go back

Return to the end of the introduction to dictionaries and carry on with the next section (Section 2.6).
Chapter 7. Functional programming or more about lists

⚠️ Caution

This chapter is under construction.
Chapter 7. Functional programming or more about lists
Chapter 8. Exceptions

8.1. General Mechanism

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

Example 8.1. Filename error

```python
g>>> 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:

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

Variable e contains the cause of the error:

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

8.2. Python built-in exceptions
Python predefines several exceptions (Figure 8.1).

**Figure 8.1. 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 inappropriate value to an operation
- **TabError, IndentationError**: two kinds of **SyntaxError**
Chapter 8. Exceptions

8.3. Raising exceptions

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

```python
if something_wrong:
    raise Exception
```

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

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

Example 8.2. Raising an exception in case of a wrong DNA character

```python
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
```

8.4. 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 8.1).

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 12).

Example 8.3. 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.

```python
class AlphabetError(ValueError):
    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

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: is just a new class name.

Example 8.4. Exceptions defined in Biopython

Some Biopython modules define their own exceptions, such as:

• ParserFailureError (GenBank package)
• BadMatrix (SubsMat package)
Chapter 9. Modules and packages

9.1. Modules

A module is a piece of code contained in a file. For instance, if the file `ValSeq.py` contains the following code (adapted from Biopython module `NBRF.ValSeq`):

Example 9.1. A module

```python
# 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:

```python
import ValSeq
```

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

```python
>>> print ValSeq.valid_sequence_dict[‘RL’]
linear RNA
>>> ValSeq.find_valid_key("linear RNA")
RL
```

Exercise 9.1. Loading and using modules

Write the code needed to print the command line arguments of a program, by using the `sys` module and its `argv` variable (Solution A.14).

9.1.1. Where are the modules?

Modules are mainly stored in files that are searched:
• in PYTHONHOME, where Python has been installed (at Pasteur it is currently /local/lib/python2.2/),
• in a path, i.e a colon (':') separated list of file pathes, stored in the environment variable PYTHONPATH.

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 9.2. Creating a module for DNA utilities

Create a module dna containing the functions defined on DNA in previous exercises: complement (Exercise 6.1), ambiguous (Exercise 5.1), check_dna (Exercise 5.2), restrict (Exercise 2.3), digest and frag_len (Exercise 2.4), revcomp (Exercise 2.6), dna_translate and the standard genetic code (Exercise 2.8). (Solution A.15)

Exercise 9.3. 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/.

9.1.2. Loading

When importing a module, for example the dna module you have just created (Exercise 9.2), you "open" its namespace, which becomes available to your program:

```python
>>> import dna
>>> dna.complement('aattttt')
'ttaaaa'
>>> dna.revcomp('aattttt')
'aaaaatt'
>>> dna.dna_translate('atggacaatttttccgggacgtag')
'MASPNFSGT*'
```

You may also select specific components from the module as "opened" (Figure 9.1):
Example 9.2. Loading a module’s components

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

In such cases, other components stay hidden, and the namespace is not the one of the module, e.g:

```python
>>> 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
```

Caution

You can restrict the components being imported by an import * statement. The __all__ variable, also used for packages (Section 9.2), can explicitly list the components to be directly accessible (see Exercise 9.8).

A module is loaded only once, i.e, a second import statement will not re-execute the code inside the module (see Python reload statement in the reference guides).
% python dna.py

the module is executed within the __main__ module (i.e. not the dna module):

% python -i dna.py
>>> dna.complement('aattttt')
NameError: name ‘dna’ is not defined
>>> complement('aattttt')
'tttaaaa'
>>> revcomp('aattttt')
'aaaaatt'
>>> dna_translate('atggacaatttttcggacgtag')
'MASPNFSGT*

For this reason, the code executed at module loading time can be made dependent of the current module name:

```python
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)
```

⚠️ Exercise 9.4. Locating components in modules


⚠️ Exercise 9.5. Bio.Seq module

Write the import statement required to import the Seq class from module Bio.Seq, in order to create a sequence:

```python
> seq=Seq("actttgccatatg")
```

_seq() is a function call that creates an instance of the class Seq, so you need to be able to access to this component of the Bio.Seq module.
9.2. 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.

9.2.1. 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 9.3. Using the Bio.Fasta package

```python
>>> import Bio.Fasta
>>> handle = open("data/ceru_human.fasta")
>>> 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:

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

Issuing:
Chapter 9. Modules and packages

>>> 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__


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
```

Solution A.17

Exercise 9.7. 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
```

Solution A.18
Exercise 9.8. Import from Bio.Clustalw

Why does the following code not work?

```python
from Bio.Clustalw import *
a=ClustalAlignment()
NameError: name 'ClustalAlignment' is not defined
```

Solution A.19
Chapter 9. Modules and packages
Chapter 10. Classes: Using classes

The presentation of classes is divided in two parts. The first part (present chapter) explains how to use classes, while the second part (Chapter 12), put after the introduction to Biopython, introduces the definition of new classes.

10.1. Creating instances

You have actually already used classes, or rather objects, i.e instances of classes, throughout this tutorial: 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...

The actual direct syntax to instantiate a class, i.e to create an instance of class, is by calling a function with the same name as the class. For instance, the random Python module defines a class Random. In order to create an instance of this class, i.e an object which generates random numbers, you do:

```python
>>> from random import Random
>>> generator = Random()
```

This creates the object and makes it available from the generator variable. You can now use this object to call Random class methods:

```python
>>> generator.randrange(100)
75
```

Sometimes, the instanciation needs arguments, depending on the definition of the class. For instance, there is a class Netscape defined in the webbrowser Python module. In order to create an instance of this class, i.e a browser that is able to browse Web documents, you need to pass the path of the netscape program on your computer:

```python
from webbrowser import Netscape
browser = Netscape('/local/bin/netscape')
```

Now, you can use the browser and open a Web document by:

```python
browser.open('http://www.biopython.org/)
```

Or, if we want to directly create an instance of class Seq, one of the class defined in Biopython to create sequence objects (see Section 11.3), we do:

```python
from Bio.Seq import Seq
```
Chapter 10. Classes: Using classes

seq = Seq('gcatgacgttattacgactctgtcacgccgcggtgcgacgcgtctgctggg')

The call to perform the instanciation (i.e: Random(), Netscape('/local/bin/netscape'), etc...) actually calls the __init__ method defined in the class. Thus, passing arguments to parameters for the instanciation call (i.e __init__ method call) follows exactly the same rules as for usual functions (see Section 6.4). So, passing the alphabet as a parameter to the Seq() call for creating a protein sequence, would be written as:

    seq = Seq('MKILIGIFLFLCSTPAWAKEKHYYIGIETTWDYASDHEKKLISVDTE',
                alphabet=Alphabet.ProteinAlphabet())

10.2. Getting information on a class

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 Netscape class:

    pydoc webbrowser.Netscape

See also the embedding module, which might bring additional documentation about related components:

    pydoc webbrowser

Attributes of a class also include variables. The variables defined for the instances can however not be listed by pydoc, since they belong to the instances, not 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 interpreter and create an instance, then ask for its dictionary:

    >>> seq = Seq('gcatgacgttattacgactctgtcacgccgcggtgcgacgcgtctgctggg')
    >>> dir(seq)
    ['__add__', '__doc__', '__getitem__', '__getslice__',
     '__init__', '__len__', '__module__', '__radd__', '__repr__',
     '__str__', 'alphabet', 'count', 'data', 'tomutable',
     'tostring']

So now, we know that the seq object has a data attribute:

    >>> seq.data
When you consult the documentation of a class with the pydoc command, you get most of the time a list of strange method names, such as __add__ or __getitem__. These methods are special methods to redefine operators, and will be explained in the next chapter on classes (Chapter 12, Section 12.2).
Chapter 11. Biopython: Introduction

11.1. Introduction

Biopython is a set of modules and packages for biology, including sequence analysis, database access, etc... or parsers components. Since it is very well described by the documentation (see Section 11.2), we are not going here to describe it extensively. Rather, we provide several exercises, since we think that the best way to understand and master Biopython is by practice.

The course about Biopython is divided in two parts, separated by a chapter explaining how to define new classes in Python (Chapter 12). The first part (present chapter), attempts to cover the use of central components such as components for sequences (Seq, SeqRecord, SeqFeature), alignments (Blast, Clustalw) and database access (SwissProt, GenBank). Then, the second part (Chapter 13) presents the main concepts of parsing in Biopython, associated with exercises to build parsing classes for Enzyme entries. The last part of the Biopython presentation summarizes several of the exercises provided in this course by the study of disulfid bonds in Human Ferroxidase 3D structure and alignments (see Section 13.2).

11.2. Documentation

- http://www.biopython.org/
- API documentation [http://www.bioinformatics.org/bradstuff/bp/api/]

See also:

- Software development - Python [http://www.scripps.edu/pub/olson-web/people/sanner/home.html], by Michel Sanner at Molecular Graphics Lab (MGL), Scripps, including ViPER, a great visual environment for molecular visualization (see [http://www.scripps.edu/~sanner/python/index.html] for details).

Chapter 11. Biopython: Introduction

- Using Python to solve problems in bioinformatics [http://bonsai.ims.u-tokyo.ac.jp/~mdehoon/software/software.html], by Michiel de Hoon.
- Object-oriented parsing of biological databases with Python [http://bioinformatics.oupjournals.org/cgi/content/abstract/16/7/628] (paper about PySAT [http://www.embl-heidelberg.de/~chenna/PySAT/]).
- SciPy - Scientific tools for Python [http://www.scipy.org/].

Figure 11.1. Overview of the Biopython course

1. SwissProt
   - fetch SwissProt ref.
   - popen

2. Seq
   - mutate
   - random

3. Prosite
   - fetch Prosite SwissProt ref.
   - search for patterns in the sequence
   - regexp

4. GenBank
   - fetch GenBank SwissProt ref.
   - find complete CDS

5. Blast
   - NCBI
   - local PSI−blast
   - PHI−Blast
   - FastaAlignment

6. PDB
   - define a PDBStructure class
   - defining a class
   - compute disulfid bonds according to coordinates

7. Enzyme
   - build parsing classes
   - fetch Enzyme SwissProt ref.

8. Clustalw
   - reading
   - running
   - PSSM
   - computed conserved Cys sites

9. compare:
   - annotated disulfid bonds (PDB)
   - computed disulfids bonds (from coordinates)
   - predicted Cys conserved sites (PSSM)

11.3. Bio.Seq and Bio.SeqRecord modules
Look at chapter 2 in Biopython tutorial (PDF [http://bioweb.pasteur.fr/docs/doc-gensoft/biopython/Doc/Tutorial.pdf]) (the following of this section actually assumes that you have read it).

11.3.1. Using Seq class
The Seq class is defined in the Bio.Seq module.

Example 11.1. Building Seq sequences from strings

```python
from Bio.Seq import Seq
seq = Seq('gcatgacgttattacgactctgtcacgccgcggtgcgactgaggcgtggcgtctgctggg')
print seq
```

Most of string manipulations seen in Section 2.1 are available on Seq objects.

Exercise 11.1. Length of a Seq sequence
Display the length of a sequence, and count the number of occurrences of ‘a’.
Solution A.20

Exercise 11.2. GC content of a Seq sequence
Adapt Exercise 2.1 (display GC content) to Bio.Seq.Seq.
Solution A.21

11.3.2. Sequences reading and writing
There are several ways in Biopython to handle sequence files. The Fasta formatted sequences might be read either through the Bio.Fasta Iterator (an iterator is an object that sequentially return successive records from a data input, see Section 13.1.3 for explanations), or with the FastaReader from the Bio.Seqio.FASTA module. There are also specialized modules to read flat format files (e.g SwissProt or GenBank formatted files) (see Section 11.4 and Section 11.5).

Example 11.2. Reading a FASTA sequence with the Bio.Fasta package

```python
import Bio.Fasta
import sys

handle = open(sys.argv[1])
seq = it.next()
```
while seq:
    print seq.name
    print seq.seq
    seq = it.next()
handle.close()

Example 11.3. Reading a FASTA sequence with the Bio.SeqIO.FASTA module

from Bio.SeqIO import FASTA
import sys
handle = open(sys.argv[1])
it = FASTA.FastaReader(handle)
seq = it.next()
while seq:
    print seq.name
    print seq.seq
    seq = it.next()
handle.close()

Exercise 11.3. Write a sequence in FASTA format

Write a sequence in FASTA format using the Bio.SeqIO.FASTA module. Initialize the sequence from a string as shown in example Example 11.1.
Solution A.22

Exercise 11.4. Code reading: Bio.sequtils

11.3.3. Bio classes for sequences
Figure 11.2 describes the classes to handle sequences. A `SeqRecord` is composed of `Seq` and `SeqFeatures`.

**Figure 11.2.** Seq, SeqRecord and SeqFeatures modules and classes hierarchies
11.3.3.1. SeqRecord objects

SeqRecord is central in Biopython. SeqRecord objects are created by various Biopython components (most of them not covered yet) Figure 11.3:

• FastaReader class in Bio.SeqIO.FASTA module (as shown above in Example 11.3)
• SequenceParser class in Bio.Fasta module (used as argument for creating and Iterator in Example 11.2).
• SequenceParser class in Bio.SwissProt.Sprot module (see Section 11.4)
• get_all_seqs method of Alignment class in Bio.Align.Generic module (see Section 11.6.2)
• Bio.GenBank parser (not tested and thus not represented in Figure 11.3) (see Section 11.5)
11.3.3.2. MutableSeq objects

A MutableSeq differs from a Seq by being modifiable.

Exercise 11.5. Random mutation of a sequence

Write a function mutateseq that randomly mutate a sequence. This function could take the following parameters, (with appropriate default values): sequence, random seed, span and probability:
mutateseq(seq, span=1000, p=0.01)

Solution A.23

Exercise 11.6. Random mutation of a sequence: count codons frequency

With the `mutateseq` function from the previous exercise (Exercise 11.5) and the `codons` function (Exercise 2.5), write a program that displays the frequencies of each codon, before and after the mutation of the sequence.

Solution A.24

Example 11.4. Plotting codon frequency

The following code use a `tkplot` module written by Michiel Jan Laurens de Hoon [http://bonsai.ims.u-tokyo.ac.jp/~mdehoon/], that we have a little modified, to plot a bar chart of each codon frequency.

```python
#--------------------------------------------------------
# bar charts of codons frequencies
# - for legibility, 2 charts are built

from tkplot import *
from Numeric import *

def codon_sort(a, b):
    if a < b:
        return -1
    elif a > b:
        return 1
    else:
        return 0

labels = count.keys()
labels.sort(codon_sort)

w1 = window(plot_title='Count codons', width=1000)
y = array(count.values())[:len(count)/2]
x = arange(len(y) + 1)
w1.bar(y, x, label=labels[:len(count)/2])

w2 = window(plot_title='Count codons(2)', width=1000)
y = array(count.values())[(len(count)/2)+1:]
x = arange(len(y) + 1)
w2.bar(y, x, label=labels[(len(count)/2)+1:])
```
Chapter 11. Biopython: Introduction

(Original Tk plot module [http://bonsai.ims.u-tokyo.ac.jp/~mdehoon/software/software.html] - Python code [http://bonsai.ims.u-tokyo.ac.jp/~mdehoon/software/tkplot/tkplot.py], modified module [modules/tkplot.py] - the one that you need to get the example work."

Exercise 11.7. Random mutation of a sequence: plot codons frequency

Use the code of Example 11.4 to plot both the normal and mutated codon counts of (Exercise 11.6).

Solution A.25

11.4. Bio.SwissProt.SProt and Bio.WWW.ExPASy

11.4.1. Reading entries

You can read SwissProt entries by using the SProt.Iterator [http://www.bioinformatics.org/bradstuff/bp/api/Bio/SwissProt/SProt_Iterator.py.html] (an iterator is an object that sequentially return successive records from a data input, see Section 13.1.3 for explanations). This iterator takes 2 arguments: the handle from where to read the entry, which may be an open file or the data fetched from an url, and a parser, which actually builds the objects returned by the iterator. For this purpose, you either use the SProt.RecordParser and get a SProt.Record, or use the SProt.SequenceParser and get a Bio.Seq.Seq sequence (see Section 13.1 for more information on parsing in Biopython).

In the following example, the entry is fetched from a local file, provided on the command line:

Example 11.5. Fetching a SwissProt entry from a file

```python
# reading a SwissProt entry from a file
from Bio.SwissProt import SProt
from sys import *

handle = open(argv[1])
sp = SProt.Iterator(handle, SProt.RecordParser())
record = sp.next()
print record.entry_name
print record.sequence

(you can try it on data/ceru_human.sp)
```

\(^2\) See also Chapter 14 on Graphics.
Exercise 11.8. Code reading: connecting with ExPASy and parsing SwissProt records

A second example is given by the script swissprot.py [http://bioweb.pasteur.fr/docs/doc-gensoft/biopython/Doc/examples/swissprot.py] provided within the Biopython distribution.

Exercise 11.9. SwissProt to FASTA

Write a function `sp_fasta` which converts a SwissProt file in FASTA format and use the `FastaWriter` defined in module `Bio.SeqIO.FASTA`. The function defined could be called this way:

```python
call_sp_fasta('data/ceru_human.sp', stdout)
```

You can store this function in a `sprot` module (`sprot.py`).

Tip
Look again at Example 11.3 which uses the `Bio.SeqIO.FASTA` module.

Tip
To read the SwissProt entry, you will need to use another parser than in Example 11.5, since the record you need must be compatible with the `FastaWriter` class (see Figure 11.3).

Look at Figure 11.3 to understand how to use the `FastaWriter` class in conjunction with a SwissProt parser.

Solution A.26

Exercise 11.10. Fetch an entry from a local SwissProt database

Changes Example 11.5 to fetch the entry from a local SwissProt database via the `golden` program (available from ftp://ftp.pasteur.fr/pub/GenSoft/ unix/db_soft/golden/). Write this as a function `get_sprot_entry_local` that you can add in the `sprot` module (`sprot.py`).

Solution A.27

Tip
Use the Python `os.popen` function.

11.4.2. Regular expressions in Python

An detailed presentation of Python regular expressions is available here: Regular Expression HOWTO [http://py-howto.sourceforge.net/regex/regex.html].
Chapter 11. Biopython: Introduction

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

```python
>>> import re
Before searching for a pattern, you must first compile it:

```python
>>> expression = '[AP]{1,2}D'
>>> p = re.compile(expression)
You then issue a `search`, for instance in the small sequence `seq`, by:

```python
>>> seq = "RPAD"
>>> result = p.search(seq)
To get the occurrences, you can ask for the start and end of the match in the searched text:

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

**Example 11.6. Searching for the occurrence of PS00079 and PS00080 Prosite patterns in the Human Ferroxidase protein**

```python
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].[0,8]G[LM]...[LIVMFYW]' ❶
p = re.compile(PS00079) ❷
result = p.search(seq.tostring()) ❸
print PS00079
print result.start(), result.end(), seq[result.start():result.end()] ❹
```

---

---

1. The regular expression is stored in a string.
2. The regular expression is compiled in a pattern.
3. The compiled pattern is searched in the sequence.
4. The result of the search is printed.
import sys
import re
from Bio.SwissProt import SProt

sp = open(sys.argv[1])
it = SProt.Iterator(sp, SProt.SequenceParser())
seq = it.next().seq
sp.close()

PS00080 = '(?P<copper3>H)CH...H...[AG](?P<copper1>[LM])'

p = re.compile(PS00080)
result = p.search(seq.tostring())
print PS00080
print result.start(), result.end(), seq[result.start():result.end()]

print 'copper type 3 binding residue: ', result.group('copper3')

print 'copper type 1 binding residue: ', result.group('copper1')

The regular expression now contains 2 identifiers: copper1 and copper3.
You can print the sub-parts of the result identified by variables: copper1 and copper3.

To get information about the re module, see pydoc, but also the sre module (Support for regular expressions), for which re is a wrapper.

Exercise 11.11. Enzymes referenced in a SwissProt entry

Write a function get_enzyme_ref which extracts the enzyme reference from SwissProt entry (you find it in the SwissProt description field). You can add this function in your sprot module (sprot.py). (Solution A.28)

11.4.3. Prosite

11.4.3.1. Prosite Dictionary

Biopython defines several dictionaries to access biological databases. Having a dictionary means that you can fetch an entry by:

```python
e = 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 Exercise 11.10). The entry fetched above is actually a string. In order to have the dictionary return a record, you must rather create it like this:


11.4.3.2. Prosite patterns

Exercise 11.12. Print the pattern of a Prosite entry

Write a function `get_prosite_pattern` that returns a string containing the pattern of a Prosite entry (provided as an id):

```python
get_prosite_pattern('PS00079')
```

Solution A.29

Exercise 11.13. Display the Prosite references of a SwissProt protein.

The SwissProt entry contains references to databases, including potential references to the Prosite [http://www.expasy.ch/prosite/] database (see SProt.Record class documentation).

Write a function `get_prosite.refs` that extracts the references to Prosite from a SwissProt entry (provided as a handle) (data [data/ceru_human.sp]).

The functions `get_prosite_pattern`, defined in Solution A.29 and the function `get_prosite.refs` can be used combined to display the patterns of the Prosite references given in a SwissProt entry. Write the statements to achieve this task.

```python
get_prosite_pattern('PS00079')
```

You can also add these functions in the sprot module (sprot.py).

Solution A.30

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.
Exercise 11.14. Search for occurrences of a protein PROSITE patterns in the sequence

Now, you know how to fetch a Prosite entry, how to extract a Prosite reference from a SwissProt entry, and how to search for pattern occurrences in a sequence. Search for the occurrences of the prosite_refs patterns in the sequence seq. Display:

• the pattern itself
• the start and end position in the sequence
• the corresponding sub-sequence.

Solution A.31

11.5. Bio.GenBank

First, look at the section on GenBank in the Biopython tutorial: http://www.biopython.org/docs/tutorial/Tutorial.html.

11.5.1. Reading entries

11.5.1.1. NCBIDictionary

Example 11.7. Using a NCBIDictionary

You can use the GenBank.NCBIDictionary to access a GenBank entry by its genbank ID (accession numbers do not work yet):

```
from Bio import GenBank
ncbi_dict = GenBank.NCBIDictionary()
gb_entry = ncbi_dict[id]
```

where gb_entry is a string.

The GenBank.NCBIDictionary may also be combined with a parser, producing either (there is a bug in the current Biopython release that disables this feature):

• a GenBank.Record instance:

```
from Bio import GenBank
record_parser = GenBank.RecordParser()
ncbi_dict = GenBank.NCBIDictionary(paper=record_parser)
gb_record = ncbi_dict[id]
```
Chapter 11. Biopython: Introduction

• or a `SeqRecord` instance:

```python
from Bio import GenBank
feature_parser = GenBank.FeatureParser()
c.ncbi_dict = GenBank.NCBIDictionary(parser=feature_parser)
seqrecord = ncbi_dict[id]
```

### 11.5.1.2. Iterator

#### Example 11.8. GenBank Iterator class

You can also use the `GenBank.Iterator` class to browse a file containing several GenBank entries:

```python
from Bio import GenBank

gb_file = argv[1]
gb_handle = open(gb_file, 'r')
feature_parser = GenBank.FeatureParser()
gb_iterator = GenBank.Iterator(gb_handle, feature_parser)
while 1:
    cur_record = gb_iterator.next()
    if cur_record is None:
        break
    print cur_record.seq
```

**Exercise 11.15. Extracting the complete CDS from a GenBank entry**

Write a function `get_complete_cds`:

```python
rec = get_gbrec(id)
cds = get_complete_cds(rec)
```

that returns the DNA sequence of the complete CDS. `get_gbrec` returning a `SeqRecord` (see examples in `gb_refs.py` [exemples/gb_refs.py]).

Solution A.32

### 11.6. Running Blast and Clustalw

#### 11.6.1. Blast
Chapter 11. Biopython: Introduction

Look at chapter 3 in Biopython tutorial [http://www.bioinformatics.org/bradstuff/bp/tut/Tutorial.html] (the following of this section actually assumes that you have read it).

Exercise 11.16. Local Blast, run and display results

Run a local Blast on the SwissProt databank, with Blast parameter E set to 1, number of descriptions and alignments set to 100, and save the result in a file (query [data/ceru_human.fasta]).
Solution A.33

Exercise 11.17. Remote Blast, run and save results

Run a remote Blast on the SwissProt databank, with Blast parameter E set to 1, number of descriptions and alignments set to 100, and save the result in a file (query [data/ceru_human.fasta]).
Solution A.34

Exercise 11.18. Remote Blast, parse results

Parse the Blast result saved in previous exercise (Exercise 11.17). Only display hits (not HSP) having and Expect value equal to 0.0. (NCBI Blast report [data/ceru_human.fasta.blast])
Solution A.35

Tip

There is a class which stores the description of the hit, including the Expect value (which is not necessarily the same as the Expect value of each HSP).

Exercise 11.19. Local PSI-Blast

Run a blastpgp locally on the NRProt databank. Display only HSPs which expect value is above a given threshold.
Solution A.36

Exercise 11.20. Search Prosite patterns with PHI-blast

Use the patterns associated to the Prosite references found in a SwissProt entry with PHI-blast.
Start from get_prosite_refs and get_prosite_pattern (Exercise 11.13) to get the patterns (these functions should have been saved in module sprot.py [exercises/sprot.py]). You will then have to provide the patterns to PHI-Blast in a "hitfile" (-k parameter) to run a PHI-Blast (see Solution A.36).
Solution A.37
Exercise 11.21. Running FASTA

How would write a run_fasta function to run a FASTA search:

```python
result = run_fasta(query_file, 'gbmam')
```

Write the run_fasta, knowing that the appropriate fasta command line to search a protein database is for instance:

```
fasta_t -q data/ceru_human.fasta /local/databases/fasta/gpmam
```

Result can be provided as text (i.e not as Python classes). Solution A.38

Tip

Use the Python `os.popen` function.

11.6.2. Clustalw

11.6.2.1. Loading a Clustalw file

Example 11.9. Loading a Clustalw file

The following code loads a file in Clustalw format (sample alignment in Custalw format [data/example.aln]):

```python
import Bio.Clustalw
from Bio.Alphabet import IUPAC
from sys import *
align = Bio.Clustalw.parse_file(argv[1], alphabet=IUPAC.protein) ❶
for seq in align.get_all_seqs():
    print seq.description
```

❶ The default alphabet seems to be nucleic.
11.6.2.2. Running Clustalw

Exercise 11.22. Doing a Clustalw alignment

Run the following code (completing the statements for loading the appropriate components) (sequences [data/clustalw.data]).

```python
cline = MultipleAlignCL(argv[1])
cline.set_output('data/test.aln')
print "Command line: ", cline

align = do_alignment(cline)
for seq in align.get_all_seqs():
    print seq.description
    print seq.seq
```

1 Construction of the command line to run Clustalw.
3 The get_all_seqs method returns SeqRecord objects (defined in the Bio.SeqRecord module).

Solution A.39

Exercise 11.23. Align Blast HSPs

Create a Fasta formatted sequences file from the HSPs of a Blast report and align these sequences. Starting from the Blast parsing in Exercise 11.19, create a Bio.Fasta.FastaAlign.FastaAlignment from the HSPs, remove the gaps, and align the sequences. Only keep the 10 first hits.

Solution A.40

11.6.2.3. Extracting information from alignments

Example 11.10. Get the consensus sequence of an alignment

```python
import Bio.Clustalw
import Bio.Align.AlignInfo
from Bio.Alphabet import IUPAC
from sys import *

align = Bio.Clustalw.parse_file(argv[1], alphabet=IUPAC.protein)
consensus = align_info.dumb_consensus()
```
print "Consensus: ", consensus.tostring()

(alignment [data/ceru_human.blastp-edit.aln])

Exercise 11.24. Get the PSSM from an alignment

First get the PSSM (Position Specific Score Matrices - you should look at the section 3.5.4 in Biopython tutorial [http://bioweb.pasteur.fr/docs/doc-gensoft/biopython/Doc/Tutorial.html] for explanations) from an alignment (loaded from a Clustalw format as in Example 11.9):

```python
align = Bio.Clustalw.parse_file(sys.argv[1], alphabet=IUPAC.protein)
ref_seq = align.get_seq_by_num(0)
pssm = align_info.pos_specific_score_matrix(ref_seq, chars_to_ignore = ['X'])
```

A PSSM is related to one of the sequences in the alignment (here the first one)

Using this PSSM, then display the positions in the alignment that have a percent identity above a given threshold. Notice positions with conserved cysteins (Cys).

Solution A.41

Exercise 11.25. Plotting Cys conserved positions

You can use Tkinter to display various plots. The following code shows how to create a plot widget, given 2 tuples (`vector_x` and `vector_y`, containing the x and y axes values.

Use this code and Exercise 11.24 to plot the Cys conserved positions of the alignment.

```python
# plot of Cys positions

from Numeric import *
from Tkinter import *
import Pmw

vector_y = ❶
vector_x =
root = Tk()
frame = Frame(root)

# vector_y must be a tuple

# See also Chapter 14 on Graphics.
```
Fill this with the code necessary to get tuples containing the plot values (percent of Cys).

Solution A.42

11.6.3. Running other bioinformatics programs under Pise

You can run several other programs interfaced under the Pise [http://www.pasteur.fr/recherche/unites/sis/Pise/] system, by using this API [http://www.pasteur.fr/recherche/unites/sis/Pise/#pisepython]. You can find here DNA data [data/dnaseq].

Example 11.11. Running the EMBOSS cusp program

The following example shows how to run the EMBOSS [http://www.hgmp.mrc.ac.uk/Software/EMBOSS/] cusp [http://bioweb.pasteur.fr/seqanal/interfaces/cusp.html] program to create a codon usage table:

```python
from Pise import PiseFactory
from Bio.SeqIO import FASTA
import sys

handle = open(sys.argv[1])
it = FASTA.FastaReader(handle)
seq = it.next()
handle.close()

factory = PiseFactory()
cusp = factory.program('cusp')
cusp.sequence(seq)

job = cusp.run()
if job.error():
    print "Error: " + job.error_message()
else:
    print "Output:\n", job.content("outfile.out")
```
Chapter 12. Classes: Defining a new class

12.1. Basic class definition

Up to now, you have used many objects, and even directly done some classes instanciations (explained in (Chapter 10). The next question is: how do you define new kinds of objects? In Python, you do this by defining a class, which describes of how the corresponding class of objects will behave and what operations, or methods, or more generally what attributes will be available on them.

You define a class by:

• the class classname statement
• describing its attributes, e.g methods and variables

Example 12.1. A sequence class

The following example defines a sequence class (simplified from Biopython Seq class). It defines several methods:

• __init__, called at instance creation
• tostring
• tomutable
• count

class Seq:

    def __init__(self, data, alphabet = Alphabet.generic_alphabet):
        self.data = data
        self.alphabet = alphabet

    def tostring(self):
        return self.data

    def tomutable(self):
        return MutableSeq(self.data, self.alphabet)

    def count(self, item):
        return len([x for x in self.data if x == item])

This method is always called when creating a new instance of class Seq. It’s the constructor.
The first argument passed to a class method call is always the object itself. Thus, the first parameter of a class method must be a variable pointing to the object itself, thus enabling to access to its attributes from inside the body of the method. `self` is just a naming convention.

Use of the `self` variable and the `.` (dot) operator to access the `data` attribute.

**Exercise 12.1. A class to store PDB residues**

Define a class to store PDB residues. A residue has: a name, a position in the sequence, and a list of atoms. An atom has a name and coordinates. Define 2 methods: `add_residue` and `add_atom` that you will use as follows:

```python
struct = PDBStructure()
residue = struct.add_residue(name = "ILE", posseq = 1 )
struct.add_atom(residue, name = "N",
               coord = (23.46800041, -8.01799965, -15.26200008))
struct.add_atom(residue, name = "CZ",
               coord = (125.50499725, 4.50500011, -19.14800072))
residue = struct.add_residue(name = "LYS", posseq = 2 )
struct.add_atom(residue, name = "OE1",
               coord = (126.12000275, -1.78199995, -15.04199982))
```

You also might need an `__init__` method to initialize the data structures.

**Tip**

The print statement should return:

```python
[[{'name': 'ILE', 'posseq': 1, 'atoms': [{'name': 'N', 'coord': (23.468000409999998, -8.0179996500000001, -15.26200008)}],
  {'name': 'CZ', 'coord': (125.50499725, 4.5050001100000001, -19.148000719999999)}},
 [ {'name': 'LYS', 'posseq': 2, 'atoms': [{'name': 'OE1', 'coord': (126.12000275, -1.7819999500000001, -15.041999819999999)}]]
```

To which kind of data structures does it correspond?

(Solution A.43)
Exercise 12.2. A class to store PDB residues (cont)

• Add 2 variables `model_id` and `chain_id` to the residue, in order to store the model ID (a PDB entry may have more than one model) and the chains.

• Add fields in the atom: `tempfactor` to store the factor of temperature, `occupancy`, `altloc`, for an alternate location, and `element`, which is the chemical name, e.g. “C” (while name, e.g. “CG2”, is the chemical name plus the position).

```python
struct = PDBStructure()
residue = struct.add_residue(model_id="1", chain_id="A",
                           name = "ILE", posseq = 1)
struct.add_atom(residue, name = "N",
                coord = (23.46800041, -8.01799965, -15.26200008),
                tempfactor=169.09, occupancy = 1.0,
                element = "N")
```

(Solution A.44)

Exercise 12.3. A class to store PDB residues (cont)

Add to your class definition the code required:

• to retrieve the residues by their name (method `get_residues_by_name`)

• to retrieve the model and chain a residue belongs to (methods `residue_model` and `residue_chain`)

• to list the residues of given chain (method `get_residues_of_chain`)

97
print "residues of name ILE:
for residue in struct.get_residues_by_name("ILE"):
    print residue
    print "model: ", struct.residue_model(residue)
    print "chain: ", struct.residue_chain(residue)

print "residues of chain B:
for residue in struct.get_residues_of_chain("B"):
    print residue

(Solution A.45)

12.2. Defining operators for classes

Standard methods enable to define the behaviour of standard operators:

- \texttt{\_\_add\_\_:} defines +
- \texttt{\_\_sub\_\_:} defines -
- \texttt{\_\_str\_\_:} defines how to convert the instance to a string representation (for e.g \texttt{print} statement)
- \texttt{\_\_getitem\_\_:} defines the access by key (object["key"])
Example 12.2. Seq operators

For instance, the Seq class defines a method `__str__` which returns a string informally representing the sequence, that will be used in `print` statement, as well as several other operators:

- `__add__` to concatenate sub-sequences by the + operator
- `__len__` for computing the length
- `__getitem__` for indexed access to a position in the sequence, etc....
- `__getslice__` for ranges in the sequence
- etc...

class Seq:

    def __repr__(self):
        return "%s(%s, %s)" % (self.__class__.__name__,
                                 repr(self.data),
                                 repr(self.alphabet))

    def __str__(self):
        if len(self.data) > 60:
            s = repr(self.data[:60] + " ...")
        else:
            s = repr(self.data)
        return "%s(%s, %s)" % (self.__class__.__name__, s,
                                 repr(self.alphabet))

    def __len__(self): return len(self.data)

    def __getitem__(self, i): return self.data[i]

    def __getslice__(self, i, j):
        i = max(i, 0); j = max(j, 0)
        return Seq(self.data[i:j], self.alphabet)

    def __add__(self, other):
        if type(other) == type(' '):
            return self.__class__(self.data + other, self.alphabet)
        elif self.alphabet.contains(other.alphabet):
            return self.__class__(self.data + other.data, self.alphabet)
        elif other.alphabet.contains(self.alphabet):
            return self.__class__(self.data + other.data, other.alphabet)
        else:
            raise TypeError, ("incompatible alphabets", str(self.alphabet),
                               str(other.alphabet))

Look at the source code of class Bio.GenBank.Dictionary in order to understand how the following code works:

```python
from Bio import GenBank
gb_dict = GenBank.NCBI Dictionary()
gb_rec = gb_dict['1617401']
print gb_rec
```

What happens here?

12.3. Inheritance

Example 12.3. biopython FastaAlignment class

The following code shows the definition of the class FastaAlignment in module Bio.Fasta.FastaAlign. This class inherits from class Bio.Align.Generic.Alignment, which defines generic methods for alignments.

```python
from Bio.Align.Generic import Alignment

class FastaAlignment(Alignment): ❶
    def __init__(self, alphabet = Alphabet.Gapped(IUPAC.ambiguous_dna)): ❷
        Alignment.__init__(self, alphabet)

    def __str__(self):
        """Print out a fasta version of the alignment info.""
        return_string = "
        for item in self._records:
            new_f_record = Fasta.Record()
            new_f_record.title = item.description
            new_f_record.sequence = item.seq.data

            return_string = return_string + str(new_f_record) + "\n"

        # have a extra newline, so strip two off and add one before returning
        return_string.rstrip(return_string) + "\n"
```

This class is a sub-class of class Bio.Align.Generic.Alignment

Call the super-class __init__ method. There is no special statement in Python to perform this (such as super).

See also class Bio.Clustal.ClustalAlignment.
Exercise 12.5. Biopython Alphabet class hierarchy

Solution A.46

Example 12.4. Exceptions class hierarchy

Exceptions are defined by inheritance (see Chapter 8 and Figure 8.1).

12.4. Classes variables

Example 12.5. Bio.Data.CodonTable class variables

The class Bio.Data.CodonTable [http://www.bioinformatics.org/bradstuff/bp/api/Bio/Data/CodonTable.py.html] relies on several variables, some of them being instances variables, other being class variables. Since they belong to the class namespace, class variables are shared by all the instances of the class.

class CodonTable:
    nucleotide_alphabet = Alphabet.generic_nucleotide
    protein_alphabet = Alphabet.generic_protein

    forward_table = {} # only includes codons which actually code
    back_table = {} # for back translations
    start_codons = []
    stop_codons = []

# Not always called from derived classes!
def __init__(self, nucleotide_alphabet = nucleotide_alphabet, 
            protein_alphabet = protein_alphabet, 
            forward_table = forward_table, back_table = back_table, 
            start_codons = start_codons, stop_codons = stop_codons):
    self.nucleotide_alphabet = nucleotide_alphabet
    self.protein_alphabet = protein_alphabet
    self.forward_table = forward_table
    self.back_table = back_table
    self.start_codons = start_codons
    self.stop_codons = stop_codons

Class variables definition (notice that they are declared outside of the methods). Notice that these definitions also use Alphabet generic_nucleotide and generic_protein class variables.

Instance variables definition (initialized at __init__()__). Instance variables are initialized with default values that are provided either by the class variables, or by the parameters.
Exercise 12.6. A class to store PDB residues (cont’)

Add a verbose functionality in the class that you can change by a switch:

```python
struct = PDBStructure()
self.verbose(1)
```

You can then add some verbose code into the methods of the class, such as:

```python
def add_atom(self, residue, name, coord, tempfactor, occupancy, element):
    if self.verbose():
        print "add_atom: ", residue: ", residue['name'],
        residue['posseq'], " name: ", name, " coord: ",
        coord, " tempfactor: ", tempfactor, " occupancy: ",
        occupancy, " element: ", element
```


```python
from Numeric import array, Float0
coord=array((x, y, z), Float0)
```

Solution A.47
Chapter 13. Biopython, continued

13.1. Parsers

13.1.1. Introduction

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 Bio.Blast.Record objects, directly available to the program. There are already several parsers in Biopython: Clustalw and Blast (standalone and NCBI) parsers, parsers for databases records (SwissProt, GenBank, Medline, SCOP, NBRF, Prosite, InterPro, Rebase, Kabat, ...). One of the reasons there are so many parsers available is that Biopython relies on Martel [http://www.dalkescientific.com/Martel/] (by Andrew Dalke), a scanner generator.

The aim of this section is not to describe all these tools, but rather to explain how they work, in order for you to be able to build one for your own programs or databases, or for programs not having their parser, yet (see also Parser.txt [http://bioweb.pasteur.fr/docs/docsoft/biopython/Doc/Parser.txt] from the Biopython documentation for a design overview of parsers). For this purpose, we will build a simple parser for the Enzyme [http://www.expasy.ch/enzyme/] database, that is only able to store the enzyme ID and the references of an entry to other databases. There is already a Bio.Enzyme package in Biopython, which defines a _Scanner class, that we are going to use, but no parser (yet).

In Biopython, parsing is often organized according to an event/callback model, one component, often called the scanner, generating parsing events when encountering a tag, and another component, often called the consumer, receiving and handling these events. Generally, you feed data to scan to the scanner through a handle, which can be an open file or an http stream.

scanner.feed(handle)

The scanner has to know about which consumer to call, which can be achieved by having a standard consumer for a given type of data. You can also provide a consumer as a parameter to the scanner:

scanner.feed(handle, consumer)

This architecture has the advantage of dividing the tasks of scanning the text and deciding what to do with the recognized text elements. Writing your own consumer enables you to build your own data structures.

At a higher level, a parser component may wrap the two other components in one class, providing a simpler component to the programmer, since he or she just has to call the parser:

parser.parse(handle)

to process the text. Such a parser might be implemented like this:
class Parser(AbstractParser): ❶
    def __init__(self):
        self._scanner = _Scanner() ❷
        self._consumer = _Consumer()

    def parse(self, handle):
        self._scanner.feed(handle, self._consumer)
        return self._consumer.data

❶ AbstractParser: see below for explanation about Biopython classes to support parsing.
❷ Wrapped scanner and consumer.

Biopython provides a support for defining new parsers and consumers classes through the Bio.ParserSupport module (AbstractParser and AbstractConsumer classes).

Example 13.1. Using SProt.RecordParser and SProt.SequenceParser

For instance, the Bio.SwissProt.SProt module defines (Figure 13.1):

• a scanner: class Bio.SwissProt.SProt._Scanner

• 2 consumers:

  • class Bio.SwissProt.SProt._RecordConsumer

  • and Bio.SwissProt.SProt._SequenceConsumer

• 2 parsers: classes Bio.SwissProt.SProt.RecordParser and Bio.SwissProt.SProt.SequenceParser, that we have already used in several occasions (Section 11.4.1).
Figure 13.1. Parsers class hierarchy
Using the SequenceParser, you get Seq objects, while using the RecordParser, you get SeqRecord objects.

```python
from Bio.SwissProt import SProt
from sys import *

fh = open(argv[1])
sp = SProt.Iterator(fh, SProt.RecordParser())
record = sp.next()
for feat in record.features:
    if feat[0] == 'DOMAIN':
fh.close()

fh = open(argv[1])
sp = SProt.Iterator(fh, SProt.SequenceParser())
sequence = sp.next()
print "sequence: ", sequence.seq
fh.close()
```

1 Reading the file with SProt.RecordParser. This enables to access to the annotations and features.
2 Re-reading the file with SProt.SequenceParser.

### 13.1.2. Exercises: building parsing classes for Enzyme

We start by defining a simple consumer to receive the scanner events, first for only one entry (Exercise 13.1), then for a file containing several entries (Exercise 13.2). After this, we will wrap this consumer and the scanner in a parser class (Exercise 13.3). The next step is an iterator, which sequentially return "records" (Exercise 13.5). In order to be able to do a search in a enzyme database, we will then add a lookup method to the iterator (Exercise 13.6), then define a proper dictionary class, such as the one that we have seen in Example 11.7 (Exercise 13.7). The last step will be to pack all this classes into a module (Exercise 13.8), and to use it to fetch the enzyme entry referenced in a SwissProt record, and display all the related proteins (Exercise 13.9).

To build a consumer, you need to know which events the scanner will generate. Biopython distribution contains documentation on this topic. See for instance a copy at: http://bioweb.pasteur.fr/docs/doc-gensoft/biopython/Doc/.

**Exercise 13.1. EnzymeConsumer, reading one entry from a file**

Define a class EnzymeConsumer storing the references of the enzyme entry. Use the class Enzyme._Scanner for reading the entry from a file (data [data/enzyme.1.16.3.1]). EnzymeConsumer should be a sub-class of AbstractConsumer from module Bio.ParserSupport. Thanks to this, you will not have to provide all the callback methods called by the scanner, but only the methods that are of interest for you.

```python
handle = open(argv[1])
```
Chapter 13. Biopython, continued

```python
scanner = Enzyme._Scanner()
consumer = EnzymeConsumer()
scanner.feed(handle, consumer)
print "results: ", consumer._references
```

(Solution A.48)

**Exercise 13.2. EnzymeConsumer, reading n entries from a file**

Define the same class as in Exercise 13.1, but stores the references for several enzyme entries (data [data/enzymes]).

```python
for id in consumer._references.keys():
    print id, consumer._references[id]
```

(Solution A.49)

**Exercise 13.3. EnzymeParser**

Define a parser to wrap the scanner and the consumer previously defined (Exercise 13.2).

```python
handle = open(argv[1])
parser = EnzymeParser()
references = parser.parse(handle)
for id in references.keys():
    print id, references[id]
```

(Solution A.50)

### 13.1.3. Iterator

An iterator is an object that sequentially return successive records from a data input:

```python
iterator = Iterator(handle)
record = iterator.next()
while record:
    print record
    record = iterator.next()
```

From with this data input, the iterator `next` method provides a parser with the lines corresponding to a record, from which the parser will build a record, e.g (simplified):
def next(self):
# 1) read the appropriate lines (until end of record)
# 2) call the parser
    return self._parser.parse(lines)

You might need to convert the lines back into a handle before passing them to the parser, since the parser rather
takes a handle. You can use the Bio.File.StringHandle class for this purpose.

Read the definition of the Bio.Swissprot.SProt.Iterator

13.1.4. Exercises: building parsing classes for Enzyme (cont)

Exercise 13.5. EnzymeIterator
Define a class EnzymeIterator based on the parser previously defined (Exercise 13.3). The parser just return
a dictionary as a "record", with an 'id' and a 'references' keys.

handle = open(argv[1])
iterator = EnzymeIterator(handle)
record = iterator.next()
while record:
    print record['id'], record['references']
    record = iterator.next()
handle.close()

The 'references' value could be look this:

[{'ac': 'P00450,', 'id': 'CERU_HUMAN;'},
 {'ac': 'Q61147,', 'id': 'CERU_MOUSE;'},
 {'ac': 'P13635,', 'id': 'CERU_RAT'}]
(Solution A.51)

Exercise 13.6. EnzymeIterator with lookup
Add to the class EnzymeIterator previously defined (Exercise 13.5) a lookup method, that goes
through the file looking for an entry of a given ID. Try on the real enzyme database (at Pasteur:
/local/databases/release/Enzyme/enzyme.dat). You can use the getopt module to pass the
database and id from the command line.

108
13.1.5. Dictionary

In Section 12.2, we have seen how to define operators for a class. Namely, the \texttt{__getitem__} method is a way to define an operator to provide an indexed access to data.

\textit{Exercise 13.7. EnzymeDictionary}

Define a class \texttt{EnzymeDictionary} based on the iterator previously defined (Exercise 13.5) (now, it is not anymore the \texttt{EnzymeIterator} class which handles the lookup). The way to use the dictionary is as following (db is the database filename, e.g \texttt{/local/databases/release/Enzyme/enzyme.dat}, id the enzyme id, e.g \texttt{1.16.3.1}):

```python
enzyme = EnzymeDictionary(db)
record = enzyme[id]
print record['references']
```

You may improve the code above by handling the \texttt{KeyError} exception.

(Solution A.53)

13.1.6. Using the parsers classes

\textit{Exercise 13.8. EnzymeParsing module}

Define an \texttt{EnzymeParsing} module. (Solution A.54)
Exercise 13.9. Fetching enzymes referenced in a SwissProt entry and display related proteins

Re-using Exercise 11.11 to find the enzyme number from the description text of a SwissProt entry, fetch the corresponding enzyme data. Return the list of SwissProt records referenced by the enzyme entry, and display their entry name and description. (Solution A.55)

13.1.7. Building parsing classes for phylogenetic trees

Exercise 13.10. Scanner

Write a Scanner for a Phylip formatted tree, such as this one [data/tree]. The Consumer can be written later (see Exercise 13.11). You can first just assume the following abstract Consumer class, called by your scanner at appropriate steps:

```python
from Bio.ParserSupport import *

class Consumer(AbstractConsumer):
    def start_tree(self, rooted=False):
        pass
    def end_tree(self):
        pass
    def begin_node(self):
        pass
    def end_node(self):
        pass
    def leaf(self, name):
        pass
    def branch_length(self, l):
        pass
```

You will also need an is_rooted function, that takes a string as parameter, to test whether the tree to be parsed is rooted or not. (Solution A.56)

Exercise 13.11. Consumer

Now write a Consumer class for building a tree. The consumer heavily depends of course on the type of tree that you need, since it is the consumer that builds and defines the resulting instance. Let us assume that we want a
very simple representation, such as a basic `Node` class for representing the tree, that would include some attributes such as:

• `name`
• `children`
• `length`

You might also need a `Stack` class for handling internal nodes recursively. (Solution A.57).

Exercise 13.12. Parser

Now write a `Parser` class to put everything together (see Exercise 13.3). This class can be used like this:

```python
treefile = sys.argv[1]
fh = open(treefile)
parser = Parser()
tree = parser.parse(fh)
print tree
```

(Solution A.58).

13.2. Practical: studying disulfid bonds in Human Ferroxidase 3D structure and alignments

13.2.1. Working with PDB

Exercise 13.13. Fetch a PDB entry from the RCSB Web server

Look at the code of the `Bio.WWW.ExPASy` module, e.g. function `get_sprot_raw` and define a function `get_pdb_entry_remote` that returns a handle (something that can be opened by `open()`) on a given PDB entry.

The url that you need is: http://www.rcsb.org/pdb/cgi/export.cgi/%s.pdb?format=PDB&compression=None&pdbId=%s.

Try your code with the 1KCW [http://www.rcsb.org/pdb/cgi/export.cgi/1KCW.pdb?format=PDB&compression=None&pdbId=1KCW] ident, which is the PDB entry corresponding to the CERU_HUMAN protein we are studying.

Solution A.59

Exercise 13.14. Define a PDBStructure class

You can start from Exercise 12.3. In summary, this class should define the following methods (constructors):

• `__init__`
• `__str__`
• `set_id`
• `set_pdb_ident`
• `add_dbref`
• `add_ssbond`
• `add_residue`
• `add_atom`

and (selectors):

• `get_residues`
• `get_atoms`
• `get_ssbonds`
• `get_residues_by_name`
• `get_residues_of_chain`
• `residue_model`
• `residue_chain`

Solution A.60
Exercise 13.15. Define a PDBConsumer class

In Exercise 12.3, the structure was built "by hand". Indeed, we had to write all the statements to add residues and atoms in our program:

```python
residue = struct.add_residue(model_id, chain_id, name = "ILE",
                          posseq = 1 )
struct.add_atom(residue, name = "N",
                coord = (23.46800041, -8.01799965, -15.26200008) ,
                tempfactor=169.09, occupancy = 1.0, element = "N")
```

We don’t really want to do this for real data. Instead, we now want to load the structure from a PDB file.

The PDBConsumer class we would like to build for this purpose roughly follows the scanner/consumer scheme that we have seen previously (Section 13.1). Our consumer’s job is to build a PDBStructure object as it receives parsing events. As a "scanner", you can actually use this PDBParser [modules/PDBParser.py] provided by Thomas Hamelryck (thamelry@vub.ac.be).

Solution A.61

Tip

The methods the PDBConsumer class will contain thus should correspond to the "events" (or callbacks) of the PDBParser class. For instance, the statement in PDBParser:

```python
self.structure_builder.set_ssbond(_from,_to)
```

calls the set_ssbond method of the consumer (here structure_builder is an equivalent of our "consumer").

Tip

When using this PDBParser class, the actual structure is returned to the program as follows:

```python
parser=PDBParser(PDBConsumer())
struct = parser.get(id, file)
```

1. Parser instanciation: the consumer is passed as an argument to the __init__ of the parser.
2. Structure creation: the get method in the parser takes an id and a filename as arguments.
You also have to know that the *get* method in the *PDBParser* needs to call a *get* method in the consumer, whose only task is to actually return the *PDBStructure* just built.

### 13.2.2. Study of disulfid bonds

**Exercise 13.16. Compute disulfid bonds in 1KCW**

Add a method in the *PDBStructure* called *disulfid_bridges*. Then write a program using the parser and checking for corresponding annotations in the PDB entry. (data [data/pdb1kcw.ent])

Solution A.62

**Exercise 13.17. Compare 3D disulfid bonds with Cys positions in the alignment (take #1).**

Compare annotated and computed disulfid bonds in 1KCW with cystein positions in the alignment. Take the code written in Exercise 11.24 to get position with a high-level of cysteins and check if they correspond to the bonds in the 3D structure.

Solution A.63

**Exercise 13.18. Compare 3D disulfid bonds with Cys positions in the alignment (take #2).**

The positions in the alignment and in the structure are somewhat different. Write a method *pdb2seq_pos* which use the DBREF lines of the PDB entry (see method *add_dbref*). Then, use can use this *pdb2seq_pos* method to display the actual positions in the alignment.

Solution A.64
Chapter 14. Graphics in Python

Note

Graphical programming is not the main purpose of this course. We just list here some resources (both tutorials and software) related to graphics, mainly:

• 2-dimensional graphics, chart/graph generation and scientific plots,
• GUI toolkits.

14.1. Tutorials


• The Python Short Course [http://www.wag.caltech.edu/home/rpm/python_course/] has a section about Tkinter (Python GUIs with Tkinter [http://www.wag.caltech.edu/home/rpm/python_course/Lecture_6.htm]).


• Tkinter help [http://www.isd197.org/sibley/cs/icp/tips/tkinter_html]: links to resources.

• Handbook of the Physics Computing Course [http://users.ox.ac.uk/~sann1276/handbook/handbook.html] (by Michael Williams) with a section on graphical programming (on Gnuplot).

Read the code and comments provided in Chapter Graphs and Charts (chapter 11) [http://www.manning.com/Grayson/] of the book cited above, Python and Tkinter Programming.

### 14.2. Software

- Python Tkinter Resources [http://www.python.org/topics/tkinter/]
- Gnuplot.py [http://gnuplot-py.sourceforge.net/]
- Vaults of Parnassus [http://www.vex.net/parnassus/]: see the "Graphics" section, which has a "GUI" (Graphical User Interfaces) section.
- Plot-resources [http://starship.python.net/crew/jhauser/plot-res.html], a collection of links to plotting resources for Python on Python Starship [http://starship.python.net/] site.
- Biggles [http://sourceforge.net/projects/biggles/]: a 2D scientific plotting package for Python, geared toward the production of publication-quality plots.
- Piddle [http://piddle.sourceforge.net/]: module for creating two-dimensional graphics in a manner that is both cross-platform and cross-media
- PLplot [http://plplot.sourceforge.net/]: scientific graphics package
- GGobi [http://www.ggobi.org/] Data Visualization System
- GDChart [http://www.fred.net/brv/chart/]: chart/graph generation in GIF format
14.3. Summary of examples and exercises with some graphics in this course

- Plotting codon frequency (Example 11.4). This example uses a Tkinter canvas to draw a bar chart. A documentation on the Tkinter canvas can be found here [http://www.pythonware.com/library/tkinter/introduction/canvas.htm].

- Plotting Cys conserved positions (Exercise 11.25). This example uses the Pmw.Blt [http://pmw.sourceforge.net/doc/Blt.html] package to draw a plot representing Cys conservation at each position of an alignment.
Chapter 14. Graphics in Python
Appendix A. Solutions

A.1. Introduction to basic types in Python

Solution A.1. GC content ()

```python
>>> (count(dna, 'c') + count(dna, 'g')) / len(dna)
0
```

```python
>>> (count(dna, 'c') + count(dna, 'g')) / float(len(dna))
0.64077669902912626
```

```python
>>> (count(dna, 'c') + count(dna, 'g')) * 100.0 / len(dna)
64.077669902912618
```

```python
>>> (count(dna, 'c') + count(dna, 'g')) / len(dna) * 100.0
0.0
```

```python
>>> gc = (count(dna, 'c') + count(dna, 'g')) / float(len(dna)) * 100
>>> ".%.2f" % gc
'64.08'
```

1 Why does this solution not work? (Section 2.7)
2 This is an example of the \% string formating operator (for more explanation see Section 6.2.4).

Solution A.2. DNA complement ()

```python
>>> replace(replace(replace(replace(replace(replace(dna, 'a', 'x'),
't', 'a'), 'x', 't'), 'c', 'x'), 'g', 'c'), 'x', 'g')
cgtactgcgaataatctgagacagtcggcgccacgctgactccgcaccgcagacgacccggaaatgaa
gcggaggcgcggaggcgcgggacgtaaggcaaggacaccggagc'
```

```python
>>> t=maketrans("AGCTagct", "TCGAtcga")
>>> translate(dna, t)
cgtactgcgaataatctgagacagtcggcgccacgctgactccgcaccgcagacgacccggaaatgaa
gcggaggcgcggaggcgcgggacgtaaggcaaggacaccggagc'
```

1 This is a simple version using replace.
2 This version use a translation table.
Solution A.3. Restriction site occurrences as a list

```python
def restrict(dna, enz):
    "find all start positions of a restriction site"
    res = []
    site = dna.find(enz)
    while site != -1:
        res.append(site)
        site = dna.find(enz, site + 1)
    return res
```

Solution A.4. Restriction digest

```python
def digest(dna, enzlist):
    "returns a list containing the cut positions when cutting dna with all enzymes in enzlist"

    Lcuts = []
    # get all cut positions
    for enz, pcut in enzlist:
        print enz, pcut
        start = 0
        stop = dna.find(enz)
        while stop != -1:
            Lcuts.append(stop + pcut)
            stop = dna.find(enz, stop+1)
    # sort
    Lcuts.sort()
    return Lcuts

def frag_len(Lcuts):
    "get fragment lengths from a list containing the cutting positions of an restriction digest sorted by order + start(=0) and end(=dna length) of the dna sequence"

    Lres = []
    start = Lcuts[0]
    for end in Lcuts[1:]:
        Lres.append(end-start)
        start = end
```
Appendix A. Solutions

return Lres

Solution A.5. Get the codon list from a DNA sequence ()

def codons(s, frame=0):
    codons=[]
    end=len(s[frame:]) - (len(s[frame:]) % 3) - 1
    for i in range(frame, end, 3):
        codons.append(s[i:i+3])
    return codons

Solution A.6. Reverse Complement of DNA ()

from string import *

def revcomp(dna):
    """ reverse complement of a DNA sequence """
    comp = dna.translate(maketrans("AGCTagct", "TCGAtcga"))
    lcomp = list(comp)
    lcomp.reverse()
    return join(lcomp, ")

Solution A.7. Translate a DNA sequence ()

standard = { 'ttt': 'F', 'tct': 'S', 'tat': 'Y', 'tgt': 'C',
             'ttc': 'F', 'tcc': 'S', 'tac': 'Y', 'tgc': 'C',
             'tta': 'L', 'tca': 'S', 'taa': '*', 'tca': 'S',
             'ttg': 'L', 'tcg': 'S', 'tag': '*', 'tcg': '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',}
Appendix A. Solutions

def dna_translate(cdna, code=standard):
    """ translate a cDNA sequence to a protein """
    prot = ""
    for i in xrange(0, len(cdna), 3):
        prot += code.get(cdna[i:i+3], "?")
    return prot

def dna_translate2(cdna, code=standard):
    """ translate a cDNA sequence to a protein """
    return ''.join([code.get(cdna[i:i+3], "?")
                    for i in xrange(0, len(cdna), 3)])

    This is a special syntax named list comprehension. It creates a list and populates it with the results of the first expression by replacing i with all values of the for loop (see also Chapter 7).

Solution A.8. Write a sequence in fasta format ()

def write_fasta(fh, seq, id="", desc="", width=60):
    """ write a sequence in fasta format.
    The following parameters can be specified:
    fh - file descriptor
    seq - sequence as a string
    id - sequence id (default is no id)
    desc - sequence description (default is no description)
    width - number of characters per sequence line (default 60)"
    print >>fh, ">%s %s" % (id, desc)
    for i in xrange(0, len(seq), width):
        print >>fh, "%s" % seq[i:i+width]

Solution A.9. Header function ()

import string

def header(title):
    """splits a fasta header in ID and description of a sequence
    if one of the two is not given None is returned instead"
    id = desc = None
Appendix A. Solutions

A.2. Control Flow

Solution A.10. Count ambiguous bases ()

```python
def ambiguous(dna):
    "returns the number of ambiguous characters in a dna sequence"
    nb = 0
    for i in dna:
        if i not in 'atgc':
            nb += 1
    return nb
```

Solution A.11. Verify DNA bases ()

# first version

```python
def check_dna(dna, alphabet='atgc'):
    """ using break and continue """
    for base in dna:
        if base not in alphabet:
            break
    else:
        return "dna ok"
```

# second version

```python
def check_dna2(dna, alphabet='atgcATGC'):
```
*** loop without break or continue ***

```python
ok = 1
for base in dna:
    if base in alphabet:
        pass ❶
    else:
        ok = 0

if ok:
    return "dna ok"
```

❶ pass is the empty statement.

**A.3. Functions**

**Solution A.12. DNA complement function ()**

```python
from string import *

def complement(dna):
    "function to calculate the complement of a DNA sequence" ❶
    tab = maketrans("AGCTagct", "TCGAtcga")
    return translate(dna, tab)
```

❶ If the first statement of a function is a string this string is the documentation of the function. It can be accessed by `func.func_doc`.

```python
>>> complement.func_doc
'function to calculate the complement of a DNA sequence'
```
Appendix A. Solutions

Solution A.13. Variable number of arguments ()

def all_2_digests(*enzymes):
    """generate all possible digests with 2 enzymes """
    digests = []
    for i in range(len(enzymes)):
        for k in range(i+1, len(enzymes)):
            digests.append([enzymes[i], enzymes[k]])
    return digests

A.4. Modules and packages

Solution A.14. Loading and using a module: print the command line arguments

Exercise 9.1

import sys
print sys.argv

Solution A.15. Creating a module

Exercise 9.2

In order to create a module called dna, just put the your functions definitions in a file called dna.py (dna.py [exercises/dna.py]).

Do a:

pydoc dna


Exercise 9.5

from Bio.Seq import Seq
seq=Seq("actttgccatag")
Solution A.17. Bio.SwissProt package

Exercise 9.6

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

```python
from Bio.WWW import * ❶
from Bio.SwissProt import SProt
expasy = ExPASy.get_sprot_raw('CERU_HUMAN')
sp = SProt.Iterator(expasy, SProt.RecordParser())
record = sp.next()
print record.keywords
```

❶ This statement import "all" components from the Bio.WWW package, including the ExPASy module (see: pydoc Bio.WWW and pydoc Bio.WWW.ExPASy, and llok at the __all__ and __path__ in the DATA section).

Solution A.18. Using a class from a module

Exercise 9.7

Why does the following code issue an error?

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

The reason is that Bio.SubsMat.FreqTable is the module containing FreqTable, not the class (see pydoc Bio.SubsMat.FreqTable). The FreqTable class is available as Bio.SubsMat.FreqTable.FreqTable. The import statement should be:

```python
from Bio.SubsMat.FreqTable import FreqTable
```

(see also: Exercise 9.5).

Solution A.19. Import from Bio.Clustalw

Exercise 9.8

Why does the following code not work?

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

126
Appendix A. Solutions

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

Look at the `__all__` variable in the `__init__.py` module file (or with `pydoc Bio.Clustalw`). It is not empty, but it does not contain `ClustalAlignment`. The import statement should be (see Example 9.2):

```python
from Bio.Clustalw import ClustalAlignment
```

### A.5. Biopython: Introduction

#### A.5.1. Bio.Seq package

**Solution A.20. Using a Bio.Seq.Seq sequence**

**Exercise 11.1**

Display the length of a sequence, and count the number of occurrences of `'a'`.

```python
from Bio.Seq import Seq

seq = Seq('gcatgacgttattacgactcagcgcgggtgacgtgagcgtgctgctggg')
print len(seq)
print seq.count('a')
```


**Exercise 11.2**

Display GC content.

```python
from Bio.Seq import Seq

seq = Seq('gcatgacgttattacgactcagcgcgggtgacgtgagcgtgctgctggg')
gc = seq.count('c') + seq.count('g') / float(len(seq)) * 100
print gc
```

**Solution A.22. Write a sequence in FASTA format**

**Exercise 11.3**

Write a sequence in FASTA format using the `Bio.Seqio.FASTA` module.
from Bio.SeqIO import FASTA
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord
from sys import *

dna = Seq('gcatgacgttattacgactctgtcgcctgaggctggtccggtgcgactgaggcgtggcgtctgctggg') ❶
seq = SeqRecord(dna, id = 'my_seq', description= 'a random sequence') ❷
out = FASTA.FastaWriter(stdout)
out.write(seq)

❶ Creation of the Seq object required to create a SeqRecord object.
❷ Creation of the SeqRecord object.

Solution A.23. Random mutation of a sequence

Exercise 11.5

from random import *

def mutateseq(seq, rseed=0, span=10, p=0.1, verbose=0):
    if rseed == 0:
        seed()
    else:
        seed(rseed)
    for t in range(0,span):
        r=randrange(0,1/p) ❶
        if r == 0:
            newbase = choice(['a', 'c', 't', 'g'])
            position = randrange(0,len(seq)-1)
            oldbase = seq[position]
            seq[position] = newbase ❷
            if verbose:
                print t, seq.toseq().tostring(), position, "(%s -> %s)" % (oldbase, newbase)

❶ Pick one value (here 0) among the possible integer values (which should be equally distributed during the span given).
❷ The sequence must be from class MutableSeq, of course.
Solution A.24. Random mutation of a sequence: count codons frequency

Exercise 11.6

import Bio.Fasta
from sys import *
from string import *
from dna import codons
from mutateseq import mutateseq

file = argv[1]
handle = open(file)
count = {}
count_random = {}
seq = it.next()
while seq:
    for codon in codons(seq.seq.tostring()):
        if count.has_key(codon):
            count[codon] += 1
        else:
            count[codon] = 0
    mutableseq = seq.seq.tomutable()
    mutateseq(mutableseq, span=1000, p=0.1)
    for codon in codons(mutableseq.tostring()):
        if count_random.has_key(codon):
            count_random[codon] += 1
        else:
            count_random[codon] = 0
    seq = it.next()

handle.close()

l=count.items()
l.sort()
print "count: ", l
l=count_random.items()
l.sort()
print "random: ", l

Initialization of the dictionaries count and count_random.
Test of the existence of a dictionary key.
Solution A.25. Random mutation of a sequence: plot codons frequency

Exercise 11.7

```python
# bar charts of codons frequencies
# - for legibility, 2 charts are built
# - both random and normal frequencies are displayed

from tkplot import *
from Numeric import *

def codon_sort(a,b):
    if a < b:
        return -1
    elif a > b:
        return 1
    else:
        return 0

for codon in count.keys():
    if not count_random.has_key(codon):
        count_random[codon] = 0

for codon in count_random.keys():
    if not count.has_key(codon):
        count[codon] = 0

labels=count.keys()
labels.sort(codon_sort)

w1=window(plot_title='Count codons',width=1000)
y=array(count.values())[:len(count)/2]
x=arange(len(y)+1)
w1.bar(y,x,label=labels[:len(count)/2])

w2=window(plot_title='Count codons(2)',width=1000)
y=array(count.values())[(len(count)/2)+1:]
x=arange(len(y)+1)
w2.bar(y,x,label=labels[(len(count)/2)+1:])

y=array(count_random.values())[:len(count_random)/2]
x=arange(len(y)+1)
w1.bar(y,x,label=labels[:len(count_random)/2])

y=array(count_random.values())[(len(count_random)/2)+1:]
x=arange(len(y)+1)
w2.bar(y,x,label=labels[(len(count_random)/2)+1:])
```

130
A.5.2. Bio.SwissProt.SProt and Bio.WWW.ExPASy

Solution A.26. SwissProt to FASTA

Exercise 11.9

```python
from Bio.SeqIO import FASTA
from Bio.SwissProt import SProt
from sys import *

def convert_sp_fasta(infile, outfile):
    """
    convert a SwissProt file into a Fasta formatted file
    """
    in_h = open(infile)
    sp = SProt.Iterator(in_h, SProt.SequenceParser())
    out_h = FASTA.FastaWriter(outfile)
    sequence = sp.next()
    out_h.write(sequence)
```

(full code [exercises/seqrandom_count_codons_plot.py])
Solution A.27. Fetch an entry from a local SwissProt database

Exercise 11.10

```python
from Bio.SwissProt import SProt
from os import *

def get_sprot_entry_local(sprot_id):
    cmd = "golden sp:" + sprot_id
    golden = popen(cmd, 'r')
    iterator = SProt.Iterator(golden, SProt.RecordParser())
    entry = iterator.next()
    golden.close()
    return entry
```

Solution A.28. Enzymes referenced in a SwissProt entry

Exercise 11.11

```python
import re

def get_enzyme_ref(record):
    description = record.description
    enzyme_re = re.compile(r'\(EC\s+(?P<id>(\[[\w\.]\]+))\).*')
    m = enzyme_re.search(record.description)
    return m.group('id')
```

Solution A.29. Print the pattern of a Prosite entry.

Exercise 11.12

```python
import Bio.Prosite
```

def get_prosite_pattern(id):
    record=prosite[id]
    return record.pattern

You can also use a local Prosite database, and use the golden program to fetch entries:

from Bio.Prosite import Iterator, RecordParser
from os import popen

def get_prosite_pattern_local(id):
    cmd="golden prosite:" + id
    handle=popen(cmd, 'r')
    iterator=Iterator(handle,RecordParser())
    record=iterator.next()
    handle.close()
    return record.pattern

Solution A.30. Display the Prosite patterns of a SwissProt protein.

Exercise 11.13

The function is defined as follows:

from Bio.SwissProt import SProt

def get_prosite_refs(handle):
    sp = SProt.Iterator(handle, SProt.RecordParser())
    refs=[]
    record = sp.next()
    for ref in record.cross_references:
        if ref[0] == 'PROSITE':
            refs.append(ref[1])
    return refs

To display the patterns of the Prosite references given in a SwissProt entry, you can do like this:
import sys
from Bio.SwissProt import SProt
from sprot import get_prosite_refs, get_prosite_pattern

sp = open(sys.argv[1])
prosite_refs = get_prosite_refs(sp)
sp.close()

for id in prosite_refs:
    print id
    pattern = get_prosite_pattern(id)
    print pattern

Solution A.31. Search for occurrences of a protein PROSITE patterns in the sequence

Exercise 11.14

import sys
from Bio.Prosite import Pattern
from Bio.SwissProt import SProt
from sprot import get_prosite_refs, get_prosite_pattern

# prosite refs
sp = open(sys.argv[1])
prosite_refs = get_prosite_refs(sp)
sp.close()

# sequence
sp = open(sys.argv[1])
iterator = SProt.Iterator(sp, SProt.SequenceParser())
seq = iterator.next().seq
sp.close()

for id in prosite_refs:
    print id
    pattern = get_prosite_pattern(id)
    print pattern
    p = Pattern.compile(pattern)
m = p.search(seq)
    print "[", m.start(), ":", m.end(), "]", seq[m.start():m.end()]

A.5.3. GenBank

Solution A.32. Extracting the complete CDS from a GenBank entry

Exercise 11.15

```python
import string

def get_complete_cds(record):
    # record should be an instance of Bio.SeqRecord.Record
    if string.find(record.description, 'complete cds') == -1:
        return None
    for feature in record.features:
        if feature.type == 'CDS':
            seq = record.seq
            return seq[feature.location.start.position:feature.location.end.position]
    return ""
```

A.5.4. Blast

Solution A.33. Local Blast

Exercise 11.16

```python
from Bio.Blast import *
from Bio.SeqIO import FASTA
import sys

query_file = sys.argv[1]

# blast
if len(sys.argv) > 2:
    E_VALUE_THRESH=sys.argv[2]
else:
```
E_VALUE_THRESH=0
done={}
blast_parser = NCBIStandalone.BlastParser()
blastcmd='/local/gensoft/bin/scripts/blastall'
blast_out, error_info = NCBIStandalone.blastall(blastcmd=blastcmd,
program='blastp',
database='swissprot',
infile=query_file,
expectation=1,
descriptions=10,
alignments=10)

blast_record = blast_parser.parse(blast_out)
for (description,alignment) in zip(blast_record.descriptions,blast_record.alignments):
    hsp_nb = 0
    for hsp in alignment.hsps:
        hsp_nb = hsp_nb + 1
        if hsp.expect <= E_VALUE_THRESH:
            sbjct=hsp.sbjct.replace('-',")
            print "%s HSP %d " % (description.title, hsp_nb)

Solution A.34. Remote Blast: save results

Exercise 11.17

# run: script query_file [db [result_file]]
# from Bio.Blast import NCBIWWW
from Bio import Fasta
from sys import *

query_file = open(argv[1])
if len(argv) >= 3:
    # e.g: nr
    db = argv[2]
else:
    db = 'swissprot'
if len(argv) >= 4:
    result_file = argv[3]
else:
    result_file = argv[1] + ".blast"
Appendix A. Solutions

fasta = Fasta.Iterator(query_file)
query = fasta.next()
query_file.close()

results_handle = NCBIWWW.blast('blastp', db,
        query, expect=1,
        format_type='HTML',
        descriptions = 100, alignments = 100)
blast_results = results_handle.read()

save_file = open(result_file, 'w')
save_file.write(blast_results)
save_file.close()

print "Results saved in : ", result_file

Solution A.35. Remote Blast: parse results

Exercise 11.18

from Bio.Blast import NCBIWWW
from sys import *

blast_results = open(argv[1])
blast_parser = NCBIWWW.BlastParser()
record = blast_parser.parse(blast_results)

for (description,alignment) in zip(record.descriptions,record.alignments):
    if description.e == 0.0:
        for hsp in alignment.hsps:
            print '\n****Alignment****
            print 'sequence:', alignment.title
            print 'length:', alignment.length
            print 'e value:', hsp.expect
            print hsp.query[0:75] + '...'
            print hsp.match[0:75] + '...'
            print hsp.sbjct[0:75] + '...

❶ The zip Python function merge 2 lists with one item from each list every 2 items. So, in the code above, you get a list with a description, an alignment, the next description, the next alignment, etc...
Solution A.36. Local PSI-Blast

Exercise 11.19
rom Bio.Blast import *
from sys import *

file = argv[1]

E_VALUE_THRESH = 0.04

blast_out, error_info = NCBIStandalone.blastpgp(
    blastcmd='/local/gensoft/bin/scripts/blastpgp',
    database='swissprot',
    infile=file,
    npasses=2)

b_parser = NCBIStandalone.PSIBlastParser()

# this creates a Bio.Blast.Record.PSIBlast
b_record = b_parser.parse(blast_out)

for round in b_record.rounds:
    for alignment in round.alignments:
        for hsp in alignment.hsps:
            if hsp.expect < E_VALUE_THRESH:
                print '****Alignment****
                print 'sequence:', alignment.title
                print 'length:', alignment.length
                print 'e value:', hsp.expect
                print hsp.query[0:75] + '...'
                print hsp.match[0:75] + '...'
                print hsp.sbjct[0:75] + '...

Just adding a pattern file to this query makes it a PHI-Blast:

blast_out, error_info = NCBIStandalone.blastpgp(
    blastcmd=blastcmd,
    database='swissprot',
    infile=queryfile,
    hit_infile=patternfile)

where patternfile contains a Prosite entry (pattern file [data/ceru_human-pattern1.dat]).
Solution A.37. Search Prosite patterns with PHI-blast

Exercise 11.20

```python
from Bio.Blast import *
from Bio.WWW import *
from Bio.SeqIO import FASTA
import Bio.Prosite
from Bio import File
from sys import *
import os
import tempfile
import sprot

sp_id = argv[1]
if len(argv) > 2:
    db = argv[2]
else:
    db='swissprot'

# blast config
blast_parser = NCBIStandalone.PSIBlastParser()
E_VALUE_THRESH = 0.04
blastcmd='/local/gensoft/bin/scripts/blastpgp'

# utilities to build files for PHI-blast
def write_query(id):
    expasy = ExPASy.get_sprot_raw(id)
    sp = SProt.Iterator(expasy, SProt.SequenceParser())
    sequence = sp.next()
    fasta_file = tempfile.mktemp()
    fasta_handle=open(fasta_file,'w')
    fasta_out = FASTA.FastaWriter(fasta_handle)
    fasta_out.write(sequence)
    fasta_handle.close
    return fasta_file

def write_pattern(ref, pattern):
    patternfile = tempfile.mktemp()
    f=open(patternfile,'w')
    f.write("ID " + ref)
    f.write("\n")
    f.write("PA " + pattern)
    f.close
    return patternfile

# get SP entry and PROSITE references
print >>stderr, "Fetching ", sp_id, " from ExPASy..."
```
expasy = ExPASy.get_sprot_raw(sp_id)
prosite_refs = sprot.get_prosite_refs(expasy)
expasy.close()
queryfile = write_query(sp_id)

# actual phi-blasting of each PROSITE pattern
for ref in prosite_refs:
    pattern = sprot.get_prosite_pattern(ref)
    print >>stderr, "Doing ", ref, " ...."
    print >>stderr, pattern
    patternfile = write_pattern(ref, pattern)
    print >>stderr, "+-------------------------------"
    print >>stderr, "Running blastpgp..."
    blast_out, error_info = NCBIStandalone.blastpgp(
        blastcmd=blastcmd,
        database=db,
        infile=queryfile,
        hit_infile=patternfile)

    blast_record = blast_parser.parse(blast_out)
    for round in blast_record.rounds:
        for alignment in round.alignments:
            for hsp in alignment.hsps:
                if hsp.expect < E_VALUE_THRESH:
                    print '****Alignment****'
                    print 'sequence:', alignment.title
                    print 'length:', alignment.length
                    print 'e value:', hsp.expect
                    print hsp.query[0:75] + '....'
                    print hsp.match[0:75] + '....'
                    print hsp.sbjct[0:75] + '....'

    os.unlink(patternfile)

os.unlink(queryfile)

Solution A.38. Run FASTA

Exercise 11.21

from os import *
import string

DB_ROOT = '/local/databases/fasta'
def run_fasta(query, db):
    cmd="fasta_t -q %s %s/%s" % (query,DB_ROOT,db)
    fasta = popen(cmd, 'r')
    lines = fasta.readlines()
    fasta.close()
    if not lines:
        return None
    result = string.join(lines, "")
    return result

You can use it this way:

result=run_fasta('data/ceru_human.fasta', 'gpmam')

A.5.5. Clustalw

Solution A.39. Doing a Clustalw alignment

Exercise 11.22

import os
from Bio.Clustalw import MultipleAlignCL
from Bio.Clustalw import do_alignment
from sys import *
cline = MultipleAlignCL(argv[1])
cline.set_output(argv[2])
print "Command line: ", cline
align = do_alignment(cline)
for seq in align.get_all_seqs():
    print seq.description
    print seq.seq

Solution A.40. Align Blast HSPs

Exercise 11.23

from Bio.Blast import *
from Bio.Fasta.FastaAlign import FastaAlignment
from Bio import Alphabet
from Bio.Alphabet import IUPAC
from Bio.Clustalw import MultipleAlignCL
from Bio.Clustalw import do_alignment
from Bio.SeqIO import FASTA
import sys
import os

fasta_seqs = FastaAlignment(alphabet=IUPAC.protein)

# first, put the entire query sequence in the fasta set of sequences
fasta_handle = open(sys.argv[1])
seq = FASTA.FastaReader(fasta_handle).next()
fasta_handle.close()
fasta_seqs.add_sequence(descriptor=seq.description,
                        sequence=seq.seq.tostring())

# file for saved seqs from blast run
if len(sys.argv) > 2:
    alig_f = sys.argv[2]
else:
    alig_f = sys.argv[1] + '.seqs'

# file for saved alignment
if len(sys.argv) > 3:
    clustalw_out = sys.argv[3]
else:
    clustalw_out = sys.argv[1] + '.aln'

# blast
E_VALUE_THRESH=0
done={}
blast_parser = NCBIStandalone.BlastParser()
blastcmd='/local/gensoft/bin/scripts/blastall'
blast_out, error_info = NCBIStandalone.blastall(blastcmd=blastcmd,
                                              program='blastp',
                                              database='swissprot',
                                              infile=sys.argv[1],
                                              expectation=1,
                                              descriptions=10,
                                              alignments=10)

blast_record = blast_parser.parse(blast_out)
for (description,alignment) in zip(blast_record.descriptions,blast_record.alignments):
    hsp_nb = 0
    for hsp in alignment.hsps:
        hsp_nb = hsp_nb + 1
        if hsp.expect <= E_VALUE_THRESH:
            sbjct=hsp.sbjct.replace('-','
            title = description.title
            if done.has_key(title):
Solution A.41. Get a PSSM from an alignment

Exercise 11.24

```python
import Bio.Clustalw
import Bio.Align.AlignInfo
from Bio.Alphabet import IUPAC
from sys import *

if len(argv) == 2:
    threshold=40.0
else:
    threshold=argv[2]

align = Bio.Clustalw.parse_file(argv[1], alphabet=IUPAC.protein)
alig_len = align.get_alignment_length()
ref_seq = align.get_seq_by_num(0)
pssm = align_info.pos_specific_score_matrix(ref_seq, chars_to_ignore = ['X'])
max = len(align_info.get_column(0))

# ----------------------
print "Conservation above %d: " % threshold
for pos in xrange(alig_len):
    for letter in pssm[pos].keys():
        percent = (pssm[pos][letter] / max) * 100.0
        print letter, percent
```

---

Appendix A. Solutions
if percent > threshold:
    print "%d %s %3.2f%s" % (pos, letter, percent, '%')

Solution A.42. Plotting Cys conserved position

Exercise 11.25

Change the loop of Exercise 11.24 like this:

```python
y = []
for pos in xrange(alig_len):
    max_percent = 0
    for letter in pssm[pos].keys():
        percent = (pssm[pos][letter] / max) * 100.0
        if letter == 'C' and percent > max_percent:
            max_percent = percent
    y.append(max_percent)
```
Figure A.2. Cys conserved positions
A.6. Classes

Solution A.43. Define a PDB structure class

Exercise 12.1

class PDBStructure:
    def __init__(self):
        self._residues=[]

    def add_residue(self, name, posseq):
        residue = {'name': resname,
                   'posseq': posseq,
                   'atoms': []}
        self._residues.append(residue)
        return residue

    def add_atom(self, residue, name, coord):
        atom = {'residue': residue,
                'name': name,
                'coord': coord}
        residue['atoms'].append(atom)
        return atom

The residue is an anonymous dictionary being returned as a result to the method call, in order for the user of the class to pass it as an argument to the next add_atom method call.

Is is the residue structure passed as argument that is actually changed, not a copy of it.
Solution A.44. Define a PDB structure class (cont)

Exercise 12.2

class PDBStructure:
    def __init__(self):
        self.residues = []
        self._ssbonds = []
        self.dbrefs = ""

    def add_residue(self, model_id, chain_id, name, posseq):
        residue = {'model_id': model_id,
                   'chain_id': chain_id,
                   'name': name,
                   'posseq': posseq,
                   'atoms': []}
        self.residues.append(residue)
        return residue

    def add_atom(self, residue, name, coord, tempfactor, occupancy, altloc, element):
        atom = {'residue': residue,
                'name': name,
                'coord': coord,
                'tempfactor': tempfactor,
                'occupancy': occupancy,
                'altloc': altloc,
                'element': element}
        residue['atoms'].append(atom)
        return atom

(complete code for testing [exercises/PDBStructure_start.py])

Solution A.45. Define a PDB structure class (cont)

Exercise 12.3

Additional methods definitions:

    def get_residues(self):
        return self.residues

    def get_residues_by_name(self, self, name):
        result = []
        for residue in self.residues:
            if residue['name'] == name:
Appendix A. Solutions

result.append(residue)
return result

def get_residues_of_chain(self, chain_id):
    result = []
    for residue in self.residues:
        if residue['chain_id'] == chain_id:
            result.append(residue)
    return result

def residue_model(self, residue):
    return residue['model_id']

def residue_chain(self, residue):
    return residue['chain_id']

You can use them like this:

if __name__ == '__main__':
    print "--------------testing my class-------------------"  
    struct = PDBStructure()
    model_id = 0
    chain_id = "A"

    residue = struct.add_residue(model_id, chain_id, name = "ILE",
posseq = 1 )
    struct.add_atom(residue, name = "N",
      coord = (23.46800041, -8.01799965, -15.26200008) ,
      tempfactor=169.09, occupancy = 1.0, altloc = 0,
      element = "N")
    struct.add_atom(residue, name = "CZ",
      coord = (125.50499725, 4.50500011, -19.14800072),
      tempfactor=169.09, occupancy = 1.0, altloc = 0,
      element = "C")

    residue = struct.add_residue(model_id, chain_id, name = "LYS",
posseq = 2 )
    struct.add_atom(residue, name = "OE1",
      coord = (126.12000275, -1.78199995, -15.04199982),
      tempfactor= 83.69, occupancy = 1.0, altloc = 0,
      element = "O")

    chain_id = "B"

    residue = struct.add_residue(model_id, chain_id, name = "HIS",
posseq = 1 )
    struct.add_atom(residue, name = "N",
      coord = (23.46800041, -8.01799965, -15.26200008) ,
Appendix A. Solutions

tempfactor=169.09, occupancy = 1.0, altloc = 0,
  element = "N")
struct.add_atom(residue, name = "CB",
  coord = (125.50499725, 4.50500011, -19.14800072),
  tempfactor=169.09, occupancy = 1.0, altloc = 0,
  element = "C")
residue = struct.add_residue(model_id, chain_id, name = "ILE",
  posseq = 2 )
struct.add_atom(residue, name = "N",
  coord = (23.46800041, -8.01799965, -15.26200008),
  tempfactor=169.09, occupancy = 1.0, altloc = 0,
  element = "N")
struct.add_atom(residue, name = "CZ",
  coord = (125.50499725, 4.50500011, -19.14800072),
  tempfactor=169.09, occupancy = 1.0, altloc = 0,
  element = "C")
print "residues of name ILE:"
for residue in struct.get_residues_by_name("ILE"):
  print residue
  print "model: ", struct.residue_model(residue)
  print "chain: ", struct.residue_chain(residue)

print "residues of chain B:"
for residue in struct.get_residues_of_chain("B"):
  print residue

(complete code for testing [exercises/PDBStructure_next.py])
Solution A.46. Bio.Alphabet class hierarchy

Exercise 12.5

Figure A.3. Biopython Alphabet class hierarchy

```
Alphabet
  |__________|_________
  |            |           |
  | Alphabet   | ProteinAlphabet |
  | size = None| letters = None |
  |__________|_____________
  |            |            |
  | SingleLetterAlphabet | DNAAlphabet |
  | size = 1   | letters = |
  |__________|__________|
  |            |            |
  | NucleotidAlphabet | RNAAlphabet |
  | letters = |
  |__________|__________|
  |            |            |
  | SecondaryStructure |
  | size = 1   | letters = |
  |__________|__________|
  |            |            |
  | AlphabetEncoder |
  | alphabet    |
  | new_letters |
  |__________|
  |            |
  | Gapped     |
  | gap_char = |
  |__________|

IUPAC
  |__________|_________
  |            |           |
  | IUPACProtein | ExtendedIUPACProtein |
  | letters = |
  |__________|_____________
  |            |            |
  | IUPACAmbiguousDNA | IUPACUnambiguousDNA |
  | letters = |
  |__________|_____________
  |            |            |
  | IUPACAmbiguousRNA | IUPACUnambiguousRNA |
  | letters = |
  |__________|_____________
  |            |            |
  | IUPACProtein |
  | id: protein |
  |__________|
  |            |
  | ExtendedIUPACProtein |
  | id: extended_protein |
  |__________|
  |            |
  | IUPACAmbiguousDNA |
  | id: ambiguous_dna |
  |__________|
  |            |
  | IUPACUnambiguousDNA |
  | id: unambiguous_dna |
  |__________|
  |            |
  | IUPACAmbiguousRNA |
  | id: ambiguous_rna |
  |__________|
  |            |
  | IUPACUnambiguousRNA |
  | id: unambiguous_rna |
  |__________|
```

Solution A.47. Define a PDB structure class (cont’)

Exercise 12.6

With an instance variable:

```python
class PDBStructure:
```
Appendix A. Solutions

```python
def verbose(self, state=None):
    if state == None:
        return self._verbose
    else:
        self._verbose = state

def __init__(self):
    self.residues = []
    self._verbose = 0

def add_residue(self, model_id, chain_id, name, posseq):
    if self.verbose():
        print "add_residue: ", " model_id: ",
        model_id, " chain_id: ", chain_id, " name: ",
        name, " posseq: " posseq

    residue = {
        'model_id': model_id,
        'chain_id': chain_id,
        'name': name,
        'posseq': posseq,
        'atoms': []
    }
    self.residues.append(residue)
    return residue
```

With a class variable:

```python
class PDBStructure:
    _verbose = 0

def verbose(self, state=None):
    if state == None:
        return PDBStructure._verbose
    else:
        PDBStructure._verbose = state

def __init__(self):
    self.residues = []

def add_residue(self, model_id, chain_id, name, posseq):
    if self.verbose():
        print "add_residue: ", " model_id: ",
        model_id, " chain_id: ", chain_id, " name: ",
        name, " posseq: " posseq

    residue = {
        'model_id': model_id,
        'chain_id': chain_id,
        'name': name,
```
Appendix A. Solutions

You can use it with:

```python
if __name__ == '__main__':
    print "--------------testing my class-------------------"
    struct = PDBStructure()
    print "verbose on"
    struct.verbose(1)
    model_id = 0
    chain_id = "A"
    residue = struct.add_residue(model_id, chain_id, name = "ILE",
                                 posseq = 1)
    struct.add_atom(residue, name = "N",
                    coord = (23.46800041, -8.01799965, -15.26200008),
                    tempfactor=169.09, occupancy = 1.0, altloc = 0,
                    element = "N")
    struct.add_atom(residue, name = "CZ",
                    coord = (125.50499725, 4.50500011, -19.14800072),
                    tempfactor=169.09, occupancy = 1.0, altloc = 0,
                    element = "C")
    residue = struct.add_residue(model_id, chain_id, name = "LYS",
                                 posseq = 2)
    struct.add_atom(residue, name = "OE1",
                    coord = (126.12000275, -1.78199995, -15.0419982),
                    tempfactor= 83.69, occupancy = 1.0, altloc = 0,
                    element = "O")
    print "verbose off"
    struct.verbose(0)
    chain_id = "B"
    residue = struct.add_residue(model_id, chain_id, name = "HIS",
                                 posseq = 1)
    struct.add_atom(residue, name = "N",
                    coord = (23.46800041, -8.01799965, -15.26200008),
                    tempfactor=169.09, occupancy = 1.0, altloc = 0,
                    element = "N")
    struct.add_atom(residue, name = "CB",
                    coord = (125.50499725, 4.50500011, -19.14800072),
```
Appendix A. Solutions

A.7. Biopython, continued

A.7.1. Enzyme

Solution A.48. EnzymeConsumer, reading one entry from a file

Exercise 13.1

```python
from Bio.ParserSupport import *

class EnzymeConsumer(AbstractConsumer):
    def __init__(self):
        self._references = ""

    def databank_reference(self, line):
        self._references += line

Which can be used this way:

handle = open(argv[1])
scanner = Enzyme._Scanner()
consumer = EnzymeConsumer()
scanner.feed(handle, consumer)
print "results: ", consumer._references
```

```python
residue = struct.add_residue(model_id, chain_id, name = "ILE",
                           posseq = 2)
struct.add_atom(residue, name = "N",
                coord = (23.46800041, -8.01799965, -15.26200008),
                tempfactor = 169.09, occupancy = 1.0, altloc = 0,
                element = "N")
struct.add_atom(residue, name = "CZ",
                coord = (125.50499725, 4.50500011, -19.14800072),
                tempfactor = 169.09, occupancy = 1.0, altloc = 0,
                element = "C")
```

print "\n-----------------Structure:------------------\n", struct

print "\n-----------------Structure:------------------\n", struct

print "\n-----------------Structure:------------------\n", struct

print "\n-----------------Structure:------------------\n", struct

Solution A.49. EnzymeConsumer, reading n entries from a file

Exercise 13.2

from Bio.ParserSupport import *
class EnzymeConsumer(AbstractConsumer):
    def __init__(self):
        self._references = {}
    def end_record(self):
        self._references[self._id] = self._refs
    def databank_reference(self, line):
        self._refs += line
    def identification(self, line):
        self._id = line
        self._refs = ""

Which can be used this way:

handle = open(argv[1])
scanner = Enzyme._Scanner()
consumer = EnzymeConsumer()
scanner.feed(handle, consumer)

for id in consumer._references.keys():
    print id, consumer._references[id]

Solution A.50. EnzymeParser

Exercise 13.3

from Bio import Enzyme
from Bio.ParserSupport import *
class EnzymeParser(AbstractParser):
    def __init__(self):
        self._scanner = Enzyme._Scanner()
        self._consumer = EnzymeConsumer()

    def parse(self, handle):
        self._scanner.feed(handle, self._consumer)

    def parse(self, handle):
        self._scanner.feed(handle, self._consumer)
Appendix A. Solutions

return self._consumer._references

This can be used like this:

from sys import *
from EnzymeParser import EnzymeParser

handle = open(argv[1])
parser = EnzymeParser()
references = parser.parse(handle)

for id in references.keys():
    print id, references[id]

(complete classes definition [exercises/EnzymeParser.py])

Solution A.51. EnzymeIterator

Exercise 13.5

The iterator might be defined like this:

import re
import string

class EnzymeIterator:
    blank = re.compile(r'^\s*$')

    def __init__(self, handle, parser=EnzymeParser()):
        self._parser = parser
        self._uhandle = File.UndoHandle(handle)

    def next(self):
        lines = ""
        while 1:
            line = self._uhandle.readline()
            if not line:
                break
            if EnzymeIterator.blank.match(line):
                break
            lines += line
            if line[:2] == '//':
                break
            if not lines:
return None

if self._parser is not None:
    return self._parser.parse(File.StringHandle(lines))
return lines

The code to use the iterator may be:

```python
from enzyme_iterator import EnzymeIterator
from sys import *

handle = open(argv[1])
iterator = EnzymeIterator(handle)
record = iterator.next()
while record:
    print record['id'], record['references']
    record = iterator.next()

handle.close()
```

The consumer and the parsers may be the same as before (Exercise 13.3). However, in order to get this output:

```python
[{'ac': 'P00450,', 'id': 'CERU_HUMAN;'},
 {'ac': 'Q61147,', 'id': 'CERU_MOUSE;'},
 {'ac': 'P13635,', 'id': 'CERU_RAT'}]
```

we need to change the `databank_reference` method:

```python
def databank_reference(self, line):
    items = line.split()[1:]
    for i in range(0, len(items)):
        if (i % 2) == 0:
            ac = items[i]
            if ac[:-1] == ',' or ac[:-1] == ';':
                ac = ac[:-1]
        else:
            id = items[i]
            if id[:-1] == ',' or id[:-1] == ';':
                id = id[:-1]
            self._references.append( { 'ac': ac, 'id': id } )
```

(Complete classes definition [exercises/enzyme_iterator.py])
Appendix A. Solutions

Solution A.52. EnzymeIterator with lookup

Exercise 13.6

The consumer and the parser are the same than in Exercise 13.5. The iterator defines an additional method, `lookup`, to search in the database:

```python
import re
import string

class EnzymeIterator:

    def __init__(self, handle, parser=EnzymeParser()):
        self._parser = parser
        self._uhandle = File.UndoHandle(handle)

    def lookup(self, id):
        ID = re.compile(r'ID\s*(?P<id>(\[\w.\]+))\s*')
        while 1:
            record = self.next()
            if not record:
                break
            m = ID.match(record['id'])
            if m.group('id') == id:
                return record
        return None

    def next(self):
        blank = re.compile(r'^\s*$')
        lines = []

        start = 1
        while 1:
            line = self._uhandle.readline()
            if start:
                while line[:2] == 'CC':
                    line = self._uhandle.readline()
                if start:
                    while line[:2] == '//':
                        line = self._uhandle.readline()
                        start = 0
            if line[:2] == '//' or line[:2] == '}' or not line:
                break
            if blank.match(line):
                break
            lines.append(line)
            if line[:2] == '//' or line[:2] == '}' or not line:
                break
```

157
if not lines:
    return None

data = string.join(lines, ",")
if self._parser is not None:
    return self._parser.parse(File.StringHandle(data))
return data

The code to use the iterator may be:

from enzyme_iterator_db import EnzymeIterator
import getopt
from sys import *
from string import *

ENZYMEDB = '/local/databases/release/Enzyme/enzyme.dat'
o, id = getopt.getopt(argv[1:], 'd:')[0]
for k,v in o:
    opts[k] = v
if opts.has_key('-d'):
    db = opts['-d']
else:
    db = ENZYMEDB
if len(id) < 1:
    usage(); sys.exit("provide an id to search")
else:
    id=id[0]

database = open(db)
iterator = EnzymeIterator(database)
record = iterator.lookup(id)
if record:
    print record["id"], join(record["references"],"")
else:
    print "id: ", id, " not found in database ", db, ""
database.close()

(solution classes definition [exercises/enzyme_iterator_db.py])

Solution A.53. EnzymeDictionary

Exercise 13.7
The iterator is the same than in Exercise 13.6.

```python
from enzyme_iterator_db import EnzymeIterator
import re

class EnzymeDictionary:
    _ID = re.compile(r'ID\s*(?P<id>(\[\w\.]+))\s*')

    def __init__(self, db):
        self._index = {}
        self._db = db
        self._handle = open(self._db)
        self._iterator = EnzymeIterator(self._handle)
        self._index = {}

    def __getitem__(self, id):
        if self._index.has_key(id):
            return self._index[id]
        while 1:
            record = self._iterator.next()
            if not record:
                self._handle.close()
                break
            self._index[record['id']] = record
            if id == record['id']:
                break
        return self._index[id]
```

The code to use the dictionary may be:

```python
from enzyme_dictionary import EnzymeDictionary
import getopt
from sys import *
from string import *

ENZYMEDB = '/local/databases/release/Enzyme/enzyme.dat'

o, id = getopt.getopt(argv[1:], 'd:')[0]
for k,v in o:
    opts[k] = v
if opts.has_key('-d'):
    db = opts['-d']
else:
```

159
db = ENZYMEDB
if len(id) < 1:
    usage(); sys.exit("provide an id to search")
else:
    id=id[0]
enzyme = EnzymeDictionary(db)
try:
    record = enzyme[id]
    print record['id'], join(record['references'],"")
except KeyError, e:
    print "key not found: ", e
print "end of lookup for ", enzyme[id]['id']

Solution A.54. EnzymeParsing module
Exercise 13.8
Create an EnzymeParsing.py file containing the required classes (the one used in Exercise 13.7 preferably).

The code to use the dictionary may be for instance:

```python
>>> from EnzymeParsing import EnzymeDictionary
>>> enzyme = EnzymeDictionary('/local/databases/release/Enzyme/enzyme.dat')
>>> print enzyme['1.1.1.5']
{'references': [{"ac": 'Q48436', 'id': 'BUDC_KLEPN'}, {"ac": 'Q04520', 'id': 'BUDC_KLETE'}], 'id': 'ID 1.1.1.5'}
```

(complete module definition [exercises/EnzymeParsing.py])

Solution A.55. Fetching enzymes referenced in a SwissProt entry and related proteins
Exercise 13.9
Fetch the enzyme entry and the corresponding SwissProt references.

```python
from EnzymeParsing import EnzymeDictionary
import sprot
from sys import *
```
Appendix A. Solutions

```python
sp_record = sprot.get_sprot_entry_local(argv[1])
enzyme_id = sprot.get_enzyme_ref(sp_record)

enzyme = EnzymeDictionary(’/local/databases/release/Enzyme/enzyme.dat’)
enzyme_record = enzyme[enzyme_id]
for ref in enzyme_record[’references’]:
    sp_id = ref[’id’]
    sp_r = sprot.get_sprot_entry_local(sp_id)
    print sp_r.entry_name, sp_r.description
```

### A.7.2. Building parsing classes for phylogenetic trees

**Solution A.56. Write a Scanner for a Phylip formatted tree**

Exercise 13.10

```python
from Bio import File
import re

class Scanner:
    def feed(self, handle, consumer):
        if isinstance(handle, File.UndoHandle):
            uhandle = handle
        else:
            uhandle = File.UndoHandle(handle)
        self._scan_tree(uhandle, consumer)

    def _scan_tree(self, self, uhandle, consumer):
        tree_text = “”.join(uhandle.readlines()).replace(“\n”, “”)
        tree_text = re.sub(“\,”, “”, tree_text)
        print “text: “, tree_text
        pos = 0
        rooted = is_rooted(tree_text)
        consumer.start_tree(rooted)

        while 1:
            c = tree_text[pos]
            if c == ’(‘:
                consumer.begin_node()
                pos += 1
            elif c == ’)’:
                consumer.end_node()
                pos += 1
```

161
elif c == ',':
    pos += 1
elif c == ':':
    # ready to process branch length
    pos += 1
    c = tree_text[pos]
    length = ''
    while re.match('[\.,\d]', c):
        length += c
        pos += 1
        c = tree_text[pos]
    consumer.branch_length(float(length))
elif c == ';':
    consumer.end_tree()
    break
elif c == '\":'
    pos += 1
else:
    name = ''
    while re.match('[\w]', c):
        name += c
        pos += 1
        c = tree_text[pos]
    consumer.leaf(name)

The definition of an is_rooted function would be:

def is_rooted(tree):
    pos = 0
    c = tree[pos]
    depth = 0
    comma = 0
    for pos in range(0, len(tree)):
        c = tree[pos]
        if c == '\(':
            depth += 1
            elif c == '\)':
                depth -= 1
            elif c == ',':
                if depth == 1:
                    comma += 1
            return comma == 1

Solution A.57. Write a Consumer for building a tree

Exercise 13.11
Let us first see a Consumer class.

```python
from Bio.ParserSupport import *

class Consumer(AbstractConsumer):
    def __init__(self):
        self.stack = Stack()
        self.data = None

    def start_tree(self, rooted=False):
        self.rooted = rooted
        self.pos = 0
        self.depth = 0
        self.stack.empty()

    def begin_node(self):
        node = Node()
        self.stack.push(node)
        self.depth = self.depth + 1

    def end_node(self):
        # 3 cases: depth 1 and unrooted tree (3 nodes)
        # or internal node (2 nodes)
        # or (depth 1 and rooted tree) (2 nodes)
        if self.depth == 1 and not self.rooted:
            right = self.stack.pop()
            middle = self.stack.pop()
            left = self.stack.pop()
            parent = self.stack.top()
            parent.left = left
            parent.middle = middle
            parent.right = right
        else:
            # internal node or rooted tree
            right = self.stack.pop()
            left = self.stack.pop()
            parent = self.stack.top()
            parent.left = left
            parent.right = right
            self.depth -= 1

    def leaf(self, name):
        node = Node()
        node.name = name
        self.stack.push(node)

    def branch_length(self, l):
        node = self.stack.top()
        node.length = l
```

163
def end_tree(self):
    self.data = self.stack.pop()

Now, this is our version of a simple Node class:

class Node:
    ""
    An over simplified representation of a tree (just nodes) ""
    def __init__(self, name=None, left=None, middle=None, right=None, length=None):
        self.name = name
        self.left = left
        self.right = right
        self.length = length
        self.middle = middle

    def __str__(self):
        if self.name is None:
            if self.middle is not None:
                return "(" + str(self.left.__str__()) + "," + str(self.middle.__str__()) + "," + str(self.right.__str__()) + ")"
            else:
                return "(" + str(self.left.__str__()) + "," + str(self.right.__str__()) + ':' + str(self.length) + ")"
        else:
            return str(self.name) + ':' + str(self.length)

The following provides a definition of a simplistic Stack class:

class Stack:
    def __init__(self):
        self._l = []

    def push(self, item):
        self._l.append(item)

    def pop(self):
        last = self._l[-1]
        self._l = self._l[:-1]
        return last

    def top(self):
        return self._l[-1]

    def empty(self):
        return self._l[-1]
self._l = []

Solution A.58. Write a Parser that put everything together

Exercise 13.12

import sys
from Bio.ParserSupport import *

class Parser(AbstractParser):
    def __init__(self):
        self._scanner = Scanner()
        self._consumer = Consumer()

    def parse(self, handle):
        self._scanner.feed(handle, self._consumer)
        return self._consumer.data

A.7.3. PDB

Solution A.59. Get a PDB entry from RCSB Web server

Exercise 13.13

import urllib
import string
from Bio import File

def get_pdb_entry_remote(id):
    fullcgi = "http://www.rcsb.org/pdb/cgi/export.cgi/%s.pdb?format=PDB&compression=None" % id

    #print fullcgi
    handle = urllib.urlopen(fullcgi)

    uhandle = File.UndoHandle(handle)

    if not uhandle.peekline():
        raise IOError, "no results"
Appendix A. Solutions

```
return uhandle
def get_pdb_entry_local(id):
    id = string.lower(id)
    filename = "data/pdb" + id + ".pdb"
    try:
        print "trying to open ", filename
        handle = open(filename)
        except IOError, e:
            print e
            filename = "data/" + id + ".pdb"
            try:
                print "trying to open ", filename
                handle = open(filename)
                except IOError, e:
                    print e
                    filename = "data/pdb" + id + ".ent"
                    try:
                        print "trying to open ", filename
                        handle = open(filename)
                        except IOError, e:
                            print e
                            return None

    return handle

Solution A.60. Define a PDBStructure class

Exercise 13.14

Add the following code to the code already written in Exercise 12.6:

```
def __str__(self):
    for residue in self._residues:
        print residue
    return ""
def set_id(self, structure_id):
    self._structure_id = structure_id
def set_pdb_ident(self, pdb_ident):
    self.pdb_ident = pdb_ident
def add_dbref(self, dbref):
    self.dbrefs += dbref
Appendix A. Solutions

```python
def add_ssbond(self, _from, _to):
    self._ssbonds.append({'from': _from, 'to': _to})
```

(complete class definition [exercises/PDBStructure1.py]).

**Solution A.61. Define a PDBConsumer class**

Exercise 13.15

```python
# PDBConsumer creates instances of PDBStructure
#
from Bio.ParserSupport import *
from PDBParser import PDBParser
from PDBStructure import PDBStructure
import sys
class PDBConsumer(AbstractConsumer):
    _verbose = 0
    def __init__(self):
        self._current_struct = None
    def set_id(self, structure_id):
        # start a new structure
        if self._verbose:
            print "set_id: ", structure_id
        self._current_struct = PDBStructure()
        self._current_struct.set_id(structure_id)
    def set_pdb_ident(self, pdb_ident):
        if self._verbose:
            print "set_pdb_ident: ", pdb_ident
        self._current_struct.set_pdb_ident(pdb_ident)
    def set_symmetry(self, spacegroup, cell):
        pass
    def init_model(self, model_id):
        if self._verbose:
            print "init_model: ", model_id
        self._current_model_id = model_id
    def init_chain(self, chain_id):
        pass
```
if self._verbose:
    print "init_chain: ", chain_id
self._current_chain_id = chain_id

def set_anisou(self, anisou):
    pass

def set_sigatm(self, sigatm):
    pass

def set_siguij(self, siguij):
    pass

def set_ssbond(self, _from, _to):
    self._current_struct.add_ssbond(_from, _to)

def set_dbref(self, line):
    self._current_struct.add_dbref(line)

def init_residue(self, name, field, posseq, icode):
    if self._verbose:
    residue = self._current_struct.add_residue(self._current_model_id,
                                              self._current_chain_id,
                                              name, field, posseq, icode)
    self._current_residue = residue

def init_atom(self, name, coord, tempfactor, occupancy, altloc, element):
    if self._verbose:
    self._current_struct.add_atom(self._current_residue,
                                   name, coord, tempfactor,
                                   occupancy, altloc, element)

def get(self):
    return self._current_struct

Solution A.62. Compute disulfid bonds in 1KCW

Exercise 13.16 You first need to complete the PDBStructure class by adding a disulfid_bridges method:
Appendix A. Solutions

BRIDGE_DIST=8.0

def dist(self, a1, a2):
    dx = a1[0] - a2[0]
    dy = a1[1] - a2[1]
    return math.sqrt(dx*dx + dy*dy + dz*dz)

def disulfid_bridges(self):
    sulfurs=[]
    for cys_residue in self.get_residues_by_name('CYS):
        #print "cys: ",cys_residue['name'], cys_residue['posseq']
        for atom in cys_residue['atoms']:
            if atom['name'] == 'SG':
                sulfurs.append({'posseq': cys_residue['posseq'],
                                'atom': atom})

    result=[]
    nb = len(sulfurs)
    for i in xrange(nb):
        for j in xrange(i+1, nb):
            d = self.dist(sulfurs[i]['atom']['coord'], sulfurs[j]['atom']['coord'])
            if d < self.BRIDGE_DIST:
                print "residue %d in contact with residue %d (distance:%.3f)." % (sulfurs[i]['posseq'], sulfurs[j]['posseq'], d)
                print 	 , sulfurs[i]['atom']['coord'], 
	 , sulfurs[j]['atom']['coord']
                result.append({'from': sulfurs[i]['posseq'],
                                'to': sulfurs[j]['posseq'],
                                'dist': d})

    return result

Then, you can use the class from:

#
# Compute disulfide bonds.
# - search for sulfur (S) atoms in Cys residues of the structure
# - compute distance between all of them
# - displays residue pairs (position) where distance < BRIDGE_DIST
#
from PDBParser import PDBParser
from PDBConsumer import PDBConsumer
from PDBStructure import PDBStructure
import sys

if __name__ == '__main__':
    p = PDBParser(PDBConsumer())
    struct = p.get("scratch", sys.argv[1])

    detected = struct.disulfid_bridges()
    for annot in struct._ssbonds:
        found = 0
        for detect in detected:
            if annot['from'] == detect['from'] and annot['to'] == detect['to']:
                print annot, " also detected: ", detect['dist']
                found = 1
                break
        if not found:
            print annot, " not found"

(complete class definition [exercises/PDBStructure2.py])

Solution A.63. Compare 3D disulfid bonds with Cys positions in the alignment (take #1).

Exercise 13.17

import Bio.Clustalw
from Bio.Seq import Seq
import Bio.Align.AlignInfo
from Bio.WWW import *
from Bio.Alphabet import IUPAC
from Bio.SwissProt import SProt
import sys
from os import *
import string
from WWWPDB import *
from PDBParser import PDBParser
from PDBConsumer import PDBConsumer

def get_pdb_entries(sprot):
    refs = []
    for ref in sprot.cross_references:
        if ref[0] == 'PDB':
            refs.append(ref[1])
Appendix A. Solutions

return refs

def get_sprot_entry_remote (sprot_id):
    expasy = ExPASy.get_sprot_raw(sprot_id)
    iterator = SProt.Iterator(expasy, SProt.RecordParser())
    entry = iterator.next()
    expasy.close()
    return entry

def get_sprot_entry_local (sprot_id):
    cmd="golden sprot:" + sprot_id
    print "Fetching entry by ", cmd
    golden = popen(cmd, ’r’)  
    iterator = SProt.Iterator(golden, SProt.RecordParser())
    entry = iterator.next()
    golden.close()
    return entry

def align2seqpos(seq,col):
    "returns the original sequence position from a gapped sequence position"
    s=list(seq.tostring())
    gaps = 0
    for i in xrange(len(s)):
        if i >= col:
            break
        if s[i] == ‘-’:
            gaps = gaps + 1
    result = col - gaps
    return result

def get_seq_description(alignment,seq_nb):
    return alignment._records[seq_nb].description

***
open alignment and create pssm
***

align = Bio.Clustalw.parse_file(sys.argv[1], alphabet=IUPAC.protein)
ref_seq = align.get_seq_by_num(0)
pssm = align_info.pos_specific_score_matrix(ref_seq, chars_to_ignore = [‘X’])
max = len(align_info.get_column(0))
alig_len = align.get_alignment_length()

***
fetch PDB entry from swissprot references
***
seq_id = get_seq_description(align,0)

171
print "Swissprot ID: ", seq_id
try:
    seq_record = get_sprot_entry_remote(seq_id)
except IOError, e:
    #print "Remote access not available: ", e
    seq_record = get_sprot_entry_local(seq_id)
refs = get_pdb_entries(seq_record)
print "PDB reference: ", refs
try:
    pdb_handle = get_pdb_entry_remote(refs[0])
except IOError, e:
    print e
    pdb_handle = get_pdb_entry_local(refs[0])

p = PDBParser(PDBConsumer())
struct = p.get_handle("scratch", pdb_handle)

# comparison
print "--- detected bonds from PDB coordinates:-------------"
detected = struct.disulfid_bridges()
for detect in detected:
    print detect

print "--- detected bonds from alignment: ----------"
for pos in xrange(alig_len):
    percent = (pssm[pos]["C"] / max) * 100.0
    if percent > 40.0:
        print "potential disulfid? ", pos + 1, " %: ", percent

Solution A.64. Compare 3D disulfid bonds with Cys positions in the alignment (take #2).

first add a method pdb2seq_pos into class PDBStructure:

```python
def pdb2seq_pos(self):
```

```
DBREF 1KCW 1 338 SWS P00450 CERU_HUMAN 20 357
DBREF 1KCW 347 474 SWS P00450 CERU_HUMAN 366 493
DBREF 1KCW 483 804 SWS P00450 CERU_HUMAN 502 903
DBREF 1KCW 892 1040 SWS P00450 CERU_HUMAN 904 1059
```
Appendix A. Solutions

```python
lines = self.dbrefs.split("\n")
items = lines[0].split()
pos_pdb = string.atoi(items[7])
pos_sws = string.atoi(items[2])
if pos_sws < pos_pdb:
    return pos_pdb - pos_sws
else:
    return pos_sws - pos_pdb
```

and use it as follows:

Exercise 13.18

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

import Bio.Clustalw
from Bio.Seq import Seq
import Bio.Align.AlignInfo
from Bio.WWW import *
from Bio.Alphabet import IUPAC
from Bio.SwissProt import SProt
import sys
from os import *
import string
from WWWPDB import *
from PDBParser import PDBParser
from PDBConsumer import PDBConsumer

def get_pdb_entries(sprot):
    refs=[]
    for ref in sprot.cross_references:
        if ref[0] == 'PDB':
            refs.append(ref[1])
    return refs

def get_sprot_entry_remote (sprot_id):
    expasy = ExPASy.get_sprot_raw(sprot_id)
    iterator = SProt.Iterator(expasy, SProt.RecordParser())
    entry = iterator.next()
    expasy.close()
    return entry

def get_sprot_entry_local (sprot_id):
    cmd="golden sprot:" + sprot_id
    print "Fetching entry by ",cmd
    golden = popen(cmd, 'r')
```
iterator = SProt.Iterator(golden, SProt.RecordParser())
entry = iterator.next()
golden.close()
return entry

def align2seqpos(seq,col):
    "returns the original sequence position from a gapped sequence position"
    s = list(seq.tostring())
    gaps = 0
    for i in xrange(len(s)):
        if i >= col:
            break
        if s[i] == '-':
            gaps = gaps + 1
    #print "gaps: ", gaps
    result = col - gaps
    return result

def get_seq_description(alignment,seq_nb):
    return alignment._records[seq_nb].description

    open alignment and create pssm
    
    align = Bio.Clustalw.parse_file(sys.argv[1], alphabet=IUPAC.protein)
    ref_seq = align.get_seq_by_num(0)
    pssm = align_info.pos_specific_score_matrix(ref_seq, chars_to_ignore = ['X'])
    max = len(align_info.get_column(0))
    alig_len = align.get_alignment_length()

    fetch PDB entry from swissprot references
    
    seq_id = get_seq_description(align,0)
    print "Swissprot ID: ", seq_id
    try:
        seq_record = get_sprot_entry_remote(seq_id)
        except IOError, e:
            print "Remote acces not available: ", e
            seq_record = get_sprot_entry_local(seq_id)
        refs = get_pdb_entries(seq_record)
        print "PDB reference: ", refs
        try:
            pdb_handle = get_pdb_entry_remote(refs[0])
            except IOError, e:
                print e
Appendix A. Solutions

```python
pdb_handle = get_pdb_entry_local(refs[0])
p = PDBParser(PDBConsumer())
struct = p.get_handle("scratch", pdb_handle)

# comparison
print "--- detected bonds from PDB coordinates:----------"
detected = struct.disulfid_bridges()
for detect in detected:
    print detect

diffpos = struct.pdb2seq_pos()
print "difference in PDB and sequence position: ", diffpos

print "--- detected bonds from alignment: ----------"
for pos in xrange(alig_len):
    percent = (pssm[pos]["C"] / max) * 100.0
    vector_y.append(percent)
    if percent > 40.0:
        print "potential disulfid? ", pos + 1, " %: ", \
        percent, " (seq: ", align2seqpos(ref_seq, pos) + 1, \
        " PDB seq: ", align2seqpos(ref_seq, pos) + 1 - diffpos, ")"
```

(complete PDBStructure class definition [exercises/PDBStructure.py])
Appendix A. Solutions
Appendix B. Bibliography

Bibliography


Appendix B. Bibliography