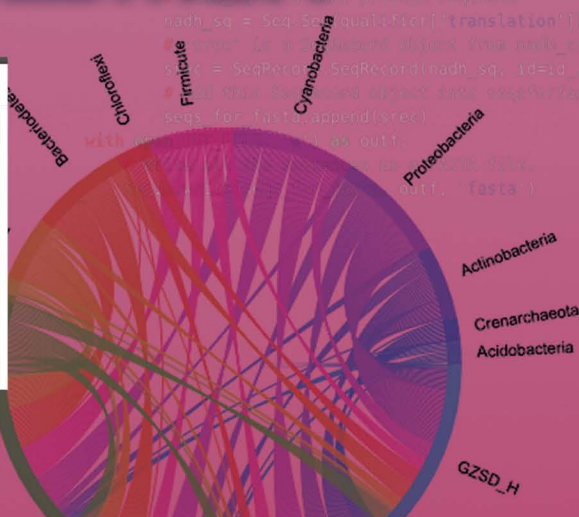
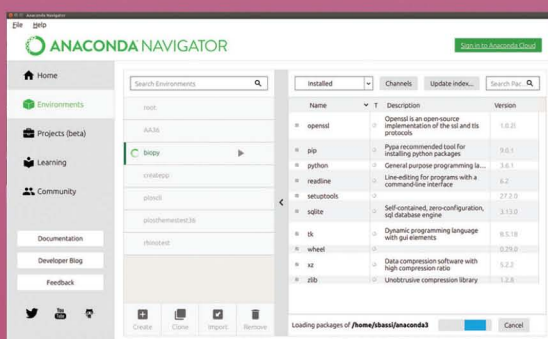


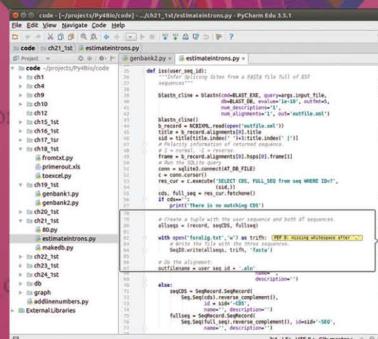
Chapman & Hall/CRC
Mathematical and Computational Biology Series

PYTHON FOR BIOINFORMATICS SECOND EDITION



```
def sortmarkers(crms, end):  
    """Sort markers into chromosomes"""  
    i = 0  
    crms_o = [[] for r in range(len(end))] # Sort the marker positions.  
    crms_o = [[] for r in range(len(end))] # Sort the marker positions.  
    for crm in crms:  
        # Add the marker position to the chromosome  
        crms_o[crms.index(crm)].append(crm[1])  
    crms_o = [[] for r in range(len(end))] # Sort the marker positions.  
    for pos in order:  
        for mark in crms[i]:  
            if pos == mark[1]:  
                crms_o[i].append(mark[0])  
            i += 1  
    return crms_o
```

biopython



SEBASTIAN BASSI

CRC Press
Taylor & Francis Group

A CHAPMAN & HALL BOOK



PYTHON FOR BIOINFORMATICS

SECOND EDITION

CHAPMAN & HALL/CRC

Mathematical and Computational Biology Series

Aims and scope:

This series aims to capture new developments and summarize what is known over the entire spectrum of mathematical and computational biology and medicine. It seeks to encourage the integration of mathematical, statistical, and computational methods into biology by publishing a broad range of textbooks, reference works, and handbooks. The titles included in the series are meant to appeal to students, researchers, and professionals in the mathematical, statistical and computational sciences, fundamental biology and bioengineering, as well as interdisciplinary researchers involved in the field. The inclusion of concrete examples and applications, and programming techniques and examples, is highly encouraged.

Series Editors

N. F. Britton

*Department of Mathematical Sciences
University of Bath*

Xihong Lin

*Department of Biostatistics
Harvard University*

Nicola Mulder

*University of Cape Town
South Africa*

Maria Victoria Schneider

European Bioinformatics Institute

Mona Singh

*Department of Computer Science
Princeton University*

Anna Tramontano

*Department of Physics
University of Rome La Sapienza*

Proposals for the series should be submitted to one of the series editors above or directly to:

CRC Press, Taylor & Francis Group

3 Park Square, Milton Park
Abingdon, Oxfordshire OX14 4RN
UK

Published Titles

An Introduction to Systems Biology: Design Principles of Biological Circuits

Uri Alon

Glycome Informatics: Methods and Applications

Kiyoko F. Aoki-Kinoshita

Computational Systems Biology of Cancer

*Emmanuel Barillot, Laurence Calzone,
Philippe Hupé, Jean-Philippe Vert, and
Andrei Zinovyev*

Python for Bioinformatics, Second Edition

Sebastian Bassi

Quantitative Biology: From Molecular to Cellular Systems

Sebastian Bassi

Methods in Medical Informatics: Fundamentals of Healthcare Programming in Perl, Python, and Ruby

Jules J. Berman

Chromatin: Structure, Dynamics, Regulation

Ralf Blossey

Computational Biology: A Statistical Mechanics Perspective

Ralf Blossey

Game-Theoretical Models in Biology

Mark Broom and Jan Rychtář

Computational and Visualization Techniques for Structural Bioinformatics Using Chimera

Forbes J. Burkowski

Structural Bioinformatics: An Algorithmic Approach

Forbes J. Burkowski

Spatial Ecology

*Stephen Cantrell, Chris Cosner, and
Shigui Ruan*

Cell Mechanics: From Single Scale- Based Models to Multiscale Modeling

*Arnaud Chauvière, Luigi Preziosi,
and Claude Verdier*

Bayesian Phylogenetics: Methods, Algorithms, and Applications

Ming-Hui Chen, Lynn Kuo, and Paul O. Lewis

Statistical Methods for QTL Mapping

Zehua Chen

An Introduction to Physical Oncology: How Mechanistic Mathematical Modeling Can Improve Cancer Therapy Outcomes

*Vittorio Cristini, Eugene J. Koay,
and Zhihui Wang*

Normal Mode Analysis: Theory and Applications to Biological and Chemical Systems

Qiang Cui and Ivet Bahar

Kinetic Modelling in Systems Biology

Oleg Demin and Igor Goryanin

Data Analysis Tools for DNA Microarrays

Sorin Draghici

Statistics and Data Analysis for Microarrays Using R and Bioconductor, Second Edition

Sorin Drăghici

Computational Neuroscience: A Comprehensive Approach

Jianfeng Feng

Biological Sequence Analysis Using the SeqAn C++ Library

Andreas Gogol-Döring and Knut Reinert

Gene Expression Studies Using Affymetrix Microarrays

Hinrich Göhlmann and Willem Talloen

Handbook of Hidden Markov Models in Bioinformatics

Martin Gollery

Meta-analysis and Combining Information in Genetics and Genomics

Rudy Guerra and Darlene R. Goldstein

Differential Equations and Mathematical Biology, Second Edition

D.S. Jones, M.J. Plank, and B.D. Sleeman

Knowledge Discovery in Proteomics

Igor Jurisica and Dennis Wigle

Introduction to Proteins: Structure, Function, and Motion

Amit Kessel and Nir Ben-Tal

Published Titles (continued)

RNA-seq Data Analysis: A Practical Approach

*Eija Korpelainen, Jarno Tuimala,
Panu Somervuo, Mikael Huss, and Garry Wong*

Introduction to Mathematical Oncology

*Yang Kuang, John D. Nagy, and
Steffen E. Eikenberry*

Biological Computation

Ehud Lamm and Ron Unger

Optimal Control Applied to Biological Models

Suzanne Lenhart and John T. Workman

Clustering in Bioinformatics and Drug Discovery

John D. MacCuish and Norah E. MacCuish

Spatiotemporal Patterns in Ecology and Epidemiology: Theory, Models, and Simulation

*Horst Malchow, Sergei V. Petrovskii, and
Ezio Venturino*

Stochastic Dynamics for Systems Biology

Christian Mazza and Michel Benaïm

Statistical Modeling and Machine Learning for Molecular Biology

Alan M. Moses

Engineering Genetic Circuits

Chris J. Myers

Pattern Discovery in Bioinformatics: Theory & Algorithms

Laxmi Parida

Exactly Solvable Models of Biological Invasion

Sergei V. Petrovskii and Bai-Lian Li

Computational Hydrodynamics of Capsules and Biological Cells

C. Pozrikidis

Modeling and Simulation of Capsules and Biological Cells

C. Pozrikidis

Cancer Modelling and Simulation

Luigi Preziosi

Introduction to Bio-Ontologies

Peter N. Robinson and Sebastian Bauer

Dynamics of Biological Systems

Michael Small

Genome Annotation

*Jung Soh, Paul M.K. Gordon, and
Christoph W. Sensen*

Niche Modeling: Predictions from Statistical Distributions

David Stockwell

Algorithms for Next-Generation Sequencing

Wing-Kin Sung

Algorithms in Bioinformatics: A Practical Introduction

Wing-Kin Sung

Introduction to Bioinformatics

Anna Tramontano

The Ten Most Wanted Solutions in Protein Bioinformatics

Anna Tramontano

Combinatorial Pattern Matching Algorithms in Computational Biology Using Perl and R

Gabriel Valiente

Managing Your Biological Data with Python

*Allegra Via, Kristian Rother, and
Anna Tramontano*

Cancer Systems Biology

Edwin Wang

Stochastic Modelling for Systems Biology, Second Edition

Darren J. Wilkinson

Big Data Analysis for Bioinformatics and Biomedical Discoveries

Shui Qing Ye

Bioinformatics: A Practical Approach

Shui Qing Ye

Introduction to Computational Proteomics

Golan Yona

Chapman & Hall/CRC Mathematical and Computational Biology Series

PYTHON FOR BIOINFORMATICS

SECOND EDITION

SEBASTIAN BASSI



CRC Press

Taylor & Francis Group

Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

MATLAB® is a trademark of The MathWorks, Inc. and is used with permission. The MathWorks does not warrant the accuracy of the text or exercises in this book. This book's use or discussion of MATLAB® software or related products does not constitute endorsement or sponsorship by The MathWorks of a particular pedagogical approach or particular use of the MATLAB® software.

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

© 2018 by Taylor & Francis Group, LLC
CRC Press is an imprint of Taylor & Francis Group, an Informa business

No claim to original U.S. Government works

Printed on acid-free paper
Version Date: 20170626

International Standard Book Number-13: 978-1-1380-3526-3 (Hardback)

This book contains information obtained from authentic and highly regarded sources. Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, please access www.copyright.com (<http://www.copyright.com>) or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. CCC is a not-for-profit organization that provides licenses and registration for a variety of users. For organizations that have been granted a photocopy license by the CCC, a separate system of payment has been arranged.

Trademark Notice: Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation without intent to infringe.

Library of Congress Cataloging-in-Publication Data

Names: Bassi, Sebastian, author.
Title: Python for bioinformatics / Sebastian Bassi.
Description: Second edition. | Boca Raton : CRC Press, 2017. | Series: Chapman & Hall/CRC mathematical and computational biology | Includes bibliographical references and index.
Identifiers: LCCN 2017014460 | ISBN 9781138035263 (pbk. : alk. paper) | ISBN 9781138094376 (hardback : alk. paper) | ISBN 9781315268743 (ebook) | ISBN 9781351976961 (ebook) | ISBN 9781351976954 (ebook) | ISBN 9781351976947 (ebook)
Subjects: LCSH: Bioinformatics. | Python (Computer program language)
Classification: LCC QH324.2 .B387 2017 | DDC 570.285--dc23
LC record available at <https://lccn.loc.gov/2017014460>

Visit the Taylor & Francis Web site at
<http://www.taylorandfrancis.com>

and the CRC Press Web site at
<http://www.crcpress.com>

Contents

List of Figures	xvii
List of Tables	xxi
Preface to the First Edition	xxiii
Preface to the Second Edition	xxv
Acknowledgments	xxix
SECTION I Programming	
CHAPTER 1 ■ Introduction	3
1.1 WHO SHOULD READ THIS BOOK	3
1.1.1 What the Reader Should Already Know	4
1.2 USING THIS BOOK	4
1.2.1 Typographical Conventions	4
1.2.2 Python Versions	5
1.2.3 Code Style	5
1.2.4 Get the Most from This Book without Reading It All	6
1.2.5 Online Resources Related to This Book	7
1.3 WHY LEARN TO PROGRAM?	7
1.4 BASIC PROGRAMMING CONCEPTS	8
1.4.1 What Is a Program?	8
1.5 WHY PYTHON?	10
1.5.1 Main Features of Python	10
1.5.2 Comparing Python with Other Languages	11
1.5.3 How Is It Used?	14
1.5.4 Who Uses Python?	15
1.5.5 Flavors of Python	15
1.5.6 Special Python Distributions	16
1.6 ADDITIONAL RESOURCES	17

CHAPTER 2 ■ First Steps with Python	19
2.1 INSTALLING PYTHON	20
2.1.1 Learn Python by Using It	20
2.1.2 Install Python Locally	20
2.1.3 Using Python Online	21
2.1.4 Testing Python	22
2.1.5 First Use	22
2.2 INTERACTIVE MODE	23
2.2.1 Baby Steps	23
2.2.2 Basic Input and Output	23
2.2.3 More on the Interactive Mode	24
2.2.4 Mathematical Operations	26
2.2.5 Exit from the Python Shell	27
2.3 BATCH MODE	27
2.3.1 Comments	29
2.3.2 Indentation	30
2.4 CHOOSING AN EDITOR	32
2.4.1 Sublime Text	32
2.4.2 Atom	33
2.4.3 PyCharm	34
2.4.4 Spyder IDE	35
2.4.5 Final Words about Editors	36
2.5 OTHER TOOLS	36
2.6 ADDITIONAL RESOURCES	37
2.7 SELF-EVALUATION	37
CHAPTER 3 ■ Basic Programming: Data Types	39
3.1 STRINGS	40
3.1.1 Strings Are Sequences of Unicode Characters	41
3.1.2 String Manipulation	42
3.1.3 Methods Associated with Strings	42
3.2 LISTS	44
3.2.1 Accessing List Elements	45
3.2.2 List with Multiple Repeated Items	45
3.2.3 List Comprehension	46
3.2.4 Modifying Lists	47

3.2.5	Copying a List	49
3.3	TUPLES	49
3.3.1	Tuples Are Immutable Lists	49
3.4	COMMON PROPERTIES OF THE SEQUENCES	51
3.5	DICTIONARIES	54
3.5.1	Mapping: Calling Each Value by a Name	54
3.5.2	Operating with Dictionaries	56
3.6	SETS	59
3.6.1	Unordered Collection of Objects	59
3.6.2	Set Operations	60
3.6.3	Shared Operations with Other Data Types	62
3.6.4	Immutable Set: Frozenset	63
3.7	NAMING OBJECTS	63
3.8	ASSIGNING A VALUE TO A VARIABLE VERSUS BINDING A NAME TO AN OBJECT	64
3.9	ADDITIONAL RESOURCES	67
3.10	SELF-EVALUATION	68
CHAPTER 4 ■ Programming: Flow Control		69
4.1	IF-ELSE	69
4.1.1	Pass Statement	74
4.2	FOR LOOP	75
4.3	WHILE LOOP	77
4.4	BREAK: BREAKING THE LOOP	78
4.5	WRAPPING IT UP	80
4.5.1	Estimate the Net Charge of a Protein	80
4.5.2	Search for a Low-Degeneration Zone	81
4.6	ADDITIONAL RESOURCES	83
4.7	SELF-EVALUATION	83
CHAPTER 5 ■ Handling Files		85
5.1	READING FILES	86
5.1.1	Example of File Handling	87
5.2	WRITING FILES	89
5.2.1	File Reading and Writing Examples	90
5.3	CSV FILES	90

5.4	PICKLE: STORING AND RETRIEVING THE CONTENTS OF VARIABLES	94
5.5	JSON FILES	96
5.6	FILE HANDLING: OS, OS.PATH, SHUTIL, AND PATH.PY MODULE	98
5.6.1	path.py Module	100
5.6.2	Consolidate Multiple DNA Sequences into One FASTA File	102
5.7	ADDITIONAL RESOURCES	102
5.8	SELF-EVALUATION	103
CHAPTER 6 ■ Code Modularizing		105
6.1	INTRODUCTION TO CODE MODULARIZING	105
6.2	FUNCTIONS	106
6.2.1	Standard Way to Make Python Code Modular	106
6.2.2	Function Parameter Options	110
6.2.3	Generators	113
6.3	MODULES AND PACKAGES	114
6.3.1	Using Modules	115
6.3.2	Packages	116
6.3.3	Installing Third-Party Modules	117
6.3.4	Virtualenv: Isolated Python Environments	119
6.3.5	Conda: Anaconda Virtual Environment	121
6.3.6	Creating Modules	124
6.3.7	Testing Modules	125
6.4	ADDITIONAL RESOURCES	127
6.5	SELF-EVALUATION	128
CHAPTER 7 ■ Error Handling		129
7.1	INTRODUCTION TO ERROR HANDLING	129
7.1.1	Try and Except	131
7.1.2	Exception Types	134
7.1.3	Triggering Exceptions	135
7.2	CREATING CUSTOMIZED EXCEPTIONS	136
7.3	ADDITIONAL RESOURCES	137
7.4	SELF-EVALUATION	138
CHAPTER 8 ■ Introduction to Object Orienting Programming (OOP)		139
8.1	OBJECT PARADIGM AND PYTHON	139

8.2	EXPLORING THE JARGON	140
8.3	CREATING CLASSES	142
8.4	INHERITANCE	145
8.5	SPECIAL METHODS	149
8.5.1	Create a New Data Type Using a Built-in Data Type	154
8.6	MAKING OUR CODE PRIVATE	154
8.7	ADDITIONAL RESOURCES	155
8.8	SELF-EVALUATION	156
CHAPTER 9 ■ Introduction to Biopython		157
<hr/>		
9.1	WHAT IS BIOPYTHON?	158
9.1.1	Project Organization	158
9.2	INSTALLING BIOPYTHON	159
9.3	BIOPYTHON COMPONENTS	162
9.3.1	Alphabet	162
9.3.2	Seq	163
9.3.3	MutableSeq	165
9.3.4	SeqRecord	166
9.3.5	Align	167
9.3.6	AlignIO	169
9.3.7	ClustalW	171
9.3.8	SeqIO	173
9.3.9	AlignIO	176
9.3.10	BLAST	177
9.3.11	Biological Related Data	187
9.3.12	Entrez	190
9.3.13	PDB	194
9.3.14	PROSITE	196
9.3.15	Restriction	197
9.3.16	SeqUtils	200
9.3.17	Sequencing	202
9.3.18	SwissProt	205
9.4	CONCLUSION	207
9.5	ADDITIONAL RESOURCES	207
9.6	SELF-EVALUATION	209

SECTION II Advanced Topics

CHAPTER 10 ■ Web Applications	213
10.1 INTRODUCTION TO PYTHON ON THE WEB	213
10.2 CGI IN PYTHON	214
10.2.1 Configuring a Web Server for CGI	215
10.2.2 Testing the Server with Our Script	215
10.2.3 Web Program to Calculate the Net Charge of a Protein (CGI version)	219
10.3 WSGI	221
10.3.1 Bottle: A Python Web Framework for WSGI	222
10.3.2 Installing Bottle	223
10.3.3 Minimal Bottle Application	223
10.3.4 Bottle Components	224
10.3.5 Web Program to Calculate the Net Charge of a Protein (Bottle Version)	229
10.3.6 Installing a WSGI Program in Apache	232
10.4 ALTERNATIVE OPTIONS FOR MAKING PYTHON-BASED DYNAMIC WEB SITES	232
10.5 SOME WORDS ABOUT SCRIPT SECURITY	232
10.6 WHERE TO HOST PYTHON PROGRAMS	234
10.7 ADDITIONAL RESOURCES	235
10.8 SELF-EVALUATION	236
CHAPTER 11 ■ XML	237
11.1 INTRODUCTION TO XML	237
11.2 STRUCTURE OF AN XML DOCUMENT	241
11.3 METHODS TO ACCESS DATA INSIDE AN XML DOCUMENT	246
11.3.1 SAX: cElementTree Iterparse	246
11.4 SUMMARY	251
11.5 ADDITIONAL RESOURCES	252
11.6 SELF-EVALUATION	252
CHAPTER 12 ■ Python and Databases	255
12.1 INTRODUCTION TO DATABASES	256
12.1.1 Database Management: RDBMS	257
12.1.2 Components of a Relational Database	258

12.1.3	Database Data Types	260
12.2	CONNECTING TO A DATABASE	261
12.3	CREATING A MYSQL DATABASE	262
12.3.1	Creating Tables	263
12.3.2	Loading a Table	264
12.4	PLANNING AHEAD	266
12.4.1	PythonU: Sample Database	266
12.5	SELECT: QUERYING A DATABASE	269
12.5.1	Building a Query	271
12.5.2	Updating a Database	273
12.5.3	Deleting a Record from a Database	273
12.6	ACCESSING A DATABASE FROM PYTHON	274
12.6.1	PyMySQL Module	274
12.6.2	Establishing the Connection	274
12.6.3	Executing the Query from Python	275
12.7	SQLITE	276
12.8	NOSQL DATABASES: MONGODB	278
12.8.1	Using MongoDB with PyMongo	278
12.9	ADDITIONAL RESOURCES	282
12.10	SELF-EVALUATION	284
CHAPTER 13 ■ Regular Expressions		285
13.1	INTRODUCTION TO REGULAR EXPRESSIONS (REGEX)	285
13.1.1	REGEX Syntax	286
13.2	THE RE MODULE	287
13.2.1	Compiling a Pattern	290
13.2.2	REGEX Examples	292
13.2.3	Pattern Replace	294
13.3	REGEX IN BIOINFORMATICS	294
13.3.1	Cleaning Up a Sequence	296
13.4	ADDITIONAL RESOURCES	297
13.5	SELF-EVALUATION	298
CHAPTER 14 ■ Graphics in Python		299
14.1	INTRODUCTION TO BOKEH	299
14.2	INSTALLING BOKEH	299
14.3	USING BOKEH	301

14.3.1	A Simple X-Y Plot	303
14.3.2	Two Data Series Plot	304
14.3.3	A Scatter Plot	306
14.3.4	A Heatmap	308
14.3.5	A Chord Diagram	309

SECTION III Python Recipes with Commented Source Code

CHAPTER 15 ■ Sequence Manipulation in Batch 315

15.1	PROBLEM DESCRIPTION	315
15.2	PROBLEM ONE: CREATE A FASTA FILE WITH RANDOM SEQUENCES	315
15.2.1	Commented Source Code	315
15.3	PROBLEM TWO: FILTER NOT EMPTY SEQUENCES FROM A FASTA FILE	316
15.3.1	Commented Source Code	317
15.4	PROBLEM THREE: MODIFY EVERY RECORD OF A FASTA FILE	319
15.4.1	Commented Source Code	320

CHAPTER 16 ■ Web Application for Filtering Vector Contamination 321

16.1	PROBLEM DESCRIPTION	321
16.1.1	Commented Source Code	322
16.2	ADDITIONAL RESOURCES	326

CHAPTER 17 ■ Searching for PCR Primers Using Primer3 329

17.1	PROBLEM DESCRIPTION	329
17.2	PRIMER DESIGN FLANKING A VARIABLE LENGTH REGION	330
17.2.1	Commented Source Code	331
17.3	PRIMER DESIGN FLANKING A VARIABLE LENGTH REGION, WITH BIOPYTHON	332
17.4	ADDITIONAL RESOURCES	333

CHAPTER 18 ■ Calculating Melting Temperature from a Set of Primers 335

18.1	PROBLEM DESCRIPTION	335
18.1.1	Commented Source Code	336
18.2	ADDITIONAL RESOURCES	336

CHAPTER 19 ■ Filtering Out Specific Fields from a GenBank File 339

19.1	EXTRACTING SELECTED PROTEIN SEQUENCES	339
------	---------------------------------------	-----

19.1.1	Commented Source Code	339
19.2	EXTRACTING THE UPSTREAM REGION OF SELECTED PROTEINS	340
19.2.1	Commented Source Code	340
19.3	ADDITIONAL RESOURCES	341
CHAPTER 20 ■ Inferring Splicing Sites		343
20.1	PROBLEM DESCRIPTION	343
20.1.1	Infer Splicing Sites with Commented Source Code	345
20.1.2	Sample Run of Estimate Intron Program	347
CHAPTER 21 ■ Web Server for Multiple Alignment		349
21.1	PROBLEM DESCRIPTION	349
21.1.1	Web Interface: Front-End. HTML Code	349
21.1.2	Web Interface: Server-Side Script. Commented Source Code	351
21.2	ADDITIONAL RESOURCES	353
CHAPTER 22 ■ Drawing Marker Positions Using Data Stored in a Database		355
22.1	PROBLEM DESCRIPTION	355
22.1.1	Preliminary Work on the Data	355
22.1.2	MongoDB Version with Commented Source Code	358
SECTION IV Appendices		
APPENDIX A ■ Collaborative Development: Version Control with GitHub		365
A.1	INTRODUCTION TO VERSION CONTROL	366
A.2	VERSION YOUR CODE	367
A.3	SHARE YOUR CODE	375
A.4	CONTRIBUTE TO OTHER PROJECTS	381
A.5	CONCLUSION	382
A.6	METHODS	384
A.7	ADDITIONAL RESOURCES	384
APPENDIX B ■ Install a Bottle App in PythonAnywhere		385
B.1	PYTHONANYWHERE	385
B.1.1	What Is PythonAnywhere	385
B.1.2	Installing a Web App in PythonAnywhere	385

APPENDIX C ■ Scientific Python Cheat Sheet	393
C.1 PURE PYTHON	394
C.2 VIRTUALENV	400
C.3 CONDA	402
C.4 IPYTHON	403
C.5 NUMPY	405
C.6 MATPLOTLIB	410
C.7 SCIPY	412
C.8 PANDAS	413
Index	417

List of Figures

2.1	Anaconda install in macOS.	21
2.2	Anaconda Python interactive terminal.	23
2.3	PyCharm Edu welcome screen.	35
3.1	Intersection.	60
3.2	Union.	61
3.3	Difference.	61
3.4	Symmetric difference.	62
3.5	Case 1.	65
3.6	Case 2.	66
5.1	Excel formatted spreadsheet called <code>sampladata.xlsx</code> .	93
8.1	IUPAC nucleic acid notation table.	147
9.1	Anatomy of a BLAST result.	181
10.1	Our first CGI.	216
10.2	CGI accessed from local disk instead from a web server.	217
10.3	<code>greeting.html</code> : A very simple form.	217
10.4	Output of CGI program that processes <code>greeting.html</code> .	218
10.5	Form <code>protcharge.html</code> ready to be submitted.	220
10.6	Net charge CGI result.	222
10.7	Hello World program made in Bottle, as seen in a browser.	224
10.8	Form for the web app to calculate the net charge of a protein.	229
11.1	Screenshot of XML viewer.	244
11.2	Codebeautify, a web based XML viewer.	245
12.1	Screenshot of PhpMyAdmin.	258
12.2	Creating a new database using phpMyAdmin.	262
12.3	Creating a new table using phpMyAdmin.	264

12.4	View of the Student table.	266
12.5	An intentionally faulty “Grades” table.	267
12.6	A better “Grades” table.	267
12.7	Courses table: A lookup table.	268
12.8	Modified “Grades” table.	268
12.9	Screenshot of SQLite manager.	277
12.10	View from a MongoDB cloud provider.	281
14.1	A circle with Bokeh.	302
14.2	Four circles with Bokeh.	303
14.3	A simple plot with Bokeh.	305
14.4	A two data series plot with Bokeh.	306
14.5	Scatter plot graphics.	308
14.6	A heatmap out of a microarray experiment.	310
14.7	A chord diagram.	312
16.1	HTML form for sequence filtering.	327
16.2	HTML form for sequence filtering.	328
21.1	Muscle Web interface.	350
22.1	Product of Listing 22.2, using the demo dataset (NODBDEMO).	356
A.1	The git add/commit process.	369
A.2	Working with a local repository.	370
A.3	Working with both a local and remote repository as a single user.	379
A.4	Contributing to open source projects.	383
B.1	“Consoles” tab.	386
B.2	The “Web” tab.	386
B.3	Upgrading domain type option.	387
B.4	Select a web framework screen, select Bottle.	388
B.5	Select a Python and Bottle version.	389
B.6	Form to enter the path of the web app.	390
B.7	The sample web app is ready to use.	390
B.8	The “File” tab.	391
B.9	Form to create a new directory in PythonAnywhere.	391
B.10	View and upload files into your account.	391

B.11	Front-end of the program to calculate charge of a protein using Bottle and hosted in PythonAnywhere.	392
------	--	-----



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

List of Tables

2.1	Arithmetic-Style Operators	26
3.1	Common List Operations	48
3.2	Methods Associated with Dictionaries	58
9.1	Sequence and Alignment Formats	175
9.2	Blast programs	178
9.3	eUtils	191
10.1	Frameworks for Web Development	233
12.1	Students in Python University	259
12.2	Table with primary key	260
12.3	MySQL Data Types	261
13.1	REGEX Special Sequences	287
A.1	Resources	367



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Preface to the First Edition

This book is a result of the experience accumulated during several years of working for an agricultural biotechnology company. As a genomic database curator, I gave support to staff scientists with a broad range of bioinformatics needs. Some of them just wanted to automate the same procedure they were already doing by hand, while others would come to me with biological problems to ask if there were bioinformatics solutions. Most cases had one thing in common: Programming knowledge was necessary for finding a solution to the problem. The main purpose of this book is to help those scientists who want to solve their biological problems by helping them to understand the basics of programming. To this end, I have attempted to avoid taking for granted any programming-related concepts. The chosen language for this task is Python.

Python is an easy-to-learn computer language that is gaining traction among scientists. This is likely because it is easy to use, yet powerful enough to accomplish most programming goals. With Python the reader can start doing real programming very quickly. Journals such as *Computing in Science and Engineering*, *Briefings in Bioinformatics*, and *PLOS Computational Biology* have published introductory articles about Python. Scientists are using Python for molecular visualization, genomic annotation, data manipulation, and countless other applications.

In the particular case of the life sciences, the development of Python has been very important; the best exponent is the Biopython package. For this reason, Section II is devoted to Biopython. Anyhow, I don't claim that Biopython is the solution to every biology problem in the world. Sometimes a simple custom-made solution may better fit the problem at hand. There are other packages like BioNEB and CoreBio that the reader may want to try.

The book begins from the very basic, with Section I ("Programming"), teaching the reader the principles of programming. From the very beginning, I place a special emphasis on practice, since I believe that programming is something that is best learned by doing. That is why there are code fragments spread over the book. The reader is expected to experiment with them, and attempt to internalize them. There are also some spare comparisons with other languages; they are included only when doing so enlightens the current topic. I believe that most language comparisons do more harm than good when teaching a new language. They introduce information that is incomprehensible and irrelevant for most readers.

In an attempt to keep the interest of the reader, most examples are somehow related to biology. In spite of that, these examples can be followed even if the reader doesn't have any specific knowledge in that field.

To reinforce the practical nature of this book, and also to use as reference

material, Section IV is called “Python Recipes with Commented Source Code.” These programs can be used as is, but are intended to be used as a basis for other projects. Readers may find that some examples are very simple; they do their job without too many bells and whistles. This is intentional. The main reason for this is to illustrate a particular aspect of the application without distracting the reader with unnecessary features, as well as to avoid discouraging the reader with complex programs. There will always be time to add features and customizations once the basics have been learned.

The title of Section III (“Advanced Topics”) may seem intimidating, but in this case, advanced doesn’t necessarily mean difficult. Eventually, everyone will use the chapters in this section [especially relational database management system—RDBMS— and XML]. An important part of the bioinformatics work is building and querying databases, which is why I consider knowing a RDBMS like MySQL to be a relevant part of the bioinformatics skill set. Integrating data from different sources is one of tasks most frequently performed in bioinformatics. The tool of choice for this task is XML. This standard is becoming a widely used platform for data interchange between applications. Python has several XML parsers and we explain most of them in this book.

Appendix B, “Selected Papers,” provides introductory level papers on Python. Although there is some overlapping of subjects, this was done to show several points of view of the same subject.

Researchers are not the only ones for whom this book will be beneficial. It has also been structured to be used as a university textbook. Students can use it for programming classes, especially in the new bioinformatics majors.

Preface to the Second Edition

The first edition of *Python for Bioinformatics* was written in 2008 and published in 2009. Even after eight years, the lessons in this book are still valuable. This is quite an accomplishment in a field that evolves at such a fast pace. In spite of its usefulness, the book is showing its age and would greatly benefit from a second edition.

The predominant Python version is 3.6, although Python 2.7 is still in use in production systems. Since there are incompatibilities between these versions, lot of effort was made to make all code in the book Python 3 compatible.

Not only has the software changed in these past eight years, but enterprise attitude and support toward Open Source Software in general and Python in particular has changed dramatically. There are also new computing paradigms that can't be ignored such as collaborative development and cloud computing.

In the original book, [Chapter 14](#) was called “Collaborative Development: Version Control” and was based on **Bazaar**, a software that follows the currently used distributed development workflow but is not what is being used by most developers today. By far the most software development is done with **Git** at **GitHub**. This chapter was rewritten to focus on current practices.

Web development is another area that changed significantly. Although this is not a book about web development, the chapter “Web Applications” now reflects current usage of long-running processes and frameworks instead of CGI/WSGI and middleware-based applications. Frameworks were discussed as a side note in this chapter, but now the chapter is based around a framework (**Bottle**) and leave the old method as a historical footnote.

In databases, the **NoSQL** gained lot of traction, from being a bullet point in the first edition, now has its own section using **MongoDB**, and a Python recipe was changed to use this NoSQL database.

Graphical libraries have improved since 2009, and there are great quality competing graphic libraries available for Python. There is a whole chapter devoted to **Bokeh**, a free interactive visualization library.

Another change that is reflected in this book is the usage of **Anaconda** and **Jupyter Notebooks** (with all code in a cloud notebook provided by Microsoft Azure¹).

¹See <https://notebooks.azure.com/py4bio/libraries/py3.us>

Regarding source code, there is a GitHub repository at <https://github.com/Serulab/Py4Bio> where you can download all the code and sample files used in this book.

There are corrections in every chapter. Sometimes there were actual mistakes, but most of the corrections were related to the Python 3 upgrade and in keeping with current good practices. Regarding corrections, I expect that this book may need corrections, so I made a web page where the readers can get updates. Please take a look at <http://py3.us> and subscribe to the low volume mailing list while at it.

Apart from software evolution and paradigms shifts, I also gained development experience and changed my views on pedagogical matters. During these years I worked in a genome sequencing project at an international consortium and as a senior software developer in an NYSE listed company (Globant). In the last 5 years I worked for several well-known clients such as Salesforce and National Geographic. I am currently working at PLOS (Public Library of Science).

By request of MATLAB, I include their contact information:

MATLAB ® is a registered trademark of The MathWorks, Inc. For product information please contact: The MathWorks, Inc. 3 Apple Hill Drive Natick, MA, 01760-2098 USA Tel: 508-647-7000 Fax: 508-647-7001 E-mail: info@mathworks.com Web: www.mathworks.com

Regarding the logo of Biopython, that is used in the cover, here it is usage license (this covers all Biopython files, including its logo):

Biopython is currently released under the "Biopython License Agreement" (given in full below). Unless stated otherwise in individual file headers, all Biopython's files are under the "Biopython License Agreement".

Some files are explicitly dual licensed under your choice of the "Biopython License Agreement" or the "BSD 3-Clause License" (both given in full below). This is with the intention of later offering all of Biopython under this dual licensing approach.

Biopython License Agreement

Permission to use, copy, modify, and distribute this software and its documentation with or without modifications and for any purpose and without fee is hereby granted, provided that any copyright notices appear in all copies and that both those copyright notices and this permission notice appear in supporting documentation, and that the names of the contributors or copyright holders not be used in advertising or publicity pertaining to distribution of the software without specific prior permission.

THE CONTRIBUTORS AND COPYRIGHT HOLDERS OF THIS SOFTWARE DISCLAIM ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL THE CONTRIBUTORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING

FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

BSD 3-Clause License

Copyright (c) 1999-2017, The Biopython Contributors All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission. THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Acknowledgments

A project such as this book couldn't be done by just one person. For this reason, there is a long list of people who deserve my thanks. In spite of the fact that the average reader doesn't care about the names, and at the risk of leaving someone out, I would like to acknowledge the following people: my wife Virginia Gonzalez (Vicky) and my son Maximo Bassi, who had to contend with my virtual absence during more than a year. Vicky also assisted me in uncountable ways during manuscript preparation. My parents and professors taught me important lessons. My family (Oscar, Graciela, and Ramiro) helped me with the English copyediting, along with Hugo and Lucas Bejar. Vicky, Griselda, and Eugenio also helped by providing a development abstraction layer, which is needed for writers and developers.

I would like to thank the people in the local Python community (<http://www.python.org.ar>): Facundo Batista, Lucio Torre, Gabriel Genellina, John Lenton, Alejandro J. Cura, Manuel Kaufmann, Gabriel Patiño, Alejandro Weil, Marcelo Fernandez, Ariel Rossanigo, Mariano Draghi, and Buanzo. I would choose Python again just for this great community. I also thank the people at Biopython: Jeffrey Chang, Brad Chapman, Peter Cock, Michiel de Hoon, and Iddo Friedberg. Peter Cock is specially thanked for his comments on the Biopython [chapter. I](#) also thank Shashi Kumar and Pablo Di Napoli who helped me with the L^AT_EX₂ ϵ issues, and Sunil Nair who believed in me from the first moment. Also people at Globant who trusted in me, like Guido Barosio, Josefina Chausovsky, Lucas Campos, Pablo Brenner and Guibert Englebienne. Globant co-workers such as Pedro Mourelle, Chris DeBlois, Rodrigo Obi-Wan Iloro, Carlos Del Rio and Alejandro Valle. People at PLOS, Jeffrey Gray and Nick Peterson.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

I

Programming



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Introduction

CONTENTS

1.1	Who Should Read This Book	3
1.1.1	What the Reader Should Already Know	4
1.2	Using this Book	4
1.2.1	Typographical Conventions	4
1.2.2	Python Versions	5
1.2.3	Code Style	5
1.2.4	Get the Most from This Book without Reading It All	6
1.2.5	Online Resources Related to This Book	7
1.3	Why Learn to Program?	7
1.4	Basic Programming Concepts	8
1.4.1	What Is a Program?	8
1.5	Why Python?	10
1.5.1	Main Features of Python	10
1.5.2	Comparing Python with Other Languages	11
	Readability	12
	Speed	13
1.5.3	How Is It Used?	14
1.5.4	Who Uses Python?	15
1.5.5	Flavors of Python	15
1.5.6	Special Python Distributions	16
1.6	Additional Resources	17

The most effective way to do it, is to do it.

Amelia Earhart

1.1 WHO SHOULD READ THIS BOOK

This book is for the life science researcher who wants to learn how to program. He/she may have previous exposure to computer programming, but this is not necessary to understand this book (although it surely helps).

This book is designed to be useful to several separate but related audiences, students, graduates, postdocs, and staff scientists, since all of them can benefit from knowing how to program.

Exposing students to programming at early stages in their career helps to boost their creativity and logical thinking, and both skills can be applied in research. In order to ease the learning process for students, all subjects are introduced with the minimal prerequisites. There are also questions at the end of each chapter. They can be used for self-assessing how much you've learned. The answers are available to teachers in a separate guide.

Graduates and staff scientists having actual programming needs should find its several real-world examples and abundant reference material extremely valuable.

1.1.1 What the Reader Should Already Know

Since this book is called *Python for Bioinformatics*, it has been written with the following assumptions in mind:

- No programming knowledge is assumed, but the reader is required to have minimum computer proficiency to be able to use a text editor and handle basic tasks in your operating system (OS). Since Python is multi-platform, most instructions from this book will apply to the most common operating systems (Windows, macOS and Linux); when there is a command or a procedure that applies only to a specific OS, it will be clearly noted.
- The reader should be working (or at least planning to work) with bioinformatics tools. Even low-scale handmade jobs, such as using the NCBI BLAST to ID a sequence, aligning proteins, primer searching, or estimating a phylogenetic tree will be useful to follow the examples. The more familiar the reader is with bioinformatics, the better he will be able to apply the concepts learned in this book.

1.2 USING THIS BOOK

1.2.1 Typographical Conventions

There are some typographical conventions I have tried to use in a uniform way throughout the book. They should aid readability and were chosen to tell apart user-made names (or variables) from language keywords. This comes in handy when learning a new computer language.

Bold: Objects provided by Python and by third-party modules. With this notation it should be clear that **round** is part of the language and not a user-defined name. Bold is also used to highlight parts of the text. **There is no way** to confuse one bold usage with the other.

Mono-spaced font: User declared variables, code, and filenames. For example: `sequence = 'MRVLLVALALLALAASATS'`.

Italics: In commands, it is used to denote a variable that can take different values. For example, in `len(iterable)`, “iterable” can take different values. Used in

text, it marks a new word or concept. For example “One such fundamental data structure is a *dictionary*.”

The content of lines starting with \$ (dollar sign) are meant to be typed in your operating system console (also called **command prompt** in Windows or **terminal** in macOS).

↵ : Break line. Some lines are longer than the available space in a printed page, so this symbol is inserted to mean that what is on the next line in the page represents the same line on the computer screen. Inside code, the symbol used is `<=`.

1.2.2 Python Versions

The current version of Python at this moment is 3.6.1. There is a 2.7.12 version that is maintained¹ because there are still a sizable number of applications in production using the 2.7 branch. Versions 3.x and 2.x are slightly different, at the point of being incompatible. Python 3 is more efficient than Python 2 in many aspects. Large websites such as Instagram migrated from Python 2.7 to Python 3.6 to save in CPU and memory consumption by up to 30%. This book uses Python 3.6.

The only scenario where you may need to use Python 2.7, apart from maintenance of old code, is when there is no availability of a specific library for Python 3. In this case, before starting a project in Python 2.7, try to search for a replacement library. For example, you want to connect with a MySQL database and you are told to use **MySQLdb**, since this package is not Python 3 compatible; instead of using Python 2.7, use **mysqlclient** or **mysql-connector-python**, both works with Python 3.

1.2.3 Code Style

Python source code that appears in this book is presented as **listings**. Each line of these listings is numbered. These numbers are not intended to be typed; they are used to reference each line in the text. You don't need to copy the code from the book, since it can be downloaded from the GitHub repository at <https://github.com/Serulab/Py4Bio>.

Code can be formatted in several ways and still be valid to the Python interpreter. This following code is syntactically correct:

```
def GetAverage(X):
    avG=sum(X)/len(X)
    " Calculate the average "
    return avG
```

Also this one:

¹Python 2.7.x has an end-of-life date in 2020. There will be no Python 2.8. For more information see <https://www.python.org/dev/peps/pep-0373/>.

```
def get_average(items):
    """ Calculate the average
    """
    average = sum(items) / len(items)
    return average
```

The former code sample follows most accepted coding styles for Python.² Throughout the book you will find mostly code formatted as the second sample. Some code in the book will not follow accepted coding styles for the following reasons:

- There are some instances where the most didactic way to show a particular piece of code conflicts with the style guide. On those few occasions, I choose to deviate from the style guide in favor of clarity.
- Due to size limitation in a printed book, some names were shortened and other minor drifts from the coding styles have been introduced.
- To show that there is more than one way to write the same code. Coding style is a guideline, and enforcement is not made at a language level, so some programmers don't follow it thoroughly. You should be able to read "bad" code, since sooner or later you will have to read other people's code.

1.2.4 Get the Most from This Book without Reading It All

- If you want to **learn how to program**, read the first section, from [Chapter 1](#) to [Chapter 8](#). The Regular Expressions (REGEX) chapter ([Chapter 13](#)) can be skipped if you don't need to deal with REGEX.
- If you know Python and just want to **know about Biopython**, read first [Chapter 9](#) (from page 158 to page 209). It is about Biopython modules and functions. Then try to follow programs found in Section III (from page 315 to page 363).
- There are three appendixes that can be read in an independent way. Appendix A (Collaborative Development: Version Control with GitHub) reproduces a paper called "A Quick Introduction to Version Control with Git and GitHub." Appendix B shows how to install a web application using Python Anywhere. Appendix C is a reference material that can be used as a cheat sheet when you need a quick answer without having to read a chapter.

²The official Python style guide is located at <https://www.python.org/dev/peps/pep-0008>, and a more easy-to-read style guide is located at <http://docs.python-guide.org/en/latest/writing/style>.

1.2.5 Online Resources Related to This Book

The book website is at <http://py3.us>. In this site you will find errata, a mailing list to keep updated about Python and links to source code repositories. Regarding source code, the official source code repository of this book is at GitHub (<https://github.com/Serulab/Py4Bio>). From this site you can inspect online or download all the code used in this book. To download all scripts, go to the “Clone or download” green button and press it. If you have Git installed in your machine (and know how to use it³), clone the repository using this address: `git@github.com:Serulab/Py4Bio.git`. Another alternative is to click on “Download ZIP”. Once you have the repository in your machine, go to the `code` folder, where there are a set of folders, each one has the scripts related to the chapter. Each script in the book has a name and this corresponds with the filename. There is another folder called `notebooks`, and it contains Jupyter notebooks that can be run locally. For more information on how to run a Jupyter notebook, please see <http://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/execute.html>.

Another online resource are the Jupyter Notebooks available at Microsoft Azure Notebook website (<https://notebooks.azure.com/py4bio/libraries/py3.us>). The same notebooks that are in the book repository, can be used online in this site.

1.3 WHY LEARN TO PROGRAM?

Many of the tasks that a researcher performs with his or her computer are repetitive: Collect data from a Web page, convert files from one format to another, execute or interpret hundreds of BLAST results, primer design, look for restriction enzymes, etc. In many cases it is evident that these are tasks that can be performed with a computer, with less effort on our part and without the possibility of errors caused by tiredness or distractions.

An important consideration when you’re evaluating whether or not to create a program is the apparent time lost in the definition and formulation of the problem, implementing it with code, and then debugging it (correcting errors). It is incorrect to consider problem definition and evaluation as a waste of time. It is generally at this precise point in the process where we understand thoroughly the problem that we face. It is common that during the attempt to formulate a problem, we realize that many of our initial assumptions were mistaken. It also helps us to detect when it is necessary to restart the planning process. When this happens, it is better that it happens at the planning stage than when we are in the middle of the project. In these cases, the planning of the program represents time saved. Another advantage to take into account is that the time that is invested to create a program once is compensated by the speed with which the tasks are performed every time we run it.

³In Appendix A there is a tutorial on how to use GitHub

Not only can it automate the procedures that we do manually, but it will also be able to do things that would otherwise not be possible.

Sometimes it is not very clear if a particular task can be done by a program. Reading a book such as this one (including the examples) will help you identify which tasks are feasible to automate with software and which ones are better done manually.

1.4 BASIC PROGRAMMING CONCEPTS

Before installing Python, let's review some programming fundamentals. If you have some previous programming experience, you may want to skip this section and jump straight to [Chapter 2](#) "Installing Python." This section introduces basic concepts such as *instructions*, *data types*, *variables*, and some other related terminology that is used throughout this book.

1.4.1 What Is a Program?

Computers only know what you tell them. The way to tell them to do something is by a program. A *program* is a set of ordered instructions designed to command the computer to do something. The word "ordered" is there because is not enough to declare what to do, but the actual order of directions should also be stated.⁴

A program is often characterized as a recipe. A typical recipe consists of a list of ingredients followed by step-by-step instructions on how to prepare a dish. This analogy is reflected in several programming websites and tutorials with the words "recipe" and "cookbook." A laboratory protocol is another useful analogy. A protocol is defined as a "predefined written procedural method in the design and implementation of experiments."

Here is a typical protocol, followed almost every day in several molecular laboratories:

Listing 1.1: Protocol for Lambda DNA digestion

Restriction Digestion of Lambda DNA

Materials

5.0 mcL	Lambda DNA (0.1 g/L)
2.5 mcL	10x buffer
16.5 mcL	H ₂ O
1.0 mcL	EcoRI

⁴There are *declarative* languages that state what the program should accomplish, rather than describing how to accomplish it. Most computer languages (Python included) are *imperative* instead of *declarative*.

Procedure

Incubate the reagents at 37°C for 1 hr.

Add 2.5 mcL loading dye and incubate for another 15 minutes.

Load 20 mcL of the digestion mixture onto a minigel

There are at least two components of a protocol: materials or ingredients, and procedures. A procedure provides specific order like **incubate**, add, mix, **load** and many others. The same goes for a computer program. The programmer gives specific order to the computer: **print**, **read**, **write**, **add**, **multiply**, **round**, and others.

While protocol procedures correlate with program instructions, materials are the *data*. In protocols, procedures are applied to materials: Mix 2.5 μL of buffer with 5 μL of Lambda DNA and 16.5 μL of H_2O , load 20 μL onto a minigel. In a program, instructions are applied to data: print the text string “Hello”, add two integer numbers, round a float number.

As a protocol can be written in different languages (like English, Spanish, or French), there are different languages to program a computer. In science, English is the *de facto* language. Due to historical, commercial and practical reasons, there is no such equivalent in computer science. There are several languages, each with its own strong points and weakness. For reasons that will make sense shortly, Python was the computer language chosen for this book.

Let’s see a simple Python program:

Listing 1.2: sample.py: Sample Python Program

```
1 seq_1 = 'Hello,'
2 seq_2 = ' you!'
3 total = seq_1 + seq_2
4 seq_size = len(total)
5 print(seq_size)
```

Note: The numbers at the beginning of the each line are for reference only, they are not meant to be typed.

This small program can be read as “Name the string **Hello**, as **seq_1**. Name the string **you!** as **seq_2**. Add the strings named **seq_1** and **seq_2** and call the result as **total**. Get the length of the string called **total** and name this value as **seq_size**. Print the value of **seq_size**.” This program prints the number 11.

As shown, there are different types of data (often called “data types” or just “types”). Numbers (integers or float), text string, and other data types are covered in [Chapter 3](#). In `print(seq_size)`, the instruction is **print** and **seq_size** is the name of the data. Data is often represented as *variables*. A *variable* is a name that stands for a value that may vary during program execution. With variables, a programmer can represent a generic command like “round n” instead of “round 2.9.” This way he can take into account a non-fixed (hence *variable*) value. When

the program is executed, “n” should take a specific value since there is no way to “round n.” This can be done by assigning a value to a variable or by binding a name to a value.⁵ The difference between “assign a value to a variable” and “bind a name to a value” is explained in detail in [Chapter 3](#) (from page 64). In any case, it is expressed as:

```
variable_name = value
```

Note that **this is not an equality** as seen in mathematics. In an equality, terms can be interchanged, but in programming, the term on the right (**value**) takes the name of the term on the left (**variable_name**). For example,

```
seq_1 = 'Hello'
```

After this assignment, the variable `seq_1` can be used, like,

```
print(seq_1)
```

This is translated as “print out the value called `seq_1`”. This command returns “Hello” because this is the value of this variable.

1.5 WHY PYTHON?

Let’s have a look at some Python features worth pointing out.

1.5.1 Main Features of Python

- **Readability:** When we talk about readability, we refer as much to the original programmer as any other person interested in understanding the code. It is not an uncommon occurrence for someone to write some code then return to it a month later and find it difficult to understand. Sometimes Python is called a “human-readable language.”
- **Built-in features:** Python comes with “batteries included.” It has a rich and versatile standard library that is immediately available, without the user having to download separate packages. With Python you can, with few lines, read and write XML and JSON files, parse and generate email messages, extract files from a zip archive, open a URL as if were a file, and many other possibilities that in other languages, it would require a third-party library.
- **Availability of third-party modules for a broad spectrum of activities.** Data visualization⁶ and plotting, PDF generation, bioinformatics analysis,⁷ image

⁵In Python the latter form is used.

⁶Matplotlib (<http://matplotlib.org/>) and Bokeh (<http://bokeh.pydata.org/en/latest/>) are the most used.

⁷Biopython library to make your own bioinformatics applications (<http://biopython.org/>).

processing,⁸ machine learning,⁹ game development, interface with popular databases,¹⁰ and application software are only a handful of examples of modules that can be installed to extend Python functionality.

- High-level built-in data structures: Dictionaries, sets, lists, tuples, and others. These are very useful to model real-world data. Third-party modules such as NumPy and SciPy can also extend the structures to kd-trees, n-dimensional arrays, matrix operations, time-series, image objects, and more.
- Multiparadigm: Python can be used as a “classic” procedural language or as “modern” object-oriented programming (OOP) language. Most programmers start writing code in a procedural way and when they need to, they upgrade to OOP. Python doesn’t force programmers to write OOP code when they just want to write a simple script.
- Extensibility: If the built-in methods and available third-party modules are not enough for your needs, you can easily extend Python, even in other programming languages. There are some applications written mostly in Python but with a processor demanding routine in C or FORTRAN. Python can also be extended by connecting it to specialized high-level languages like R or MATLAB¹¹.
- Open source: Python has a liberal open source license that makes it freely usable and distributable, even for commercial use.
- Cross platform: A program made in Python can be run under any computer that has a Python interpreter. This way, a program made under Windows 10 can run unmodified in Linux or OSX. Python interpreters are available for most computer and operating systems, and even some devices with embedded computers like the Raspberry Pi.
- Thriving community: Python is nowadays the programming language to use for scientists and researchers.¹² This translates into more libraries for your projects and people you can go to for support.

1.5.2 Comparing Python with Other Languages

You may be wondering why you should use Python, and not more well-known languages like Java, PHP, or C++. It is a good question. A programming language

⁸Scikit-image paper: <http://peerj.com/articles/453>

⁹scikit-learn website: <http://scikit-learn.org/stable/>

¹⁰<https://wiki.python.org/moin/DatabaseProgramming>

¹¹MATLAB® is a registered trademark of The MathWorks, Inc. For product information please contact: The MathWorks, Inc. 3 Apple Hill Drive Natick, MA, 01760-2098 USA. Tel: 508-647-7000. Fax: 508-647-7001. E-mail: info@mathworks.com. Web: www.mathworks.com.

¹²<http://www.nature.com/news/programming-pick-up-python-1.16833>

can be regarded as a tool, and choosing the best tool for the job makes a lot of sense.

Readability

Nonprofessional programmers tend to value the learning curve as much as the legibility of the code (both aspects are tightly related).

A simple “hello world” program in Python looks like this:

```
print("Hello world!")
```

Compare it with the equivalent code in Java:

```
public class Hello
{
    public static void main(String[] args) {
        System.out.printf("Hello world!");
    }
}
```

Let’s see a code sample in C language. The following program reads a file (`input.txt`) and copies its contents into another file (`output.txt`):

```
#include <stdio.h>
int main(int argc, char **argv) {
    FILE *in, *out;
    int c;
    in = fopen("input.txt", "r");
    out = fopen("output.txt", "w");
    while ((c = fgetc(in)) != EOF) {
        fputc(c, out);
    }
    fclose(out);
    fclose(in);
}
```

The same program in Python is shorter and easier to read:

```
with open("input.txt") as input_file:
    with open("output.txt") as output_file:
        output_file.writelines(in)
```

Let’s see a Perl program that calculates the average of a series of numbers:

```

sub avg(@_) {
    $sum += $_ foreach @_;
    return $sum / @_ unless @_ == 0;
    return 0;
}
print avg((1..5))."\n";

```

The equivalent program in Python is:

```

def avg(data):
    if len(data)==0:
        return 0
    else:
        return sum(data)/len(data)
print(avg([1,2,3,4,5]))

```

The purpose of this Python program could be almost fully understood by just knowing English.

Python is designed to be a highly readable language.¹³ The use of English keywords, and the use of spaces to limit code blocks and its internal logic (indentation), contribute to this end. It's possible to write hard-to-read code in Python, but it requires a deliberate effort to obfuscate the code.¹⁴

Speed

Another criterion to consider when choosing a programming language is execution speed. In the early days of computer programming, computers were so slow that some differences due to language implementation were very significant. It could take a week for a program to be executed in an interpreted language, while the same code in a compiled language could be executed in a day. This performance difference between interpreted and compiled languages still has the same proportion, but it is less relevant. This is because a program that took a week to run, now takes less than ten seconds, while the compiled one takes about one second. Although the difference seems important (at least one order of magnitude), it is not so relevant if we consider the time it takes to develop it.

This does not mean that execution speed does not need to be considered. A 10X speed difference can be crucial in some high-performance computing operations. Sometimes a lot of improvements can be achieved by writing optimized code. If the code is written with speed optimization in mind, it is possible to obtain results quite

¹³Other languages are regarded as “write only,” since once written it is very difficult to understand it.

¹⁴A simple `print 'Hello World'` program could be written, if you are so inclined, as `print "".join([chr((L>=65 and L<=122) and (((L>=97) and (L-96) or (L-64))-1)+13)%26+((L>=97) and 97 or 65)) or L) for L in [ord(C) for C in 'Uryyb Jbeyq!']])` (py3.us/1).

similar to the ones that could be obtained in a compiled language. In the cases where the programmer is not satisfied with the speed obtained by Python, it is possible to link to an external library written in another language (like C or Fortran). This way, we can get the best of both worlds: the ease of Python programming with the speed of a compiled language.

1.5.3 How Is It Used?

Python has a wide range of applications. From cell phones to web servers, there are thousands of Python applications in the most diverse fields. There is Python code powering Wikipedia robots, helping design next generation special effects at Industrial Light & Magic,¹⁵ embedded in D-link modems and routers,¹⁶ and it is the scripting language of the OpenOffice suite¹⁷.

Some languages are strong in one niche (like PHP for web applications, Java for desktop programs), but Python can't be typecast easily.

With a single code base, Python desktop applications run with a native look and feel on multiple platforms. Well-known examples of this category include the **BitTorrent** p2p client/server, **Calibre**, an Ebook manager, **Sage Math** (a mathematics software system), the **Dropbox** client, and more.

As a language for building web applications, Python can be found in high traffic sites like Reddit, NationalGeographic, Instagram, and NASA. There are specialized software for building web sites (called webframeworks) in Python like **Django**, **Web2Py**, **Pyramid**, **Flask**, and **Bottle**.

From system administration to data analysis, Python provides a broad range of tools to this end:

- Generic Operating System Services (os, io, time, curses)
- File and Directory Access (os.path, glob, tempfile, shutil)
- Data Compression and Archiving (zipfile, gzip, bz2)
- Interprocess Communication and Networking (subprocess, socket, ssl)
- Internet (email, mimetools, rfc822, cgi, urllib)
- String Services (string, re, codecs, unicodedata)

Python is gaining momentum as the default computer language for the scientific community. There are several libraries oriented toward scientific users, such as **SciPy**¹⁸ and **Anaconda**.¹⁹ Both distributions integrate modules for linear algebra,

¹⁵<https://www.python.org/about/success/ilm/>

¹⁶<https://www.python.org/about/success/dlink/>

¹⁷<http://wiki.services.openoffice.org/wiki/Python>

¹⁸<https://www.scipy.org>

¹⁹<https://www.continuum.io/anaconda-overview>

signal processing, optimization, statistics, genetic algorithms, interpolation, ODE solvers, special functions, etc.

Python has support for parallel programming with pyMPI and 2D/3D scientific data plotting.

Python is known to be used in wide and diverse fields like engineering, electronics, astronomy, biology, paleomagnetism, geography, and many more.

1.5.4 Who Uses Python?

Python is used by several companies, from small and unknown shops up to big players in their fields like Google, National Geographic, Disney, NASA, NYSE, and many more.

It is one of the four “official languages” of Google among Java, C++ and Go. They have web sites made in Python, stand-alone programs and even hosting solutions.²⁰ As a confirmation that Google is taking Python seriously, in December 2005 they hired Guido van Rossum, the creator of Python. It may not be Google’s main language, but this shows that they are a strong supporter of it.

Even Microsoft, a company not known for their support of open source programs, developed a version of Python to run their “.Net” platform (**IronPython**) and also developed a the **Python Tools for Visual Studio**,²¹ a Free, open source plugin that turns Visual Studio into a Python IDE.

Many well-known Linux distributions already use Python in their key tools. Ubuntu Linux “prefers the community to contribute work in Python.” Python is so tightly integrated into Linux that some distributions won’t run without a working copy of Python.

1.5.5 Flavors of Python

Although in this book I refer to Python as a programming language, Python is actually a language definition. What we use most of the time is a specific implementation, CPython, that is the Python language definition implemented in C. Since this implementation is the most used, we just call Python to the CPython implementation.

The most relevant Python implementations are: CPython, PyPy,²² Stackless,²³ Jython²⁴ and IronPython.²⁵ This book will focus on the standard Python version (CPython), but it is worth knowing about the different versions.

- CPython: The most used Python version, so the terms CPython and Python are used interchangeably. It is made mostly in C (with some modules made

²⁰<https://cloud.google.com/appengine/>

²¹<https://www.visualstudio.com/vs/python/>

²²<http://codespeak.net/pypy/dist/pypy/doc/home.html>

²³<http://www.stackless.com>

²⁴<http://www.jython.org/Project>

²⁵<http://ironpython.net>

in Python) and is the version that is available from the official Python Web site (<http://www.python.org>).

- **PyPy:** A Python version made in Python. It was conceived to allow programmers to experiment with the language in a flexible way (to change Python code without knowing C). It is mostly an experimental platform.
- **Stackless:** Another experimental Python implementation. The aim of this implementation doesn't focus on flexibility like PyPy; instead, it provides advanced features not available in the "standard" Python version. This is done in order to overcome some design decisions taken early in Python development history. Stackless allows custom-designed Python application to scale better than CPython counterparts. This implementation is being used in the EVE Online massively multi-player online game, *Civilization IV*, *Second Life*, and *Twisted*.
- **Jython:** A Python version written in Java. It works in a JVM (Java Virtual Machine). One application of Jython is to add the Jython libraries to their Java system to allow users to add functionality to the application. A very well known learning 3D programming environment (Alice²⁶) uses Jython to let the users program their own scripts.
- **IronPython:** Python version adapted by Microsoft to run on ".Net" and ".Mono" platform. .Net is a technology that competes with Java regarding "write once, runs everywhere."

1.5.6 Special Python Distributions

Apart from Python implementations, there are some special adaptations of the original CPython that are packaged for specific purposes. They are called Python bundles or distributions. Most of them brings to the table 3rd party software such as editors, visualization modules and the Jupyter Notebook. This is a web application that allows you to create and share documents that contain live code, equations, visualizations and explanatory text. Here is a list of most useful distributions²⁷:

- **ActivePython:**²⁸ Aimed at enterprise users, ActiveState provides a precompiled, supported, quality-assured Python distribution that makes it easy for corporations to comply with policy requirements to have supported open source products. From a technical standpoint it offers all modern Python versions with most used external modules already pre-installed. It also has its own package management and external modules repository (PyPM²⁹)

²⁶ Alice is available for free at <http://www.alice.org>.

²⁷ For a complete list of Python implementations and distributions see <https://www.python.org/download/alternatives>

²⁸ <http://www.activestate.com/activepython>

²⁹ <https://code.activestate.com/pypm/>

- **Enthought Canopy:**³⁰ Another all-in-one Python solution. Includes over 450 core scientific analytic and Python packages, like NumPy, SciPy, IPython, 2D and 3D visualization, database adapters, and others. Also includes a Code Editor with Jupyter Notebook Support. It has some add ons such a graphical package manager that notifies you of updates, installs with one click and helps you roll back package versions. Everything is available as a single-click installer for the three major operating systems. This bundle is suitable for scientific users, and it is made by the same people who made NumPy and SciPy. There are different licenses like a free academic one, and various paid commercial enterprise licenses.
- **WinPython:**³¹ It defines itself as a free open-source portable distribution of the Python programming language for Windows 7/8/10 and scientific and educational usage. Also includes packages suitable for scientists, data scientists, and education (NumPy, SciPy, Sympy, Matplotlib, Pandas, pyqtgraph, etc.). Uses Spyder (Scientific PYthon Development EnviRonment) as the default editor and it is portable in the sense the user can move the WinPython directory and all settings are kept. You can have multiples copies of isolated and self-consistent WinPython installations.
- **Anaconda:**³² A Python and R distribution for scientific computing. Includes over 720 packages for data preparation, data analysis, data visualization, machine learning, and interactive data science. It shares the objective and user type with Enthought Canopy. Also comes with Spyder as the default code editor. It has several products that differentiates it from other Python distribution, like Repository, Accelerate, Scale, Mosaic, Notebooks and Fusion. Most of these services are available only to the expensive subscriptions. If you don't use any of these services you still get an excellent all-in-one Python distribution. Continuum, the company behind Anaconda is a institutional partner of Project Jupyter, which means that they support the development of Jupyter Notebook, a web application to run Python code in a browser.

You may be wondering which one to use (or just use the standard “plain vanilla” Python). There is no single and correct answer to this question, since it will depend on your needs, work habits, budget, and personal preferences. Personally I tend to use the standard Python in servers and Anaconda in the computers I use for software development.

1.6 ADDITIONAL RESOURCES

- Interactive notebooks: Sharing the code. Interactive notebooks: Sharing the code. The free IPython notebook makes data analysis easier to record, under-

³⁰<https://www.enthought.com/products/canopy/>

³¹<http://winpython.github.io/>

³²<https://www.continuum.io/anaconda-overview>

stand and reproduce. Helen Shen. Nature 515, 151–152 (06 November 2014)
doi:10.1038/515151a
<https://goo.gl/HfBJ12>

- Python for feature film:
<http://dgovil.com/blog/2016/11/30/python-for-feature-film/>
- Alternative Python implementations:
<https://www.python.org/download/alternatives/>
- IPython: an interactive computing environment.
<http://ipython.org/>
- bpython: A fancy interface to the Python interpreter for Unix-like operating systems:
<https://www.bpython-interpreter.org>
- Python history, a blog by Guido van Rossum:
<http://python-history.blogspot.com>

First Steps with Python

CONTENTS

2.1	Installing Python	20
2.1.1	Learn Python by Using It	20
2.1.2	Install Python Locally	20
	Installing Anaconda	20
2.1.3	Using Python Online	21
2.1.4	Testing Python	22
2.1.5	First Use	22
2.2	Interactive Mode	23
2.2.1	Baby Steps	23
2.2.2	Basic Input and Output	23
	Output: Print	23
	Input: input	24
2.2.3	More on the Interactive Mode	24
2.2.4	Mathematical Operations	26
	Division	27
2.2.5	Exit from the Python Shell	27
2.3	Batch Mode	27
2.3.1	Comments	29
2.3.2	Indentation	30
2.4	Choosing an Editor	32
2.4.1	Sublime Text	32
2.4.2	Atom	33
2.4.3	PyCharm	34
2.4.4	Spyder IDE	35
2.4.5	Final Words about Editors	36
2.5	Other Tools	36
2.6	Additional Resources	37
2.7	Self-Evaluation	37

The journey of a thousand miles begins with one step.

Lao Tzu

2.1 INSTALLING PYTHON

2.1.1 Learn Python by Using It

This section shows how to install Python to start running your own programs. Learning by doing is the most efficient way of learning. It is better than just passively reading a book (even this book). You will find “Python interactive mode” very rewarding in this sense, since it can answer your questions faster than a book or a search engine. As a bonus, the answers you get from the Python interactive mode are definitive.

For these reasons I suggest installing Python before continuing to read this book.

2.1.2 Install Python Locally

Python is pre-installed in macOS and most Linux distributions. In Windows you have to download the **Windows x86-64 web-based installer** from the Python download page (<https://www.python.org/downloads/windows/>) and then install it. Installation is pretty straightforward if you are used to installing Windows programs. You should double-click the installer file and run the Python Install Wizard. Accept the default settings and you will have Python installed in a few minutes without hassle.

Remember that as an alternative to the official installer, you can download and install one of the Python distributions mentioned on page 16. My personal preference at this moment is the Anaconda distribution, but you don’t need to install Anaconda to follow this book, any Python distribution will do it.

Installing Anaconda

The following instruction is common for macOS and Windows (see Linux installation in the next paragraph). Download the graphical installer¹ appropriate for your OS and double click on it.² It will show an installer with the title “Welcome to the Anaconda 3 installer.” Press **Continue** in the **Introduction**, **Read Me** and **License**. In **Destination Select** press **Continue** if you want to install it only for the current user. The next step is pressing **Install**. The installation will take some minutes and you will see a screen like the one in [figure 2.1](#). In macOS, when you close the installer you are offered to move the installer to the trash, doing this will delete only the installer.

To install Anaconda in Linux, download the Linux version from <https://www.continuum.io/downloads#linux>, you will get a file with a name similar to `Anaconda3-4.3.1-Linux-x86_64.sh`. In a terminal, run

```
$ bash Anaconda3-4.3.1-Linux-x86_64.sh
```

¹<https://www.continuum.io/downloads>

²If using Mac, the macOS version must be 10.12.3 (Sierra) or later.

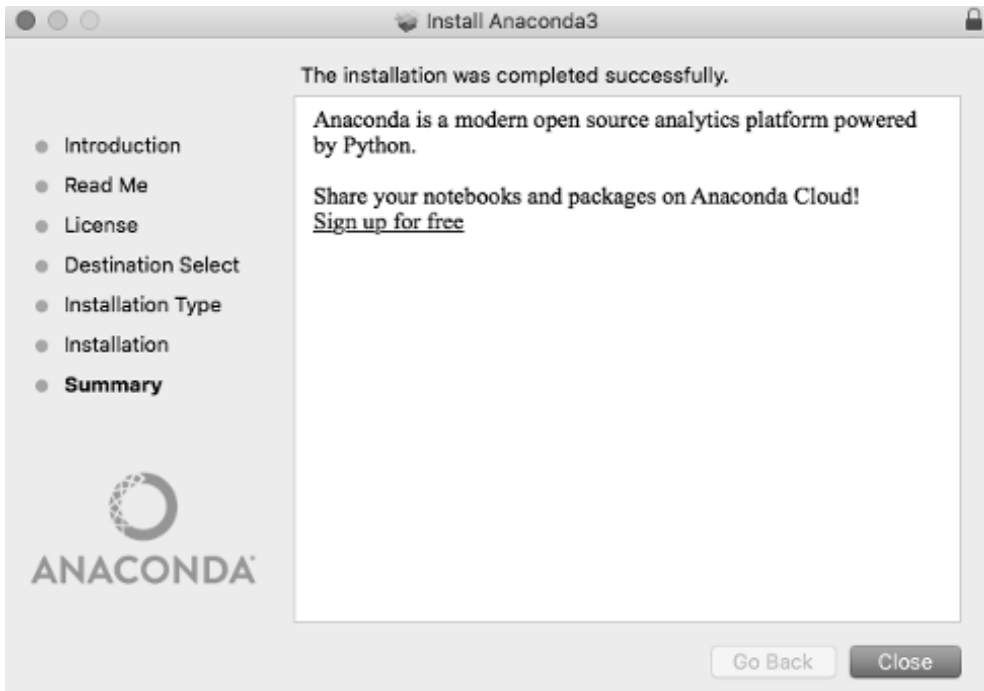


Figure 2.1 Anaconda install in macOS.

and follow the instructions. If you answer “yes” to the last question, your default Python will be the Anaconda Python. Note that this change will take into effect when you open a new terminal.

2.1.3 Using Python Online

Another way to try Python for learning or even for running programs is by using an online service. PythonAnywhere (<https://www.pythonanywhere.com>) is a service that allows you to run different versions of Python. You will need a browser and an Internet connection. PythonAnywhere provides a free service for running Python scripts online. They have several plans from \$5/month with unlimited Python or Bash consoles, a subdomain to host a web application, 1 Gb of storage, databases and more.³ Their free “Beginner” service is good enough for exploration and learning.

Another service that is worth trying is Microsoft Azure Notebooks. It is in preview state at this moment (June 2017) but I’ve been using it for some months and it seems stable enough to recommend it⁴. This is not a service where you

³For more information on PythonAnywhere plans, please see <https://www.pythonanywhere.com/pricing>.

⁴You can try the code in the book in this platform at <https://notebooks.azure.com/py4bio/libraries/py3.us>. The link is also in book web page (<http://py3.us>)

access to a Python console to type away any Python command, but it uses “Jupyter Notebook”, which is a web application that looks like a webpage with live code. This won’t replace a developer environment but can be used for learning and for presentations. Another web service where you can run simple Python programs (like those featured in the first five chapters of this book) is Rep.it (<https://repl.it/languages/python3>).

2.1.4 Testing Python

Once Python is installed, you should make sure it works. On Windows, double-click on the Python icon. Linux and macOS⁵ users could open a terminal and then type `python`.

You should see a screen like this one:⁶

```
Python 3.6.0 |Anaconda 4.3.1 (64-bit)| (Dec 23 2016, 12:22:00)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

This is the *Python console*⁷ and it is used for programming in *interactive mode*. This mode will be explained in the next section.

2.1.5 First Use

There are two ways to use Python: interactive and batch mode. These methods are complementary and they are used with different purposes. Interactive mode allows the programmer to get an immediate answer to each instruction. In batch mode, instructions are stored in one or more files and then executed. Interactive mode is used mostly for small tests while most programs are run in batch mode.

Interactive mode can be invoked by executing `python` or within some Python editors like Spyder, PyCharm, IDLE and others. It also can be used online at sites like <https://repl.it/languages/python3> or <https://www.pythonanywhere.com/>. I recommend installing Python in you own machine rather than using it online.⁸

If using Anaconda, run the Anaconda Navigator⁹ and choose QtConsole; in this case, Python interactive mode will look like this in Figure 2.2.

If you are not using Anaconda, from your terminal or command prompt, type `python`.

⁵In macOS the terminal is located under the Applications/Utilities folder.

⁶This output could vary from system to system depending on Python version, base operating system, and options set during compilation.

⁷The technical name is REPL, from Read-Eval-Print Loop, but most people call it *Python interpreter*, *Python shell* or *Python terminal*

⁸If you are in a machine without Admin or root rights and can’t install Python, the online option is a good alternative.

⁹Look for the Anaconda Navigator icon or run in from the command line with `anaconda-navigator`.

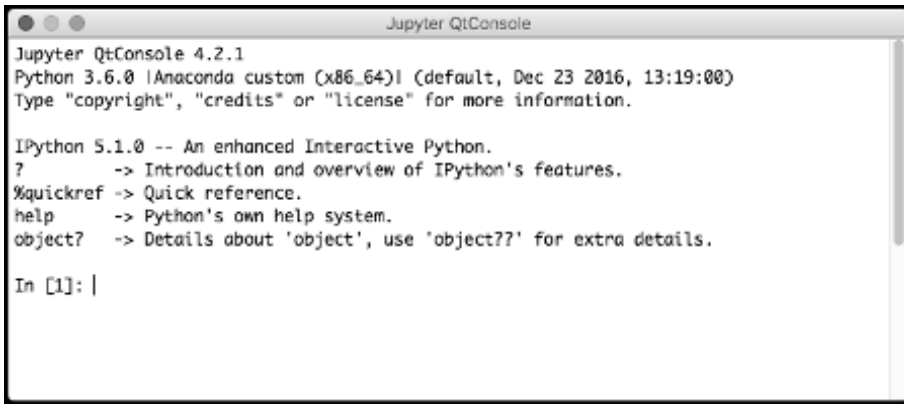


Figure 2.2 Anaconda Python interactive terminal.

Let's learn some Python basics using the interactive mode.

2.2 INTERACTIVE MODE

2.2.1 Baby Steps

The following code shows how to command the interpreter to print the string “Hello world!”¹⁰

```
>>> print('Hello World!')
Hello World!
```

Note the three greater-than characters (>>>); this is the Python prompt of the interactive mode. It is already there, you don't need to type it. This means that Python is ready to execute commands or evaluate expressions.

2.2.2 Basic Input and Output

Output: Print

From Python 3, **print** is a function. A function is a reusable code that can perform a specific task. Each function may receive one or more values called parameters. In the case of **print("Hello World!")**, the name of the function is **print** and the parameter is the string **"Hello World!"**. We will see functions with some detail in [Chapter 6](#).

The print function can receive several elements:

```
>>> print('Hello', 'World!')
```

¹⁰There is a tradition among programmers to show how a language works by printing the string “Hello world”. Python programmers are not immune to this custom. See what happens when you type this statement in the interactive mode: **import __hello__**.

Hello World!

By default it prints all string separated with a whitespace, but you can change the the separator with a parameter named **sep**:

```
>>> print('Hello', 'World!', sep=';')
Hello;World!
```

Redirect the output to a file:

```
>>> print("Hello","World!", sep="," , file=filehandle)
```

We will see how to handle files in [Chapter 6](#).

To change the end on the output, use parameter *end*. Changing the end of the output in this case adds two carrier returns (or *enter*):

```
>>> print("Hello", "World!", sep=";", end='\n\n')
Hello;World!
```

Input: input

To input data in a running program you can use **input**. The following command takes a string of data from the user and returns it to a variable called *name*. In the following code, after typing the string, the variable is entered and the content of the variable is displayed:

```
>>> name = input("Enter your name: ")
Enter your name: Seba
>>> name
'Seba'
```

Most of the time you will not use the **input** function since there are more practical ways to enter data, like reading it from a file, from a web page or from the output of another program.

2.2.3 More on the Interactive Mode

Interactive mode can be used as a calculator:

```
>>> 1+1
2
```

When '+' is used on strings, it returns a concatenation:

```
>>> '1'+'1'
'11'
>>> "A string of " + 'characters'
'A string of characters'
```

Note that single (') and double (") quotes can be used in an indistinct way, as long as they are used with consistency. That is, if a string definition is started with one type of quote, it must be finished with the same kind of quote.¹¹

Different data types can't be added:

```
>>> 'The answer is ' + 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not int
```

Only elements of the same type can be added. To convert this into a sum of strings, the number must be converted into a string; this is done with the **str()** function:

```
>>> 'The answer is ' + str(42)
'The answer is 42'
```

The same result can be archived with “String Formatting Operations”:¹²

```
>>> 'The answer is {0}'.format(42)
'The answer is 42'
```

Note that the opposite transformation (from string to integer instead of from integer to string) can be done with the **int()** function:

```
>>> 1 + '1'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> 1 + int('1')
2
```

You can assign **names** to any Python element, and then refer to them later:

```
>>> number = 42
>>> 'The answer is {0}'.format(number)
'The answer is 42'
```

¹¹In [Chapter 3](#) there is a detailed description of strings.

¹²For more information, read PEP-3101 at (<http://www.python.org/dev/peps/pep-3101>).

TABLE 2.1 Arithmetic-Style Operators

Symbol	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation
%	Modulus (remainder)

Names should contain only letters, numbers, and underscores (`_`), but they can't start with numbers. In other programming languages names are called variables.

A command worth noting is `dir()`, it tells the names available in the current environment. Try:

```
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', <=
'__package__', '__spec__', 'number']
```

Names with the form `__name__` are specials and will be discussed later, but see that all variable names you defined are there.

2.2.4 Mathematical Operations

Any standard mathematical operation can be done in the Python shell:

```
>>> 12*2
24
>>> 30/3
10.0
>>> 2**8/2+100
228.0
```

Double star (`**`) stands for “elevated to the power of” and the inverted slash (`/`) is the division operation. So this expression means: $2^8 : 2 + 100$. In [Table 2.1](#) there is a list of arithmetic-style operators supported by Python.

Note that the operator precedence is the same as used in math. An easy way to remember precedence order is with the acronym **PEMDAS**:

P Parentheses have the highest precedence and are used to set the order of expression evaluation. This is why $2 * (3-2)$ yields 2 and $(3-1) ** (4-1)$ yields 8. Parentheses can also be used to make expressions easier to read.

E Exponentiation is the second in order, so $2**2+1$ is 5 and not 8.

MD Multiplication and Division share the same precedence. $2*2-1$ yields 3 instead of 2.

AS Addition and Subtraction also share the same (latest) order of precedence.

Last but not least, operators with the same precedence are evaluated from left to right. So $60/6*10$ yields 100 and not 1.

Division

There are currently two division operators. The standard one, `/` is called **true division** and returns the mathematical result of the division.¹³

```
>>> 10/4
2.5
```

There is also `//`, which is called **floor division**, it returns the integer part of the division:

```
>>> 10//4
2
```

2.2.5 Exit from the Python Shell

To exit from the Python shell, in MacOS or Linux, use CTRL-D (that is, press Control and D simultaneously). In Windows, press CTRL-Z and Enter. Another alternative, that works in any operating system, is to use the **exit()** function:

```
$ python
Python 3.5.1 |Anaconda 2.4.1 (64-bit)| (Dec 7 2015, 11:16:01)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()
$
```

2.3 BATCH MODE

Although the interactive interpreter is very useful, most nontrivial programs are stored in files. The code used in an interactive session can be accessed only when the session is active. Each time that an interactive session is closed, all typed code is gone. In order to have code persistence, programs are stored in text files. When a program is executed from such a text file, rather than line by line in an interactive interpreter, it is called **batch mode**. These are regular text files usually with the “.py” extension. These files can be generated with any standard text editor.¹⁴

¹³In Python 2.x the division symbol (`/`) is used to return the integer part of the division.

¹⁴Any text editor can be used for Python programming, but it is highly advisable to use a specialized text editor instead of a generic one. At the end of this chapter there is a section devoted to choosing an editor.

An optional feature of Python scripts under a Unix-like system is a first line with the path to the Python interpreter. If the Python interpreter is located at `/usr/bin/python` (a typical location in Linux), the first line will be:

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

This is called **shebang** and it is a Unix convention that allows the operating system to know what the interpreter is for the program and this interpreter can be executed without the user having to explicitly invoke the python interpreter.¹⁵ Invoking a Python program without this line causes the operating system to try to execute the program as a shell script.

Let's suppose that you have this very simple program:

Listing 2.1: `hello.py`: A “Hello World!” program

```
1 print("Hello World!")
```

Remember not to type the number at the beginning, it is there only for reference. This program will work from the command line only if it is called as an argument of the Python interpreter:

```
$ python hello.py
Hello World!
```

But if you want to run it as a standalone program, you will see something like this:

```
$ ./hello.py
./hello.py: line 1: syntax error near unexpected token <=
'Hello world!''
./hello.py: line 1: 'print('Hello world!')'
```

This error message is sent by the shell when trying to execute the program as a system script (without invoking Python). It can be avoided by editing the first line of the program:

Listing 2.2: `hello2.py`: Hello World! with shebang

```
1 #!/usr/bin/python
2 print("Hello World!")
```

This version works as it were an executable binary file:¹⁶

¹⁵You can specify an interpreter path to select a particular Python version when there is more than one version installed.

¹⁶In Linux and macOS you have to make sure that the file has executable permission, which is done with `chmod a+x hello.py`.

```
$ ./hello2.py
Hello World!
```

If you want to invoke the first available Python interpreter instead of a specific interpreter, use `#!/usr/bin/env python`. Use the path to a specific Python interpreter only when you want to run your program with a particular version (like `/usr/bin/python2.7`).

In Windows, this line is ignored since the interpreter is launched according to the file extension (`.py`).

Python can also be executed from within the text editor, provided that the editor has this functionality. In most editors you can launch a program with the *F5* key.

Another special comment that is usually found in Python code is the “encoding comment.” This line defines the character encoding for the rest of the document and it takes this form:

```
# -*- coding: ENCODING -*-
```

Where `ENCODING` may be, for example, `ascii`, `latin1`, `8859-1`, `UTF-8` and others. So an encoding line for a source code with characters encoding in `UTF-8` will have this line:

```
# -*- coding: UTF-8 -*-
```

Without encoding comment, Python’s parser will assume `UTF-8` (or `ASCII` in Python prior to version 3).

2.3.1 Comments

If you tried this program in any editor with syntax coloring capabilities, you may have noted that the first line (`#!/usr/bin/python`) has a different color. That is due to the use of the “`#`” symbol. This character has special significance for Python. It is used to identify lines that aren’t executed by the interpreter. As a result, these lines are called “comments.” Comments don’t add functionality to the program but help the developer or other readers of the code. Let’s rewrite the previous code with a comment:

Listing 2.3: Hello World! with comments

```
#!/usr/bin/env python
# The next line prints the string "Hello World!"
print("Hello World!")
```

The comment in this particular code is rather pointless since there is no doubt about the function of “`print`.” In other programs there is code that is not so easy to

understand and where a comment can improve code readability. It is customary to put the comments before the code you are referring to. Comments are made mostly to help someone else to understand our code, but they can be useful even for the same programmer who sees the code sometime after writing and doesn't remember the purpose of a routine.

Comments can also be used to disable part of the code (this is called “comment-out” in programming jargon). This is usually done for debugging purposes. When trying alternative codes to accomplish a task, it is better to have an inactive part of the code until you are sure which code you will use. It is easier to uncomment an inactive code than retype something that was deleted. This is such a common task that all Python editors have tools to comment-out or to uncomment an entire block of texts.¹⁷

Tip: Extensions in Python.

Python files have the **.py** extension, but you could also find other extensions that are related to Python:

- **py**: Standard Python files.
 - **pyc**: “Compiled” Python files. When you import a Python module for the first time, it gets compiled into byte code, so the next time it starts faster. Compile can be forced from Python with the **compile_dir** function in the **compileall** module. Note that **.pyc** files load faster but do not run faster.
 - **pyo**: “Optimized” code. This is generated by running the Python interpreter with the *-o flag*. Don't be fooled with the name, most code will run at same speed even with the *-o flag* enabled.
 - **pyw**: It is a standard Python file with an extension that makes Windows execute them with `pythonw.exe` instead of `python.exe`. `Pythonw.exe` doesn't launch the DOS console, so it is preferred for graphical programs under Windows.
-

2.3.2 Indentation

One of the first things that stands out to software developers about Python is its code indentation system. Non-programmers must be wondering at this point what is indentation of the source code. Here is some C code that is not indented:

```
if (attr == -1){while (x<5){
printf("Waiting...\n");wait(1);
```

¹⁷In the Python default editor (IDLE) this tool is under the *Format* menu.

```
x = x+1;}printf("Everything is OK\n");}
else {printf("There is an error\n");}
```

The indented version of the same program portion (or “code snippet”):

```
if (attr == -1) {
    while (x<5) {
        printf("Waiting...\n");
        wait(1);
        x = x+1;}
    printf("Everything is OK\n");}
else {
    printf("There is an error\n");}
```

Even without knowing C we can say that the second program is more readable than the first one. In a programming language like C or Java, code blocks that are executed as an entity, are separated with braces. This way the interpreter knows that, for example, `printf("Everything is OK\n");` is within the `if` structure but not within `while`. The logical relations between the elements are clearer in an indented program than one without indentation. Inspect the following code snippet in Python, where there are no braces but there are code blocks that are defined by indentation.

```
if attr==-1:
    while x<5:
        print("Waiting...")
        wait(1)
        x = x + 1
    print("Everything is OK")
else
    print("There is an error")
```

It is not important at this time if you don’t understand this program. The purpose of this example is to show one of the most striking aspects of language. This is considered an advantage because when the structure of the code is clear enough, there will be less chance to introduce coding mistakes. Some say that it is annoying having to maintain the code this way, but this is not the case. Most text editors deal with code indentation in an automatic way so there is no burden on the programmer. Another criticism to the mandatory indentation is the deep nesting of the code; some statements are placed at the far right. There are programming tools to avoid writing code with too many levels of indentation (such as writing modular code). Using these tools appropriately is a desired skill to have and it is independent of the programming language that you use.¹⁸ Forcing the programmer

¹⁸Linus Torvalds, the creator of the Linux kernel, has said, “If you need more than 3 levels of indentation, you’re screwed anyway, and should fix your program.”

to use indentation is a feature that goes along with one aspect of Python’s design philosophy: Readability counts.¹⁹ As Oliver Fromme wrote in “Python: Myths about Indentation”:²⁰ “Python forces you to use indentation that you would have used anyway, unless you wanted to obfuscate the structure of the program.”

2.4 CHOOSING AN EDITOR

In principle, any text editor can be used to program in Python. Nothing prevents you programming in Notepad (if you are so inclined) or any lightweight text editor, although there is a lot to gain by using an editor designed for Python programming.

Choosing an editor is not a trivial matter; in fact it is a matter of controversy among software developers to the point of leading to “editor wars.”²¹ It may not be something worth fighting for, but choosing the best editor for your needs could boost your productivity.

Following is a short review of popular editors used by Python developers.

2.4.1 Sublime Text

Sublime Text²² is arguably one of the most used text editors. After using it a couple of hours is easy to understand why. It is lean, fast and powerful, features not easily found in the same program. The user interface (UI) is minimalist but not frustrating. The default color palette used for syntax highlight is pleasing to the eye. It has several nice features than once you get used to, you don’t want to switch to another editor without those features. Some prominent features:

- **Minimap:** There is an overview of the whole document in the right side of the editor, and the developer can scroll on it and inspect the large portions of code quickly.
- **Fast global search:** A common problem for developers working with a large code base is text searching in all project files. Instead of resorting to command line utilities like *find* and *grep*, there is an option to do it in a fast and intuitive way without leaving the editor.
- **Performance:** Sublime text is fast by all accounts (start-up time, zero latency). Even when opening large files, the editor won’t drag you down.
- **Column selection:** This allows us to select portions of text in column form, and after selecting a column, multiple insertion points will appear so you can enter text in multiple positions at once.

¹⁹Please see <http://www.python.org/dev/peps/pep-0020> for more information on the guiding principles for Python’s design.

²⁰http://www.secnetix.de/~olli/Python/block_indentation.hawk.

²¹See https://en.wikipedia.org/wiki/Editor_war.

²²Available at <https://www.sublimetext.com>.

- **Extensible:** By using plugins you can extend editor functionality to suit your needs. There is a plugin called *Package Control*²³ that allows the user to search, download and install other plugins without leaving Sublime. Plugins are written in Python, which also explains why this editor is popular within Python developers.
- **Truly multiplatform:** Sublime looks the same on all three major platforms. Some key shortcuts will change in order to follow each operating system UI guidelines.

In most aspects, Sublime is the best text editor for software development. Although it has a problem that may be a deal breaker for some or just a annoyance for others: Sublime is closed source software and a commercial license is needed to use it. If you can pay the license (\$70 at the moment) and don't have problem with closed source software, this may be the ideal multi-purpose text editor.

2.4.2 Atom

Atom is a text editor made by the people at GitHub.²⁴ The idea behind this editor is to make a product convenient as Sublime and TextMate, but extensible and flexibly like Emacs and Vim. At first Atom looks like Sublime. This is not by chance; their creators copied Sublime UI in order to attract its users. Apart from the UI, most of the functionality and shortcuts are kept, so the transition should be easy. There are some differences, like the underlying technology. While the Sublime core is made in C++ with Python as an extension language, Atom is based on Chromium (the open source version of Google Chrome). The speed difference is noticeable. Atom is slower, so it is not advisable to use it with old or underpowered machines. The advantage of using Chromium is that the editor can be customized easily with JavaScript, HTML, and CSS. This makes Atom an excellent editor for web developers. Since Atom is made by GitHub, it has some Git integration not seen in other editors²⁵.

- **Smart autocompletion:** Autocompletion in interpreted languages like Python are not perfect, but Atom does a good job offering adequate options while you type.
- **Modular design:** It comes with basic functionality and then install packages from the settings page. The package selection (more than 6200) is very similar to Sublime, and most used packages are already ported to Atom. Apart from packages, there are themes to control look and feel, and with more than 2100 themes at this moment you won't get bored.

²³Download Package Control from <https://packagecontrol.io>.

²⁴<https://atom.io>

²⁵You can access to some of the most common Git operations without leaving Atom. For more information, read <http://blog.atom.io/2017/05/16/git-and-github-integration-comes-to-atom.html>.

- **Open Source:** You can access the source code on GitHub and be part of the developer community. In this case, open source also means free. Development is sponsored by GitHub.
- **Truly multiplatform:** As with Sublime, it looks the same on all three major platforms.

Atom is the best alternative if you can't or don't want to pay for the Sublime license. It is not perfect. The speed in a lot of cases is an issue. With large document, it eats RAM in a way that may slow down your computer at to the point of being unresponsive. Without any doubt, Atom is worth trying, test it with your hardware and settings to see if it sluggish.

2.4.3 PyCharm

This is a Python editor used mostly in commercial/professional settings. There is a new “educational” version.²⁶ It advertises itself as a “Free, Easy & Professional Tool to Learn Programming with Python.” This edition is free and has support for course creation and distribution, so it comes with a library of courses and you may also access material created by another users. I didn't use it to learn Python, so I can't tell if it is suitable for this purpose, but as an advanced user I found that this educational aspect doesn't affect my normal developer workflow, in fact, it help me by displaying tips each time there was some room for improvement. This educational edition does not support different web development technologies, remote development capabilities or additional languages.

These features are included in **PyCharm**:

- **Intelligent Python Assistance:** Smart code completion, code inspections, on-the-fly error highlighting and quick fixes, along with automated code refactorings. This last point is much appreciated by professionals, while the error highlighting is useful for novices. For example if you import a module and don't use through the file, it is marked in gray. If you call a method that doesn't exist, it is highlighted in yellow.
- **Scientific Tools:** Integrates with Jupyter Notebook, has an interactive Python console, and supports Anaconda as well as multiple scientific packages including matplotlib and NumPy.
- **Built-in Developer Tools:** A huge collection of tools out of the box: an integrated debugger and test runner, Python profiler, a built-in terminal, and integration with major VCS and built-in database tools.
- **Web Development Frameworks:** Web Development Frameworks: PyCharm offers great framework-specific support for modern web development

²⁶ Available at <https://www.jetbrains.com/pycharm-edu>



Figure 2.3 PyCharm Edu welcome screen.

frameworks such as Django, Flask, Google App Engine, Pyramid, and web2py. Pro edition only.

2.4.4 Spyder IDE

Spyder IDE is an open source cross-platform IDE for scientific programming. It is included with the Anaconda distribution, but can be downloaded and used as a standalone editor. It also supports plugins, although the selection of plugins are not as big as Sublime and Atom. In part this is not an issue because Spyder is a Python IDE so it doesn't need plugins the way Sublime and Atom do in order to have Python functions. Even if it is a Python IDE, it is not limited to Python; it supports several languages such as C, C++, and Fortran (because its origin as a development platform for science).

It includes most of the features of PyCharm (code completion works really well) so in a way is a good alternative. When you highlight a word, it automatically highlights all instances of that word. The interactive console support both Python and IPython (a better interactive console). In my experience it is not so stable; sometimes the process running in the console get deattached from the code in the editor and you must restart the IDE.

Availability: Spyder IDE comes package inside Anaconda and WinPython. So if you use any of these Python distributions, you already have Spyder IDE. If you want to install Spyder without installing a Python distribution, you can install it using PIP:

```
pip install spyder
```

If you are using Conda, you can run it from the Conda Navigator or from the command line:

```
conda install spyder
```

2.4.5 Final Words about Editors

There is no editor that is unquestionably better than all others in all areas. As multipurpose editors, Sublime and Atom are good choices. If you are comfortable with closed source software, Sublime is the best choice (provided you can pay the license fee). As an alternative, Atom provides most benefits of Sublime if you have a powerful machine.

As Python-specific editors, both Spyder and PyCharm have everything you may need for Python programming. Both IDEs come with integrated terminal emulation that comes handy when debugging (and that may be a considerable amount of time). They also play well with Anaconda Python distribution, in fact, Spyder is part of Anaconda.

All mentioned editors work with Windows, but if you are a Windows developer you may consider Visual Studio, which started supporting Python recently.²⁷ It is so new that there is not much feedback on how it performs, but Windows developers are feeling at home with this product.

I am not going to recommend a particular editor as I consider this a personal choice. My recommendation is that you try all that you can and choose the one that best fits your needs. I use Atom for general purpose file editing (this book is made in L^AT_EX using Atom with a plugin that helps L^AT_EX editing), but for Python I tend to use PyCharm Edu.

2.5 OTHER TOOLS

Besides code editors, developers tend to use other tools that helps them do their work. Although you may not need them at the beginning of your learning process, it's worth mentioning some useful tool you may use along your way.

- **Jupyter Notebook** (<http://jupyter.org>): A web application that runs locally and allows you to create and share documents that contain Python code that can be run inside the web page. Besides equations, it can show text and interactive graphics. All code in this book is also available in this format. For more information see the book web page at <http://py3.us>.
- **Kite** (<https://kite.com>): It defines itself as “The smart copilot for programmers”. It is an utility that integrates into your code editor and provides code

²⁷See <https://www.visualstudio.com/vs/python/>.

completions based on the most used options in the web, documentation and examples. See their presentation video in their web page. Currently available for Windows and macOS, Linux is “almost done” at the time of writing.

- **QuantifiedCode** (<https://QuantifiedCode.com>): An online tool that reads your code from your software repository and display comments and very useful hints about your code. This is not a free service,
- **Pylint** (<https://www.pylint.org>): A command line utility that analyses your code and outputs comments on how to improve it. It checks if the code follows the Python style guide, it detects errors and duplicated code that could be refactored. .
- **Pylama** (<https://github.com/klen/pylama>): Wraps several tools, like Pylint and others, to provide source code analysis. It is not a user friendly software, but it is powerful.
- **ptpython** (<https://github.com/jonathanslenders/ptpython>): A replacement for the Python interactive interpreter (or REPL). It adds syntax highlighting, multiline editing and autocompletion to the Python terminal.

2.6 ADDITIONAL RESOURCES

- Using the Python Interpreter.
<https://docs.python.org/3/tutorial/interpreter.html>
- IPython: An interactive computing environment.
<http://ipython.org/>
- PyFormat: Format values in strings using **format()** and **%**.
<https://pyformat.info>
- Wikipedia article: “Comparison of text editors.”
http://en.wikipedia.org/wiki/Comparison_of_text_editors
- Python Anywhere: “Host, run, and code Python in the cloud!”
<https://www.pythonanywhere.com>
- Repl.it: An online Python interpreter that allows code sharing.
<https://repl.it/languages/python3>

2.7 SELF-EVALUATION

1. Define: Program, instruction, and variable.
2. What is the difference between Python and cPython?

3. Name some Python implementations.
4. What is the advantage of having both single and double quotes?
5. What is `format()`?
6. What is an RPEL?
7. When would you use batch mode instead of the interactive shell?
8. What is indentation? Why is it mandatory in Python?
9. What is a comment in a source code?
10. Is there a valid reason to comment out working source code?
11. What is a “shebang”?
12. What is an “encoding comment.” and when should you use it?

Basic Programming: Data Types

CONTENTS

3.1	Strings	40
3.1.1	Strings Are Sequences of Unicode Characters	41
3.1.2	String Manipulation	42
3.1.3	Methods Associated with Strings	42
3.2	Lists	44
3.2.1	Accessing List Elements	45
3.2.2	List with Multiple Repeated Items	45
3.2.3	List Comprehension	46
3.2.4	Modifying Lists	47
	Adding	47
	Removing	47
3.2.5	Copying a List	49
3.3	Tuples	49
3.3.1	Tuples Are Immutable Lists	49
3.4	Common Properties of the Sequences	51
	Indexing	51
	Slicing	52
	Membership Test	53
	Concatenation	53
	len, max, and min	53
	Turn a Sequence into a List	54
3.5	Dictionaries	54
3.5.1	Mapping: Calling Each Value by a Name	54
3.5.2	Operating with Dictionaries	56
	Dictionaries Are Made of Keys and Values	56
	Query Dictionary Values	57
	Erasing Elements	58
3.6	Sets	59
3.6.1	Unordered Collection of Objects	59
	Creating a Set	59
3.6.2	Set Operations	60
	Intersection	60
	Union	60

Difference	61
Symmetric Difference	62
3.6.3 Shared Operations with Other Data Types	62
Maximum, Minimum, and Length	62
Converting a Set into a List	62
3.6.4 Immutable Set: Frozenset	63
3.7 Naming Objects	63
3.8 Assigning a Value to a Variable versus Binding a Name to an Object	64
3.9 Additional Resources	67
3.10 Self-Evaluation	68

As mentioned in the previous chapter, some data structures are shared between different computer languages, but some of them are language specific. That is why data types somehow define a computer language. Python has its own characteristic data types.

One such fundamental data structure is a *sequence*. Inside a sequence, the elements have a sequential order. For example the **string**, which is an ordered sequence of characters. Other sequences are **lists** and **tuples**.¹ Although fundamental differences exist between these types of sequences, they share common properties. Sequence elements have an order, can be indexed, can be sliced, and can be iterated. Don't worry if you don't understand some of these terms. Just keep on reading. We'll see all these points during this chapter.

Apart from *sequences*, there are also *unordered* data types: *dictionaries* and *sets*. A *dictionary*² stores relationships between a key and a value, while a *set* is just an unordered collection of values. The next pages are focused on ordered (string, list, and tuple) and unordered types (dictionary and set).

3.1 STRINGS

A string is a sequence of symbols delimited by a single quote ('), double quotes ("), triple single quotes ('''), or triple double quotes ("""). Therefore, the following strings are equivalent:

```
"This is a string in Python"
'This is a string in Python'
'''This is a string in Python'''
"""This is a string in Python"""
```

It may seem a little bit redundant to have so many ways to delimit a string.

¹There are more sequence types not covered in this book. For more information on other sequence types see *Additional Resources* at the end of the chapter.

²Also classified as a mapping data type.

The advantage of having both single (') and double (") quote delimiters is that we can insert a single quote in a string delimited for double quote, and vice versa:

```
"A single quote (') inside a double quote"
'Here we have "double quotes" inside single quotes'
```

The important thing to remember is that if we begin a string with a type of quote, we must finish it with the same type of quote. The following string is not valid:

```
>>> "Mixing quote types leads to the dark side'
File "<stdin>", line 1
    "Mixing quote types leads to the dark side'
                                     ^
```

SyntaxError: EOL while scanning single-quoted string

Note: EOL stands for **end-of-line**.

Regarding strings enclosed by triple quotes, we can use them to indicate multi-line strings (also known as a block string):

```
"""Hi! I'm a
multiline
    string"""
```

The character '\n' represents an end-of-line (EOL) character. Therefore, the code above could be written in one line as:

```
"Hi! I'm a\nmultiline\n    string"
```

You can use triple quotation marks to build and format a string just like you'd expect to see it displayed. There are other uses for triple quoted strings such as documentation.

3.1.1 Strings Are Sequences of Unicode Characters

Since Python 3, all string sequences are Unicode characters by default. So this is a valid string:

```
>>> 'In Python 3, strings are Unicode: こんにちは 世界'
'In Python 3, strings are Unicode: こんにちは 世界'
```

If you are wondering about Unicode, it is an industry standard that “provides the basis for processing, storage and interchange of text data in any language in all modern software.”³ Since Python 3 has built-in support for Unicode, you just use it without the need to do explicit declarations or conversions. However, you may need to take some precautions when reading or writing data from or to external sources.

³Taken from the Unicode FAQ at http://www.unicode.org/faq/basic_q.html.

3.1.2 String Manipulation

Strings are immutable. Once a string is created, it can't be modified. If you need to change a string, what you can do is to make a derived string. This is done using the string as a parameter in a function and then gets the returned value. In the following example there is a string that represents an amino-acid sequence and it is called *signal_peptide*:

```
>>> signal_peptide = 'MASKATLLLAFTLLFATCIA'
```

To get a lower-case version of the string, use the method **lower()**:

```
>>> signal_peptide.lower()
'maskatllllaftlllfatcia'
```

Despite having obtained the lower-case string, the original string has not been modified:

```
>>> signal_peptide
'MASKATLLLAFTLLFATCIA'
```

If we want this new lower case string to have the same name as the previous one, we need to assign it:

```
>>> signal_peptide = signal_peptide.lower()
>>> signal_peptide
'maskatllllaftlllfatcia'
```

The net effect is like we modified the string. It's time to see some methods associated with strings.

3.1.3 Methods Associated with Strings

replace(old,new[,count]): Allows us to replace a portion of a string (*old*) with another (*new*). If the optional argument *count* is used, only the first *count* occurrences of *old* will be replaced:

```
>>> dna_seq = 'GCTAGTAATGTG'
>>> m_rna_seq = dna_seq.replace('T','U')
>>> m_rna_seq
'GCUAGUAAUGUG'
```

count(sub[, start[, end]]): Counts how many times the substring *sub* appears, between the *start* and *end* positions (if available). Let's see how it can be used to calculate the CG content⁴ of a sequence:

⁴CG content is the amount of cytosine and guanine in a DNA sequence. CG content is related to the DNA melting temperature and other physical properties.

```
>>> dna_seq
'GCTAGTAATGTG'
>>> c = dna_seq.count("C")
>>> g = dna_seq.count("G")
>>> (c+g)/len(dna_seq)*100
41.66666666666667
```

find(sub[,start[,end]]): Returns the position of the substring *sub*, between the *start* and *end* positions (if available). If the substring is not found in the string, this method returns the value -1:

```
>>> m_rna_seq
'GCUAGUAAUGUG'
>>> m_rna_seq.find('AUG')
7
>>> m_rna_seq.find('GGG')
-1
```

index(sub[,start[,end]]): Works like *find()*. The difference is that *index* will raise a *ValueError* exception when the substring is not found. This method is recommended over *find()* because the value -1 could be interpreted as a valid value, while a *ValueError* returned by *index()* can't be taken as a valid value.

split([sep [,maxsplit]]): Separates the “words” of a string and returns them in a list. If a separator (*sep*) is not specified, the default separator will be a white space:

```
>>> 'This string has words separated by spaces'.split()
['This', 'string', 'has', 'words', 'separated', 'by', 'spaces']
```

When white space is not the data separator, we have to specify a custom separator:

```
>>> "Alex Doe,5555-2333,nobody@example.com".split()
['Alex', 'Doe,5555-2333,nobody@example.com']
```

In this case the separator is a comma (“,”), so we have to state it explicitly:

```
>>> "Alex Doe,5555-2333,nobody@example.com".split(",")
['Alex Doe', '5555-2333', 'nobody@example.com']
```

Bioinformatics Application: Parsing BLAST Files.

One of the most used bioinformatics programs is NCBI-BLAST (this program is reviewed from page 177).

The BLAST standalone executable can generate output as a “tab separated file” (by using the argument `-m 8`). This output file can be parsed by using `split('\t')`.

The inverse function of `split()` is `join()`:

join(seq): Joins the sequence using a string as a “glue character”:

```
> ''.join(['Alex Doe', '5555-2333', 'nobody@example.com'])
'Alex Doe;5555-2333;nobody@example.com'
```

To join a sequence without any glue character, use empty quotes (`""`):

```
>>> ''.join(['A','C','A','T'])
'ACAT'
```

For a complete description of string methods, see `help(str)` in the console.

3.2 LISTS

Lists are one of the most versatile object types in Python. A list is an ordered collection of objects. It is represented by elements separated by commas and enclosed between square brackets.

We already have seen a list as a result of applying the `split()` function:

```
>>> 'Alex Doe,5555-2333,hi@example.com'.split(',')
['Alex Doe', '5555-2333', 'hi@example.com']
```

This is a three-element list, `'Alex Doe'`, `'5555-2333'`, and `'hi@example.com'`, all of them strings.

The next code shows how to define and name a list:

```
>>> first_list = [1, 2, 3, 4, 5]
```

This is a list with five elements. In this case, all the elements are of the same type (integer). A list can hold different kinds of elements:

```
>>> other_list = [1, 'two', 3, 4, 'last']
```

A list can even contain another list:

```
>>> nested_list = [1, 'two', first_list, 4, 'last']
>>> nested_list
[1, 'two', [1, 2, 3, 4, 5], 4, 'last']
```

An empty list is defined with empty brackets:

```
>>> empty_list = []
>>> empty_list
[]
```

An empty list doesn't have any use as is, but sometimes we may want to define an empty list to add elements at a later time.

3.2.1 Accessing List Elements

As other sequence data types, you can get list elements by an index starting at zero.

```
>>> first_list = [1, 2, 3, 4, 5]
>>> first_list[0]
1
>>> first_list[1]
2
```

Negative numbers are used to access lists from the right:

```
>>> first_list = [1, 2, 3, 4, 5]
>>> first_list[-1]
5
>>> first_list[-4]
2
```

Another way of obtaining lists is by turning a non-list object into a list by using the built-in function `list()`:

```
>>> aseq = "atggctaggc"
>>> list(aseq)
['a', 't', 'g', 'g', 'c', 't', 'a', 'g', 'g', 'c']
```

3.2.2 List with Multiple Repeated Items

If you want to initialize a list with the same item repeated multiple times, you can use the `*` operator like this:

```
>>> samples = ['red'] * 5
>>> samples
['red', 'red', 'red', 'red', 'red']
```

This can be used to pre-populate a list with empty values and can be useful when working with big lists and the number of elements is known beforehand. Defining a list with a fixed size is more efficient than creating an empty list and expanding it as needed:

```
>>> samples = [None] * 5
>>> samples
[None, None, None, None, None]
```

3.2.3 List Comprehension

There is another way to define a list. A list can be created from another list. As in mathematics where you can define a set by enumerating all its elements (enumeration) or by describing properties shared by its members (comprehension), in Python a list can be created by both methods.

A set defined by enumeration,

$$A = \{0, 1, 2, 3, 4, 5\}$$

A list defined by enumeration in Python,

```
>>> a = [0, 1, 2, 3, 4, 5]
```

A set defined by comprehension,

$$B = \{3 * x / x \in A\}$$

This is equivalent to

$$B = \{0, 3, 6, 9, 12, 15\}$$

A list defined by comprehension in Python,

```
>>> [3*x for x in a]
[0, 3, 6, 9, 12, 15]
```

Any Python function or method can be used to define a list by comprehension. For example from a list of strings, let's make a list with the same elements but without trailing and leading white spaces:

```
>>> animals = [' King Kong', ' Godzilla ', 'Gamera ']
>>> [x.strip() for x in animals]
['King Kong', 'Godzilla', 'Gamera']
```

We can add a conditional statement (**if**) to narrow the result set:

```
>>> animals = [' King Kong', ' Godzilla ', 'Gamera ']
>>> [x.strip() for x in animals if 'i' in x]
['King kong', 'Godzilla']
```

3.2.4 Modifying Lists

Unlike strings, lists can be modified⁵ by adding, removing, or changing their elements:

Adding

There are three ways to add elements into a list: **append**, **insert**, and **extend**.

append(*element*): Adds an element at the end of the list.

```
>>> first_list.append(99)
>>> first_list
[1, 2, 3, 4, 5, 99]
```

insert(*position*,*element*): Inserts the element *element* at the position *position*.

```
>>> first_list.insert(2,50)
>>> first_list
[1, 2, 50, 3, 4, 5, 99]
```

extend(*list*): Extends a list by adding a list to the end of the original list.

```
>>> first_list.extend([6,7,8])
>>> first_list
[1, 2, 50, 3, 4, 5, 99, 6, 7, 8]
```

This is the same as using the + symbol:

```
>>> [1,2,3]+[4,5]
[1, 2, 3, 4, 5]
```

Removing

There are three ways to remove elements from a list.

pop([*index*]): Removes the element in the index position and returns it to the point where it was called. Without parameters, it returns the last element.

```
>>> first_list
[1, 2, 50, 3, 4, 5, 99, 6, 7, 8]
>>> first_list.pop()
8
>>> first_list.pop(2)
50
>>> first_list
[1, 2, 3, 4, 5, 99, 6, 7]
```

⁵Lists are called “mutables” in Python jargon.

TABLE 3.1 Common List Operations

Properties	Description
<code>l.append(x)</code>	Adds the <i>x</i> element to list <i>s</i>
<code>l.count(x)</code>	Counts how many times <i>x</i> is in <i>s</i>
<code>l.index(x)</code>	Returns the location of <i>x</i> in list <i>s</i>
<code>l.remove(x)</code>	Removes the element <i>x</i> from list <i>s</i>
<code>l.reverse()</code>	Reverses list <i>s</i>
<code>l.sort()</code>	Sorts list <i>s</i>

remove(*element*): Removes the element specified in the parameter. In the case where there is more than one copy of the same object in the list, it removes the first one, counting from the left. Unlike **pop()**, this function does not return anything.

```
>>> first_list.remove(99)
>>> first_list
[1, 2, 3, 4, 5, 6, 7]
```

Trying to remove a nonexistent element raises an error.⁶

```
>>> first_list
[1, 2, 3, 4, 5, 6, 7]
>>> first_list.remove(10)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: list.remove(x): x not in list
```

Another way of removing an element of a list is using the command **del**, for what:

```
del first_list[0]
```

This has a similar effect to:

```
first_list.pop(0)
```

with the difference that **pop()** returns the extracted element to where it was called, while **del** just deletes it.⁷

Table 3.1 summarizes other properties of lists.

⁶In Chapter 7 there is a more detailed description of exceptions.

⁷The object is not deleted. What actually happens is that the reference between the object and its name is lost. For the programmer, this action has the same effect as if it were deleted (it is not possible to gain access to the object). Eventually, the “Python garbage collector” will eliminate it in a transparent and automatic way.

3.2.5 Copying a List

Copying a list can be tricky. In the following code we try to copy the list *a* into *b*, but when I modify *b* by removing an element, this element is also removed from *a*:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b.pop()
3
>>> a
[1, 2]
```

As seen, “=” doesn’t copy the values, it copies a reference to the original object.⁸ To copy a list you must use the **copy** method in the **copy** module:

```
>>> import copy
>>> a = [1, 2, 3]
>>> b = copy.copy(a)
>>> b.pop()
3
>>> a
[1, 2, 3]
```

There is a way to accomplish the same without using the **copy** module:

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> b.pop()
3
>>> a
[1, 2, 3]
```

3.3 TUPLES

3.3.1 Tuples Are Immutable Lists

A tuple is a collection of ordered objects with the characteristic that once created, it cannot be modified. That is why they are referred to as “immutable lists.” Python objects can be divided into mutable and immutable. As the name implies, immutable objects cannot be modified after they are created. You can easily tell a tuple from a list because the tuple’s elements are enclosed between parentheses instead of square brackets:

```
>>> point = (23, 56, 11)
```

⁸For a detailed review of what is going on under the hood, please see page 64.

`point` is a tuple with three elements (23, 56, and 11).

When the tuple has only one element, you should use a trailing comma:

```
lone_element_tuple = (5,)
```

This is done to sort the ambiguity of having (5) that means 5 (number five) since parentheses around an expression are ignored. With the trailing comma and parentheses the Python interpreter can tell that it is a tuple and not an expression.

You are not allowed to add or to remove elements from a tuple:

```
>>> point.append(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'append'
>>> point.pop()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'pop'
```

In a certain way, a tuple is like a limited list (limited in the sense that we cannot modify it). So what are tuples good for? Why not just use a list instead?

There is a conceptual difference between the data types stored as a tuple and the data stored as a list. Lists should hold a variable quantity of objects of the same data type. A list containing the file names of all the files in a directory can be stored in a list. They are all of the same type (string), and the number of elements in the list changes according to each directory. The element ordering inside this list is not relevant.

On the other hand, a typical example of a tuple is a coordinate system. In a three-dimensional coordinate system, each point is referred to by a three-element tuple (x, y, z). The number of elements for each tuple does not change (since there are always three coordinates), and each position is important since each point corresponds to a specific axis.

We can say the same thing regarding the elements that are returned from a function or a dictionary key.⁹ Another advantage of the tuple is it can be used to make safer code - the information we don't want to change stays "write-protected" in an immutable tuple.

Tuples takes less memory than lists. When working with big datasets, this can be significant. Also the speed of operations involving tuples are faster than that of lists. While this may be true in specific cases, this fact alone shouldn't be a major consideration when choosing between a list or a tuple.

⁹This will make sense after seeing functions and dictionaries.

3.4 COMMON PROPERTIES OF THE SEQUENCES

Since sequences share common properties, let's see them together. You can apply these properties to lists, tuples, and strings.

Indexing

Indexing was discussed when covering lists, but for the sake of completeness, it is also covered here. Since the elements in the sequences are ordered, we can gain access to any element through an index that begins at zero:

```
>>> point = (23, 56, 11)
>>> point[0]
23
>>> point[1]
56
>>> sequence = 'MRVLLVALALLALAASATS'
>>> sequence[0]
'M'
>>> sequence[5]
'V'
>>> parameters = ['UniGene', 'dna', 'Mm.248907', 5]
>>> parameters[2]
'Mm.248907'
```

We can also access the elements of a sequence from the right by using negative numbers:

```
>>> point[-1]
11
>>> point[-2]
56
>>> my_sequence[-2]
'T'
>>> my_sequence[-4]
'S'
>>> my_sequence[-1]
5
```

To access an element that is inside a sequence, which is itself inside another sequence, you need to use another index:

```
>>> seqdata = ('MRVLLVALALLA', 12, '5FE9EEE8EE2DC2C7')
>>> seqdata[0][5]
'V'
```

The first index (0) indicates we're accessing the first element of `seqdata`. The second index (5) refers to the 6th element ('V') of the first element ('MRVLLVALALLA')

Slicing

You can select a portion of a sequence using *slice notation*. Slicing consists of using two indexes separated by a colon (:). These indexes represent a position in the existing space **between** the elements. The string "Python" can be represented as,

```
+---+---+---+---+---+---+
| P | y | t | h | o | n |
+---+---+---+---+---+---+
0   1   2   3   4   5   6
```

```
>>> my_sequence="Python"
>>> my_sequence[0:2]
'Py'
```

When omitting the first sub-index, the index value defaults to the first position (0):

```
>>> my_sequence[:2]
'Py'
```

On the other hand, when the second sub-index is omitted, the index value defaults to the last position (-1):

```
>>> my_sequence = "Python"
>>> my_sequence[4:6]
'on'
>>> my_sequence[4:]
'on'
```

There is a third, optional index to skip positions (step argument):

```
>>> my_sequence[1:5]
'ytho'
>>> my_sequence[1:5:2]
'yh'
```

A step with a negative number is used to count backwards. So -1 (in the third position) can be used to invert a sequence:

```
>>> my_sequence[::-1]
'nohtyP'
```

Note that slicing always returns another sequence.

Membership Test

You can verify whether an element belongs to a sequence, using the **in** keyword:¹⁰

```
>>> point = (23, 56, 11)
>>> 11 in point
True
>>> my_sequence = 'MRVLLVALALLALAASATS'
>>> 'X' in my_sequence
False
```

Concatenation

You can concatenate two or more sequences of the same class using the “+” sign:

```
>>> point = (23, 56, 11)
>>> point2 = (2, 6, 7)
>>> point + point2
(23, 56, 11, 2, 6, 7)
>>> dna_seq = 'ATGCTAGACGTCCTCAGATAGCCG'
>>> tata_box = 'TATAAA'
>>> tata_box + dna_seq
'TATAAAATGCTAGACGTCCTCAGATAGCCG'
```

Sequences of different types can't be concatenated:

```
>>> point + tata_box
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate tuple (not "str") to tuple
```

len, max, and min

len() returns the length (the number of items) of a sequence:

```
>>> point = (23, 56, 11)
>>> len(point)
3
>>> my_sequence = 'MRVLLVALALLALAASATS'
>>> len(my_sequence)
19
```

max() and **min()** applied over a sequence of numbers return, as expected, the maximum and the minimum value:

¹⁰Note that the **in** keyword was used on page 59

```
>>> point
(23, 56, 11)
>>> max(point)
56
>>> min(point)
11
```

max() and **min()** applied to strings return a character according to the maximum or minimum value of its ASCII code:

```
>>> my_sequence = 'MRVLLVALALLALAASATS'
>>> max(my_sequence)
'V'
>>> min(my_sequence)
'A'
```

Turn a Sequence into a List

To convert a sequence (like a tuple or a string) into a list, use the **list()** method:

```
>>> tata_box = 'TATAAA'
>>> list(tata_box)
['T', 'A', 'T', 'A', 'A', 'A']
```

Using a list provides us with methods to indirectly modify a string. Since lists, unlike strings, are mutable, we can convert a string to a list, modify this list and then convert it back into a string (with **str()**).¹¹ This process is not efficient, so I suggest that whenever possible, use string properties to obtain another string.

3.5 DICTIONARIES

3.5.1 Mapping: Calling Each Value by a Name

Dictionaries are a special data type not present in all programming languages. The main characteristic of a dictionary is that it stores arbitrary indexed unordered data types.

This example shows us why this data type is called a dictionary:

```
>>> iupac = {'A': 'Ala', 'C': 'Cys', 'E': 'Glu'}
>>> print('C stands for the amino acid {}'.format(iupac['C']))
C stands for the amino acid Cys
```

¹¹By using the method **join()** as it was described on page 44.

`iupac` is the name of a dictionary with three elements. It was defined by enclosing *key:value* pairs between curly brackets (`{}`).

This dictionary works as a translation table that allows us to translate between the one-letter amino acid code to a three-letter code. Every element consists of a pair *key:value*. The key is the index used to retrieve the value:

```
>>> iupac['E']
'Glu'
```

Not every object can be used as a dictionary key. Only immutable objects like strings, tuples and numbers can be used as keys. If the tuple contains any mutable object, it cannot be used as a key.

A dictionary can also be created from a sequence with **dict**:

```
>>> rgb = [('red','ff0000'), ('green','00ff00'), ('blue','0000ff')]
>>> colors_d = dict(rgb)
>>> colors_d
{'red': 'ff0000', 'blue': '0000ff', 'green': '00ff00'}
```

dict also accepts *name=value* pairs in the keyword argument list:

```
>>> rgb = dict(red='ff0000', green='00ff00', blue='0000ff')
>>> rgb
{'blue': '0000ff', 'green': '00ff00', 'red': 'ff0000'}
```

Another way to initialize a dictionary is to create an empty dictionary and add elements as needed:

```
>>> rgb = {}
>>> rgb['red'] = 'ff0000'
>>> rgb['green'] = '00ff00'
>>> rgb
{'green': '00ff00', 'red': 'ff0000'}
```

len(), returns the number of elements in the dictionary:

```
>>> len(iupac)
3
```

To add values to a dictionary,

```
>>> iupac['S'] = 'Ser'
>>> len(iupac)
4
```

Dictionaries are unordered because they don't keep track of the order of their elements. When you request to see the contents of the dictionary, you may or may not get the elements in the same order as they were entered:

```
>>> iupac = {'A':'Ala','C':'Cys','E':'Glu'}
>>> iupac
{'E': 'Glu', 'C': 'Cys', 'A': 'Ala'}
```

When entering a new element, it is not inserted at a particular place (like the end):

```
>>> iupac['X'] = 'Xaa'
>>> iupac
{'E': 'Glu', 'X': 'Xaa', 'C': 'Cys', 'A': 'Ala'}
```

Don't rely on a dictionary to keep track of element order. If you need an ordered dictionary, you must use **OrderedDict**¹²:

```
>>> from collections import OrderedDict
>>> d = OrderedDict()
>>> d['a'] = 'A'
>>> d['b'] = 'B'
>>> d['c'] = 'C'
>>> d
OrderedDict([('a', 'A'), ('b', 'B'), ('c', 'C')])
```

For more information on OrderedDict, see PEP-372 at <https://www.python.org/dev/peps/pep-0372>.

Built-in data types as those found in this chapter are enough for most users, but for more advanced uses there is a third-party module for dealing with ordered content: **SortedContainers**¹³. Check it out when you need a fast and memory optimized solution for sorted containers when dealing with large amounts of data.

3.5.2 Operating with Dictionaries

As lists, dictionaries have their own methods.

Dictionaries Are Made of Keys and Values

To get the keys or values of a dictionary, there are methods like **keys()** and **values()**:

¹²In Python 3.6, dictionaries are ordered, but this is considered an implementation detail and should not be relied upon. This may change in Python 3.7, so please use OrderedDict if you want to keep order.

¹³<http://www.grantjenks.com/docs/sortedcontainers/>

```
>>> iupac
{'E': 'Glu', 'X': 'Xaa', 'C': 'Cys', 'A': 'Ala'}
>>> iupac.keys()
dict_keys(['E', 'X', 'C', 'A'])
>>> iupac.values()
dict_values(['Glu', 'Xaa', 'Cys', 'Ala'])
```

Note that these methods do not return a list (that was their behavior before Python 3), but they return a special object called **dictionary views**. This object shows you the current keys or values, so if it changes in the dictionary, it will change in the **dictionary view**:

```
>>> iupac.values()
dict_values(['Glu', 'Xaa', 'Cys', 'Ala'])
>>> iupac.keys()
dict_keys(['E', 'X', 'C', 'A'])
>>> iupac_keys = iupac.keys()
>>> iupac_vals = iupac.values()
>>> iupac.pop('X')
'Xaa'
>>> iupac_keys
dict_keys(['E', 'A', 'C'])
>>> iupac_vals
dict_values(['Glu', 'Cys', 'Ala'])
```

This can be useful when you need the keys or values in multiple parts of your program without the need of recalculating the keys or values each time they are used.

Another way of accessing the elements of a dictionary is by using **items()**, which returns a **dictionary view** with a tuple for every key/value pair:

```
>>> iupac = {'E': 'Glu', 'X': 'Xaa', 'C': 'Cys', 'A': 'Ala'}
>>> iupac.items()
dict_items([('E', 'Glu'), ('A', 'Ala'), ('C', 'Cys'), ('X', 'Xaa')])
```

Query Dictionary Values

To query a value from a dictionary without the risk of invoking an exception, use **get(k,x)**. *K* is the key of the element to extract, while *x* is the element that will be returned in case *k* is not found as a key of the dictionary.

```
>>> iupac = {'E': 'Glu', 'X': 'Xaa', 'C': 'Cys', 'A': 'Ala'}
>>> iupac.get('A','No translation available')
'Ala'
>>> iupac.get('Z','No translation available')
'No translation available'
```


TABLE 3.2 Methods Associated with Dictionaries

Properties	Description
<code>len(d)</code>	Number of elements of <i>d</i>
<code>d[k]</code>	The element from <i>d</i> that has a <i>k</i> key
<code>d[k] = v</code>	Set <i>d</i> [<i>k</i>] to <i>v</i>
<code>del d[k]</code>	Remove <i>d</i> [<i>k</i>] from <i>d</i>
<code>d.clear()</code>	Remove all items from <i>d</i>
<code>d.copy()</code>	Copy <i>d</i>
<code>k in d</code>	True if <i>d</i> has a key <i>k</i> , else False
<code>k not in d</code>	Equivalent to <code>not k in d</code>
<code>d.has_key(k)</code>	Equivalent to <code>k in d</code> , use that form in new code
<code>d.items()</code>	A copy of <i>d</i> 's list of (key, value) pairs
<code>d.keys()</code>	A copy of <i>d</i> 's list of keys
<code>d.update([b])</code>	Updates (and overwrites) key/value pairs from <i>b</i>
<code>d.fromkeys(seq[, value])</code>	Creates a new dictionary with keys from <i>seq</i> and values set to <i>value</i>
<code>d.values()</code>	A copy of <i>d</i> 's list of values
<code>d.get(k[, x])</code>	<i>a</i> [<i>k</i>] if <i>k</i> in <i>d</i> , else <i>x</i>
<code>d.setdefault(k[, x])</code>	<i>a</i> [<i>k</i>] if <i>k</i> in <i>d</i> , else <i>x</i> (also setting it)
<code>d.pop(k[, x])</code>	<i>d</i> [<i>k</i>] if <i>k</i> in <i>d</i> , else <i>x</i> (and remove <i>k</i>)
<code>d.popitem()</code>	Remove and return an arbitrary (key, value) pair

If you omit *x*, and there is no *k* key in the dictionary, the method returns **None**.

```
>>> iupac.get('Z')
None
```

Erasing Elements

To erase elements from a dictionary, use the **del** instruction:

```
>>> iupac = {'E': 'Glu', 'X': 'Xaa', 'C': 'Cys', 'A': 'Ala'}
>>> del iupac['A']
>>> iupac
{'C': 'Cys', 'X': 'Xaa', 'E': 'Glu'}
```

Table 3.2 summarizes the properties of dictionaries.

3.6 SETS

3.6.1 Unordered Collection of Objects

This type of data is also not commonly found in other programming languages. A **set** is a structure frequently found in mathematics. It is similar to a list, with two outstanding differences: *its elements do not preserve an implied order and every element is unique*.

The most common uses of sets are membership testing, duplicate removal, and the application of mathematical operations: intersections, unions, differences, and symmetrical differences.

Creating a Set

Sets are created with the instruction `set()`:

```
>>> first_set = {'CP0140.1', 'XJ8113.5', 'EF3616.3'}
```

It is also possible to create an empty set and then add the elements as needed:

```
>>> first_set = set()
>>> first_set.add('CP0140.1')
>>> first_set.add('XJ8113.5')
>>> first_set.add('EF3616.3')
>>> first_set
{'CP0140.1', 'XJ8113.5', 'EF3616.3'}
```

You can also define a set by comprehension, as in list comprehension (see page 46):

```
>>> {2*x for x in [1,2,3]}
{2, 4, 6}
```

Since a **set** does not accept repeated elements, there is no effect when you try to add an element that is already in the set:

```
>>> first_set.add('CP0140.1')
>>> first_set
{'CP0140.1', 'XJ8113.5', 'EF3616.3'}
```

In the case of set comprehension:

```
>>> {2*x for x in [1,1,2,2,3,3]}
{2, 4, 6}
```

This property can be used to remove duplicated elements from a list:

```
>>> uniques = {2,2,3,4,5,3}
>>> uniques
{2, 3, 4, 5}
```

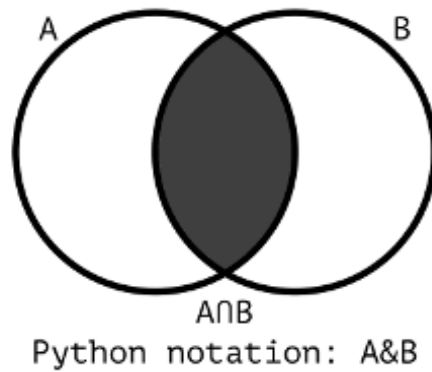


Figure 3.1 Intersection.

3.6.2 Set Operations

Intersection

To get the common elements in two sets (as shown in [Figure 3.1](#)), use the operator `intersection()`:

```
>>> first_set = {'CP0140.1', 'XJ8113.5', 'EF3616.3'}
>>> other_set = {'EF3616.3'}
>>> common = first_set.intersection(other_set)
>>> common
{'EF3616.3'}
```

It is equivalent to `&`:

```
>>> common = first_set & other_set
>>> common
{'EF3616.3'}
```

Union

The union of two (or more) sets is the operator **union** (as seen in [Figure 3.2](#)) and its abbreviated form is `|`:

```
>>> first_set = {'CP0140.1', 'XJ8113.5', 'EF3616.3'}
>>> other_set = {'AB7416.2'}
>>> first_set.union(other_set)
{'CP0140.1', 'XJ8113.5', 'EF3616.3', 'AB7416.2'}
>>> first_set | other_set
{'CP0140.1', 'XJ8113.5', 'EF3616.3', 'AB7416.2'}
```

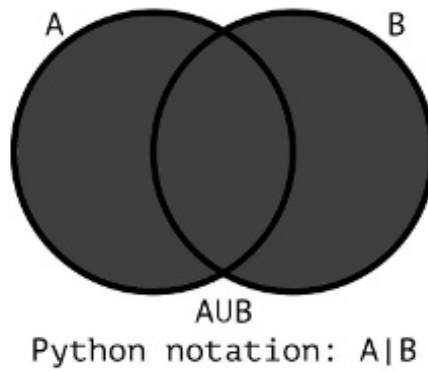


Figure 3.2 Union.

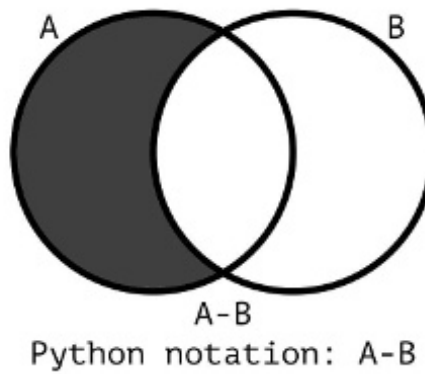


Figure 3.3 Difference.

Difference

A **difference** is the resulting set of elements that belongs to one set but not to the other (See [Figure 3.3](#)). Its shorthand is `-`:

```
>>> first_set.difference(other_set)
{'CP0140.1', 'XJ8113.5', 'EF3616.3'}
>>> first_set - other_set
{'CP0140.1', 'XJ8113.5', 'EF3616.3'}
>>> other_set - first_set
{'AB7416.2'}
```

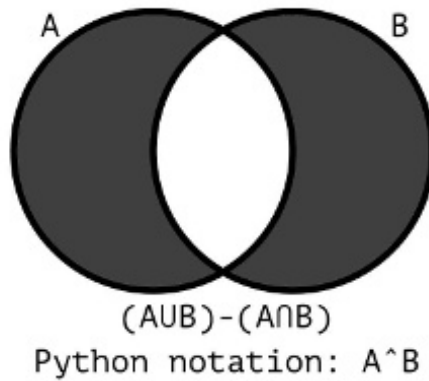


Figure 3.4 Symmetric difference.

Symmetric Difference

A symmetric difference refers to those elements that are not a part of the intersection (see [Figure 3.4](#)); its operator is `symmetric_difference` and it is shortened as `^`:

```
>>> first_set.symmetric_difference(other_set)
{'CP0140.1', 'XJ8113.5', 'EF3616.3', 'AB7416.2'}
>>> first_set ^ other_set
set(['EF3616.2', 'CP0140.1', 'CP0140.2', 'EF3616.1'])
```

3.6.3 Shared Operations with Other Data Types

Maximum, Minimum, and Length

Sets share some properties with sequences, such as `max`, `min`, `len`, `in`, etc. As we can expect, these properties work in the same way.

Converting a Set into a List

As with strings, sets can be turned into lists with the function `list()`:

```
>>> first_set
{'CP0140.1', 'XJ8113.5', 'EF3616.3'}
>>> list(first_set)
['CP0140.1', 'XJ8113.5', 'EF3616.3']
```

Type `help(set())` in the console to see all methods associated with sets.

3.6.4 Immutable Set: Frozenset

Frozenset is the immutable version of set. Its contents cannot be changed, so methods like `add()` and `remove()` are not available. It is generated with the **frozenset** object that takes an iterable as input:

```
>>> fs = frozenset(['a','b'])
>>> fs
frozenset({'a', 'b'})
>>> fs.remove('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'remove'
>>> fs.add('c')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
```

Since frozensets are immutable, they can be used as a dictionary key.

3.7 NAMING OBJECTS

Valid names should contain letters, numbers, and underscores (`_`), but they can't start with numbers. They also can't be "language reserved words" such as:

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
asser	else	import	pass	
break	except	in	raise	

Here is a sample of invalid names, with an explanation in a comment:

```
>>> 23crm = "1"      # Start with a number
>>> 23 = "1"         # Start with a number
>>> Var? = "value"   # Has an invalid character (?).
>>> $five = 5         # Has an invalid character ($)
>>> for = 123         # Has a reserved word
>>> if = "data"       # Has a reserved word
```

We've seen several name assignments up to this point:

```
>>> my_sequence = 'MRVLLVALALLALAASATS'
>>> first_list = [1,2,3,4,5]
```

```
>>> d= {1:'a',2:'b',3:'c'}
>>> k = d.keys()
>>> point = (23,56,11)
>>> first_set = {'CP0140.1','XJ8113.5','EF3616.3'}
>>> fs = frozenset(['a','b'])
```

Those are valid names. There are also naming conventions that must be followed to improve code readability. These conventions are part the Python Style Guide.¹⁴ According to this guide, names should be lowercase, with words separated by underscores as necessary to improve readability.

3.8 ASSIGNING A VALUE TO A VARIABLE VERSUS BINDING A NAME TO AN OBJECT

The following statement can be thought of as a variable assignment:

```
>>> a = 3
>>> b = [1,2,a]
```

Also this one:

```
>>> b = [1,2,a]
```

Translated into English, they mean: “Let the variable **a** have a value of 3” and “Let the variable **b** have a list with three elements: 1, 2, and **a** (that has the value 3).”

Printing **b** seems to confirm both statements:

```
>>> b
[1, 2, 3]
```

So by changing the value of **a**, the value of **b** should also change:

```
>>> a = 5
>>> b
[1, 2, 3]
```

What happened here? If you know another programming language, you may think that “Python is storing the value instead of the reference to the value.” That is not exactly the case, so I urge you to keep on reading.

The following statements seem to work in a different way:

```
>>> c = [1, 2, 3]
>>> d = [5, 6, c]
```

¹⁴Available at <https://www.python.org/dev/peps/pep-0008>

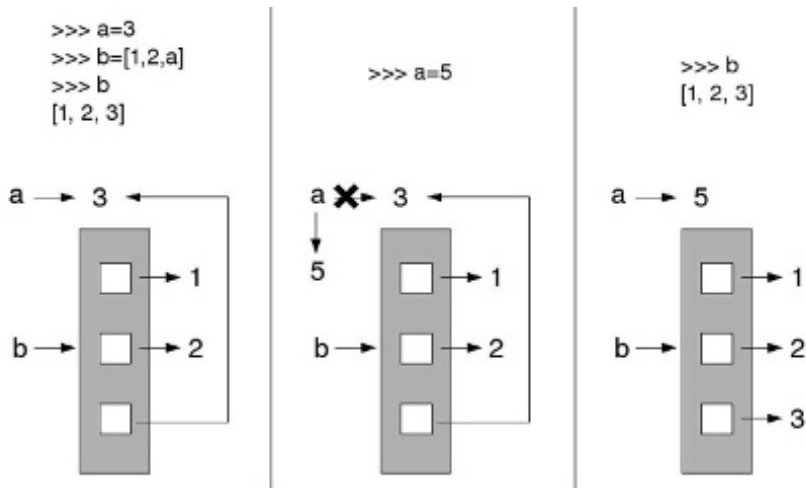


Figure 3.5 Case 1.

Translated into English, they mean: “Let the variable `c` be a list with three elements: 1, 2 and 3” and “Let the variable `d` be a list with three elements: 5, 6, and `c` (that is a list with three elements: 1, 2, and 3).”

This can be confirmed by printing both variables:

```
>>> c
[1, 2, 3]
>>> d
[5, 6, [1, 2, 3]]
```

Let’s change the value of `c` to see what happens with `d`:

```
>>> c.pop()
3
>>> c
[1, 2]
>>> d
[5, 6, [1, 2]]
```

In this case, changing one variable, does change the other variable. This seems like an inconsistent behavior. If we think of all these variable assignments as a binding names with objects, what seems inconsistent starts to make sense. Try following the next explanation using [Figure 3.5](#).

```
>>> a = 3
>>> b = [1, 2, a]
```

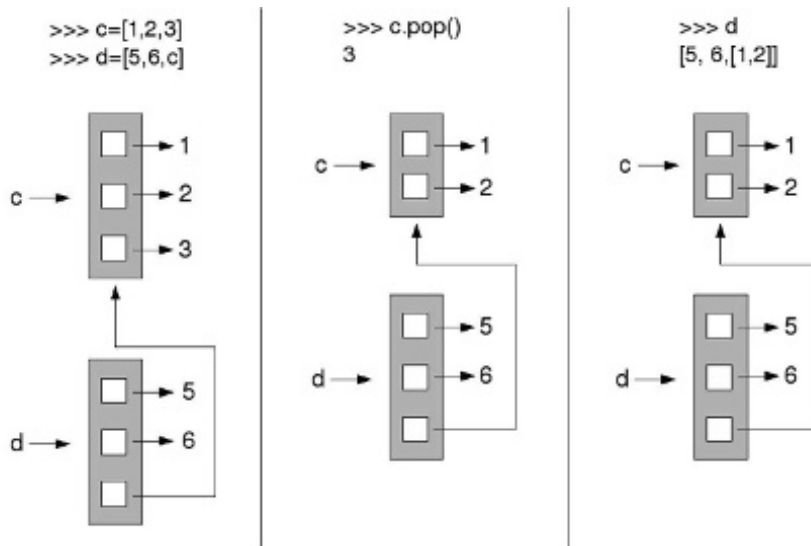



Figure 3.6 Case 2.

Translated into English, they mean: “Let the object 3 be called *a*” and “Let the list with three elements (1, 2 and *a*) be called *b*.”

Printing *b* seems to confirm both statements:

```
>>> b
[1, 2, 3]
```

Then we create a new object (5) and name it *a*. So the previous reference (*a*=3) is destroyed (this is represented by a cross in the arrow from *a* to 3). The name *a* is not bound to 3 anymore, now *a* is bound to 5. What about *b*?

```
>>> a = 5
>>> b
[1, 2, 3]
```

Since the third position in the list called *b* was not altered, *b* remains unmodified. We only changed the binding between *a* and 3.

The next sample case can also be explained by taking into account that there is no variable assignments in Python, but names that bind objects. In this case you should follow [Figure 3.6](#).

```
>>> c = [1, 2, 3]
>>> d = [5, 6, c]
```

Translated into English, they mean, “Let the list with three elements: 1, 2 and 3 be called *c*” and “Let the list with three elements: 5, 6 and *c* (which is the name of a

list of three elements: 1, 2 and 3) be called `d`.” This can be confirmed by requesting the contents or both names:

```
>>> c
[1, 2, 3]
>>> d
[5, 6, [1, 2, 3]]
```

In the next step, modify the list called `c` by removing the last element and see what happens with `d`:

```
>>> c.pop()
3
>>> print c
[1, 2]
>>> print d
[5, 6, [1, 2]]
```

This time, `c` was modified (and not just a relationship). Since the actual value of `c` was altered, this is reflected every time it is called. See [Figure 3.6](#) in case of doubt.

Even if names are bound to objects and there is no variable assignment in Python, force of habit is strong and most texts (even this book) use the terms *variables* and *names* interchangeably.

3.9 ADDITIONAL RESOURCES

- Learn to program using python: variables and identifiers.
<http://www.developer.com/lang/other/article.php/626321>
- Python 101—Introduction to Python.
<http://ascii-world.wikidot.com/python-101>
- Beginning Python for bioinformatics.
<http://www.onlamp.com/pub/a/python/2002/10/17/biopython.html>
- Text Processing Services.
<https://docs.python.org/3/library/text.html>
- Python 3 Unicode HOWTO.
<https://docs.python.org/3/howto/unicode.html>
- Adding a built-in set object type.
<http://www.python.org/dev/peps/pep-0218/>
- Python built-in types.
<https://docs.python.org/3/library/stdtypes.html>

- Revamping `dict.keys()`, `.values()`, and `.items()`.
<http://www.python.org/dev/peps/pep-3106/>
- Python dictionaries with recursive dot notation access.
<https://github.com/cdgriffith/Box>

3.10 SELF-EVALUATION

1. Which are the principal data types in Python?
2. What is the difference between a list and a tuple? When would you use each one?
3. What is a set and when would you use it?
4. How do you test if an element is inside a list?
5. What is a dictionary?
6. What data type can be used as a key in a dictionary?
7. What is a “dictionary view”?
8. Can you iterate over an unordered sequence?
9. Sort the data types below according to the following criteria:
 - Mutable–immutable
 - Sorted–unsorted
 - Sequence–mapping

Data types to sort: lists, tuples, dictionaries, sets, strings.

10. What is the difference between a set and a frozenset?
11. How do you convert any iterable data type into a list?
12. How do you create a dictionary from a list?
13. How do you create a list from a dictionary?

Programming: Flow Control

CONTENTS

4.1	If-Else	69
4.1.1	Pass Statement	74
4.2	For Loop	75
4.3	While Loop	77
4.4	Break: Breaking the Loop	78
4.5	Wrapping It Up	80
4.5.1	Estimate the Net Charge of a Protein	80
4.5.2	Search for a Low-Degeneration Zone	81
	First Version	81
	Version with While	82
	Version without List of Subchains	82
4.6	Additional Resources	83
4.7	Self-Evaluation	83

In order to be able to do something useful, programs must have some mechanism to manage how and when instructions are executed. In the same way that traffic lights control vehicular flow in a street, flow control structures direct that a code portion is executed at a given time.

Python has only three flow control structures. There is one conditional and two iteration structures. A conditional structure (**if**) determines, after an expression evaluation, whether a block of code is executed or not. Iteration structures (**for** and **while**) allow multiple executions of the same code portion. How many times is the code associated with an iteration structure executed? A **for** cycle executes a code block many times as elements are available in a specified iterable element, while the code under a **while** cycle is executed until a given condition turns false.¹

4.1 IF-ELSE

The most classic control structure is the conditional one. It acts upon the result of an evaluation. If you know any other computer language, chances are that you are familiar with **if-else**.

¹This is equivalent to saying that the condition is executed **while** the condition is true.

If evaluates an expression. If the expression is true, the block of code just after the **if** clause is executed. Otherwise, the block under **else** is executed.

A basic schema of an **if-else** condition,

```
if EXPRESSION:
    BLOCK1
else:
    BLOCK2
```

EXPRESSION must be an expression that returns **True** or **False**. Like all comparison operators: $x < y$ (less than), $x > y$ (greater than), $x == y$ (equal to), $x != y$ (not equal to), $x <= y$ (less than or equal to), $x >= y$ (greater than or equal to).

Let's see an example:

Listing 4.1: ifelse1.py: Basic if-else sample

```
1 height = float(input('What is height? (in meters): '))
2 if height > 1.40:
3     print('You can get in')
4 else:
5     print('This ride is not for you')
```

Program output,

```
What is height? (in meters): 1.80
You can get in
```

Tip: About the code in this book.

You don't have to type the code in [Listing 4.1](#) (or any other from this book). It is available to download from its GitHub repository at <https://github.com/Serulab/Py4Bio>. It also can be used online at Microsoft Azure Notebooks (<https://notebooks.azure.com/library/py3.us>). Both links are also available at the book's website (<http://py3.us/>).

Try to execute the code (either locally or online) rather than just read it from this book.

Another example,

Listing 4.2: ifelse2.py: if-else in action

```

1 three_letter_code = {'A':'Ala','N':'Asn','D':'Asp','C':'Cys'}
2 aa = input('Enter one letter: ')
3 if aa in three_letter_code:
4     print('The three letter code for {0} is {1}'.format(aa,
5         three_letter_code[aa]))
6 else:
7     print('Sorry, I don't have it in my dictionary')
```

Program output,

```

Enter one letter: A
The three letter code for A is: Ala
```

To evaluate more than one condition, use **elif**:

```

if EXPRESSION1:
    BLOCK1
elif EXPRESSION2:
    BLOCK2
elif EXPRESSION3:
    BLOCK3
else:
    BLOCK4
```

You can use as many **elif** as conditions you want to evaluate. Take into account that once a condition is evaluated as true, the remaining conditions are not checked.

The following program evaluates more than one condition using **elif**:

Listing 4.3: elif1.py: Using elif

```

1 dna = input('Enter your primer sequence: ')
2 seqsize = len(dna)
3 if seqsize < 10:
4     print('The primer must have at least ten nucleotides')
5 elif seqsize < 25:
6     print('This size is OK')
7 else:
8     print('The primer is too long')
```

This program (**elif1.py**) asks for a string with a DNA sequence entered with the keyboard at runtime. This sequence is called *dna*. In line 2 its size is calculated

and this result is bound to the name *seqsize*. In line 3 there is an evaluation. If *seqsize* is lower than ten, the message “The primer must have at least ten nucleotides” is printed. The program flows goes to the end of this **if** statement, without evaluating any other condition in this **if** statement. But if it is not true (for example, if the sequence length was 15), it would execute the next condition and its associated block in case that condition is evaluated as true. If the sequence length were of a value greater than 10, the program would skip line 4 (the block of code associated with the first condition) and would evaluate the expression in line 5. If this condition is met, it will print “This size is OK”. If there is no expression that evaluates as true, the **else** block is executed.

Tip: What Is True?

Remember that the statement after the *if condition* is executed only when the expression is evaluated as *True*. So the questions “What is True?” (and “What is False?”) are relevant.

What is *True*?:

- Nonempty data structures (lists, dictionaries, tuples, strings, sets). Empty data structures count as *False*.
- 0 and *None* count as *False* (while other values count as *True*).
- Keyword *True* is *True* and *False* is *False*.

If you have a doubt if an expression is *True* or *False*, use **bool()**:

```
>>> bool(1=='1')
False
```

Conditionals can be nested:

Listing 4.4: nested.py: Nested if

```
1 dna = raw_input('Enter your DNA sequence: ')
2 seqsize = len(dna)
3 if seqsize < 10:
4     print('Your primer must have at least ten nucleotides')
5     if seqsize == 0:
6         print('You must enter something!')
7 elif seqsize < 25:
8     print('This size is OK')
9 else:
10    print('Your primer is too long')
```

In line 5 there is a condition inside another.

Note the double equal sign (“==”) instead of the single equal. Double equal is used to compare values, while the equal sign is used to assign values:

```
>>> answer=42
>>> answer
42
>>> answer==3
False
>>> answer==42
True
```

The nested **if** introduced in [Listing 4.4](#), can be avoided:

Listing 4.5: elif2.py: Nested if

```
1 dna = raw_input('Enter your DNA sequence: ')
2 seqsize = len(dna)
3 if seqsize == 0:
4     print('You must enter something!')
5 elif 0 < seqsize < 10:
6     print('Your primer must have at least ten nucleotides')
7 elif seqsize < 25:
8     print('This size is OK')
9 else:
10    print('Your primer is too long')
```

See how the expression is evaluated in line 5. This leads us to think about inserting multiple statements in one **if**, like in [Listing 4.6](#):

Listing 4.6: multiplepart.py: Multiple part condition

```
1 x = 'N/A'
2 if x != 'N/A' and 5 < float(x) < 20:
3     print('OK')
4 else:
5     print('Not OK')
```

This expression is evaluated from left to right. If one part of the expression is false, the following parts are not evaluated. Since `x='N/A'`, the program will print 'Not OK' (because the first condition is false). Look what happens when the same expression is written in reverse order.

This listing (multiplepart2.py),

Listing 4.7: multiplepart2.py: Multiple part condition, inverted

```
1 x = 'N/A'
2 if 5 < float(x) < 20 and x != 'N/A':
3     print('OK')
4 else:
5     print('Not OK')
```

returns:

```
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    if 5 < float(x) < 20 and x != 'N/A':
ValueError: could not convert string to float: 'N/A'
```

The **ValueError** is returned because the string 'N/A' can't be converted to float. In [Listing 4.7](#), `x` is also evaluated as 'N/A', but there is no **ValueError** because this part of the expression is skipped before evaluation.

4.1.1 Pass Statement

Sometimes there is no need for an alternative choice in an **if statement**. In this case you just can avoid using **else**:

```
if EXPRESSION:
    BLOCK
# Rest of the program...
```

To make the same code more readable, Python provides the **pass** statement. This statement is like a placeholder; it has no other purpose than to put something when a statement is required syntactically. The following code produces the same output as the former code:

```
if CONDITION:
    BLOCK
else:
    pass
# Rest of the program...
```

Advanced Tip: Conditional Expressions.

Sometimes comes in handy: the availability of a special syntax to write an **if condition** in one line

```
expression1 if condition else expression2
```

This line will take the value of *expression1*, if *condition* is true; otherwise, it will take the value of *expression2*.

This syntax allows us to write:

```
>>> total = 5
>>> items = 2
>>> print('Average = {0}'.format(total/items if items != 0 else 'N/A'))
Average = 2.5
```

instead of,

```
>>> total = 5
>>> items = 2
>>> if items != 0:
...     print('Average = {0}'.format(total/items))
... else:
...     print('Average = N/A')
...
Average = 2.5
```

4.2 FOR LOOP

This control structure allows code to be repeatedly executed while keeping a variable with the value of an iterable object.² The generic form of a **for loop** is,

```
for VAR in ITERABLE:
    BLOCK
```

For example:

```
for each_item in some_list:
    # Do something with each_item
    print(each_item)
```

Note the colon at the end of the first line. This is mandatory. As the indentation of the block of code the colon is part of the **for loop**. This structure results in the repetition of *BLOCK* as many times as elements are in the iterable object. On each iteration, *VAR* takes the value of the current element in *ITERABLE*. In the following code, **for** walks through a list (**bases**) with four elements. On each iteration, *x* takes the value of one of the elements in the list.

```
>>> bases = ['C', 'T', 'G', 'A']
>>> for x in bases:
...     print(x)
...
```

²The most common iterable objects are lists, tuples, strings, and dictionaries. Files and custom-made objects can also be iterable.

```
C
T
G
A
```

To know the position on the iterable you are iterating, the method **enumerate** will return the index of the iterable along with the value.

```
>>> bases = ['C', 'T', 'G', 'A']
>>> for n, x in enumerate(bases):
...     print(n, x)
...
0 C
1 T
2 G
3 A
```

In other languages, the **for** loop is used to allow a block of code to run a number of times while changing a counter variable. This behavior can be reproduced in Python by iterating over a list of numbers:

```
>>> for n in [0, 1, 2, 3, 4]:
...     print(n)
...
0
1
2
3
4
```

Another alternative to iterate through a list of numbers, is to generate them with the built-in function³ **range(*n*)**. This function returns an iterable object. Which each time you call it, returns a number, from 0 to the first parameter entered in the function minus one (that is, *n*-1).

```
>>> for x in range(4):
...     print(x)
...
0
1
2
3
```

³All built-in functions are described in <https://docs.python.org/3.6/library/functions.html>.

The following code calculates the molecular weight of a protein based on its individual amino acids.⁴ Since the amino acid is stored in a string, the program will walk through each letter by using **for**.

Listing 4.8: `protwfor.py`: Using **for** to figure the weight of a protein

```

1 prot_seq = input('Enter your protein sequence: ')
2 prot_weight = {'A':89, 'V':117, 'L':131, 'I':131, 'P':115,
3               'F':165, 'W':204, 'M':149, 'G':75, 'S':105,
4               'C':121, 'T':119, 'Y':181, 'N':132, 'Q':146,
5               'D':133, 'E':147, 'K':146, 'R':174, 'H':155}
6 total_weight = 0
7 for aa in prot_seq:
8     total_weight = total_weight + prot_weight.get(aa.upper(), 0)
9 total_weight = total_weight - (18 * (len(prot_seq) - 1))
10 print('The net weight is: {0}'.format(total_weight))

```

Code explanation: On the first line the user is requested to enter a protein sequence (for example, MKTFVLHIFIFALVAF). The string returned by **input** is named **protseq**. From line 2 to 5, a dictionary (**protweight**) with the amino acid weights is initialized. A **for loop** is used in line 7 to iterate over each element in **protseq**. In each iteration, **aa** takes a value from an element from **protseq**. This value is used to search in the **protweight** dictionary. After the cycle, **totalW** will end up with the sum of the weight of all amino acids. In line 9 there is a correction due to the fact that each bond involves the loss of a water molecule (with molecular weight of 18). The last line prints out the net weight.

4.3 WHILE LOOP

A loop is very similar to **for** since it also executes a code portion in a repeated way. In this case there is no iteration over an object, so this loop doesn't end when the iteration object is traversed, but when a given condition is not true.

Model of **while loop**:

```

while EXPRESSION:
    BLOCK

```

It is very important to take into account that there should be an instruction inside the block to make the while condition false. Otherwise, we could enter into an infinite loop.

⁴Amino acids are the building blocks of the proteins. Each amino acid (represented by a single letter) has an individual weight. Since each amino acid bond releases a water molecule (with a weight of 18 u), the weight of all the water molecules released is subtracted from the total.

```
>>> a = 10
>>> while a < 40:
...     print(a)
...     a += 10
...
10
20
30
```

A way to exit from a **while** loop is using **break**. In this case the loop is broken without evaluating the loop condition. **break** is often used in conjunction with a condition that is always true:

```
>>> a = 10
>>> while True:
...     if a < 40:
...         print(a)
...     else:
...         break
...     a += 10
...
10
20
30
```

This is done to ensure the block inside the loop is executed at least once. In other languages there is a separate loop type for these cases (**do while**), but it is not present in Python.⁵

4.4 BREAK: BREAKING THE LOOP

break is used to escape from a loop structure. We've just seen a usage example with **while**, but it can also be used under **for**.

It is not easy at first to realize where using a break statement actually makes sense.

Take, for example, [Listing 4.9](#):

Listing 4.9: `seachinlist.py`: Searching a value in a list of tuples

```
1 color_code = [('red', 1), ('green', 2), ('blue', 3), ('black', 4)]
2 name = 'blue'
3 for color_pair in color_code:
4     if name == color_pair[0]:
```

⁵A proposal to add this structure to Python was rejected in 2013.

```

5         code = color_pair[1]
6 print(code)

```

In this code there is a **for loop** to iterate over `color_code` list. For each element, that is, for each tuple, it checks for the first element. When it matches our query (`name`), the program stores the associated code in `code`.

So the output of this program is “3.”

The problem with this program is that the whole sequence is walked over, even if we don’t need to. In this case, the condition in line 4 is evaluated once per each element in `color_code` when it is clear that once the match is positive there is no need to keep on testing. You can save some time and processing power by breaking the loop just after the positive match:

Listing 4.10: `searchinlist2.py`: Searching a value in a list of tuples

```

1 color_code = [('red',1), ('green',2), ('blue',3), ('black',4)]
2 name = 'blue'
3 for color_pair in color_code:
4     if name == color_pair[0]:
5         code = color_pair[1]
6         break
7 print(code)

```

This code is identical to [Listing 4.9](#) with the exception of the `break` statement in line 6. The output is the same as before, but this time you don’t waste CPU cycles iterating over a sequence once the element is found. The time saved in this example is negligible, but if the program has to do it several times over a big list or file (you can also iterate over a file), **break** can speed it up in a significant way.

The use of **break** can be avoided, but the resulting code is not legible as in [Listing 4.10](#):

Listing 4.11: `searchinlist3.py`: Searching a value in a list of tuples

```

1 color_code = [('red',1), ('green',2), ('blue',3), ('black',4)]
2 name = 'blue'
3 i = 0
4 while name != color_code[i][0]:
5     i += 1
6 code = color_code[i][1]
7 print(code)

```

In a case like this, with a list that can easily fit in memory, it is a better idea to create a dictionary and query it:

Listing 4.12: `seachindict.py`: Searching a value in a list of tuples using a dictionary

```
1 color_code = [('red',1), ('green',2), ('blue',3), ('black',4)]
2 name = 'blue'
3 color_code_d = dict(color_code)
4 print(color_code_d[name])
```

4.5 WRAPPING IT UP

Now we will combine **if**, **for**, **while** and the data type seen up to this point. Here I present some small programs made with the tools we've just learned:

4.5.1 Estimate the Net Charge of a Protein

At a fixed pH, it is possible to calculate the net charge of a protein summing the charge of its individual amino acids. This is an approximation since it doesn't take into account if the amino acids are exposed or buried in the protein structure. This program also fails to take into account the fact that cysteine adds charge only when it is not part of a disulfide bridge. Since it is an approximate value the obtained values should be regarded as an estimation. Here is the first version of `protnetcharge.py`:

Listing 4.13: `protnetcharge.py`: Net charge of a protein

```
1 prot_seq = input('Enter protein sequence: ').upper()
2 charge = -0.002
3 aa_charge = {'C':-.045,'D':-.999,'E':-.998,'H':.091,
4             'K':1,'R':1,'Y':-.001}
5 for aa in prot_seq:
6     if aa in aa_charge:
7         charge += aa_charge[aa]
8 print(charge)
```

The **if** statement in line 6 can be avoided with `get()`:

Listing 4.14: `protnetcharge2.py`: Net charge of a protein using `get`

```
1 prot_seq = input('Enter protein sequence: ').upper()
2 charge = -0.002
3 aa_charge = {'C':-.045,'D':-.999,'E':-.998,'H':.091,
4             'K':1,'R':1,'Y':-.001}
5 for aa in prot_seq:
```

```

7     charge += aa_charge.get('aa', 0)
8 print(charge)

```

4.5.2 Search for a Low-Degeneration Zone

To find PCR primers, it is better to use a DNA region with less degeneration (or more conservation). This give us a better chance to find the target sequence. The aim of this program is to search for this region. Since a PCR primer has about 16 nucleotides, to give room for the primer design, the search space should be at least 45 nucleotides long. We should find a 15 amino acid region in the input sequence. 15-amino acids provides a search region of 45 nucleotides (3 nucleotides per amino acid).

Each amino acid is encoded by a determined number of codons. For example, valine (V) can be encoded by four different codons (GTT, GTA, GTC, GTG), while tryptophan (W) is encoded only by one codon (TGG). Hence a region rich in valines will have more variability than a region with lots of tryptophan.

A program that finds a low-degeneration region:

First Version

Listing 4.15: lowdeg.py: Search for a low degeneration zone

```

1  prot_seq = input('Protein sequence: ').upper()
2  prot_deg = {'A':4, 'C':2, 'D':2, 'E':2, 'F':2, 'G':4,
3             'H':2, 'I':3, 'K':2, 'L':6, 'M':1, 'N':2,
4             'P':4, 'Q':2, 'R':6, 'S':6, 'T':4, 'V':4,
5             'W':1, 'Y':2}
6  segs_values = []
7  for aa in range(len(prot_seq)):
8      segment = prot_seq[aa:aa + 15]
9      degen = 0
10     if len(segment)==15:
11         for x in segment:
12             degen += prot_deg.get(x, 3.05)
13         segs_values.append(degen)
14 min_value = min(segs_values)
15 minpos = segs_values.index(min_value)
16 print(prot_seq[minpos:minpos + 15])

```

Code explanation: Takes a string (`prot_seq`) entered by the user. The program uses a dictionary (`prot_deg`) to store the NUMBER of codons that corresponds to each amino acid. From line 7 to 9, we generate sliding windows of length 15. For each 15 amino acid segments, the number of codons is evaluated, then we

select the segment with less degeneration (line 14). Note that in line 10 there is a check of the size of `segment`, since when the sequence of `prot_seq` slides away, the subchain has less than 15 amino acids.

Version with While

Listing 4.16: `lowdeg2.py`: Searching for a low-degeneration zone; version with `while`

```

1 prot_seq = input('Protein sequence: ').upper()
2 prot_deg = {'A':4, 'C':2, 'D':2, 'E':2, 'F':2, 'G':4,
3            'H':2, 'I':3, 'K':2, 'L':6, 'M':1, 'N':2,
4            'P':4, 'Q':2, 'R':6, 'S':6, 'T':4, 'V':4,
5            'W':1, 'Y':2}
6 segs_values = []
7 segs_seqs = []
8 segment = prot_seq[:15]
9 a = 0
10 while len(segment)==15:
11     degen = 0
12     for x in segment:
13         degen += prot_deg.get(x, 3.05)
14     segs_values.append(degen)
15     segs_seqs.append(segment)
16     a += 1
17     segment = prot_seq[a:a+15]
18 print(segs_seqs[segs_values.index(min(segs_values))])

```

Code explanation: This version doesn't use a `for` to walk over `prot_seq`; instead, it uses `while`. Code will be executed as long as the sliding window is inside `prot_seq`.

Version without List of Subchains

Listing 4.17: `lowdeg3.py`: Searching for a low-degeneration zone without subchains

```

1 prot_seq = input('Protein sequence: ').upper()
2 prot_deg = {'A':4, 'C':2, 'D':2, 'E':2, 'F':2, 'G':4,
3            'H':2, 'I':3, 'K':2, 'L':6, 'M':1, 'N':2,
4            'P':4, 'Q':2, 'R':6, 'S':6, 'T':4, 'V':4,
5            'W':1, 'Y':2}
6 degen_tmp = max(prot_deg.values()) * 15

```

```

7 for n in range(len(prot_seq) - 15):
8     degen = 0
9     for x in prot_seq[n:n + 15]:
10         degen += prot_deg.get(x, 3.05)
11     if degen <= degen_tmp:
12         degen_tmp = degen
13         seq = prot_seq[n:n + 15]
print(seq)

```

Code explanation: In this case every degeneration value is compared with the last one (line 10), and if the current value is lower, it is stored. Note that the first time a degeneration value is evaluated, there is no value to compare it with. This problem is sorted in line 6 where a maximum theoretical value is provided.

4.6 ADDITIONAL RESOURCES

- Python tutorial: More control flow tools.
<https://docs.python.org/3.6/tutorial/controlflow.html>
- Python programming: Flow control.
http://en.wikibooks.org/wiki/Python_Programming/Flow_control
- Python Basics: Understanding The Flow Control Statements.
<https://goo.gl/ss6uNh>
- Python in a Nutshell, Second Edition. By Alex Martelli. [Chapter 4](#).
Excerpt at <http://www.devshed.com/c/a/Python/The-Python-Language>
- Python break, continue and pass Statements.
http://www.tutorialspoint.com/python/python_loop_control.htm

4.7 SELF-EVALUATION

1. What is a control structure?
2. How many control structures does Python have? Name them.
3. When would you use **for** and when would you use **while**?
4. Some languages have a **do while** control structure. How can you get a similar function in Python?
5. Explain when you would use **pass** and when you would use *break*.
6. In line 6 of [Listing 4.16](#), the condition under the **while** can be changed from `len(ProtSeq[i : i + 15]) == 15` to `i < (len(ProtSeq) - 7)`. Why?

7. Make a program that outputs all possible IP addresses, that is, from 0.0.0.0 to 255.255.255.255.
8. Make a program to solve a linear equation with two variables. The equation must have this form:

$$a_1.x + a_2.y = a_3$$

$$b_1.x + b_2.y = b_3$$

The program must ask for a_1, a_2, a_3, b_1, b_2 , and b_3 and return the value of x and y .

9. Make a program to check if a given number is a palindrome (that is, it remains the same when its digits are reversed, like 404).
10. Make a program to convert Fahrenheit temperature to Celsius and write the result with only one decimal value. Use this formula to make the conversion:
 $T_c = (5/9) * (T_f - 32)$
11. Make a program that converts everything you type into Leetspeak, using the following equivalence: 0 for O, 1 for I (or L), 2 for Z (or R), 3 for E, 4 for A, 5 for S, 6 for G (or B), 7 for T (or L), 8 for B, and 9 for P (or G and Q). So “Hello world!” is rendered as “H3770 w02ld!”
12. Given two words, the program must determine if they rhyme or not. For this question “rhyme” means that the last three letters are the same, like wiz**ard** and liz**ard**.
13. Given a protein sequence in the one-letter code, calculate the percentage of methionine (M) and cysteine (C). For example, from MFKFASAVILCLVAASSTQA the result must be 10% (1 M and 1 C over 20 amino acids).
14. Make a program like [Listing 4.17](#) but without using a predefined maximum value.

Handling Files

CONTENTS

5.1	Reading Files	86
5.1.1	Example of File Handling	87
5.2	Writing Files	89
5.2.1	File Reading and Writing Examples	90
5.3	CSV Files	90
	More Functions from the CSV Module	92
5.4	Pickle: Storing and Retrieving the Contents of Variables	94
5.5	JSON Files	96
5.6	File Handling: os, os.path, shutil, and path.py Module	98
5.6.1	path.py Module	100
5.6.2	Consolidate Multiple DNA Sequences into One FASTA File ...	102
5.7	Additional Resources	102
5.8	Self-Evaluation	103

Reading and saving files are an important part of most programs. This chapter shows how to read and write any text file. For the purposes of this book, “reading a text file” is the process of entering the data from a file into a program. The process of determining the syntactic structure of an expression to retrieve a specific part for further analysis is called **parsing**.

Take, for example, a file like this:

```
1867864656,1,BOT,[T/C],0009803928,Homo sapiens
1867864658,10,BOT,[A/T],0021792978,Homo sapiens
1867864660,100,BOT,[A/G],0069608915,Homo sapiens
```

On each line, there are different data units delimited by commas (often called data points). When the file is read, Python will recognize each line as one string, so there is the need of an extra step to recognize each of the six data points on it. This step is **parsing**. The parsing step depends on the format of the data, so there is no universal method for text parsing. This chapter shows on page 90 how to parse data separated by a special character such as a comma, a semicolon or the tab character (commonly called CSV files). There are other suitable formats for data interchange such as JSON and XML, and both will be covered.

5.1 READING FILES

Reading a file is a three-step process in Python:

1. Open the file: There is a built-in function called **open** that creates a *filehandle*. This *filehandle* is used to refer to the file during the file's lifetime. The **open** function takes two parameters: name of the file and opening mode. The file name is a string with the file name, in most cases including the system path. When the system path is included, this **absolute path** is used by the program. In case you enter just the file name (without any path), a relative path is assumed.¹ The second parameter has the following valid parameters: "r" to **read**, "w" to **write**, and "a" to **append** data at the end of a file. The default value is "r." If you want to open a file for both read and write, use "r+".

Using **open** create a file handle to read a file:

```
>>> file_handle = open('readme.txt', 'r')
```

As you can see here, **file_handle** is **not the file, but a reference to it**:

```
>>> file_handle
<_io.TextIOWrapper name='readme.txt' mode='r' encoding='UTF-8'>
```

2. Read the file: Once the file is opened, we can read its contents. The file handle has several methods to read a file; here are the most used:

read(n) :Reads *n* bytes from the file. Without parameters, it reads the whole file.²

readline() :Returns a string with only one line from the file, including '\n' as an end of line marker. When it reaches the end of the file, it returns an empty string.

readlines() :Returns a list where each element is a string with a line from the file.

Reading a file called **seqA.fas** with *read()*:

```
>>> file_handle = open('seqA.fas', 'r')
>>> file_handle.read()
>000626|HUMAN Small inducible cytokine A22.
```

¹Use **os.getcwd()** in case you need to know the current path.

²Due to the amount of memory it could take, it is not advisable to read the whole file in this way, unless you are sure of the file size. To process big files, there are better strategies like reading one line at a time.

```
MARLQTALLVVLVLLAVALQATEAGPYGANMEDSVCCRDYVRYRLPLRVVKHIFYWTSDS<=
CPRPGVLLTFRDKEICADPR
VPWVKMILNKLSQ
```

3. Close the file. Once we are done with the file, we close it by using: `filehandle.close()`. If we don't close it, Python will close it after program execution. However in most cases is better to close the file as soon as it is not needed because the number of open files is a limited resource. A way to ensure the file will be closed is to use **with**. Instead of:

```
file_handle = open('readme.txt', 'r')
# do something with the file
file_handle.read()
file_handle.close()
```

Do this:

```
with open('readme.txt', 'r') as file_handle:
    # do something with the file
    file_handle.read()
# from here on, the file is closed
```

5.1.1 Example of File Handling

Let's suppose we have a file called `seqA.fas` that contains:³

```
>000626|HUMAN Small inducible cytokine A22.
MARLQTALLVVLVLLAVALQATEAGPYGANMEDSVCCRDYVRYRLPLRVVKHIFYWTSDS<=
CPRPGVLLTFRDKEICADPR
VPWVKMILNKLSQ
```

From this file we need the name and the sequence. A first approach is to read the file with `read()`:

```
>>> with open('seqA.fas', 'r') as file_handle:
...     file_handle.read()
...
'>000626|HUMAN Small inducible cytokineA22.\nMARLQTALLVVLVL<=
LAVALQATEAGPYGANMEDSVCCRDYVRYRLPLRVVKHIFYWTSDCPRPGVLLTFRDK<=
EICADPR\nVPWVKMILNKLSQ\n'
```

³This is a FASTA file with one entry. The first line have a > followed by sequence name and description. The following lines has the sequence (DNA or amino acids). For more information on FASTA files, please see <http://www.ncbi.nlm.nih.gov/BLAST/fasta.shtml>.

In this case my goal is to have two variables, one with the sequence name and the other with the sequence itself. In [Listing 5.1](#) we can see a way to do it using `read()`:

Listing 5.1: `firstread.py`: First try to read a FASTA file

```
1 with open('seqA.fas') as fh:
2     my_file = fh.read()
3 name = my_file.split('\n')[0][1:]
4 sequence = ''.join(my_file.split('\n')[1:])
5 print('The name is : {0}'.format(name))
6 print('The sequence is: {0}'.format(sequence))
```

The first line opens the file in read mode and creates a file handle that we call `fh`. On line two, the whole file is read with `read()` and the resulting string is stored in system memory with the name `my_file`. The next step is to separate the names from the sequences. Since the name is after the “>” symbol and before the ‘\n’, this information can be used to get the data we want (line 3). The sequence is obtained by joining the elements resulting from splitting the `my_file` string, but without the first element.

The problem with this code is that it uses the `read()` function to read all the file at once. This is a potential problem if there is not enough memory available to accommodate the file’s contents. This is why it is better to use `readline()` to loop through the lines in a file object.

Listing 5.2: `fastaRead.py`: Reads FASTA file, sequentially

```
1 sequence = ''
2 with open('seqA.fas') as fh:
3     name = fh.readline()[1:-1]
4     for line in fh:
5         sequence += line.replace('\n', '')
6 print('The name is : {0}'.format(name))
7 print('The sequence is: {0}'.format(sequence))
```

Code explanation: This program adds to our protein net charge calculation program ([Listing 4.14](#)) the ability to use as input data, a FASTA format sequence, instead of entering it manually. In line 3 we grab the first line with `readline()` to retrieve the sequence name (*O00626/HUMAN Small inducible cytokineA22.*) and call it `name`. The formula **for *x* in *filehandle*** (line 4) is the most efficient way to iterate through all the lines of a file.

Listing 5.3: `netchargefile.py`: Calculate the net charge, reading the input from a file

```

1 sequence = ''
2 charge = -0.002
3 aa_charge = {'C':-.045, 'D':-.999, 'E':-.998, 'H':.091,
4              'K':1, 'R':1, 'Y':-.001}
5 with open('prot.fas') as fh:
6     fh.readline()
7     for line in fh:
8         sequence += line[:-1].upper()
9 for aa in sequence:
10     charge += aa_charge.get(aa,0)
11 print(charge)

```

Code explanation: The code is essentially the same as that in [Listing 4.14](#), with the difference in how the protein data is read; instead of using **input**, the data is read from a file (lines 5 to 8), such as in [Listing 5.2](#). Note that in line 6 we read the first line and the returned value is not stored, because it is the header and not needed for net charge estimation. When the program starts iterating the file from line 7, it starts from the second line.

5.2 WRITING FILES

Writing a file is very similar to reading it. Steps 1 and 3 are similar. The change is at the second state. Let's have a look at the entire process anyway:

1. Open the file. This is similar to opening a file for reading, only it is necessary to take into consideration the use of the open mode that corresponds to the operation that we are going to do. To create a new file, use "w" as the open mode. To append data to the end of the file, use "a."

Creating a file handle for a new file:

```
>>> fh = open('newfile.txt','w')
```

Creating a new file handle to append information to a file:

```
>>> fh = open('error.log','a')
```

2. Write data to the file. The method to write data to a file is called **write()**. It accepts as a parameter a string, which will be written to the file represented by the file handle on which the function will be applied. Schematically: *filehandle.write(string)*. Take into consideration that **write** does not add line feeds, which must be added as needed.
3. Close the file in the same way as done previously: **filehandle.close()**. As with reading files, you can use **with** to open a file for writing and close it in an implicit but safe way. See [Listing 5.4](#).

5.2.1 File Reading and Writing Examples

The code that follows will save the numbers from 1 to 5 to a file, each one on a separate line. Between each number the respective line feeds are indicated.

Listing 5.4: `Newfile.py`: Write numbers to a file.

```
1 with open('numbers.txt','w') as fh:
2     fh.write('1\n2\n3\n4\n5')
```

The program in [Listing 5.3](#) can be modified to write the result to a file, instead of displaying it on the screen:

Listing 5.5: `nettofile.py` Net charge calculation, saving results in a file

```
1 sequence = ''
2 charge = -0.002
3 aa_charge = {'C':-.045, 'D':-.999, 'E':-.998, 'H':.091,
4             'K':1, 'R':1, 'Y':-.001}
5 with open('prot.fas') as fh:
6     next(fh)
7     for line in fh:
8         sequence += line[:-1].upper()
9 for aa in sequence:
10     charge += aa_charge.get(aa, 0)
12 with open('out.txt','w') as file_out:
13     file_out.write(str(charge))
```

Code explanation: The code is similar to [Listing 5.3](#), with the addition of the functionality on the two final lines (12 and 13) to write the result to the file.

5.3 CSV FILES

While doing data processing work, it's very common to run into a file type called CSV. CSV stands for “**C**omma **S**eparated **V**alues.” These are files where the data are separated by commas, although sometimes other separators are used (such as colons, tabs, etc.). Another feature of this text file format in particular is that each line represents a separate record. All spreadsheets can be read and written in this file format, which helps to explain their popularity. Take, for example, the following file (`B1.csv`):

```
MarkerID,LenAmp,MotifAmpForSeq
TK0001,119,AG(12)
TK0002,255,TC(16)
```

```
TK0003,121,AG(5)
TK0004,220,AG(9)
TK0005,238,TC(17)
```

The line contains a description of each field. Like the information it stores, the descriptions are also separated by commas. The following lines contain the data, following the same order of the description. To get the average of the value in the second column, we can do something like this:

Listing 5.6: `csvwocsv.py`: Reading data from a CSV file

```
1 total_len = 0
2 with open('B1.csv') as fh:
3     next(fh)
4     for n, line in enumerate(fh):
5         data = line.split(',')
6         total_len += int(data[1])
7 print(total_len/n)
```

Code explanation: This is a program that walks through a file, like [Listing 5.5](#), but this time the method `split()` is used to split components of each line. In line 6 the sum of the second field (`LenAmp`) is stored (this field has the length of the sequence).

These files are so popular that Python has a module to deal with them: `csv`.

Listing 5.7: `csv1.py`: Reading data from a CSV file, using `csv` module

```
1 import csv
2 total_len=0
3 lines = csv.reader(open('B1.csv'))
4 next(lines)
5 for n, line in enumerate(lines):
6     total_len += int(line[1])
7 print(total_len / n)
```

Code explanation: This program is very similar to the previous one with the difference being that the use of the `csv` module allows us access to the contents of each line without having to use the `split` method.

One way of using the `csv` module is to convert the object returned by the reader method to a list. Doing this, we generate something similar to a matrix from a csv file, with one line of code:

```
>>> data = list(csv.reader(open('B1.csv')))
>>> data[0][2]
```

```
'MotifAmpForSeq'
>>> data[1][1]
'119'
>>> data[1][2]
'AG(12)'
>>> data[3][0]
'TK0003'
```

This way we have a two-dimensional array of the type *name[row, column]*. Taking this into consideration we can rewrite the program from [Listing 5.7](#):

More Functions from the CSV Module

The field delimiter is changed with the **delimiter** attribute. By default it is “,”, but any string can be used to delimit the fields:

```
rows = csv.reader(open('/etc/passwd'), delimiter=':')
```

For some files it is better to specify the CSV “dialect” that we are interested in. This is important because not all csv files have the same structure. CSV is not a formal standard, so each vendor may introduce some variation. These subtle differences that may spoil our data processing. In some cases the data is enclosed between quotations, in others the quotations are reserved for text data only. For the csv files generated by Excel, we have the Excel “dialect”:

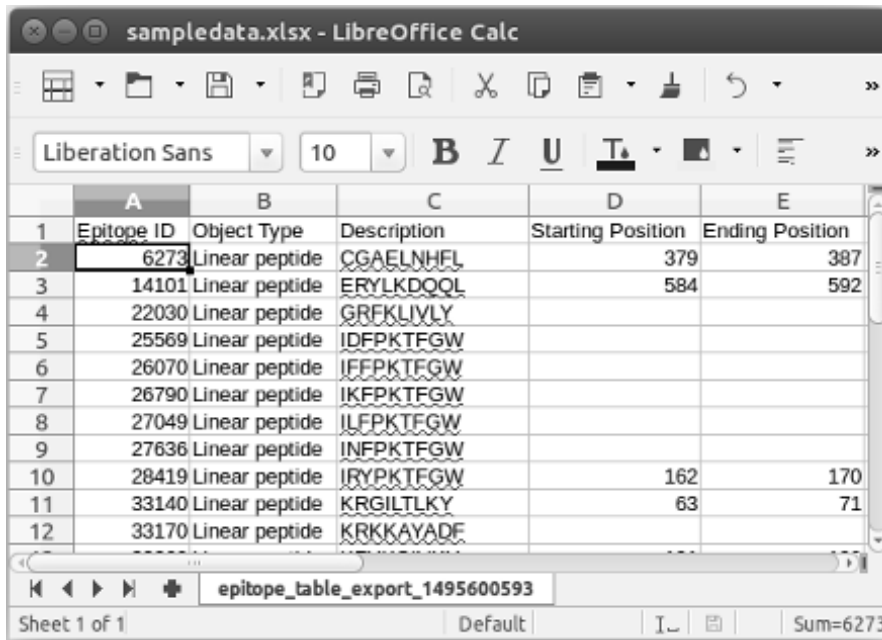
```
rows = csv.reader(open('data.csv'), dialect='excel')
```

Additionally there is a dialect for Excel csv files that uses a “tab” instead of the comma to separate data. If we aren’t sure of the dialect that our code will have to handle, the **csv** module has a class that tries to guess it: **Sniffer()**:

```
dialect = csv.Sniffer().sniff(open('data.csv').read())
rows = csv.reader(open('data.csv'), dialect=dialect)
```

There are more methods available in the **csv** module. To find out more about it, I recommend the module documentation at <https://docs.python.org/3/library/csv.html> and the PEP-305 (<https://www.python.org/dev/peps/pep-0305/>), an old but still valid document.

CSV files are very handy, but they can’t represent hierarchical data so other formats are used to store data, such as JSON and XML. Both are covered in this book.



	A	B	C	D	E
1	Epitope ID	Object Type	Description	Starting Position	Ending Position
2	6273	Linear peptide	CGAELNHFL	379	387
3	14101	Linear peptide	ERYLKDQQL	584	592
4	22030	Linear peptide	GRFKLIVLY		
5	25569	Linear peptide	IDFPKTFGW		
6	26070	Linear peptide	IFFPKTFGW		
7	26790	Linear peptide	IKFPKTFGW		
8	27049	Linear peptide	ILFPKTFGW		
9	27636	Linear peptide	INFPKTFGW		
10	28419	Linear peptide	IRYPKTFGW	162	170
11	33140	Linear peptide	KRGILTLYK	63	71
12	33170	Linear peptide	KRKKAYADF		

Figure 5.1 Excel formatted spreadsheet called `sampledata.xlsx`.

Tip: Reading and Writing Excel Files.

The `csv` module allows you to read Excel files, provided that the file is converted first to `csv`. This step can be avoided with a third-party module called **xlrd**. This module can be installed with **pip** or **conda** (see page 117 for detailed information on how to install external packages).

[Listing 5.8](#) retrieves data from an Excel file called `sampledata.xlsx` (see Figure 5.3). We want to make a dictionary (`iedb`) out of column A (keys) and Column C (values), so this program walks over both columns and completes the dictionary:

Listing 5.8: `excel1.py`: Reading an `xlsx` file with `xlrd`

```

1 import xlrd
2 iedb = {}
3 book = xlrd.open_workbook('../samples/sampledata.xlsx')
4 sh = book.sheet_by_index(0)
5 for row_index in range(1, sh.nrows): #skips first line.
6     iedb[int(sh.cell_value(rowx=row_index, colx=0))] = \
7         sh.cell_value(rowx=row_index, colx=2)
8 print(iedb)
```

`excel1.py` returns a dictionary like this:

```
{6273: 'CGAELNHFL', 14101: 'ERYLKDQQL', 22030: 'GRFKLIVLY', <=
25569: 'IDFPKTFGW', 26070: 'IFFPKTFGW', 26790: 'IKFPKTFGW', <=
27049: 'ILFPKTFGW', 27636: 'INFPKTFGW', 28419: 'IRYPKTFGW', <=
33140: 'KRGILTLKY', 33170: 'KRKKAYADF'}
```

Note that this is a sample output, the actual output is larger but was cut for brevity. Compare this output with Figure 5.3.

To write Excel files, you can use **xlwt**, which works in a similar fashion to **xlrd**. Listing 5.9 writes `list1` and `list2` in column A and B using **xlwt**.

Listing 5.9: `excel2.py`: Write an XLS file with **xlwt**

```
1 import xlwt
2 list1 = [1,2,3,4,5]
3 list2 = [234,267,281,301,331]
4 wb = xlwt.Workbook()
5 ws = wb.add_sheet('First sheet')
6 ws.write(0,0,'Column A')
7 ws.write(0,1,'Column B')
8 i = 1
9 for x,y in zip(list1,list2): #Walk two list at the same time.
10     ws.write(i,0,x) # Row, Column, Data.
11     ws.write(i,1,y)
12     i += 1
13 wb.save('mynewfile.xls')
```

For sample usage of **pyExceerator**, see Listing 18.2 on page 336.

5.4 PICKLE: STORING AND RETRIEVING THE CONTENTS OF VARIABLES

All variables created during the lifetime of a program are temporarily stored in memory and disappear when the program terminates. Python provides a module to store and retrieve from disk or any other media the contents of these variables: The **Pickle**⁴ module. **pickle** serializes any Python data structure into a byte stream. This byte stream can be saved to disk or sent over the network. At the other end, **pickle** can transform the byte stream into an object with the original internal structure. The following script generates a dictionary (`sp_dict`) and saves it into a file (named `spdict.data`) so it is available from another program.

⁴Pickle has other features than those described in this book; in order to have a more extensive view of what **pickle** has to offer, see <https://docs.python.org/3/library/pickle.html#module-pickle>.

Listing 5.10: `picklesample.py`: Basic pickle sample

```

1 import pickle
2 sp_dict = {'one':'uno', 'two':'dos', 'three':'tres'}
3 with open('spdict.data', 'wb') as fh:
4     pickle.dump(sp_dict, fh)

```

With **`pickle.dump()`**, the dictionary `sp_dict` is saved to the file referenced by the file handle (`fh`). **`pickle.dump()`** accepts four parameters. The first parameter is the object you want to store. The second is the file-like object where you want to store your object. A third argument (not used in the example) is the **protocol**, is a integer number that represents the way in which the information will be encoded. When no protocol is specified (as in this case), it defaults to **3** which is a binary backward-incompatible protocol designed for Python 3. For more information on pickle protocols, see Infobox *Protocols for Pickle*. The fourth parameter (**`fix_imports`**) when set to **True** and protocol less than 3 is used for getting backward compatibility for Python 2.

Data stored in `spdict.data` can be retrieved with **`pickle.load()`**:

```

>>> import pickle
>>> pickle.load(open('spdict.data','rb'))
{'one':'uno', 'two':'dos', 'three':'tres'}

```

The **`load`** method requires the file handle of the object we want to pick up.

Note that in both cases (**`dump`** and **`load`**) I am using **b** (for binary) for opening the file because the default protocol that I am using writes a binary file. If you don't open the file as binary, Python will try to convert it to Unicode and will fail.

Protocols for Pickle

Here is a list of five different protocols that can be used for pickling. The higher the protocol used, the more recent the version of Python needed to read the produced pickle.

1. 0 is the original “human-readable” protocol and is backward compatible with earlier versions of Python.
2. 1 is an old binary format that is also compatible with earlier versions of Python.
3. 2 provides much more efficient pickling of new-style classes.
4. 3 is the default protocol in Python 3. It has explicit support for byte objects and cannot be unpickled by Python 2.x. This is the recommended protocol when compatibility with other Python 3 versions is required.

5. 4 from Python 3.4. It adds support for very large objects, pickling more kinds of objects, and some data format optimizations⁵.

Warning Never unpickle data received from an untrusted source because the pickle module is not secure against erroneous or maliciously constructed data.

5.5 JSON FILES

JSON (JavaScript Object Notation) is a human-readable data interchange format inspired by JavaScript that is widely used in web related applications. You can convert a dictionary, a list, or almost any kind of data into a **JSON** object and then store or transmit this object over the network. Not all objects can be converted to **JSON**, since some objects are Python specific and not available in other computer languages. For example you can't convert a **set** into a **JSON**. Why would you use a less capable serializer like **JSON** when you have **pickle**? Because you need to share data and want this data to be available for any computer language. If you share a **pickle** object you are forcing the receiver to use Python. So why not use **CVS** which is also very popular? **CVS** is good for columnar data, but not for nested and dictionary-like data. To share data that holds some complex relationship and want it to be available to any programming language, it is better to use **JSON**.

This is an example of a **JSON** file:

```
{
  "contactPoint":{
    "fn":"PREUSCH, PETER\u00a0",
    "hasEmail":"mailto:preuschp@nigms.nih.gov"
  },
  "description":"<p>The Protein Data Bank (PDB) archive is the
single worldwide repository of information about the 3D
structures of large biological molecules, including proteins
and nucleic acids found in all organisms</p>\n",
  "identifier":"d9f3932a-9c55-41b3-ad3a-0b4e18ee4752",
  "keyword":[
    "national-institutes-of-health-nih"
  ],
  "language":[
    "en"
  ],
  "license":"http://opendefinition.org/licenses/odc-odbl/",
  "modified":"2016-07-18",
  "programCode":[
    "009:000"
```

⁵Refer to PEP 3154 for information about improvements brought by protocol 4.

```

],
"publisher":{
    "@type":"org:Organization",
    "name":"National Institutes of Health (NIH)"
},
"title":"Protein Data Bank (PDB)"
}

```

JSON shares the same interface as **pickle**, that is, you can read and write in a similar way.

```

>>> import json
>>> sp_dict = {'one':'uno', 'two':'dos', 'three':'tres'}
>>> with open('spdict.json', 'w') as fh:
...     json.dump(sp_dict, fh)

```

The saved file looks like this:

```
{'three': 'tres', 'one': 'uno', 'two': 'dos'}
```

To retrieve the dictionary out of a JSON file:

```

>>> import json
>>> with open('spdict.json') as fh:
...     sp_dict = json.load(fh)

```

After running the previous code, the dictionary *sp_dict* has the original content:

```

>>> sp_d
{'three': 'tres', 'one': 'uno', 'two': 'dos'}

```

Note that **JSON** can't serialize sets and other specific Python objects:

```

>>> json.dump({1,2,3,4,3,2,7}, open('test.json','wb'))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "python3.5/json/__init__.py", line 178, in dump
    for chunk in iterable:
  File "python3.5/json/encoder.py", line 436, in _iterencode
    o = _default(o)
  File "python3.5/json/encoder.py", line 180, in default
    raise TypeError(repr(o) + " is not JSON serializable")
TypeError: {1, 2, 3, 4, 7} is not JSON serializable

```

Here is a list of serializable objects: int, float, str, list, dict, True, False, and None.

5.6 FILE HANDLING: OS, OS.PATH, SHUTIL, AND PATH.PY MODULE

There are more actions with files besides reading and writing. Copy, move, delete, list, change directory, set file properties, and others can be done with **os**, **shutil**, and **path.py** modules.

The **os** module handles an interface with the Operating System. Let's see some important methods provided by this module:

getcwd(): Return a string representing the current working directory.

```
>>> import os
>>> os.getcwd()
'/home/sb'
```

chdir(path): Change the current working directory to a given path.

```
>>> os.getcwd()
'/home/sb'
>>> os.chdir('docs')
>>> os.getcwd()
'/home/sb/docs'
>>> os.chdir('..')
>>> os.getcwd()
'/home/sb'
```

listdir(dir): Return a list containing the names of the entries in the directory. To know if a name returned from **listdir** is a file or a directory, use either **os.path.isdir()** or **os.path.isfile()**.

```
>>> os.listdir('/home/sb/bioinfo/seqs')
['readme.txt', 'ms115.ab1', '.atom', 'projects', '.bash_history']
```

path.isfile(string) and **path.isdir(string)**: Check if the string passed as an argument is a file or a directory. Returns True or False.

```
>>> os.path.isfile('/home/sb')
False
>>> os.path.isdir('/home/sb')
True
```

remove(file): Remove a file. The file should exist and you should have write permission on it.

```
>>> os.remove('/home/sb/bioinfo/seqs/ms115.ab1')
```

rename(source, destination): Rename the file or directory *source* to *destination*.

```
>>> os.rename('/home/sb/seqs/readme.txt', '/home/sb/Readme')
```

mkdir(*path*): Create a directory named *path*.

```
>>> os.mkdir('/home/sb/processed-seqs')
```

Inside **os** module resides the **path** module. It contains methods related with path manipulation.

path.join(*directory1, directory2, ...*): Join two or more path name components, inserting the operating system path separator as needed. In Windows it will add "\", while in Linux and macOS it will insert "/". **path.join** will not check if the created path is valid.

```
>>> os.path.join(os.getcwd(), 'images')
'/home/images'
```

path.exists(*path*): Checks if a given *path* exists.

```
>>> os.path.exists(os.path.join(os.getcwd(), 'images'))
False
```

path.split(*path*): Returns a tuple splitting the file or directory name at the end and the rest of the path.

```
>>> os.path.split('/home/sb/seqs/ms2333.ab1')
('/home/sb/seqs', 'ms2333.ab1')
```

path.splitext(*path*): Splits out the extension of a file. It returns a tuple with the dotted extension and the original parameter up to the dot.

```
>>> os.path.splitext('/home/sb/seqs/ms2333.ab1')
('/home/sb/seqs/ms2333', '.ab1')
```

Other file-related operations like copying and removal can be found in the **shutil** module:

The most important functions are *copy*, *copy2*, and *copytree*.

copy(source, destination): Copy the file *source* to *destination*.

copy2(source, destination): Copies also the last access time and last modification (like the Unix command `cp -p`).

copytree(source, destination): Recursively copy an entire directory tree from the *source* directory to a destination directory that must not already exist.

For more information on **shutil**, see the documentation on <http://docs.python.org/lib/module-shutil.html> (or with `help(shutil)` on the Python shell).

5.6.1 path.py Module

There is an external module called **path.py** that acts as a wrapper for **os.path**. This module allows you to do most of same tasks as all other modules but with an easier to use programming interface. Since is an external module, it is not available with the regular Python installation (unless you have a Python distro like Anaconda that comes with this and other external modules). If you need to install **path.py**, use pip:

```
# pip install path.py
Collecting path.py
  Using cached path.py-9.0-py2.py3-none-any.whl
Installing collected packages: path.py
Successfully installed path.py
>>> import path
>>>
```

If there is no import error, **path.py** was installed successfully. For more information on installing third-party modules, please see page 117.

Here are some things that can be done with **path.py**. These examples assume the following directory structure:

```
/home
-- /sb
   -- xx.py
   -- /py4bio
      -- ch1.pdf
      -- ch1.tex
      -- ch2.pdf
      -- ch2.tex
      -- /imgs
         -- fig1.png
         -- fig2.png
```

- Create a new file: Create a Path object and call the **touch** method. This will generate a new file if the string you used to generate the Path object doesn't correspond with any file.

```
>>> from path import Path
>>> f = Path('/home/sb/newfile.text')
>>> f.touch()
```

- Check if a path object is a file or a directory (**isfile()**):

```
>>> f.isfile()
True
```

- Get the name, extension, and parent directory (**ext**, **name** and **parent**):

```
>>> f = Path('/home/sb/xx.py')
>>> f.ext
'.py'
>>> f.name
Path('xx.py')
>>> f.parent
Path('/home/sb')
>>> f.parent.parent
Path('/home')
```

- Get all files and directories in a directory (**files()** and **dirs()**):

```
>>> d = Path('/home/sb/py4bio')
>>> d.files()
[Path('/home/sb/py4bio/ch1.tex'), Path('/home/sb/py4bio/ch2.pdf'),
Path('/home/sb/py4bio/ch2.tex'), Path('/home/sb/py4bio/ch1.pdf')]
>>> d.dirs()
[Path('/home/sb/py4bio/imgs')]
```

You can also apply filters:

```
>>> d.files('*.pdf')
[Path('/home/sb/py4bio/ch2.pdf'), Path('/home/sb/py4bio/ch1.pdf')]
```

- Walk over all files in a directory (including files inside directories found in the parent directory), using **walk()**:

```
d = Path('/home/sb/py4bio')
for f in d.walk():
    if f.isfile():
        print(f)
```

which will return:

```

/home/sb/py4bio/ch1.tex
/home/sb/py4bio/imgs/fig1.png
/home/sb/py4bio/imgs/fig3.png
/home/sb/py4bio/imgs/fig4.png
/home/sb/py4bio/imgs/fig2.png
/home/sb/py4bio/ch2.pdf
/home/sb/py4bio/ch2.tex
/home/sb/py4bio/ch1.pdf

```

5.6.2 Consolidate Multiple DNA Sequences into One FASTA File

The following program assumes that we have a directory (`bioinfo/seqs/`) with several DNA sequences in FASTA format and we want to consolidate them in a single FASTA file called `outfile.fasta`. This file can be used, for example, as an input file for a BLAST run.

Listing 5.11: `consolidate.py`: Consolidating several files in one

```

1 from path import Path
2 d = Path('bioinfo/seqs/')
3 with open('outfile.fasta', 'w') as f_out:
4     for file_name in d.walk('*.fasta'):
5         with open(file_name) as f_in:
6             data = f_in.read()
7             f_out.write(data)

```

Code explanation: The program defines a `Path` with the directory `bioinfo/seqs/`. In line 3 we open a file (`outfile.fasta`) to save the contents of all the sequences. From line 4 we start walking over every file matching the `*.fasta` pattern. For each file we open it, read the content (line 6) and write it to `outfile.fasta` (line 7). There is no need to close any open file because they are inside a **with** statement.

5.7 ADDITIONAL RESOURCES

- File and directory access.
<https://docs.python.org/3/library/filesys.html>
- Generate temporary files and directories.
<https://docs.python.org/3/library/tempfile.html>
- CSV file API.
<http://www.python.org/dev/peps/pep-0305/>

- Tad, a program to view and analyze CSV data.
<http://tadviewer.com>
- Working with Excel files in Python.
<http://www.python-excel.org>
- The “with” statement.
<http://www.python.org/dev/peps/pep-0343/>
- JSON formatter.
<https://jsonformatter.curiousconcept.com/>
- Py YAML.
<http://pyyaml.org/>

5.8 SELF-EVALUATION

1. What is the difference between “w” and “a” modes if both allow you to write files?
2. Why we must close all files that are no longer in use?
3. Why we open files using `with`?
4. Make a program that asks a name, and then writes it to a file called `MyName.txt`.
5. Is it possible to parse csv files without csv module? If so, how is it done?
6. Why is it not recommended to read a file using `read()`?
7. What is the most efficient way to walk through a file line by line?
8. What is Pickle in Python?
9. Explain what is JSON and what limitation it has with respect to Pickle.
10. Make a program that reads all the numbers from the second column of an Excel file and prints the average of these values.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Code Modularizing

CONTENTS

6.1	Introduction to Code Modularizing	105
6.2	Functions	106
6.2.1	Standard Way to Make Python Code Modular	106
	Function Scope	108
6.2.2	Function Parameter Options	110
	Placement of Arguments	110
	Arguments with Default Values	111
	Undetermined Numbers of Arguments	111
	Undetermined Number of Keyword Arguments	112
6.2.3	Generators	113
	Creating a Generator	114
6.3	Modules and Packages	114
6.3.1	Using Modules	115
6.3.2	Packages	116
6.3.3	Installing Third-Party Modules	117
	Pip Is the Preferred Method	117
	Using System Package Management	118
	Copying to PYTHONPATH	119
6.3.4	Virtualenv: Isolated Python Environments	119
6.3.5	Conda: Anaconda Virtual Environment	121
	Manually Build and Install	124
6.3.6	Creating Modules	124
6.3.7	Testing Modules	125
	Doctest, Testing Modules in an Automatic Way	125
6.4	Additional Resources	127
6.5	Self-Evaluation	128

6.1 INTRODUCTION TO CODE MODULARIZING

With what we have seen so far, we have an interesting portfolio of resources for Python programming.¹ We can read files, do some data processing, and store its

¹If you are interested in applying what you have learned so far, I recommend the exercises in this page: <https://github.com/karan/Projects>.

results. Although programs made so far are very short, it is easy to imagine that they could grow to a size that may be difficult to manage.

There are several resources that can be used to modularize source code in a way that we may end up with a small program that calls pre-made code blocks. This approach favors code re-usability and readability. Both features also help maintenance, since you have to debug only one code implementation, regardless of how many times this code is used. As an additional advantage, it helps to improve performance, since any optimization on a modularized code benefits all the code that calls it.

For some authors, code modularizing is “The Greatest Invention in Computer Science”². I don’t know if this is the “greatest invention” or not, but certainly it is a fundamental concept that you can’t live without if you plan to do any serious programming.

Python provides several ways to modularize the source code: functions, modules, packages, and classes. This chapter covers all of them, with the exception of classes, which have their own chapter.

6.2 FUNCTIONS

6.2.1 Standard Way to Make Python Code Modular

Functions are the traditional way to modularize code. A function takes values (called arguments or parameters), executes some operation based on them and returns a value. We have already seen several Python built-in functions.³ For example `len()`, first mentioned on page 9, takes an iterable as parameter and returns a number:

```
>>> len('Hello')
5
```

Let’s see how to make our own functions. The general syntax of a function is:

```
def function_name(argument1, argument2, ...):
    """ Optional Function description (Docstring) """
    ... FUNCTION CODE ...
    return DATA
```

The code in [Listing 4.14](#) can be rewritten as a function:

Listing 6.1: `netchargefn`: Function to calculate the net charge of a protein

²Read the Steve McConnell column at <http://www.stevemcconnell.com/ieeesoftware/bp16.htm>.

³A list of all available functions in Python is available at: <https://docs.python.org/3/library/functions.html>.

```

1 def protcharge(aa_seq):
2     """Returns the net charge of a protein sequence"""
3     protseq = aa_seq.upper()
4     charge = -0.002
5     aa_charge = {'C':-.045, 'D':-.999, 'E':-.998, 'H':.091,
6                  'K':1, 'R':1, 'Y':-.001}
7     for aa in protseq:
8         charge += aa_charge.get(aa,0)
9     return charge

```

To “use” the function, it must be called with the parameter:

```

>>> protcharge('EEARGPLRGKGDQKSAVSQKPRSRGILH')
4.094

```

If we forget to pass the parameter, or if we pass an incorrect number of parameters, we get an error:

```

>>> protcharge()

```

```

Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    protcharge()
TypeError: protcharge() takes exactly 1 argument (0 given)

```

In this example, the function returns a number (of float type). If we want it to return more than one value, we can make it return a list or a tuple.⁴ The function *protcharge* (coded in [Listing 6.1](#)) could be modified to return, besides the net charge, the proportion of charged amino acids:

Listing 6.2: *netchargefn*: Function that returns two values

```

1 def charge_and_prop(aa_seq):
2     """ Returns the net charge of a protein sequence
3     and proportion of charged amino acids
4     """
5     protseq = aa_seq.upper()
6     charge = -0.002
7     cp = 0
8     aa_charge = {'C':-.045, 'D':-.999, 'E':-.998, 'H':.091,
9                  'K':1, 'R':1, 'Y':-.001}
10    for aa in protseq:

```

⁴It makes more sense to return a tuple instead of a list since for a given function there is a fixed number of parameters returned.

```

11         charge += aa_charge.get(aa,0)
12         if aa in aa_charge:
13             cp += 1
14         prop = 100.*cp/len(aa_seq)
15         return (charge,prop)

```

If we call the function with the same parameters of the last example, we get another result:

```

>>> charge_and_prop('EEARGPLRGKGDQKSAVSQKPRSRGILH')
(4.0940000000000003, 39.285714285714285)

```

Use an index to get one value:

```

>>> charge_and_prop('EEARGPLRGKGDQKSAVSQKPRSRGILH')[1]
39.285714285714285

```

All functions return something. A function can be used to “do something” instead of returning a value. In this case the value returned is **None**. For example, the following function stores the contents of a list in a text file:⁵

Listing 6.3: `convertlist.py`: Converts a list into a text file

```

1 def save_list(input_list, file_name):
2     """A list (input_list) is saved in a file (file_name)"""
3     with open(file_name, 'w') as fh:
4         for item in input_list:
5             fh.write('{0}\n'.format(item))
6     return None

```

The **return None** statement is optional. The function will return **None** without it, but Python developers prefer explicit statements than implicit assumptions.

Since Python 3, [Listing 6.3](#) can be written with the **print** function. Just replace line 5 for `print(item, file=fh)`. The “for loop” in line 4 can be avoided by using a property not seen yet. [Listing 6.6](#) on page 112 shows an alternative without the loop.

Function Scope

Variables declared inside a function are valid only inside the function. That means, if you try to access to a variable from outside the function, Python won't find it. To access the contents of a function variable from outside the function, the variable must be returned to the main program by using the **return** statement. In the following example, the variable `y`, defined inside the `duplicate` function, can't be used outside the function:

⁵For a way to save all kinds of Python data structures, see **Pickle** on page 94.

```
>>> def duplicate(x):
...     y = 1
...     print('y = {0}'.format(y))
...     return(2*x)
...
>>> duplicate(5)
y = 1
10
>>> y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'y' is not defined
```

In this case, the scope of `y` is inside the `duplicate` function. We can say that the function provides a **namespace** where the name `y` “lives.”

If the name is called inside the function but it is not defined there, Python will look for it outside the function; if it can't find it there, will return a **NameError**. Note that there is an order of preference when looking for names. First in the scope it was called, and then outside until reaching the global scope:

```
>>> def duplicate(x):
...     print('y = {0}'.format(y))
...     return(2*x)
...
>>> duplicate(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in duplicate
NameError: name 'y' is not defined
```

If the name is defined in the parent scope, it will find it:

```
>>> y = 3
>>> def duplicate(x):
...     print('y = {0}'.format(y))
...     return(2*x)
...
>>> duplicate(5)
y = 3
10
```

If the name is defined in the namespace provided by the function and outside, Python will use the first available name, that is, the one inside the function:

```
>>> y = 3
>>> def duplicate(x):
...     y = 1
...     print('y = {0}'.format(y))
...     return(2*x)
...
>>> duplicate(5)
y = 1
10
```

It can be specified inside a function that a variable is of **global** type, so its life won't be confined to the place it was defined. It is not a good idea to use global variables, since they can be modified at unexpected places. Another problem related to global variables is that Python has to keep track of its value for the entire runtime so it is not memory efficient.

```
>>> def test(x):
...     global z
...     z = 10
...     print('z = {0}'.format(z))
...     return x*2
...
>>> z = 1
>>> test(4)
z = 10
8
>>> z
10
```

6.2.2 Function Parameter Options

Placement of Arguments

Up to this point the arguments were put in the same order as originally defined. The function `savelist` can be called this way:

```
save_list([1,2,3], 'list.txt').
```

If we flip the order of the arguments (`save_list('temp.txt', [1,2,3])`) we get an error message:

```
save_list('list.txt', [1,2,3])
Traceback (most recent call last):
```

```
File "<ipython-input-5-fe7756f18e74>", line 1, in <module>
    save_list('list.txt', [1,2,3,4,5])
```

```
File "save_list1.py", line 10, in save_list
    with open(file_name, 'w') as fh:
```

```
TypeError: invalid file: [1, 2, 3, 4, 5]
```

This **TypeError** occurs because this function expects a list as the first parameter and a string as a second parameter. To call the function with the parameters in a different order than was originally defined, the parameter must be named when calling the function:

```
>>> savelist(file_name='list.txt', input_list=[1,2,3])
```

By using variable names the order of parameters is irrelevant.

Arguments with Default Values

Python allows default values in the arguments. This is done by entering the default value in the function definition:

```
def name(arg1=default_value, arg2=default_value, ... ):
```

For example the function `save_list`, which saves the contents of a list to a file, may have a default file name:

Listing 6.4: `list2textdefault.py`: Function with a default parameter

```
1 def save_list(input_list, file_name='temp.txt'):
2     """A list (input_list) is saved in a file (file_name)"""
3     with open(file_name, 'w') as fh:
4         for item in input_list:
5             fh.write('{0}\n'.format(item))
6     return None
```

Now the function can be called with only one parameter:

```
>>> save_list(['MS233', 'MS772', 'MS120', 'MS93', 'MS912'])
```

Undetermined Numbers of Arguments

Functions can have variable numbers of arguments if the final parameter is preceded by a `"*"`. Any excess arguments will be assigned to the last parameter as a tuple:

Listing 6.5: `getaverage.py`: Function to calculate the average of values entered as parameters

```

1 def average(*numbers):
2     if len(numbers)==0:
3         return None
4     else:
5         total = sum(numbers)
6         return total / len(numbers)

```

In this way the `average` function can be called with an undetermined number of arguments:

```

>>> average(2,3,4,3,2)
2.8
>>> average(2,3,4,3,2,1,8,10)
4.125

```

There is another use of the asterisk (*) in Python. From Python 3 a variable preceded by “*” becomes a list, which contains any items from the corresponding sequence that aren’t assigned to variable names.⁶ This property is used here (line 5 in Listing 6.6) to avoid using a loop to walk over all elements of L:

Listing 6.6: `list2text2.py`: Converts a list into a text file, using `print` and *

```

1 def save_list(input_list, file_name='temp.txt'):
2     """A list (input_list) is saved to a file (file_name)"""
3     with open(file_name, 'w') as fh:
4         print(*input_list, sep='\n', file=fh)
5     return None

```

Undetermined Number of Keyword Arguments

The functions can also accept an arbitrary number of arguments with keywords. In this case we use the final parameter preceded by “**” (two asterisks). The excess arguments are passed to the function as a dictionary:

Listing 6.7: `list2text2.py`: Function that accepts a variable number of arguments

```

1 def commandline(name, **parameters):
2     line = ''
3     for item in parameters:
4         line += ' -{0} {1}'.format(item, parameters[item])
5     return name + line

```

⁶This is explained in detail in PEP-3132 (<http://www.python.org/dev/peps/pep-3132>) and this Stackoverflow post: <http://stackoverflow.com/questions/6967632>.

This function can be called with a variable number of keyword parameters:

```
>>> cmdline('formatdb', t='Caseins', i='indata.fas')
'formatdb -t Caseins -i indata.fas'
>>> cmdline('formatdb', t='Caseins', i='indata.fas', p='F')
'formatdb -t Caseins -p F -i indata.fas'
```

Tip: Some Words about Docstrings.

Functions can have a text string immediately following the function definition. This line (or lines) is called “docstring.” Listing 6.6 in page 112 has a one-line docstring.

These lines are used for online help, automatic documentation generation systems, and for anyone who cares to read the source code. You can write anything inside a *docstring*, but there are written guidelines to standardize the structure of a docstring. Please refer to PEP-257 (<http://www.python.org/dev/peps/pep-0257>) for more information on Docstring format conventions.

Not only functions can have docstrings; modules and classes are expected to have its documentation as the first statement.

6.2.3 Generators

Generators are a special kind of function. Functions perform some action using variables in its local namespace. These variables are deleted after the function is executed. This process occurs each time a function is called. To avoid this, there is a special kind of function called a **generator**. When a generator is executed, its internal state is kept, so the next time it is invoked, the values of the variables can be accessed. Sometimes they are called resumable functions. This is used to avoid returning a huge object (like a big list, tuple, etc.) at once.

Take for example a function that reads records from a file and returns a data structure with data from this file. If the file is too big (like several times the available memory), the resulting data structure may not fit in memory. A solution to this problem is to modify the function to return one record at a time. A function can't do that because it doesn't keep a state, so each time it is executed, it has to process all the data again. **Generator** are functions that can keep their internal state. They introduce a new keyword: **yield**. When a **yield *EXPRESSION*** statement is found, it returns (or yields) *EXPRESSION* back to where it was called (as a function) but keeps track of its internal values, so next time it is called, it resumes operation with the values as it had before yielding the value.

Creating a Generator

[Listing 6.8](#) has a function (`all_primes()`) that returns all prime numbers available up to a given value. It returns them all together in a list:

Listing 6.8: `allprimes.py`: Function that returns all prime numbers up to a given value

```

1 def is_prime(n):
2     """Returns True is n is prime, False if not"""
3     for i in range(2,n-1):
4         if n%i == 0:
5             return False
6     return True
7
8 def all_primes(n):
9     primes = []
10    for number in range(1,n):
11        if isprime(number):
12            primes.append(number)
13    return p

```

Function `all_primes()` from [Listing 6.8](#) can be replaced with generator `g_all_primes()`:

Listing 6.9: `allprimesg.py`: Generator that replaces `putn()` in code 6.8.

```

1 def g_all_primes(n):
2     for number in range(1,n):
3         if is_prime(number):
4             yield number

```

Note that code in [Listing 6.9](#) doesn't use a list, since there is no need for it because it yields one result at a time. Both functions can be used to walk over the results, but `all_primes()` generates a list, while `g_all_primes()` doesn't.

6.3 MODULES AND PACKAGES

A module is a file with function definitions, constants, or any type of object that you can use from other modules or from your main program. Modules also provide namespaces, so two functions may be given the same name provided that they are defined in different modules. The name of the module is taken from the name of the file. If the module filename is `my_module.py`, the module name is `my_module`.

6.3.1 Using Modules

To access the contents of a module, use **import**. Usually import is issued at the beginning of the program. It is not mandatory to place the imports at the beginning of the file, but it must be placed before calling any of the elements of the module. It is customary, however, to place the **import** statement at the beginning of the program. There are many ways to use import. The most used form is by calling a module by its name. To call the built-in module **os**, use,

```
>>> import os
```

When a module is imported for the first time, its contents are executed. If the module is imported more than once, the successive imports will not have any effect. This gives us the assurance that we can put an import statement inside a function and not worry if it is called repeatedly.

Once a module is imported, to access a function or a variable, use the name of the module as a prefix:

```
>>> os.getcwd()
'/mnt/hda2'
>>> os.sep
','
```

It is also possible to import from a module only a required function. This way we can call it without having to use the name of the module as a prefix.

```
>>> from os import getcwd
>>> getcwd()
'/mnt/hda2'
```

To import all the contents of a module, use the “*” operator (asterisk):

```
>>> from os import *
>>> getcwd()
'/mnt/hda2'
>>> sep
','
```

Warning: Don’t use the **from module import *** unless you know what you are doing. The problem with importing all the elements of the module is that it may produce conflicts with the names already defined in the main program (or defined in other modules and imported the same way). In Python programming standards, wildcard imports are equivalent to the dark side of the force. They’re quicker, easier, and more seductive, but dangerous.

It is also possible to import a module using a different name:

```
>>> import xml.etree.ElementTree as et
>>> tree = et.parse('/home/sb/bioinfo/smallUniprot.xml')
```

Don't worry if you don't know what `xml.etree.ElementTree` is, we will look at this in the XML chapter, but from this moment, take into account that this entire name (`xml.etree.ElementTree`) is called "ET."

6.3.2 Packages

A **package** is a group of modules with some characteristics in common. They are directories with the modules or other directories inside. Also contains a special file named `__init__.py`. This file indicates that the directory it contains is a Python package and can be imported as a module.

Bio/	Top-level package
<code>__init__.py</code>	Initialize the sound package
Align/	Subpackage for Alignment related software
<code>__init__.py</code>	
AlignInfo.py	
Alphabet/	Subpackage for amino-acid alphabets
<code>__init__.py</code>	
IUPAC.py	
Reduced.py	
Blast/	Subpackage for Blast parsers
<code>__init__.py</code>	
Applications.py	
NCBIStandalone.py	
NCBIWWW.py	
NCBIXML.py	
ParseBlastTable.py	
Record.py	

The `__init__.py` files are required to make Python treat the directories as containing packages. In most cases, `__init__.py` is an empty file, but it can also execute initialization code for the package.

Users of the package can import individual modules from the package, for example:

```
import Bio.Blast.Applications
```

Even if there are differences between modules and packages, both terms are used interchangeably.

6.3.3 Installing Third-Party Modules

Python comes with several modules (built-in modules). These modules are bundled with Python so they are ready to use as soon as you have a working Python interpreter.⁷

There are also third-party modules that extend Python functionality, as mentioned on page 11. Installation can be as easy as copying a single file to a specific location up to executing several programs in a predetermined order. It depends on the complexity of the modules. Modules range from one file to several files spanned in multiple directories that interact with other programs; in this case it is called a package. So there is no unique way to install every external module available to Python.

Pip Is the Preferred Method

Most packages support **pip installation**. **pip** is the native way to install Python packages and is the preferred method.⁸ For this kind of installation you need **pip** and **setuptools**. Most likely you already have them installed (as it comes with Python 3.4 binaries), if not; install it with:

```
sudo apt install python-pip
```

Note that in Ubuntu the package is called `python3-pip`.

Then upgrade it to the latest version.

On Linux or macOS:

```
$ pip install -U pip setuptools
```

In Ubuntu the command is:

```
$ pip3 install -U pip setuptools
```

On Windows:

```
$ python -m pip install -U pip setuptools
```

Once installed and updated, Python modules can be installed with:

```
$ pip install MODULE_NAME
```

For example, to install **xlrd**, a package to read Excel files:

⁷Check <http://docs.python.org/library/index.html> for a complete description of the Python Standard Library including built-in modules.

⁸There is another method called **easy_install** that was featured in the first edition of this book, but **pip** has more features so **easy_install** was removed from this book. To compare both systems, please check https://packaging.python.org/pip_easy_install/.

```

$ pip install xlrd
Collecting xlrd-1.0.0
  Downloading xlrd-1.0.0.tar.gz
Building wheels for collected packages: xlrd
  Running setup.py bdist_wheel for xlrd ... done
  Stored in directory: /home/sb/.cache/pip/wheels/55/e2/c6f97024749<=
ea24f67400fb4c55eab7f2b49cbf39379805ef5
Successfully built xlrd
Installing collected packages: xlrd
Successfully installed xlrd-1.0.0

```

You need a working Internet connection for the above command to run. Pip will retrieve the package from the PyPi repository at <https://pypi.python.org>. To find out what packages are available to install using pip, see the list in <https://pypi.python.org/pypi?%3Aaction=browse>.

Another caveat to take into account is who will use the installed package. To make the package available for all Python users on the machine, you must be an administrator user or install it with sudo:

```
$ sudo pip install xlrd
```

But the preferred method of installation is as a user and inside a virtual environment (see page 119).

Using System Package Management

Some Python modules can be installed as any other program you install in your computer, such as double-clicking in an installer (Windows/macOS) or using the system package management like apt-get in Ubuntu.

The advantage of using system package management is that you can keep track of installed Python modules the same way you keep track of every other software in your system. Upgrades and uninstallations are easier and without nasty consequences such as orphan files or broken installations. This method also has its drawbacks, like a gap between current package version and the version available in your Linux distribution repository. Some modules develop at a fast pace, sometimes so fast that package managers can't keep up to date. For example, Ubuntu users who want to install Biopython using apt-get, at time of writing, are limited to version 1.66 when 1.69 is the last version available at Biopython website. Another problem is that in some systems you need administration rights to use package management. Windows installers do not provide all required software and do not search for it in an automatic way so you may need to install some prerequisite software before running the installer. The main problem involving package management is that sometimes the required package is not available. For all these reasons, this is not the recommended way to install new packages.

Copying to PYTHONPATH

This is not the most frequent module installation procedure, but it is mentioned first because it is very simple. Just copy the module where Python searches for modules. Where does Python search for modules? There are three places:

- In the same directory where the program that will call the module is located.
- In the same directory where the Python executable is located. This directory is different on each operating system.⁹
- In a directory created especially for our modules. In this case, it must be specified in the environment variable **PYTHONPATH** or in the variable **sys.path**. This final variable lists all the paths where Python should look for a module. To add a directory to **sys.path**, you should modify it as you would do with any list, using the **append** method:

```
>>> import sys
>>> sys.path
['/usr/lib/python3.5', '/usr/lib/python3.5/plat-x86_64-<=
linux-gnu', '/usr/lib/python3.5/lib-dynload', '/usr/loc<=
al/lib/python3.5/dist-packages', '/usr/lib/python3/dist<=
-packages']
>>> sys.path.append('/home/sb/MyPyModules')
```

6.3.4 Virtualenv: Isolated Python Environments

virtualenv is a program that creates isolated Python environments. Each environment created with **virtualenv** has its own set of external (third-party) modules. This allows you to have multiple independent projects each in its own environment, so there is no conflict with incompatible dependences. A project may require a module in version X and another project may need the same module but in version Y. Since you can't install two different versions of the same module in the same Python installation, you need a way to isolate each installation. This is what **virtualenv** provides. When should you use **virtualenv**? Short answer: Always. Long answer: Each time you start working with a new project, it is better to have a dedicated Python environment. This also has the benefit of being able to reproduce the setup for a particular program in another machine when needed. If you have Python3.6 or greater, you don't need **virtualenv** as a stand alone program, since it is included with Python.

With a Python version previous to 3.6, install **virtualenv** using pip:

⁹On Windows, it is usually found at C:\program files\Python, while on Linux it is found at /usr/bin/python. To find the path to the Python executable in *nix, use `which python`.

```
# pip install virtualenv
```

Note that in Windows **pip** may be located in the **Scripts** directory. Once installed, a **virtualenv** is created this way:

```
$ virtualenv <DIRECTORY>
```

If you have Python 3.6 or greater, you don't need to install **virtualenv**, to create a new virtual environment with Python 3.6 or greater. Instead, just do:

```
python3 -m venv <DIRECTORY>
```

Example with an older version of Python:

```
$ virtualenv bioinfo
Using base prefix '/usr'
New python executable in /home/sb/bioinfo/bin/python3
Also creating executable in /home/sb/bioinfo/bin/python
Installing setuptools, pip, wheel...done.
$
```

or with Python 3.6:

```
python3 -m venv bioinfo
```

This will generate a **bioinfo** directory inside the current directory. Once created, it is time to activate it. macOS and Linux:

```
$ source <DIRECTORY>/bin/activate
```

In the previous example, the activation command would be:

```
$ source bioinfo/bin/activate
```

Note that in Windows, the virtual environment is activated this way:

```
<DIRECTORY>\Scripts\activate
```

After activating the virtual environment, the prompt will change to:

```
(bioinfo)$
```

This is used to indicate that the **virtualenv** is activated and every package you install from that point will be available only inside this **virtualenv**.

To install a package, proceed in the same way as before (by using **pip**) but inside the environment, for example:

```
(bioinfo)$ pip install xlrd
```

This way, the **xlrd** package will be available only in the **bioinfo** environment and will not interfere with any other Python installation.

Once you are done working with the virtual environment, you should deactivate it to return to your standard prompt:

```
(bioinfo)$ deactivate
$
```

In windows:

```
(bioinfo)> \path\to\env\bin\deactivate.bat
>
```

6.3.5 Conda: Anaconda Virtual Environment

If you are using Anaconda Python distribution, you should use **conda create** instead of **virtualenv**. If you are using the regular Python, just skip this section.

To create the new environment, use this command:

```
$ conda create -n NAME
```

Where **NAME** is the name you want to use for the new environment, if you want to use **bioinfo**, the command is:

```
$ conda create -n bioinfo
Fetching package metadata .....
.Solving package specifications: .
Package plan for installation in environment /home/sb/anaconda3/<=
envs/bioinfo:
```

The following empty environments will be CREATED:

```
/home/sb/anaconda3/envs/bioinfo
```

```
Proceed ([y]/n)?
```

```
#
# To activate this environment, use:
# > source activate bioinfo
#
# To deactivate this environment, use:
# > source deactivate bioinfo
#
```


To activate the environment, type:

```
$ source activate bioinfo
(bioinfo) $
```

As in the **virtualenv** environment, everything you install inside this environment will be local to the environment and won't affect any other Python installation. To install a package in the active environment, the preferred way is to use **conda install**. To install the **pillow** package:

```
(bioinfo) $ conda install pillow
```

If the package is not available in the **conda** repository, you can use **pip install**:

```
(bioinfo)$ pip install beautifulsoup4
```

My advice is to use **conda** when working with **Anaconda** and **pip** if you use the standard Python distribution.¹⁰

Conda also allows you to create a virtual environment and install packages in one command. Just add the package name after the **conda create** command:

```
$ conda create -n excelprocessing xlrd
Fetching package metadata .....
Solving package specifications: .....
```

Package plan for installation in environment /sb/anaconda3/envs/exc<=elprocessing:

The following packages will be downloaded:

package	build	
python-3.6.0	0	16.3 MB
setuptools-27.2.0	py36_0	523 KB
wheel-0.29.0	py36_0	88 KB
xlrd-1.0.0	py36_0	185 KB
pip-9.0.1	py36_1	1.7 MB
Total:		18.8 MB

The following NEW packages will be INSTALLED:

¹⁰See the article at this site for more information: <https://goo.gl/Ki8zks>.

```

openssl:    1.0.2j-0
pip:        9.0.1-py36_1
python:     3.6.0-0
readline:   6.2-2
setuptools: 27.2.0-py36_0
sqlite:     3.13.0-0
tk:         8.5.18-0
wheel:      0.29.0-py36_0
xlrd:       1.0.0-py36_0
xz:         5.2.2-1
zlib:       1.2.8-3

```

Proceed ([y]/n)? y

Fetching packages ...

```

python-3.6.0-0 100% |#####| Time: 0:00:01 10.99 MB/s
setuptools-27. 100% |#####| Time: 0:00:00 5.72 MB/s
(...)

```

Extracting packages ...

```

[ COMPLETE ] |#####| 100%

```

Linking packages ...

```

[ COMPLETE ] |#####| 100%
#

```

To activate this environment, use:

```
# > source activate excelprocessing
```

#

To deactivate this environment, use:

```
# > source deactivate excelprocessing
```

#

Activate and check that the package is installed:

```
$ source activate excelprocessing
```

```
(excelprocessing) $ python
```

```
Python 3.6.0 |Continuum Analytics, Inc.| (Dec 23 2016, 12:22:00)
```

```
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux
```

```
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>> import xlrd
```

```
>>>
```

To see all installed environments (name and path), run `conda info -envs`:

```
$ conda info --envs
```

```
# conda environments:
```

```
#
```

```

AEML                /home/sb/anaconda3/envs/AEML
bioinfo              /home/sb/anaconda3/envs/bioinfo
biopy1               /home/sb/anaconda3/envs/biopy1
excelprocessing      /home/sb/anaconda3/envs/excelprocessing
py4bio               /home/sbassi/anaconda3/envs/py4bio
root                 * /home/sbassi/anaconda3

```

The asterisk (*) shows the active environment.

Using **virtualenv** (or **conda** if using Anaconda) is so important that there will be multiple references in this book on how to use it.

Manually Build and Install

If you can't use system packages and don't want to (or can't) use **pip**, there is always a manual way to install packages. Download the module files (usually in ".tar.gz" format), unpack them and look for a **setup.py** file. In most cases, installing it is a matter of running:

```
python setup.py install
```

If there are any problems, see the **README** file. In fact, it is advisable to check the **README** file before trying to install the program (who does that?). In most cases the problem arises from missing dependencies (like you need module X to run module Y), that you will have to fulfill. That is why it is better to install Python modules with **pip** or with your system package management.

6.3.6 Creating Modules

To create a module, you have to create a file and save it with the ".py" extension. It should be saved in a directory where the Python interpreter searches for it, like those in the **PYTHONPATH** variable (see page 119 for more information).

For example, store the function **save_list** in a module and call it **utils**. For this, create the file **utils.py** with the following contents:

```

# utils.py file
def save_list(input_list, file_name='temp.txt'):
    """A list (input_list) is saved to a file (file_name)"""
    with open(file_name, 'w') as fh:
        print(*input_list, sep='\n', file=fh)
    return None

```

This way, this function (**save_list**) can be used from any program, provided that this file is saved in a location accessible from Python:

```

>>> import utils
>>> utils.save_list([1,2,3])

```

6.3.7 Testing Modules

A good programming practice involves the creation of tests to verify the correct functioning of your code.

As the modules are designed to be used from within a program, these tests must be executed only when called from the command line. This way, tests will not interfere with the normal operation of the program.

To achieve this, we need to be able to differentiate when code is being executed as a standalone program and when it is executed as a module from another program. When the code is executed as a program, the variable `__name__` takes the value `"__main__"`. As a result, the way to incorporate test code is by doing it after verifying that the program executes independently.

```
if __name__ == '__main__':
    #Do something
```

This type of test is usually included at the end of a module. In [Listing 6.10](#) (page 125) we can see a test in action.

Python provides a module that facilitates the task of testing that our code works as we expect. This module is called **doctest**.

Doctest, Testing Modules in an Automatic Way

Doctest is a module that searches for pieces of Python code inside a docstring. This code is executed as if it were an interactive Python session. The module tests if this code works exactly as shown in the docstring or in an external file.

In [Listing 6.10](#) we have `is_prime()`, a function that checks if a given number (`n`) is prime. Let's see how we can incorporate a test unit and run it:

Listing 6.10: prime5.py: Module with doctest

```
1 def is_prime(n):
2     """ Check if n is a prime number.
3     Sample usage:
4     >>> is_prime(0)
5     False
6     >>> is_prime(1)
7     True
8     >>> is_prime(2)
9     True
10    >>> is_prime(3)
11    True
12    >>> is_prime(4)
13    False
14    >>> is_prime(5)
```

```

15     True
16     """
17
18     if n <= 0:
19         # This is only for numbers > 0.
20         return False
21     for x in range(2, n):
22         if n%x == 0:
23             return False
24     return True
25
26 def _test():
27     import doctest
28     doctest.testmod()
29
30 if __name__ == '__main__':
31     _test()

```

Code explanation: The `is_prime(n)` function is defined from line 1 to 24, but the actual functionality starts at line 18. Up to this line, there are some tests. These tests are not executed if the program is called from another program, which is checked in line 30. If the program is executed as a standalone program, all test are run:

```

$ python prime5.py
$

```

There is no output. That is, no news is good news. Let's see what happens when we change line 21 to `"for x in range(1,n):"`:

In this case, the test fails:

```

$ python prime5.py
*****
File "./prime5.py", line 10, in __main__.is_prime
Failed example:
    is_prime(2)
Expected:
    True
Got:
    False
*****
File "./prime5.py", line 12, in __main__.is_prime
Failed example:
    is_prime(3)

```

```

Expected:
    True
Got:
    False
*****
File "./prime5.py", line 16, in __main__.is_prime
Failed example:
    is_prime(5)
Expected:
    True
Got:
    False
*****
1 items had failures:
  3 of   6 in __main__.is_prime
***Test Failed*** 3 failures.

```

Testing is so important that there is a methodology called test-driven development. It proposes to design a test for every function before starting to write code. Testing may not be perceived as a primary need for a program, but one cannot be certain that a function works unless one tests it. Testing is also useful to make sure that a change in the code has no unintended consequences.

Python has extensive support for software testing (with modules **doctest** and **unittest**), but this is out of the scope of this book. For more information on testing, see “Additional Resources.”

6.4 ADDITIONAL RESOURCES

- Modules, the Python tutorial.
<http://docs.python.org/tutorial/modules.html>
- Default parameter values in Python, by Fredrik Lundh.
<http://effbot.org/zone/default-values.htm>
- Python library reference. Unittest API.
<https://docs.python.org/3.6/library/unittest.html>
- Installing Python modules.
<http://docs.python.org/install/index.html>
- PIP.
<https://pip.pypa.io/en/stable/>
- Extreme programming. Wikipedia article.
http://en.wikipedia.org/wiki/Extreme_Programming

6.5 SELF-EVALUATION

1. What is a function?
2. How many values can return a function?
3. Can a function be called without any parameters?
4. What is a docstring and why is it related to functions and modules?
5. Does every function need to know in advance how many parameters it will receive?
6. Write a generator function.
7. Why must all optional arguments in a function be placed at the end in the function call?
8. What is a module?
9. Why are modules invoked at the beginning of the program?
10. How do you import all contents of a module? Is this procedure advisable?
11. How can you test if your code is being executed as a standalone program or called as a module?
12. What is **virtualenv** and when would you use it?

Error Handling

CONTENTS

7.1	Introduction to Error Handling	129
7.1.1	Try and Except	131
7.1.2	Exception Types	134
	How to Respond to Different Exceptions	134
7.1.3	Triggering Exceptions	135
7.2	Creating Customized Exceptions	136
	All Exceptions Derive from Exception Class	137
7.3	Additional Resources	137
7.4	Self-Evaluation	138

You can make it foolproof, but you can't make it damnfoolproof.

Naeser's law

7.1 INTRODUCTION TO ERROR HANDLING

A program rarely works as expected, at least on the first try.

Traditionally a developer would choose between one of these two strategies when faced with runtime program errors. The problem is ignored or each condition is verified where an error may occur and then he or she would write code in consequence. The first option, which is very popular, is not advisable if we want our program to be used by anyone besides ourselves. The second option, which is also known as **LBYL** (**L**ook **B**efore **Y**ou **L**ean), is time consuming and may make code unreadable. Let's have a look at an example of each strategy.

The following program reads a file (`myfile.csv`) separated by tabs and looks for a number that is found in the first column of the first line. This value is multiplied by 0.2 and that result is written to another file (`otherfile.csv`).

This version does not check for any types of errors and limits itself to its core functionality.

Listing 7.1: `wotest.py`: Program with no error checking

```
1 with open('myfile.csv') as fh:
2     line = fh.readline()
```



```

3 value = line.split('\t')[0]
4 with open('other.txt','w') as fw:
5     fw.write(str(int(value)*.2))

```

This program may do its job provided that there are no unexpected events. What does “unexpected events” mean in this context? The first line is prone to error. For example, it may be trying to open a file that doesn’t exist. In this case, when the program runs it will immediately stop after executing the first line and the user will face an error:

```

Traceback (most recent call last):
  File "wotest.py", line 1, in <module>
    with open('myfile.csv') as fh:
FileNotFoundError: [Errno 2] No such file or directory: 'myfile.csv'

```

This is a problem because the program stops, and it is not professional to show the end user a system error.

This program can fail in various places. There may be no tabs in the file, there may be letters instead of numbers, and we may not have the write permissions in the directory where we intend to write the output file.

That is what happens when the file exists but there are no tabs inside.

```

Traceback (most recent call last):
  File "wotest.py", line 6, in <module>
    fw.write(str(int(value)*.2))
ValueError: invalid literal for int() with base 10: '12,dsa\n'

```

The result is similar to the previous one. It causes the program to stop and the interpreter shows us another error message. This way we may continue with all the blocks of code that are prone to the failure.

Let’s look at the strategy of checking each condition likely to generate an error in order to prevent its occurrence (LBYL).

Listing 7.2: LBYL.py: Error handling LBYL version

```

1 import os
2 iname = input("Enter input filename: ")
3 oname = input("Enter output filename: ")
4 if os.path.exists(iname):
5     with open(iname) as fh:
6         line = fh.readline()
7         if "\t" in line:
8             value = line.split('\t')[0]
9             if os.access(oname, os.W_OK) == 0:

```

```

10         with open(oname, 'w') as fw:
11             if value.isdigit():
12                 fw.write(str(int(value)*.2))
13             else:
14                 print("Can't be converted to int")
15     else:
16         print("Output file is not writable")
17 else:
18     print("There is no TAB. Check the input file")
19 else:
20     print("The file doesn't exist")

```

This program considers almost all the possible errors. If the file that the user enters does not exist, the program will not have an abnormal termination. Instead, it will display an error message designed by the programmer that would allow the user to reenter the name of the input file.

The disadvantage of this option is that the code is both difficult to read and maintain because the error checking is mixed with its processing and with the main objective of the program. It is for this reason that new programming languages have included a specific system for the control of exceptional conditions. Contrary to LBYL, this strategy is known as EAFP (it's **easier to ask forgiveness than permission**). With Python, the statements **try**, **except**, **else** y **finally** are used.

7.1.1 Try and Except

try delimits the code that we want to execute, while **except** delimits the code that will be executed if there is an error in the code under the **try** block. Errors detected during execution are called *exceptions*. Let's look at the general outline:

```

try:
    code block 1
    # ...some error prone code...
except:
    code block 2
    # ...do something with the error...
[else:
    code block 3
    # ...to do when there is no error...
finally:
    code block 4
    #...some clean up code...]

```

This code will first try to execute the code in block 1. If the code is executed without problems, the flow of execution continues through the code in block 3 and finally through block 4. In case the code in block 1 produces an error (or raises an

exception according to the jargon), the code in block 2 will be executed and then the code in block 4. The idea behind this mechanism is to put the block of code that we believe may produce an error (block 1), inside the **try** clause. The code that is triggered when there is an exception is placed in the **except** block. This code (code block 2) deals with the exception, or in another words, it *handles the exception*. Error messages are what the user gets when exceptions are not handled:

```
>>> 0/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Optionally, it is possible to add the statement **else**, which will be executed only if the code inside **try** (code block 1) executes successfully. Note that the code below **else** can be placed in the **try** block because it would have the same effect (it would execute if there are no errors). The block inside **try** should contain only the code that may raise an exception, while we would have to leave in the block inside **else** the instructions that should be executed when the instructions inside **try** are executed without error. Note that, code inside **finally** is always executed.

For instance:

```
try:
    print(0/0)
except:
    print("Houston, we have a problem...")
```

The result is:

```
Houston, we have a problem...
```

The first thing that we take note of is that neither **else** nor **finally** is included as they are optional statements. In this case, the statement **print(0/0)** raises an exception. This exception is “caught” by the code inside **except**. This way we make sure that even after an error, the program will flow in a predictable way.

In this code, exception handling is applied to code [Listing 7.2](#):

Listing 7.3: exception.py: Similar to 7.2 but with exception handling.

```
1 try:
2     iname = input("Enter input filename: ")
3     oname = input("Enter output filename: ")
4     with open(iname) as fh:
5         line = fh.readline()
6     if '\t' in line:
7         value = line.split('\t')[0]
```

```

8     with open(oname, 'w') as fw:
9         fw.write(str(int(value)*.2))
10 except NameError:
11     print("There is no TAB. Check the input file")
12 except FileNotFoundError:
13     print("File not exist")
14 except PermissionError:
15     print("Can't write to outfile.")
16 except ValueError:
17     print("The value can't be converted to int")
18 else:
19     print("Thank you!. Everything went OK.")

```

At first look it is noticeable that this code is easier to follow than the previous version (7.2). At least the code logic is separated from the error handling. From line 10 is where the exception handling begins. According to the type of exception, it is the code that will be executed below. We will see how to distinguish between the different types of exceptions later.

[Listing 7.3](#) is an introductory example of how to apply exception handling to [Listing 7.1](#), and not a definitive guide of how to handle exceptions.

Listing 7.4: `nested.py`: Code with nested exceptions

```

1 iname = input("Enter input filename: ")
2 oname = input("Enter output filename: ")
3 try:
4     with open(iname) as fh:
5         line = fh.readline()
6 except FileNotFoundError:
7     print("File not exist")
8 if '\t' in line:
9     value = line.split('\t')[0]
10 try:
11     with open(oname, 'w') as fw:
12         fw.write(str(int(value)*.2))
13 except NameError:
14     print("There is no TAB. Check the input file")
15 except PermissionError:
16     print("Can't write to outfile.")
17 except ValueError:
18     print("The value can't be converted to int")
19 else:
20     print("Thank you!. Everything went OK.")

```

We've seen in general terms how the **try/except** clause works, and now we can go a little deeper to discuss the types of exceptions.

7.1.2 Exception Types

Exceptions can be individualized. A nonexistent variable and mixing incompatible data types are not the same type of error. The first exception is of the **NameError** type, while the second is of the **TypeError** type. A complete list of exceptions can be found in <https://docs.python.org/3.6/library/exceptions.html>.

How to Respond to Different Exceptions

It is possible to handle an error generically using **except** without a parameter:

```
d = {"A": "Adenine", "C": "Cysteine", "T": "Timine", "G": "Guanine"}
try:
    print(d[input("Enter letter: ")])
except:
    print("No such nucleotide")
```

Just because we may be able to respond generically to all errors doesn't mean that it is a good idea. This makes debugging our code difficult because an unanticipated error can pass unnoticed. This code will return a "No such nucleotide" for any type of error. If we introduce an EOF signal (end of file, CONTROL-D in some terminals), the program will output "No such nucleotide". It is useful to distinguish between the different types of abnormal events, and react in consequence. For example to differentiate an EOF from a nonexistent dictionary key:

```
d = {"A": "Adenine", "C": "Cysteine", "T": "Timine", "G": "Guanine"}
try:
    print(d[input("Enter letter: ")])
except EOFError:
    print("Good bye!")
except KeyError:
    print("No such nucleotide")
```

This way, the program prints "No such nucleotide" when the user enters a key that does not exist in `d` dictionary and "Good bye!" when it gets an EOF.

To get information about the exception that is currently being handled, use `sys.exc_info()`:

Listing 7.5: `sysexc.py`: Using `sys.exc_info()`

```
1 import sys
```

```

2
3 try:
4     0/0
5 except:
6     a,b,c = sys.exc_info()
7     print('Error name: {0}'.format(a.__name__))
8     print('Message: {0}'.format(b))
9     print('Error in line: {}'.format(c.tb_lineno))

```

This program prints:

```

Error name: ZeroDivisionError
Message: integer division or modulo by zero
Error in line: 4

```

Listing 7.6: `sysexc2.py`: Another use of `sys.exc_info()`

```

1 import sys
2
3 try:
4     x = open('random_filename')
5 except:
6     a, b = sys.exc_info()[:2]
7     print('Error name: {}'.format(a.__name__))
8     print('Error code: {}'.format(b.args[0]))
9     print('Error message: {}'.format(b.args[1]))

```

This program prints:

```

Error name: FileNotFoundError
Error code: 2
Error message: No such file or directory

```

7.1.3 Triggering Exceptions

Exceptions can be activated manually using **raise**, without the need to wait for them to occur. You may be wondering why you would want to trigger an exception. An appropriately raised exception may be more helpful to the programmer or to the user than an exception that is fired in an uncontrolled way. This is especially true when debugging programs.

This is better understood with an example. The `avg` function that follows, calculates the average of a sequence of numbers:

```

def avg(numbers):
    return sum(numbers)/len(numbers)

```

A function of this type will have problems with an empty list:

```
>>> avg([])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in avg
ZeroDivisionError: division by zero
```

The issue with this error message is that it does not tell us that it was caused by the empty list, but says that it was provoked but trying to divide by zero. By knowing how the function works, one can deduce that an empty list causes this error. However, it would be more interesting if this error points this out, without having to know the internal structure of the function. For this we can raise an error by ourselves.

```
def avg(numbers):
    if not numbers:
        raise ValueError("Please enter at least one element")
    return sum(numbers)/len(numbers)
```

In this case, the error type is closer to the actual problem.

```
>>> avg([])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in avg
ValueError: Please enter at least one element
```

We could have avoided the error if we printed a string without raising the error, but this will be against pythonic principles (“errors should not pass unnoticed”). In practice, this may cause problems because if a function returns an unanticipated value, the effects can be unpredictable. On raising the exception, we assure the error will not pass unnoticed.

In some texts or old code you will find a syntax of the form `raise "This is an error"`. These types of exceptions (called chained exceptions), are not compatible with Python 2.6 and later. The form `raise ValueError, 'A message'` is also deprecated and the preferred form is `raise ValueError('A message')`. From Python 3.0, the latter form is mandatory.¹

7.2 CREATING CUSTOMIZED EXCEPTIONS

An advantage of the exception system is that we don't have to limit ourselves to those provided by Python. We can define new exceptions to serve our needs. In

¹Please see PEP 3109 (<http://www.python.org/dev/peps/pep-3109>) regarding the rationale for this.

order to create an exception, we need to work with Object Oriented Programming (OOP), a topic that has not been covered yet. As a result, if you're reading this book from the start and need to create your own exceptions, my recommendation is that you skip the rest of this chapter and proceed directly to [Chapter 8](#). After reading [Chapter 8](#), return to this section.

All Exceptions Derive from Exception Class

Since all exceptions derive from the `Exception` class, we can make our own exception by subclassing the `Exception` class. Take for example this exception that I called `NotDNAException`. It should be raised when there is a DNA sequence with a character not belonging to 'a', 'c', 't', or 'g'. Let's see a custom exception defined:

```
class NotDNAException(Exception):
    """A user-defined exception"""
    def __init__(self, dna):
        self.dna = dna
    def __str__(self):
        for nt in self.dna:
            if nt not in 'atcg':
                return nt
```

The programmer should create a code to detect the exception:

```
dnaseq = 'agctwtacagt'
if set(dnaseq) != set('atcg'):
    raise NotDNAException(dnaseq)
else:
    print('OK')
```

If `dnaseq` is an iterable object with 'a', 'c', 't', or 'g', this code prints OK. But if `dnaseq` contains a non-DNA character, the exception will be raised. This is the result of the former code but with a 'w' in `dnaseq`:

```
Traceback (most recent call last):
  File "7_25.py", line 22, in <module>
    raise NotDNAException(dnaseq)
__main__.NotDNAException: w
```

7.3 ADDITIONAL RESOURCES

- PEP 3134 “Exception Chaining and Embedded Tracebacks.”
<https://www.python.org/dev/peps/pep-3134/>
- Python documentation. Built-in exceptions.
<https://docs.python.org/3.6/library/exceptions.html>

- Python documentation. Standard errno system symbols.
<https://docs.python.org/3.6/library/errno.html>
- C H. Swaroop. Python exceptions.
<https://python.swaroopch.com/exceptions.html>
- Ian Bicking. Re-raising exceptions.
<http://www.ianbicking.org/blog/2007/09/re-raising-exceptions.html>

7.4 SELF-EVALUATION

1. What is the meaning of LBYL and EAFP? Which one is used in Python?
2. What is an **exception**?
3. What is an “unhandled exception”?
4. When do you use **finally** and when do you use **else**?
5. Exceptions are often associated with file handling. Why?
6. How do you sort an error derived from a disk full condition from trying to write to a read-only file system?
7. Why is it not advisable to use **except:** to catch all kind of exceptions, instead of using, for example, **except IOError:**?
8. Exceptions can be raised at will. Why would you do that?
9. What is the purpose of **sys.exc_info()**?
10. Explain the purpose of this function:

```
def formatExceptionInfo():
    """ Author: Arturo 'Buanzo' Busleiman """
    cla, exc = sys.exc_info()[:2]
    excName = cla.__name__
    try:
        excArgs = exc.__dict__["args"]
    except KeyError:
        excArgs = str(exc)
    return (excName, excArgs)
```

Introduction to Object Orienting Programming (OOP)

CONTENTS

8.1	Object Paradigm and Python	139
8.2	Exploring the Jargon	140
	Classes: Object Generators	140
	Instance: Particular Implementation of Class	140
	Attributes or Instance Variables: Characteristics of Objects	141
	Methods: Behavior of Objects	141
	Class Attributes: Characteristics of Classes	141
	Inheritance: Properties Are Transmitted between the Related Classes	141
	Polymorphism	141
	Encapsulation	141
8.3	Creating Classes	142
8.4	Inheritance	145
	Introducing Some Biopython Objects	146
8.5	Special Methods	149
	8.5.1 Create a New Data Type Using a Built-in Data Type	154
8.6	Making Our Code Private	154
8.7	Additional Resources	155
8.8	Self-Evaluation	156

8.1 OBJECT PARADIGM AND PYTHON

As mentioned in the introduction of the book, Python is an object-oriented language. Unlike other languages that handle objects, Python allows us to program in a classic procedural way, without considering the objects paradigm. Sometimes this is called a “multi-paradigm language.”

We have already used objects, even without stating it in an explicit way. Data types included in Python are objects. Strings, dictionaries, and lists, are implementation of objects. Each of them has its associated functions (*methods* in the jargon) and its attributes (associated data). We have seen that **lower()** returns a string

in lower case. This is because all the objects of the class `string` have the method `lower()` associated with them.

Representing part of the real world usually is one of the goals of programming. From a bank transaction to the reconstruction of a DNA sequence, all can be expressed in a programming language. Although data types included in Python are many and varied, its capacity to include all our information modeling needs is limited. A class can be used to define new kind of data type.

For example, a dictionary can represent a translation table between nucleotides and amino acids, a string may represent a DNA sequence and a tuple can represent the space coordinates of an atom in a protein. But what data type do we use to represent a metabolic state of a cell? The different domains in a protein? The result of a BLAST run? What about an ecosystem?

There is a need to define our own data types, to be able to model any system, either biological or any other type. Although the functions are useful to modularize the code, they are not designed to fulfill this role. The functions cannot store states, since the values of variables only have life while the function is being executed. Other languages have their personalized data types, like “structs” in C or “record” in Pascal, but they do not have the same flexibility as the objects of languages based on OOP (like Java, C++ or Python). Objects have enough ductility to be able to model any type of system and its possible relations with other systems.

8.2 EXPLORING THE JARGON

The world of OOP has its own vocabulary. In this section I will try to clarify a few of the many new words such as class, method, instance, attributes, polymorphism, inheritance, etc. The definitions will not be exhaustive. Some of them will not even be exact, but the priority will be the understanding of the subject rather than being overly formal. Let’s remember that the objective of this book is to provide programming tools to solve biological problems. Keeping this in mind, the following definitions and their respective examples have been written.

Classes: Object Generators

A class is a template that is used to generate objects. Objects can contain data and have associated functions. A class can be a data type such as `string` or `set`, but also something more complex like `genome`, `people`, `sequences`, etc. Any object capable of being abstracted can be a class.

Instance: Particular Implementation of Class

An instance is the implementation of a class. For instance, if we have a class `Orca`, an instance can be `Willy`. Several instances from the same class can be created (for example, `Shamu`) and all are independent of each other.

Attributes or Instance Variables: Characteristics of Objects

Each object will have its own characteristics (or attributes), for example weight. Willy may have a weight different from Shamu, but in spite of having variations in their attributes, both instances, belong to the same class `Orca`. They share at least the “type of attributes.” We could create a class `dog`, with instances `Lassie`, `Laika` and `Rin-tin-tin`. This class can have the attribute `hair_color`, which is not going to be shared by instances of the `Orca` class.

Methods: Behavior of Objects

A method is a function associated with an object. Methods define how the objects “behave.” For example, the `DNA` class can have an instance `plasmid` with a **method** `translate` that allows translating an amino acid sequence into a protein. The notation in Python for this is: `plasmid.translate()`. This method is a function associated with a class. It could require as parameters a string with the DNA sequence and a dictionary with a translation table. Keeping with the `Orca` class, it could have an `eat` method.

Class Attributes: Characteristics of Classes

Attributes are variables associated with all the objects of a class. Whenever an object is created from a class, this object inherits the variable of the class. In the `Orca` class, `weight` can be a class attribute.

Inheritance: Properties Are Transmitted between the Related Classes

Classes can be related to each other and are not isolated entities. It is possible to have a *Mammal class* with common properties with the *Orca class* and the *Dog class*. For example, the method `reproduction` can be defined for the *Mammal class*. When we create the classes `Dog` and `Orca`, and define them as “children” of `Mammal`, it won’t be necessary to create for them the method `reproduction`. This method will be inherited from the parent class. Child classes may have their own unique methods, like `swim` and `run`.

Polymorphism

Polymorphism is the ability of different types of objects to respond to the same method with a different behavior. The same method, for example `feed`, is very different in the *Orca class* and the *Dog class*. Both will be called the same way but the result may be different. For example, you can iterate over a list, a dictionary, a file, and more in the same way, but the way Python handles the iteration changes for each type of object.

Encapsulation

Encapsulation is the ability to hide the internal operation of an object and leave access for the programmers only through their public methods. The term **encapsulation** is not associated with Python because this language does not have a **true encapsulation**. It is possible to make the access to certain methods difficult, but not to prevent it. It is not in the philosophy of Python to be in the way of the programmer. What it is possible to do in Python is to make clear which methods and properties are owned solely by a class and which are conceived to be shared. This behavior is also referred as pseudo-encapsulation or translucent encapsulation. It is up to the programmer to make a rational use of this option. This is called in Python: Protection by convention, not by legislation. See the section “Making Our Code Private” on page 154 for using this property.

8.3 CREATING CLASSES

Classes are the template of the objects. The syntax to create classes in Python is very simple:

```
class Name:
    [body]
```

Let's see a sample class:

```
class Square:
    def __init__(self):
        self.side = 1
```

This class (`Square`) has a method called `__init__`. It is a special method that doesn't return any value. It is executed whenever an instance of `Square` is created (or instantiated). It is used to customize a specific initial state. In this case it sets the value of the attribute `side`. Another peculiarity to consider is the word **self**, which is repeated as parameter of the method and as part of the name of the attribute. **Self** is a variable that is used to represent the instance of `Square`. It is possible to use another name instead of **self**, but **self** is used by convention. It is advisable to follow the convention because it makes our program easier to understand by other programmers.¹

To instantiate a class, you need to use function notation. This is like a function without parameters that returns a new instance of the class.

Let's see an example, the use of the `Square` class, with the creation of the instance `Bob`:

```
>>> Bob = Square() # Bob is an instance of Square.
>>> Bob.side #Let's see the value of side
1
```

¹There are also code analyzers that depend on this convention to work.

It is possible to change the value of the attribute `side` of the instance `Bob`:

```
>>> Bob.side = 5 #Assigning a new value to side
>>> Bob.side #Let's see the new value of side
5
```

This change is specific for the `Bob` instance. When new instances are created, the method `__init__` is executed again to assign the `side` value to the new instance:

```
>>> Krusty = Square()
>>> Krusty.side
1
```

If the variable `side` is a variable that must be accessible from all the instances of the class, it is advisable to use a **class variable**. These variables are shared by all the objects of the same class.

```
class Square:
    side = 1
```

This way, the value of `side` will be defined even before we create an instance of `Square`:

```
>>> Square.side
1
```

Of course if we created instances of `Square`, they will also have this value of `side`:

```
>>> Crab = Square()
>>> Crab.side
1
```

The class variables can have information on the instances. For example, it is possible to use them to control how many instances of a class have been created.

```
class Square:
    count = 0
    def __init__(self):
        Square.count += 1
        print("Object created successfully")
```

This version of `Square` can count the number of instances that have been created. Note that the `count` variable is accessed within the class as `Square.count` to distinguish itself from an instance variable, which is noted with the prefix `self.name`. Let's see how this object is used:

```
>>> Bob = Square()
Object created successfully
>>> Patrick = Square()
Object created successfully
>>> Square.count
2
```

Let's see another class:

```
class Sequence:
    transcription_table = {'A':'U', 'T':'A', 'C':'G' , 'G':'C'}
    def __init__(self, seqstring):
        self.seqstring = seqstring.upper()
    def transcription(self):
        tt = ""
        for letter in self.seqstring:
            if letter in 'ATCG':
                tt += self.transcription_table[letter]
        return tt
```

This class has two methods and one attribute. The method `__init__` is used to set the value of *seqstring* in each instance:

```
>>> dangerous_virus = Sequence('atggagagccttggttcttggtgtcaa')
>>> dangerous_virus.seqstring
'ATGGAGAGCCTTGTTCTTGGTGTCAA'
>>> harmless_virus = Sequence('aatgctactactattagtagaattgatgcc')
>>> harmless_virus.seqstring
'AATGCTACTACTATTAGTAGAATTGATGCCA'
```

The *Sequence* class also has a method called *transcription* that has as its only parameter the instance itself (represented by *self*). This parameter does not appear when the function is called, because it is implicit. Notice that the function *transcription* uses the class variable of the *transcription_table* (that is, a dictionary) to convert the sequence *seqstring* to its transcript equivalent:

```
>>> dangerous_virus.transcription()
'GCUAAGAGCUCGCGUCCUCAGAGUUUAGGA'
```

The methods can also have parameters. In order to show this, here is a new method (*restriction*) in the *Sequence* class. This method calculates how many restriction sites a sequence has for a given enzyme.² Therefore, this method will

²A restriction enzyme is a protein that recognizes a specific DNA sequence and produces a cut within the recognition zone.

require as a parameter the name of a restriction enzyme. Another difference is that this class will contain a dictionary that relates the name of the enzyme to the recognition sequence:

Listing 8.1: seqclass.py: Sequence class

```
class Sequence:
    transcription_table = {'A':'U', 'T':'A', 'C':'G', 'G':'C'}
    enz_dict = {'EcoRI':'GAATTC', 'EcoRV':'GATATC'}
    def __init__(self, seqstring):
        self.seqstring = seqstring.upper()
    def restriction(self, enz):
        try:
            enz_target = Sequence.enz_dict[enz]
            return self.seqstring.count(enz_target)
        except KeyError:
            return 0
    def transcription(self):
        tt = ""
        for letter in self.seqstring:
            if letter in 'ATCG':
                tt += self.transcription_table[letter]
        return tt
```

Using the Sequence class:

```
>>> other_virus = Sequence('atgatatcggagaggatatcggtgtcaa')
>>> other_virus.restriction('EcoRV')
2
```

8.4 INHERITANCE

Inheritance of classes implies that the new (child) class “inherits” the methods and attributes of the base class. The following is the syntax used to create a class that inherits from other class:

```
class DerivedClass(BaseClass):
    [body]
```

Following with the example of the *Orca* class that inherited from *Mammal* class:

Listing 8.2: orca.py: Orca class

```
class Orca(Mammal):
    """Docstring with class description"""
    # Properties here
    # Methods here
```

Let's see as an example a class called `Plasmid`³ that is based on the `Sequence` class. Because plasmid is a type of DNA sequence, we created the `Plasmid` class which inherits methods and properties from `Sequence`. We also defined methods and attributes that are exclusive to this new class, like `AbResDict` and `ABres`. The method `ABres` is used to know if our plasmid has resistance to a particular antibiotic, whereas the `AbResDict` attribute has the information of the regions that characterize the different antibiotic resistances.

Listing 8.3: `plasmid.py`: `Plasmid` class

```
class Plasmid(Sequence):
    ab_res_dict = {'Tet':'ctagcat', 'Amp':'CACTACTG'}
    def __init__(self, seqstring):
        Sequence.__init__(self, seqstring)
    def ab_res(self, ab):
        if self.ab_res_dict[ab] in self.seqstring:
            return True
        return False
```

Notice that within the method `__init__` of `Plasmid` we called the method `__init__` of `Sequence`. This is the way that our class inherits the attributes and methods of the “father” class. Let's see how the `Plasmid` class uses its own methods and those of its father (`Sequence`). The method `ABres` works in a way similar to `Restriction` with the difference that instead of giving back the position that we are looking for, it simply informs us if it is present or absent.

Introducing Some Biopython Objects

While there is a special section for Biopython ahead in this book, we will see some Biopython structures here to get familiar with them.

Class `IUPACAmbiguousDNA`: The class `IUPACAmbiguousDNA`⁴ is in the module `IUPAC`. It is a class that derives from `Alphabet` and holds the information regarding the IUPAC⁵ approved letters for DNA sequences. In this

³A plasmid is a DNA molecule that is independent of the chromosomal DNA of a microorganism.

⁴See <http://biopython.org/DIST/docs/api/Bio.Alphabet.IUPAC-module.html> for more information.

⁵IUPAC stands for International Union for Pure and Applied Chemistry; it is an international federation that regulates the nomenclature used in chemistry.

The IUPAC nucleic acid notation			
	Symbol	Meaning	Mnemonic
DNA Bases	G	Guanine	<u>G</u> uanine
	T	Thymine	<u>T</u> hymine
	A	Adenine	<u>A</u> denine
	C	Cytosine	<u>C</u> ytosine
Ambiguity Characters	R	G + A	pu <u>R</u> ine
	Y	T + C	p <u>Y</u> rimidine
	S	G + C	<u>S</u> trong interactions (3 H bonds)
	W	T + A	<u>W</u> eak interactions (2 H bonds)
	K	G + T	<u>K</u> eto
	M	A + C	a <u>M</u> ino
	D	G + T + A	Not-C (<u>D</u> follows C in alphabet)
	H	T + A + C	Not-G (<u>H</u> follows G)
	B	G + T + C	Not-A (<u>B</u> follows A)
	V	G + A + C	Not-T or U (<u>V</u> follows U)
	N	G + A + T + C	a <u>N</u> y

Figure 8.1 IUPAC nucleic acid notation table.

case (*IUPACAmbiguousDNA*) ambiguity is taken into account, that is, there are characters to encode nucleotides not fully determined in a given position. For example, if a nucleotide in a specific position can be A or G, it is encoded with an R (see Figure 8.1 for the complete IUPAC nucleic acid notation table). For this reason *IUPACAmbiguousDNA* has a class variable `letters` that holds the string 'GATCRYWSMKHBVDN'. At first sight it doesn't seem a very useful class, but in the class *Seq* its usefulness will be shown.

Class *IUPACUnambiguousDNA*: Like *IUPACAmbiguousDNA*, there is *IUPACUnambiguousDNA*. This class derives from the former, so it keeps its properties. The only difference is that this class again defines the `letters` attribute, with 'GATC' as the content.

Class *Seq*: In the module *Seq* there is a class called *Seq*.⁶ Objects from this class store sequence information. Up to this point we have represented sequences as strings. The problem with this approach is that the string holds only sequence information, and there is no metadata to tell us what kind of sequence it is (DNA, RNA, amino acids). In the *Seq* class, there are two parameters: `data`

⁶See <http://biopython.org/DIST/docs/api/Bio.Seq.Seq-class.html> for more information.

and **alphabet**. **Data** is a string with the sequence and **alphabet** is an object of the **alphabet** type. It contains information about the type of sequence alphabet. Another feature of this class is that it is “immutable,” that is, once a sequence is defined, it can’t be modified (just as a string). This way we are sure the sequence remains the same even after several manipulations. In order to change the sequence, we have to use a **MutableSeq** kind of object.

The **Seq** class defines several methods, as the most important: *complement* (returns the complement sequence), *reverse_complement* (returns the reverse complement sequence), *tomutable* (returns a **MutableSeq** object), and *tostring* (returns the sequence as a string). Let’s see it in action:⁷

```
>>> from Bio.Alphabet import IUPAC
>>> from Bio.Seq import Seq
>>> first_seq = Seq('GCTATGCAGC', IUPAC.unambiguous_dna)
>>> first_seq
Seq('GCTATGCAGC', IUPACUnambiguousDNA())
>>> first_seq.complement()
Seq('CGATACGTCG', IUPACUnambiguousDNA())
>>> first_seq.tostring()
'GCTATGCAGC'
```

This object has special methods that allow the programmer to work with a **Seq** type object as if it were a string:

```
>>> first_seq[:10] # slice a sequence
Seq('GCTAT', IUPACUnambiguousDNA())
>>> len(first_seq) # get the length of the sequence
10
>>> first_seq[0] # get one character
'G'
```

Class MutableSeq: This is an object very similar to **Seq**, with the main difference that its sequence can be modified. It has the same methods as **Seq**, with some methods tailored to handle mutable sequences.

We can create it from scratch or it can be made from a **Seq** object using the *tomutable* method:

```
>>> first_seq
Seq('GCTATGCAGC', IUPACUnambiguousDNA())
>>> AnotherSeq=first_seq.tomutable()
>>> AnotherSeq.extend("TTTTTTT")
```

⁷For running this code you need to install Biopython; see page 159 more more information.

```
>>> print(AnotherSeq)
MutableSeq('GCTATGCAGCTTTTTTT', IUPACUnambiguousDNA())
>>> AnotherSeq.pop()
'T'
>>> AnotherSeq.pop()
'T'
>>> print(AnotherSeq)
MutableSeq('GCTATGCAGCTTTTTT', IUPACUnambiguousDNA())
```

8.5 SPECIAL METHODS

Some methods have a special meaning. We have already seen the `__init__` method that is executed each time a new instance is created (or a new object is instantiated). Each special method is executed under a pre-established condition. The developer can modify how the object responds to each of these pre-established conditions.

Take for example the `__len__` method. This method is activated in an object each time the function `len(instance)` is called. What this method returns is up to the developer. Recall the *Sequence* class ([Listing 8.1](#)) and see what happens when you want to find out the length of a sequence:

```
>>> len(Sequence("ACGACTCTCGACGGCATCCACCCTCTCTGAGA"))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: Sequence instance has no attribute '__len__'
```

This was somehow expected. We didn't define what is the meaning of the length of *Sequence*. This object has several attributes and the interpreter has no way to know which attribute returns when `len(Sequence)` is required. The error message gives us a clue about the problem: "Sequence instance has no attribute `'__len__'`". Hence if we want to set a behavior for `len()` function, we have to define the special method attribute `__len__`:

```
def __len__(self):
    return len(self.seqstring)
```

This method must be included in the class definition (8.1).

Listing 8.4: seqclass2.py: Sequence class

```
class Sequence:
    transcription_table = {'A':'U', 'T':'A', 'C':'G', 'G':'C'}
    enz_dict = {'EcoRI':'GAATTC', 'EcoRV':'GATATC'}
```

```

def __init__(self, seqstring):
    self.seqstring = seqstring.upper()
def __len__(self):
    return len(self.seqstring)
def restriction(self, enz):
    try:
        enz_target = Sequence.enz_dict[enz]
        return self.seqstring.count(enz_target)
    except KeyError:
        return 0
def transcription(self):
    tt = ""
    for letter in self.seqstring:
        if letter in 'ATCG':
            tt += self.transcription_table[letter]
    return tt

```

Now that we have defined the `__len__` method, we can apply the function `len` to the *Sequence* objects:

```

>>> M13 = Sequence("ACGACTCTCGACGGCATCCACCCTCTCTGAGA")
>>> len(M13)
32

```

In the same way that we can control what is returned by `len()`, we can do it with other methods that can be programmed in a class. Let's see some of them:⁸

- `__str__` This method is invoked when the string representation of an object is required. This representation is obtained with `str(object)` or with `print object`. This way the programmer can choose how its object “looks.” For example, the translation table provided by Biopython, *Bio.Data.CodonTable*, is stored as a dictionary, but its representation appears as a table:

```

>>> import Bio.Data.CodonTable
>>> print(Bio.Data.CodonTable.standard_dna_table)
Table 1 Standard, SGC0

```

	T		C		A		G	
	+	-----	+	-----	+	-----	+	-----
T		TTT F		TCT S		TAT Y		TGT C
T		TTC F		TCC S		TAC Y		TGC C
T		TTA L		TCA S		TAA Stop		TGA Stop

⁸In <https://docs.python.org/3.6/reference/datamodel.html#special-method-names> there is a list of **Special methods**.

T		TTG L(s)		TCG S		TAG Stop		TGG W		G
--+-+-----+-----+-----+-----+--										
C		CTT L		CCT P		CAT H		CGT R		T
C		CTC L		CCC P		CAC H		CGC R		C
C		CTA L		CCA P		CAA Q		CGA R		A
C		CTG L(s)		CCG P		CAG Q		CGG R		G
--+-+-----+-----+-----+-----+--										
A		ATT I		ACT T		AAT N		AGT S		T
A		ATC I		ACC T		AAC N		AGC S		C
A		ATA I		ACA T		AAA K		AGA R		A
A		ATG M(s)		ACG T		AAG K		AGG R		G
--+-+-----+-----+-----+-----+--										
G		GTT V		GCT A		GAT D		GGT G		T
G		GTC V		GCC A		GAC D		GGC G		C
G		GTA V		GCA A		GAA E		GGA G		A
G		GTG V		GCG A		GAG E		GGG G		G
--+-+-----+-----+-----+-----+--										

- `__repr__` invoked with the `repr()` built-in function and when the object is entered into the interactive shell. It should look like a valid Python expression that could be used to re-create an object with the same value, when not possible, a string of the form `<...some useful description...>`. It is used mostly in debugging. See the same object as above but with `repr()` instead of `print()`:

```
>>> repr(Bio.Data.CodonTable.standard_dna_table)
'<Bio.Data.CodonTable.NCBICodonTableDNA instance at 0xb7da0c>'
```

- `__getitem__` is used to access an object sequentially or by using a subscript like `object[n]`. Each time you try to access an object as `object[n]`, `object.__getitem__(n)` is executed. This method requires two parameters: The object (usually `self`) and the index. There is a usage sample in [listing 8.6](#).
- `__iter__` allows walking over a sequence. With `__iter__` we can iterate the same way over many different objects such as dictionaries, lists, files, strings, and so on. The `for` statement calls the built-in function `iter` on the object being iterated over. `__iter__` defines how the items are returned when using the `__next__` special method. In the first example we create the *Straight* class, where its elements are returned in the same order as they are stored, while the *Reverse* class returns its elements using an inverted order:

Listing 8.5: `straight.py`: Straight and Reverse classes

```
class Straight:
```

```

    def __init__(self, data):
        self.data = data
        self.index = 0
    def __iter__(self):
        return self
    def __next__(self):
        if self.index == len(self.data):
            raise StopIteration
        answer = self.data[self.index]
        self.index += 1
        return answer

class Reverse:
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
        return self
    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index -= 1
        return self.data[self.index]

```

Let's see them in action:

```

>>> a = Straight("123")
>>> for i in a:
    print(i)

1
2
3
>>> b = reverse("123")
>>> for i in b:
    print(i)

3
2
1

```

- `__setitem__` is used to assign a value to a key (with the form *self[key]=value*). It is typically used to change the value of a dictionary key. In this case it is used to replace a character in a string:

```
def __setitem__(self, key, value):
    if len(value) == 1:
        self.seq = self.seq[:key] + value + self.seq[key+1:]
        return None
    else:
        raise ValueError
```

- `__delitem__` implements the deletion of objects of the form *self[key]*. It can be used with any object that supports the deletion of its elements.

Sequence class with some special methods:

Listing 8.6: `seqwitsm.py`: Sequence class with special methods attributes

```
class Sequence:
    transcription_table = {'A':'U', 'T':'A', 'C':'G', 'G':'C'}
    comp_table = {'A':'T', 'T':'A', 'C':'G', 'G':'C'}
    def __init__(self, seqstring):
        self.seqstring = seqstring.upper()
    def restriction(self, enz):
        enz_dict = {'EcoRI':'ACTGG', 'EcoRV':'AGTGC'}
        try:
            target = enz_dict[enz]
        except KeyError:
            raise ValueError('No such enzyme in out enzyme DB')
        return self.seqstring.count(target)
    def __getitem__(self, index):
        return self.seqstring[index]
    def __getslice__(self, low, high):
        return self.seqstring[low:high]
    def __len__(self):
        return len(self.seqstring)
    def __str__(self):
        if len(self.seqstring) >= 28:
            return '{0}...{1}'.format(self.seqstring[:25],
                                      self.seqstring[-3:])
        else:
            return self.seqstring
    def transcription(self):
        tt = ''
        for x in self.seqstring:
            if x in 'ATCG':
                tt += self.transcription_table[x]
        return tt
```

```

def complement(self):
    tt = ''
    for x in self.seqstring:
        if x in 'ATCG':
            tt += self.comp_table[x]
    return tt

```

8.5.1 Create a New Data Type Using a Built-in Data Type

We can create our own classes derived from built-in data types. To illustrate this point, see how to create a variant of the *dict type*. `Zdict` is a dictionary-like object; it behaves like a dictionary with one difference: Instead of raising an exception when trying to retrieve a value with a nonexistent key, it returns 0 (zero).

Listing 8.7: `zdict.py`: Extending dictionary class

```

1 class Zdic(dict):
2     """ A dictionary-like object that return 0 when a user
3         request a non-existent key.
4     """
5
6     def __missing__(self,x):
7         return 0

```

Code explanation: In line 1 we name the class and pass a data type as argument (`dict`). This means that the resulting class (`Zdic`) inherits from the `dict` type. From line 7 to 8 there is the definition of a special method: `__missing__`. This method is triggered when the user tries to retrieve a value with a non-existent key. It takes as an argument the value of the key, but in this case the program does not use such value (`x`) since it returns 0, disregarding of the key value:

```

>>> a = Zdic()
>>> a['blue'] = 'azul'
>>> a['red']
0

```

8.6 MAKING OUR CODE PRIVATE

At the beginning of this chapter it was highlighted that one of the characteristics of the OOP is encapsulation. Encapsulation is about programmers ignoring the internal operation of objects and only being able to see their available methods. Some of these methods that we will create will not be for “external consumption,” but they will serve as support for other methods of the class, which we do want

to be used from other sections of the program. Some languages allow the hiding of methods and properties. In the jargon, this is called “making a method private.” Python does not allow the hiding of a method, because it is one of its premises not to be in the way of the programmer. But it has a syntax that makes it difficult to access a method or a property from outside of a class. This is called **mangling** and its syntax consists of adding two underscores at the beginning (but not at the end) of the name of the method or attribute that we want to be private. Let’s see an example of a class that defines 2 methods, `a` and `__b`:

```
class TestClass:
    """A class with a "private" method (b)"""
    def a(self):
        pass
    def __b(self):
        # mangled to _TestClass__b
        pass
```

Trying to access `__b()` raises an error:

```
>>> my_object = TestClass()
>>> my_object.a()
>>> my_object.__b()
```

```
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    my_object.__b()
AttributeError: TestClass instance has no attribute '__b'
```

It is possible to access the method `a`, but not `__b`, at least not directly. The notation `object._Class__method` should be used. For example:

```
>>> my_object._TestClass__b()
```

You may be wondering what the point is of a privacy method that is not really private. On one hand, the methods that have this “semi-protection” are inherited/associated by the child classes (and the name space is not contaminated). On the other hand, when an object is explored using **dir**, this class of objects won’t be seen. An important thing to consider is that the protection offered by this notation is a convention on how to proceed more than an effective protection.

8.7 ADDITIONAL RESOURCES

- Python programming/OOP.
http://en.wikibooks.org/wiki/Python_Programming/OOP

- Introduction to OOP with Python.
<http://www.voidspace.org.uk/python/articles/OOP.shtml>
- *Dive into Python*, by Mark Pilgrim. Chapter 5, Objects and Object-Oriented.
http://diveintopython.org/object_oriented_framework
- Python objects, by Fredrik Lundh.
<http://www.effbot.org/zone/python-objects.htm>
- Java tutorial: lesson: object-oriented programming concepts.
<http://java.sun.com/docs/books/tutorial/java/concepts/>
- The seq object:
<http://biopython.org/wiki/Seq>

8.8 SELF-EVALUATION

1. Why is Python often characterized as a multi-paradigm language?
2. Name the main characteristics of Object-Oriented Programming (OOP).
3. Explain the following concepts: Inheritance, Encapsulation, and Polymorphism.
4. What is the difference between class attributes and instance attributes?
5. What is a special method attribute? Name at least four.
6. What is the difference between `__str__` and `__repr__`?
7. What is a private method? Are they really private in Python?
8. What is `self`? Inside a class definition, what is the difference between `self.var = 0` and `var = 0`?
9. Define a class that keeps track of how many instances have instantiated.
10. Define a new type based on a built-in type.

Introduction to Biopython

CONTENTS

9.1	What Is Biopython?	158
9.1.1	Project Organization	158
9.2	Installing Biopython	159
	In macOS/Linux	159
	In Windows	162
9.3	Biopython Components	162
9.3.1	Alphabet	162
9.3.2	Seq	163
	Seq Objects as a String	165
9.3.3	MutableSeq	165
9.3.4	SeqRecord	166
9.3.5	Align	167
9.3.6	AlignIO	169
	AlignInfo	170
9.3.7	ClustalW	171
	Passing Parameters to ClustalW	173
9.3.8	SeqIO	173
	Reading Sequence Files	174
	Writing Sequence Files	175
9.3.9	AlignIO	176
9.3.10	BLAST	177
	BLAST Running and Processing with Biopython	178
	Starting a BLAST Job	178
	Reading the BLAST Output	180
	What's in a BLAST Record Object?	180
9.3.11	Biological Related Data	187
9.3.12	Entrez	190
	eUtils at a Glance	190
	Biopython and eUtils	191
	eUtils: Retrieving Bibliography	191
	eUtils: Retrieving Gene Information	192
9.3.13	PDB	194
	Bio.PDB Module	195
9.3.14	PROSITE	196
9.3.15	Restriction	197
	Bio.Restriction Module	198

Analysis Class: All in One	199
9.3.16 SeqUtils	200
DNA Utils	200
Protein Utils	202
9.3.17 Sequencing	202
Phd Files	203
Ace Files	203
9.3.18 SwissProt	205
9.4 Conclusion	207
9.5 Additional Resources	207
9.6 Self-Evaluation	209

9.1 WHAT IS BIOPYTHON?

Biopython¹ is a package of useful modules to develop bioinformatics applications. Although each bioinformatics analysis is unique, there are some tasks that are repeated, constants shared between programs and standard file formats. This situation suggests the need for a package to deal with biological problems.

Biopython started as an idea in August of 1999; it was an initiative by Jeff Chang and Andrew Dalke. Although they came up with the idea, collaborators soon joined the project. Among the most active developers, Brad Chapman, Peter Cock, Michiel de Hoon, and Iddo Friedberg stand out. The project began to take code form in February 2000 and in July of the same year the first release was made. The original idea was to build a package equivalent to BioPerl which, back then, was the principal bioinformatics package. Although BioPerl may have been Biopython's inspiration, the conceptual differences between Perl and Python have given Biopython a particular way of doing things. Biopython is part of the family of open-bio projects (also known as Bio*), which, institutionally is a member of the Open Bioinformatics Foundation.²

9.1.1 Project Organization

It is an open source community project. Although the Open Bioinformatics Foundation takes care of administrative, economic, and legal aspects, its content is managed by the software developers and users.

The code is in the public domain and is available in its Github repository at <https://github.com/biopython/biopython>. Anyone can participate in the project. The procedure that you have to follow to collaborate on Biopython is similar to other open source projects. You have to use the software and then determine if it needs any additional features or if you want to modify any of the existing features. Before writing any code, my recommendation is to discuss your ideas on

¹ Available from <http://www.biopython.org>.

² <http://www.open-bio.org>

the development mailing list³ first. There you will find out if that feature had already been discussed and was rejected or if it was not included because nobody needed it until that time. In the case of a bug fix, you don't need to ask; just report it in the issue tracker,⁴ and if possible, add a solution proposal.

Due to the open nature of the project, tens of people have contributed code from diverse fields within Bioinformatics, from information theory to population genetics.

I was involved in Biopython as a user since 2002 and submitted my first contribution in 2003 with `lcc.py`, a function to calculate the local compositional complexity of a sequence. In 2004 I submitted code for melting point calculation of oligonucleotides. In 2007 I contributed with some functions for the `Checksum` module.⁵ My last submission was a patch to `Bio.Restriction` module (2017). In every case I found a supportive community, especially in the first submission when my coding skills were at a beginner level.

For more information concerning how to participate in the Biopython project, see the specific instructions at <http://biopython.org/wiki/Contributing>.

The Biopython code is developed under the Biopython License.⁶ It is very liberal and there are virtually no restrictions to its use.⁷

9.2 INSTALLING BIOPYTHON

In macOS/Linux

The following commands shows how to install Biopython under a `virtualenv`. First install `virtualenv` (if you don't have it already):

```
$ pip install virtualenv
Collecting virtualenv
  Using cached virtualenv-15.1.0-py2.py3-none-any.whl
Installing collected packages: virtualenv
Successfully installed virtualenv-15.1.0
You are using pip version 8.1.1, however version 9.0.1 is available.
You should consider upgrading via the 'pip install --upgrade pip' <=
command.
$ pip install --upgrade pip
Collecting pip
(...)
```

³http://biopython.org/wiki/Mailing_lists

⁴<https://github.com/biopython/biopython/issues>

⁵Bassi, Sebastian, and Gonzalez, Virginia. New checksum functions for Biopython. Available from Nature Precedings <<http://dx.doi.org/10.1038/npre.2007.278.1>> (2007).

⁶The license is included in the Biopython package and available online at <http://www.biopython.org/DIST/LICENSE>.

⁷The only condition imposed for using Biopython are related to publishing the copyright notice and not using the name of the contributors in advertising.

Create a virtualenv for Biopython (in this case the virtualenv is called `py4biovirtualenv`)

```
$ virtualenv py4biovirtualenv
Using base prefix '/usr'
New python executable in /home/sb/py4biovirtualenv/bin/python3
Also creating executable in /home/sb/py4biovirtualenv/bin/python
Installing setuptools, pip, wheel...done.
```

Activate the virtualenv

```
$ . py4biovirtualenv/bin/activate
(py4biovirtualenv) $
```

Inside the virtualenv, install *numpy* and then, *biopython*

```
(py4biovirtualenv) $ pip install numpy
Collecting numpy
(...)
Installing collected packages: numpy
Successfully installed numpy-1.11.2
(py4biovirtualenv) $ pip install biopython
Collecting biopython
(...)
Successfully installed biopython-1.68
(py4biovirtualenv) $ python
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import Bio
>>> Bio.__version__
'1.68'
```

If you are using Anaconda, instead of running **virtualenv**, use **conda create**. You need to do it only once.

```
$ conda create -n biopy python
Fetching package metadata: ....
Solving package specifications: .....
Package plan for installation in environment /sb/anaconda3/envs/biopy:
```

The following packages will be downloaded:

package		build
-----		-----

```

pip-9.0.1          |          py35_1          1.7 MB

```

```

(...)
#
# To activate this environment, use:
# $ source activate biopy
#
# To deactivate this environment, use:
# $ source deactivate
#

```

Once the conda environment (equivalent to a virtualenv) is created, you need to activate it. You do this each time you need to use the environment:

```

$ source activate biopy
discarding /sb/anaconda3/bin from PATH
prepending /sb/anaconda3/envs/biopy/bin to PATH
(biopy)$

```

Once in the conda environment, install Biopython using **conda** instead of **pip**:

```

(biopy)$ conda install biopython
Fetching package metadata: ....
Solving package specifications: .....
Package plan for installation in environment /sb/anaconda3/envs/biopy:

```

The following packages will be downloaded:

```

(...)

```

The following NEW packages will be INSTALLED:

```

biopython: 1.68-np111py35_0
mkl:       11.3.3-0
numpy:     1.11.2-py35_0

```

Proceed ([y]/n)?

```

(...)

```

Test that the Biopython package is installed:

```

(biopy)$ python
Python 3.5.2 |Continuum Analytics, Inc.| (default, Jul  2 2016)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux

```



```
Type "help", "copyright", "credits" or "license" for more information.
>>> import Bio
>>> Bio.__version__
'1.68'
```

In Windows

There are no official Biopython packages for 64-bit Windows (the most common Windows architecture), so the Windows installation is not so easy. The first step is to download an unofficial package from <http://www.lfd.uci.edu/~gohlke/pythonlibs>. There are a lot of files in this page, so choose carefully. It should match the Python version and microprocessor architecture you are using. For Python 3.6 in a 64-bit machine, download `biopython-1.68-cp36-cp36m-win_amd64.whl`. Once the file is downloaded, take note where it is because you will need it in the next step. Since Python has **pip** pre-installed since version 2.7.9, you can use it from the command prompt:

```
c:\Users\sab\AppData\Local\Programs\Python36\Scripts> pip install c:<=
\Users\sab\Downloads\biopython-1.68-cp36-cp36m-win_amd64.whl
```

This will install Biopython.

9.3 BIOPYTHON COMPONENTS

Biopython has several modules. Some facilitate tasks that are undertaken on a daily basis in most molecular biology laboratories while others have very specific objectives. What is “commonly used” will depend on the work environment of the reader, so the compilation that follows is based in my personal perspective on what I think it is most used.

As with all enumerations, it is arbitrary and it is possible that it would not reflect the interests of all readers. It’s sorted in didactic fashion with the intention that the first items will help you to understand the rest.

9.3.1 Alphabet

In bioinformatics we deal with alphabets. DNA has a 4-letter alphabet (A,C,T,G) while proteins have their 20 amino acids, each one represented by a letter of the alphabet. There are also special “alphabets” like the ones that contemplate ambiguity positions. These are positions where more than one nucleotide may be present. For example, the letter S may represent the nucleic acids C or G, and the letter H represents A, C, or T. This ambiguous alphabet in Biopython is called **ambiguous_dna**. Concerning the proteins, there is also an extended dictionary, which is the dictionary that contains amino acids that are not normally found in proteins⁸ (**ExtendedIUPACProtein**). Similarly, there is an extended alphabet for

⁸Selenocysteine and pyrrolysine are typical examples.

nucleotides (**ExtendedIUPACDNA**) that allows letters with modified bases. Going back to proteins, there is also a reduced alphabet that, taking into account common physicochemical properties, lumps together several amino acids into one letter.

There is even one alphabet that is not DNA or amino-acid based: **SecondaryStructure**. This alphabet represents domains like **Helix**, **Turn**, **Strand**, and **Coil**.

Alphabets defined by IUPAC are stored in Biopython as classes of the IUPAC module. The parent module (**Bio.Alphabet**) includes more general/generic cases. Here are some attributes of the alphabets:

```
>>> import Bio.Alphabet
>>> Bio.Alphabet.ThreeLetterProtein.letters
['Ala', 'Asx', 'Cys', 'Asp', 'Glu', 'Phe', 'Gly', 'His', '<=
'Ile', 'Lys', 'Leu', 'Met', 'Asn', 'Pro', 'Gln', 'Arg', '<=
'Ser', 'Thr', 'Sec', 'Val', 'Trp', 'Xaa', 'Tyr', 'Glx']
>>> from Bio.Alphabet import IUPAC
>>> IUPAC.IUPACProtein.letters
'ACDEFGHIKLMNPQRSTVWY'
>>> IUPAC.unambiguous_dna.letters
'GATC'
>>> IUPAC.ambiguous_dna.letters
'GATCRYWSMKHBVDN'
>>> IUPAC.ExtendedIUPACProtein.letters
'ACDEFGHIKLMNPQRSTVWYBXZJUO'
>>> IUPAC.ExtendedIUPACDNA.letters
'GATCBDSW'
```

Alphabets are used to define the content of a sequence. How do you know that sequence made of “CCGGGTT” is a small peptide with several cysteine, glycine, and threonine or it is a DNA fragment of cytosine, guanine, and thymine? If sequences were stored as strings, there would be no way to know what kind of sequence it is. This is why Biopython introduces **Seq** objects.

9.3.2 Seq

This object is composed of the sequence itself and an **alphabet** that defines the nature of the sequence.

Let’s create a sequence object as a DNA fragment:

```
>>> from Bio.Seq import Seq
>>> import Bio.Alphabet
>>> seq = Seq('CCGGGTT', Bio.Alphabet.IUPAC.unambiguous_dna)
```

Since this sequence (**seq**) is defined as DNA, you can apply operations that are permitted in DNA sequences. Seq objects have the **transcribe** and **translate** methods:

```
>>> seq.transcribe()
Seq('CCGGGUU', IUPACUnambiguousRNA())
>>> seq.translate()
BiopythonWarning: Partial codon, len(sequence) not a multiple of three.
Explicitly trim the sequence or add trailing N before translation.<=
This may become an error in future.
Seq('PG', IUPACProtein())
```

An RNA sequence can't be transcribed, but it can be translated:

```
>>> rna_seq = Seq('CCGGGUU', Bio.Alphabet.IUPAC.unambiguous_rna)
>>> rna_seq.transcribe()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/sb/Seq.py", line 520, in transcribe
    raise ValueError("RNA cannot be transcribed!")
ValueError: RNA cannot be transcribed!
>>> rna_seq.translate()
Seq('PG', IUPACProtein())
```

You can go back from RNA to DNA using the **back_transcribe** method.

```
>>> rna_seq.back_transcribe()
Seq('CCGGGTT', IUPACUnambiguousDNA())
```

Tip: The Transcribe Function in Biopython.

Note that the **transcribe** function may not work as expected by most biologists. This function replaces each occurrence of “T” in the sequence with a “U.” In biology, a transcription means replace each DNA nucleotide with its complementary nucleotide and reverse the resulting string. The **transcribe** function works this way because all biological publications show the non-template strand. Biopython assumes that you are giving the non-template strand to the function. The Bio.Seq module also has transcribe, back transcribe, and translate functions that can be used on Seq objects or strings:

```
>>> from Bio.Seq import translate, transcribe, back_transcribe
>>> dnaseq = 'ATGGTATAA'
>>> translate(dnaseq)
'MV*'
>>> transcribe(dnaseq)
'AUGGUAUAA'
>>> rnaseq = transcribe(dnaseq)
>>> translate(rnaseq)
```

```
'MV*'
>>> back_transcribe(rnaseq)
'ATGGTATAA'
```

Seq Objects as a String

Seq objects behave almost like a string, hence many string operations are allowed:

```
>>> seq = Seq('CCGGGTTAACGTA', Bio.Alphabet.IUPAC.unambiguous_dna)
>>> seq[:5]
Seq('CCGGG', IUPACUnambiguousDNA())
>>> len(seq)
13
>>> print(seq)
CCGGGTTAACGTA
```

This behavior is constantly evolving, so expect more string-like features in the next Biopython releases.

If you need a string representation of a Seq object, you can use the Python built-in `str()` function. There is also a `tostring()` method that still works but it is recommended only if you want to make your code compatible with older Biopython versions.

9.3.3 MutableSeq

Seq objects are not mutable. This is intended since you may want to keep your data without changes. This way, immutable seq matches Python string behavior. Attempting to modify it raises an exception:

```
>>> seq[0] = 'T'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: 'Seq' object does not support item assignment
```

This problem can be solved by generating a **MutableSeq** with the `tomutable()` method:

```
>>> mut_seq = seq.tomutable()
>>> mut_seq
MutableSeq('CCGGGTTAACGTA', IUPACUnambiguousDNA())
```

Introduce a change to test that it is mutable:

```
>>> mut_seq[0] = 'T'
>>> mut_seq
MutableSeq('TCGGGTAAACGTA', IUPACUnambiguousDNA())
```

You can change the sequence as if it were a list, with **append()**, **insert()**, **pop()** and **remove()**. There are also some methods specific for manipulating a DNA sequence:

```
>>> mut_seq.reverse()
>>> mut_seq
MutableSeq('ATGCAATTGGGCT', IUPACUnambiguousDNA())
>>> mut_seq.complement()
>>> mut_seq
MutableSeq('TACGTTAACCCGA', IUPACUnambiguousDNA())
>>> mut_seq.reverse_complement()
>>> mut_seq
MutableSeq('TCGGGTAAACGTA', IUPACUnambiguousDNA())
```

9.3.4 SeqRecord

The **Seq** class is important because it stores the main subject of study in bioinformatics: the sequence. Sometimes we need more information than the plain sequences, like the name, id, description, and cross references to external databases and annotations. For all this information related to the sequence, there is the **SeqRecord** class. In other words, a **SeqRecord** is a **Seq** object with associated metadata:

```
>>> from Bio.SeqRecord import SeqRecord
>>> SeqRecord(seq, id='001', name='MHC gene')
SeqRecord(seq=Seq('CCGGGTAAACGTA', IUPACUnambiguousDNA()), id='001', name='MHC gene', description='<unknown description>', dbxrefs=[])
```

SeqRecord has two main attributes:

id A string with an identifier. This attribute is optional but highly recommended.

seq A **Seq** object. This attribute is required.

There are some additional attributes:

name A string with the name of the sequence.

description A string with more information.

dbxrefs A list of strings; each string is a database cross reference id.

features A list of SeqFeature objects. This represents those Sequence Feature found in Genbank records. This attribute is usually populated when we retrieve a sequence from a GenBank file (using, for example, the **SeqIO** parser). It contains the sequence location, type, strand, and other variables.

annotations A dictionary with further information about the whole sequence. This attribute can't be set when initializing a **SeqRecord** object.

Creating a **SeqRecord** object from scratch:

```
>>> from Bio.SeqRecord import SeqRecord
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import generic_protein
>>> rec = SeqRecord(Seq('mdstnvrsgmksrkkkpkttviddddcmtcsacqs'
                        'klvkisditkvsldyintmrgntlacaacgssklkll',
                        generic_protein),
                    id = 'P20994.1', name = 'P20994',
                    description = 'Protein A19',
                    dbxrefs = ['Pfam:PF05077', 'InterPro:IPR007769',
                              'DIP:2186N'])
>>> rec.annotations['note'] = 'A simple note'
>>> print(rec)
ID: P20994.1
Name: P20994
Description: Protein A19
Database cross-references: Pfam:PF05077, InterPro:IPR007769, DIP<=
:2186N
Number of features: 0
/note=A simple note
Seq('mdstnvrsgmksrkkkpkttviddddcmtcsacqsklvkisditkvsldyint...kl<=
l', ProteinAlphabet())
```

To create a SeqRecord from a GenBank file, please see page 174.

9.3.5 Align

The Align module contains code for dealing with alignments. The central object of this module is the **MultipleSeqAlignment** class. This object stores sequence alignments. It is not meant for making alignments; it is supposed that the sequences are already aligned before storing the alignments in it.

Here is a simple two small peptide sequence alignment:

```
MHQAIFIYQIGYPLKSGYIQSIRSPEYDNW
|| |||||*||||||| ||
MH--IFIYQIGYALKSGYIQSIRSPEY-NW
```

This alignment can be stored in one object by using Biopython as in [Listing 9.1](#):

Listing 9.1: Using Align module

```
1 from Bio.Alphabet import generic_protein
2 from Bio.Align import MultipleSeqAlignment
3 from Bio.Seq import Seq
4 from Bio.SeqRecord import SeqRecord
5 seq1 = 'MHQAIFIYQIGYPLKSGYIQSIRSPEYDNW'
6 seq2 = 'MH--IFIYQIGYALKSGYIQSIRSPEY-NW'
7 seq_rec_1 = SeqRecord(Seq(seq1, generic_protein), id = 'asp')
8 seq_rec_2 = SeqRecord(Seq(seq2, generic_protein), id = 'unk')
9 align = MultipleSeqAlignment([seq_rec_1, seq_rec_2])
10 print(align)
```

Code explanation: The **MultipleSeqAlignment** class is instantiated in line 9. **align** is the name of the **MultipleSeqAlignment** object. Both sequences are added in the **MultipleSeqAlignment** object initialization as **SeqRecord** objects.

Output of previous code:

```
ProteinAlphabet() alignment with 2 rows and 30 columns
MHQAIFIYQIGYPLKSGYIQSIRSPEYDNW asp
MH--IFIYQIGYALKSGYIQSIRSPEY-NW unk
```

MultipleSeqAlignment can be treated as a list of sequences (or **SeqRecord** objects), it shares some of its methods. To add a new sequence to the alignment uses **append** and to add multiple sequences, it supports **extend**:

```
>>> seq3 = 'M---IFIYQIGYAAKSGYIQSIRSPEY--W'
>>> seq_rec_3 = SeqRecord(Seq(seq3, generic_protein), id = 'cas')
>>> align.append(seq_rec_3)
>>> print(align)
ProteinAlphabet() alignment with 3 rows and 30 columns
MHQAIFIYQIGYPLKSGYIQSIRSPEYDNW asp
MH--IFIYQIGYALKSGYIQSIRSPEY-NW unk
M---IFIYQIGYAAKSGYIQSIRSPEY--W cas
```

Note that the new **SeqRecord** objects must have the same length as the original alignment, and have alphabets compatible with the alignment's alphabet.

Another property in common with lists, is that you can retrieve an element (a row or a **SeqRecord** object) by using an integer index:

```
>>> align[0]
```

```
SeqRecord(seq=Seq('MHQAIFIYQIGYPLKSGYIQSIRSPEYDNW', ProteinAlphabet()), <=
  id='asp', name='<unknown name>', description='<unknown description>', <=
  dbxrefs=[])
```

Use Python's slice notation instead of an integer index to retrieve a sub-alignment:

```
>>> print(align[:2,5:11])
ProteinAlphabet() alignment with 2 rows and 6 columns
FIYQIG asp
FIYQIG unk
```

As in any Python sequence, you can get the length of the alignment with `len()`:

```
>>> len(align)
3
```

It also supports iterating over all its elements, returning a `SeqRecord` object for each sequence.

The following code calculates the isoelectric point of each sequence in the alignment:

```
>>> from Bio.SeqUtils.ProtParam import ProteinAnalysis
>>> for seq in align:
...     print(ProteinAnalysis(str(seq.seq)).isoelectric_point())
6.50421142578125
8.16033935546875
8.13848876953125
```

9.3.6 AlignIO

To read a file with one alignment, use `AlignIO.read()`. It requires two parameters: The file name (or file handle object) and the format of the alignment. Valid formats are clustal, emboss, fasta, fasta-m10, ig, maf, nexus, phylip, phylip-sequential, phylip-relaxed and stockholm. The `AlignIO.read()` method returns a **MultipleSeqAlignment** object.

```
>>> from Bio import AlignIO
>>> AlignIO.read('cas9al.fasta', 'fasta')
print(align)
SingleLetterAlphabet() alignment with 8 rows and 1407 columns
MDKKYSIGLDIGTNSVGWAVITDDYKVP SKKFKVLGNTDRHSIK...GGD J7M7J1
MDKKYSIGLDIGTNSVGWAVITDDYKVP SKKFKVLGNTDRHSIK...GGD A0A0C6FZC2
MDKKYSIGLDIGTNSVGWAVITDDYKVP SKKFKVLGNTDRHSIK...GGD A0A1C2CVQ9
MDKKYSIGLDIGTNSVGWAVITDDYKVP SKKFKVLGNTDRHSIK...GGD A0A1C2CV43
```



```
MDKKYSIGLDIGTNSVGWAVITDDYKVP SKKFKVLGNTDRHSIK...GGD Q48TU5
MDKKYSIGLDIGTNSVGWAVITDDYKVP SKKFKVLGNTDRHSIK...GGD M4YX12
MKKPYSIGLDIGTNSVGWAVVTDDYKVP AKKMKVLGNTDKSHIK...GGD A0A0E2EP65
-----...GED A0A150NVN1
```

For reading files with more than one alignment, use **AlignIO.parse()**: It takes the same arguments as **AlignIO.read()**, and it returns an iterator with all the alignments present in this file. It is meant to be used in a loop:

```
>>> from Bio import AlignIO
>>> for alignment in AlignIO.parse('example.aln', 'clustal'):
...     print(len(alignment))
```

```
1098
233
```

To write an alignment to disk, use **AlignIO.write()**. This method requires as the first parameter the **MultipleSeqAlignment** object, then it needs the same two parameters as **AlignIO.read()** (file name and format). Accepted formats are: clustal, fasta, maf, nexus, phylip, phylip-sequential, phylip-relaxed, stockholm. The **AlignIO.write()** method returns the number of alignments saved:

```
>>> from Bio import AlignIO
>>> AlignIO.write(aln, 'cas9al.phy', 'phylip')
1
```

There is a helper function to convert alignment files in one step: **AlignIO.convert()**. It takes four parameters: file name to read, format of the file to read, file name to write, and format of the file to write. It also returns the number of alignments saved:

```
>>> from Bio import AlignIO
>>> AlignIO.convert('cas9al.fasta', 'fasta', 'cas9al.aln', 'clustal')
1
```

AlignInfo

The **AlignInfo** module is used to extract information from alignment objects. It provides the **print_info_content** function, and the **SummaryInfo** and **PSSM** classes:

- **print_info_content()**:

Let's see them in action:

```

>>> from Bio import AlignIO
>>> from Bio.Align.AlignInfo import SummaryInfo
>>> from Bio.Alphabet import ProteinAlphabet
>>> align = AlignIO.read('cas9align.fasta', 'fasta')
>>> align._alphabet = ProteinAlphabet()
>>> summary = SummaryInfo(align)
>>> print(summary.information_content())
4951.072487965924
>>> summary.dumb_consensus(consensus_alpha=ProteinAlphabet())
Seq('MDKKYSIGLDIGTNSVGWAVITDDYKVPSKKFKVLGNTDRHSIKKNL...GGD',<=
ProteinAlphabet())
>>> summary.gap_consensus(consensus_alpha=ProteinAlphabet())
Seq('MDKKYSIGLDIGTNSVGWAVITDDYKVPSKKFKVLGNTDRHSIKKNL...GGD',<=
ProteinAlphabet())
>>> print(summary.alignment)
ProteinAlphabet() alignment with 8 rows and 1407 columns
MDKKYSIGLDIGTNSVGWAVITDDYKVPSKKFKVLGNTDRHSIK...GGD J7M7J1
MDKKYSIGLDIGTNSVGWAVITDDYKVPSKKFKVLGNTDRHSIK...GGD AOAOC6FZC2
MDKKYSIGLDIGTNSVGWAVITDDYKVPSKKFKVLGNTDRHSIK...GGD AOA1C2CVQ9
MDKKYSIGLDIGTNSVGWAVITDDYKVPSKKFKVLGNTDRHSIK...GGD AOA1C2CV43
MDKKYSIGLDIGTNSVGWAVITDDYKVPSKKFKVLGNTDRHSIK...GGD Q48TU5
MDKKYSIGLDIGTNSVGWAVITDDYKVPSKKFKVLGNTDRHSIK...GGD M4YX12
MKKPYSIGLDIGTNSVGWAVITDDYKVPAAKMKVLGNTDKSHIK...GGD AOA0E2EP65
-----...GED AOA15ONVN1
>>> print(summary.pos_specific_score_matrix())
-   A   C   D   E   F   G   H   I   K   L   M   N   P   Q
M  1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 7.0 0.0 0.0
D  1.0 0.0 0.0 6.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0
K  1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 7.0 0.0 0.0 0.0 0.0
K  1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 6.0 0.0 0.0 1.0
Y  1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
S  1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
I  1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 7.0 0.0 0.0 0.0 0.0 0.0
G  1.0 0.0 0.0 0.0 0.0 0.0 7.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
L  1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 7.0 0.0 0.0 0.0
D  1.0 0.0 0.0 7.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0

```

9.3.7 ClustalW

This module has classes and functions to interact with **ClustalW**.⁹ You may know **ClustalX**, a popular graphical multiple alignment program authored by

⁹This program is available from <http://www.clustal.org/download/current>.

Julie Thompson and Francois Jeanmougin. ClustalX is a graphical front-end for ClustalW, a command line multiple alignment program.

Biopython has support for Clustalw, with the **ClustalwCommandline** wrapper. This class can be used to construct the ClustalW command line:

```
>>> from Bio.Align.Applications import ClustalwCommandline
>>> clustalw_exe = 'clustalw2'
>>> ccli = ClustalwCommandline(clustalw_exe, <=
infile="input4align.fasta", outfile='.././aoutput.aln')
>>> print(ccli)
clustalw2 -infile=input4align.fasta -outfile='.././aoutput.aln
```

If the `clustalw` program is not in your system path, you have to specify its location when initializing the object. For example, if `clustalw` is in `c:\windows\program file\clustal\clustalw.exe`, replace the line

```
>>> clustalw_exe = 'clustalw2'

for

>>> clustalw_exe='c:\\windows\\program file\\clustal\\clustalw.exe'
```

To run the program, call the created instance:

```
>>> from Bio.Align.Applications import ClustalwCommandline
>>> clustalw_exe = 'clustalw2'
>>> ccli = ClustalwCommandline(clustalw_exe,
infile="input4align.fasta", outfile='.././aoutput.aln')
>>> ccli()
('\\n\\n\\n CLUSTAL 2.1 Multiple Sequence Alignments\\n\\n\\nSequence <=
format is Pearson\\nSequence 1: AGA92859.1    106 aa\\nSequence 2: <=
AML31452.1    116 aa\\nSequence 3: AAH03888.1    473 aa\\nSequence 4<=
: BAE71953.1    118 aa\\nStart of Pairwise alignments\\nAligning...<=
\\n\\nSequences (1:2) Aligned. Score: 88\\nSequences (1:3) Aligned<=
. Score: 93\\nSequences (1:4) Aligned. Score: 82\\nSequences (2:<=
(...))
[alignoutput.txt]\\n\\n', '')
```

The function returns a tuple with two values. The first value is what the program returns (also called the standard output) while the second is the error message, if any.

To process the output, read the file with **AlignIO.read()**:

```
>>> from Bio import AlignIO
>>> seqs = AlignIO.read('.././aoutput.aln', 'clustal')
```

```
>>> seqs[0]
SeqRecord(seq=Seq('-----QVQLQQSDAELVKPGASVKISCKVSG<=
YTFTDHTIH...---', SingleLetterAlphabet()), id='AGA92859.1', name<=
='<unknown name>', description='AGA92859.1', dbxrefs=[])
>>> seqs[1]
SeqRecord(seq=Seq('MEWSWVFLFFLSVTTGVHSQVQLQQSDAELVKPGASVKISCKVSG<=
YTFTDHTIH...PGK', SingleLetterAlphabet()), id='AAH03888.1', name<=
='<unknown name>', description='AAH03888.1', dbxrefs=[])
>>> seqs[2]
SeqRecord(seq=Seq('-----EVQLQESDAELVKPGASVKISCKVSG<=
YTFTDHSIH...---', SingleLetterAlphabet()), id='AML31452.1', name<=
='<unknown name>', description='AML31452.1', dbxrefs=[])
```

Passing Parameters to ClustalW

To pass more parameters to Clustalw, pass them when instantiating `ClustalwCommandline`. For example to change the *Gap opening penalty*, use *pwgapopen*:

```
>>> from Bio.Align.Applications import ClustalwCommandline
>>> clustalw_exe = 'clustalw2'
>>> ccli = ClustalwCommandline(clustalw_exe,
infile="input4align.fasta", outfile='.././aoutput.aln',
pwgapopen=5)
>>> print(ccli)
clustalw2 -infile=input4align.fasta -outfile='.././aoutput.aln <=
-pwgapopen=5
```

To see the rest of available parameters, do:

```
>>> from Bio.Align.Applications import ClustalwCommandline
>>> ccli = ClustalwCommandline()
>>> help(ccli)
```

or see the online manual at <https://goo.gl/dJwoJx> or at the API page: <http://biopython.org/DIST/docs/api/>.

9.3.8 SeqIO

`Bio.SeqIO` is a common interface to input and output sequence file formats. Sequences retrieved with this interface are passed to your program as `SeqRecord` objects. `Bio.SeqIO` can also read alignment file formats, and it will return each record as a `SeqRecord` object. To retrieve an alignment as an `Alignment` object, use the `Bio.AlignIO` module.

Reading Sequence Files

The method used for reading sequences is `parse(file_handle, format)`. Where `format` can be “fasta”, “genbank” or any other present in [Table 10.1](#). This parser returns a generator. The elements returned by this generator are of the **SeqRecord** type:

```
>>> from Bio import SeqIO
>>> f_in = open('.././samples/a19.gp')
>>> seq = SeqIO.parse(f_in, 'genbank')
>>> next(seq)
SeqRecord(seq=Seq('MGHHHHHHHHHHSSGHIDDDDKHMLEMDSTNVRSGMKSRKKKPKT<=
TVIDDDDDDC...FAS', IUPACProtein()), id='AAX78491.1', name='AAX784<=
91', description='unknown [synthetic construct]', dbxrefs=[])
```

When there is only one sequence in the file, use `SeqIO.read()` instead of `SeqIO.parse()`:

```
>>> f_in = open('.././samples/a19.gp')
>>> SeqIO.read(f_in, 'genbank')
SeqRecord(seq=Seq('MGHHHHHHHHHHSSGHIDDDDKHMLEMDSTNVRSGMKSRKKKPKT<=
TVIDDDDDDC...FAS', IUPACProtein()), id='AAX78491.1', name='AAX784<=
91', description='unknown [synthetic construct]', dbxrefs=[])
```

In [Listing 9.2](#) there is a script that reads a file full of sequences in FASTA format and displays the title and the length of each entry.

Listing 9.2: readfasta.py: Read a FASTA file

```
1 from Bio import SeqIO
2
3 FILE_IN = '.././samples/3seqs.fas'
4
5 with open(FILE_IN) as fh:
6     for record in SeqIO.parse(fh, 'fasta'):
7         id_ = record.id
8         seq = record.seq
9         print('Name: {0}, size: {1}'.format(id_, len(seq)))
```

Content of the input file (3seqs.fas):

```
>Protein-X [Simian immunodeficiency virus]
NYLNNLTVPDHNKCDNTTGRKGNA PGPCVQRTYVACH
>Protein-Y [Homo sapiens]
MEEPQSDPSVEPPLSQETFSDLWKLLENVLSPLPSQAMDDLMLSPDDIEQWFTEDPGPDA
>Protein-Z [Rattus norvegicus]
MKA AVLAVLVFLTGCQAWFEWQQDEPQSQWDRVKDFATVYVDAVKDSGRDYVSQFESST
```

[Listing 9.2](#) parses the file `3seqs.fas` and generates the following output:

```
(biopy169) $ python readfasta.py
Name: Protein-X, size: 38
Name: Protein-Y, size: 62
Name: Protein-Z, size: 60
```

TABLE 9.1 Sequence and Alignment Formats

Format name	Description	Alignment - Sequence
ace	Reads the contig sequences from an ACE assembly file.	S
clustal	Output from Clustal W or X	A
embl	The EMBL flat file format.	S
emboss	The “pairs” and “simple” alignment format from the EMBOSS tools.	A
fasta	A simple format where each record starts with an identifier line starting with a “>” character, followed by lines of sequence.	A/S
fasta-m10	Alignments output by Bill Pearson’s FASTA tools when used with the -m 10 command line option.	A
genbank	The GenBank or GenPept flat file format.	S
ig	IntelliGenetics file format, also used by MASE.	A/S
nexus	Used by MrBayes and PAUP. See also the module Bio.Nexus which can also read any phylogenetic trees in these files.	A
phd	Output from PHRED.	S
phylip	Used by the PHYLIP tools.	A
stockholm	Used by PFAM.	A
swiss	Swiss-Prot (UniProt) format.	S
tab	Simple two column tab separated sequence files.	S

Writing Sequence Files

SeqIO has a method for writing sequences: **write(iterable, file_handle, format)**. The first parameter that this function takes is an iterable object with **SeqRecord** objects (e.g., a list of **SeqRecord** objects). The second parameter is the file handle that will be used to write the sequences. The *format* argument works as in **parse**.

[Listing 9.3](#) shows how to read a file with a sequence as plain-text and write it as a FASTA sequence:

Listing 9.3: `rwfasta.py`: Read a file and write it as a FASTA sequence

```

1 from Bio import SeqIO
2 from Bio.Seq import Seq
3 from Bio.SeqRecord import SeqRecord
4 with open('NC2033.txt') as fh:
5     with open('NC2033.fasta','w') as f_out:
6         rawseq = fh.read().replace('\n','')
7         record = (SeqRecord(Seq(rawseq),'NC2033.txt','',''),)
8         SeqIO.write(record, f_out,'fasta')

```

Knowing how to read and write most biological file formats allows one to read a file with sequences in one format and write them into another format:

```

from Bio import SeqIO
fo_handle = open('myseqs.fasta','w')
readseq = SeqIO.parse(open('myseqs.gbk'), 'genbank')
SeqIO.write(readseq, fo_handle, "fasta")
fo_handle.close()

```

There are more examples of SeqIO usage in [Chapter 15](#).

9.3.9 AlignIO

AlignIO is the Input/Output interface for alignments. It works mostly as SeqIO, but instead of returning a SeqRecord object, it returns an Alignment object. It has three main methods: **read**, **parse**, and **write**. The first two methods are used for input and the last one for output.

- **read(handle,format[,sec_count])**: Take the file handle and the alignment format as arguments and return an Alignment object.

```

>>> from Bio import AlignIO
>>> fn = open('secu3.aln')
>>> align = AlignIO.read(fn, 'clustal')
>>> print(align)
SingleLetterAlphabet() alignment with 3 rows and 1098 columns
-----secu3
-----AT1G14990.1-CDS
GCTTTGCTATGCTATATGTTTATTACATTGTGCCTCTG...CAC AT1G14990.1-SEQ

```

The *sec_count* argument can be used with any file format although it is used mostly with FASTA alignments. It indicates the number of sequences per alignment, which is useful to determine if a file is only one alignment with 15 sequences or three alignments of 5 sequences.

- **write(iterable,handle,format)**: Take a set of Alignment objects, a file handle, and a file format, to write them into a file. You are expected to call this function with all alignments in `iterable` and close the file handle. The following code reads an alignment in Clustal format and writes it in Phylip format.

Listing 9.4: Alignments

```
fi = open('.././samples/example.aln')
with open('.././samples/example.phy', 'w') as fo:
    align = AlignIO.read(fi, 'clustal')
    AlignIO.write([align], fo, 'phylip')
```

9.3.10 BLAST

Basic Local Alignment Search Tool (BLAST) is a sequence similarity search program used to compare a user's query to a database of sequences. Given a DNA or amino acid sequence, the BLAST heuristic algorithm finds short matches between the two sequences and attempts to start alignments from these “hot spots.” BLAST also provides statistical information about an alignment such as the “expect” value.¹⁰ Note that BLAST is not a single program, but a family of programs. All BLAST programs search for match between sequences, but there is a specialized BLAST program for each type of sequence search. *blastn* for example, is used to search in nucleotide databases using a nucleotide sequence as input. When the database is protein (amino-acid) based and your input is a nucleotide sequence, the BLAST program you should use is *blastx*. This program translate the nucleotide input into a a protein and make a search against the protein database. See table 9.2 for a list of BLAST programs and when to use them.

BLAST is one of the most widely used bioinformatics research tools, since it has several applications. Here is a list of typical BLAST applications:

- Following the discovery of a previously unknown gene in one species, search other genomes to see if other species carry a similar gene.
- Finding functional and evolutionary relationships between sequences.
- Search for consensus regulatory patterns such as promoter signals, splicing sites and transcription factor binding sites.
- Infer protein structure based on previously crystallized proteins.
- Help identify members of gene families.

¹⁰The expect value (E) is a parameter that describes the number of hits one can “expect” to see by chance when searching a database of a particular size.

If you work in bioinformatics, chances are that you will need to run some BLAST queries or face the need to process BLAST queries generated by you or by another person. Biopython provides tools for both tasks:

TABLE 9.2 Blast programs

Program name	Query/database combination
blastn	nucleotide vs nucleotide.
blastp	protein vs protein.
blastx	translated nucleotide vs protein.
tblastn	protein vs translated nucleotide.
tblastx	translated nucleotide vs translated nucleotide

BLAST Running and Processing with Biopython

BLAST can be run online on the NCBI webserver or locally on your own computer. Running BLAST over the Internet is a good option for small jobs involving few sequences. Larger jobs tend to get aborted by the remote server with the message “CPU usage limit was exceeded.” Since NCBI BLAST is a public service, they have to put quotas on CPU usage to avoid overloading their servers. Another compelling reason to use a local version of BLAST is when you need to query a custom database. There is some flexibility regarding the database(s) you could use in the NCBI BLAST server, but it can’t accommodate every type and size of custom data.

For all these reasons, it is not uncommon for most research laboratories to run in-house BLAST searches.

Starting a BLAST Job

Biopython has a wrapper for each BLAST executable, so you can run a BLAST program from inside your script. The wrapper for **blastn** is a function called *NcbiblastnCommandline*, inside the **Bio.Blast.Applications** module. The wrapper for **blastx** is *NcbiblastxCommandline*, and so on. We will see how to use *NcbiblastnCommandline* but this can be applied to all other wrappers as well.

Here is the *NcbiblastnCommandline* syntaxis:

```
NcbiblastnCommandline(blast executable, program name, database,<=
input file, [align_view=7], [parameters])
```

This function returns a tuple with two file objects. The first one is the actual result while the second one is the BLAST error message (if any). Most parameters are self-explanatory. [Listing 9.5](#) will make it clear:

Listing 9.5: runblastn.py: Running a local NCBI BLAST

```
1 from Bio.Blast.Applications import NcbiblastnCommandline as blastn
```

```

2 BLAST_EXE = '/home/sb/opt/ncbi-blast-2.6.0+/bin/blastn'
3 f_in = 'seq3.txt'
4 b_db = 'db/samples/TAIR8cds'
5 blastn_cline = blastn(cmd=BLAST_EXE, query=f_in, db=b_db,
6                       evalue=.0005, outfmt=5)
7 rh, eh = blastn_cline()

```

The BLAST program is run in line 5. To retrieve the result, you have to read the returned file-like object `rh`, as already seen in [Chapter 5](#):

```

>>> rh.readline()
<?xml version="1.0"?>
>>> rh.readline()
'<!DOCTYPE BlastOutput PUBLIC "-//NCBI//NCBI BlastOutput/EN"<=
"http://www.ncbi.nlm.nih.gov/dtd/NCBI_BlastOutput.dtd">\n'

```

The output is in XML format. This information can be parsed using the tools learned in [Chapter 11](#) or with the tools provided by Biopython (more on this in the next section). There is also a way to avoid dealing with the XML output by forcing `NcbiblastnCommandline` to use plain text as output. This is done by using `-outfmt 0`¹¹ as an optional parameter in the command line or in the Biopython function. This will result in an easier-to-read (by a human) but hard-to-parse (by a computer) output. If the last sentence seems strange, bear with me for a few paragraphs to understand why a “human-readable” format may not be suitable for automated processing.

The `eh` filehandle stores the error message returned by `NcbiblastnCommandline`. In this case it is empty (since there was no error):

```

>>> eh.readline()
''

```

The function call in line 5 is the equivalent of entering the following statement in the command line:

```
$ ./blastn -query seq3.fasta -db db/TAIR8cds -outfmt 5
```

All parameters in this command line can be matched up with a parameter in the Biopython `NcbiblastnCommandline` function. The last parameter (`-outfmt 5`) forces the result to output XML. Other BLAST output formats like plain text and HTML tend to change from version to version, making keeping an up-to-date parser very difficult.¹² Human-readable text tends to be non structured. These reasons is why an easy to read output ends up being harder to parse.

¹¹-outfmt 0 to 4 will generate different type of human readable outputs.

¹²There is an official statement from NCBI about this: “NCBI does not advocate the use of the plain text or HTML of BLAST output as a means of accurately parsing the data.”

There are many aspects of a blast query that can be controlled via optional parameters that are appended at the end of the function call. Check this appendix at the NCBI BLAST User Manual at <https://www.ncbi.nlm.nih.gov/books/NBK279675> for more information.

Once you have the BLAST result as a file object, you may need to process it. If you plan to store the result for later processing, you need to save it:

```
>>> fh = open('testblast.xml', 'w')
>>> fh.write(rh.read())
>>> fh.close()
```

Most of the time you will need to extract some information from the BLAST output. For this purpose the NCBI XML parser, featured in the next subsection, comes in handy.

Reading the BLAST Output

Parsing the contents of a BLAST file is something any bioinformatician has to deal with. Biopython provides a useful parser in the `Bio.Blast.NCBIXML` module (called `parse`). With this parser, the programmer can extract any significant bit from a BLAST output file. This parser takes as input a file object with the BLAST result and returns an iterator for each record inside the file. In this context, `record` represents an object with all the information of each BLAST result (assuming that the BLAST file has the result of several BLAST queries inside¹³). Since it returns an iterator, you can retrieve BLAST records one by one using a *for loop*:

```
from Bio.Blast import NCBIXML
for blast_record in NCBIXML.parse(rh):
    # Do something with blast_record
```

What's in a BLAST Record Object?

Every bit of information present in a BLAST file can be retrieved from the *blast record object*. Here is the big picture: A BLAST record contains the information of the BLAST run. This information is divided in two groups. First there are fixed features such as the characteristics of the program, query sequence, and database (like program name, program version, query name, database length, name). The other group of data is related to the alignments (or hits). Each hit is the alignment between the query sequence and the target found. In turn, each alignment may have more than one HSP (High-scoring Segment Pairs). An HSP is a segment of an alignment. [Figure 9.1](#) should make these concepts more accessible.

¹³This is a bug in BLAST versions prior to 2.2.14 with the way it formats the XML results for multiple queries, so you must use newer NCBI BLAST versions.

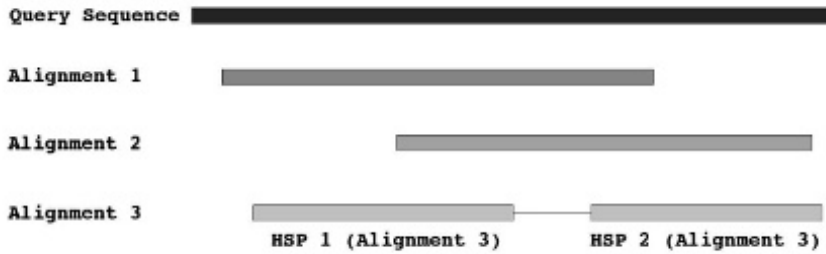


Figure 9.1 Anatomy of a BLAST result. This query sequence has three alignments. Each alignment has at least one HSP. Note that an alignment (or hit) can have more than one HSP like the “Alignment 3.”

The BLAST record object mirrors this structure. It has an *alignments* property which is a list of (BLAST) alignment objects. Each alignment object has the information of the hit (*hit_id*, *hit_definition*, *title*) and a list (*hsps*) with the information of each HSP. The data associated with each HSP is usually the most requested information from a BLAST record (like bit score, E value, position). Let’s see a plain-text BLAST output in [Listing 9.6](#):

Listing 9.6: A BLAST output

BLASTX 2.6.0+

Reference: Stephen F. Altschul, Thomas L. Madden, Alejandro A. Schaffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman (1997), "Gapped BLAST and PSI-BLAST: a new generation of protein database search programs", Nucleic Acids Res. 25:3389-3402.

Database: Non-redundant UniProtKB/SwissProt sequences
466,658 sequences; 175,602,800 total letters

Query= sample X

Length=257

	Score	E
	(Bits)	Value
Sequences producing significant alignments:		

```
P04252.1 RecName: Full=Bacterial hemoglobin; ... 93.6 1e-34
Q9RC40.1 RecName: Full=Flavohemoprotein; AltN... 66.2 2e-17
Q8ETH0.1 RecName: Full=Flavohemoprotein; AltN... 66.6 1e-16
```

```
>P04252.1 RecName: Full=Bacterial hemoglobin; AltName:
Full=Soluble cytochrome O
Length=146
```

```
Score = 93.6 bits (231), Expect(2) = 1e-34,
Method: Compositional matrix adjust.
Identities = 45/45 (100%), Positives = 45/45 (100%),
Gaps = 0/45 (0%)
Frame = +3
```

```
Query 123 VAAAHYPIVGQELLGAIKEVLGDAATDDILDAWGKAYGVIADVFI 257
          VAAAHYPIVGQELLGAIKEVLGDAATDDILDAWGKAYGVIADVFI
Sbjct 90 VAAAHYPIVGQELLGAIKEVLGDAATDDILDAWGKAYGVIADVFI 134
```

```
Score = 72.8 bits (177), Expect(2) = 1e-34,
Method: Compositional matrix adjust.
Identities = 36/36 (100%), Positives = 36/36 (100%),
Gaps = 0/36 (0%)
Frame = +2
```

```
Query 2 PKALAMTVLAAAQNIENLPAILPAVKKIIVKHCQAG 109
          PKALAMTVLAAAQNIENLPAILPAVKKIIVKHCQAG
Sbjct 54 PKALAMTVLAAAQNIENLPAILPAVKKIIVKHCQAG 89
```

```
>Q9RC40.1 RecName: Full=Flavohemoprotein; AltName:
Full=Flavohemoglobin;
AltName: Full=Hemoglobin-like protein; AltName: Full=Nitric
oxide dioxygenase; Short=NO oxygenase; Short=NOD
Length=411
```

```
Score = 66.2 bits (160), Expect(2) = 2e-17,
Method: Composition-based stats.
Identities = 28/45 (62%), Positives = 37/45 (82%),
Gaps = 0/45 (0%)
Frame = +3
```

```
Query 123 VAAAHYPIVGQELLGAIKEVLGDAATDDILDAWGKAYGVIADVFI 257
```

```

      +      YPIVG+ LL A++EVLGDAA+DD+L+AW +AY +IADVFI
Sbjct  94  IKPEQYPIVGENLLAAMREVLGDAASDDVLEAWREAYELIADVFI 138

```

```

Score = 41.6 bits (96), Expect(2) = 2e-17,
Method: Composition-based stats.
Identities = 19/32 (59%), Positives = 26/32 (81%),
Gaps = 0/32 (0%)
Frame = +2

```

```

Query   2   PKALAMTVLAAAQNIENLPAILPAVKKIAVKH  97
          P+ALA ++ AAA++I+NL AILP V +IA KH
Sbjct  58  PQALANSIYAAA EHIDNLEAILPVVSRIAHKH  89

```

```

>Q8ETH0.1 RecName: Full=Flavohemoprotein;
AltName: Full=Flavohemoglobin;
AltName: Full=Hemoglobin-like protein; AltName: Full=Nitric
oxide dioxygenase; Short=NO oxygenase; Short=NOD
Length=406

```

```

Score = 66.6 bits (161), Expect(2) = 1e-16,
Method: Composition-based stats.
Identities = 31/45 (69%), Positives = 37/45 (82%),
Gaps = 0/45 (0%)
Frame = +3

```

```

Query  123  VAAAHYPIVGQELLGAIKEVLGDAATDDILDAWGKAYGVIADVFI 257
          +      YPIVG+ LL AIKEVLGDAATD+I++AW KAY VIAD+FI
Sbjct  96  IKPEQYPIVGKYLIIAIKEVLGDAATDEIIEAW EKAYFVIADIFI 140

```

```

Score = 39.3 bits (90), Expect(2) = 1e-16,
Method: Composition-based stats.
Identities = 22/31 (71%), Positives = 23/31 (74%),
Gaps = 0/31 (0%)
Frame = +2

```

```

Query   5   KALAMTVLAAAQNIENLPAILPAVKKIAVKH  97
          KALA TV AAA NIE L  ILP VK+IA KH
Sbjct  61  KALANTVYAAAAANIEKLEEILPHVKQIAHKH  91

```

Lambda	K	H	a	alpha
0.318	0.134	0.401	0.792	4.96

Gapped

Lambda	K	H	a	alpha	sigma
0.267	0.0410	0.140	1.90	42.6	43.6

Effective search space used: 4334628608

Database: Non-redundant UniProtKB/SwissProt sequences

Posted date: May 14, 2017 12:32 PM

Number of letters in database: 175,602,800

Number of sequences in database: 466,658

Matrix: BLOSUM62

Gap Penalties: Existence: 11, Extension: 1

Neighboring words threshold: 12

Window for multiple hits: 40

[Listing 9.6](#) is the product of a *blastx* of a DNA query sequence against the SwissProt protein database, using these settings:

```
./blastx -db db/swissprot -query sampleX.fas -evalue 1e-15 -outfmt 5
```

default program settings.¹⁴ Note that there are three alignments in this result. The first and last alignments have only one HSP, while the second one, has two HSPs.

See in [Listing 9.7](#) how to retrieve the name of all the target sequence names:

Listing 9.7: BLASTparser1.py: Extract alignments title from a BLAST output

```
1 from Bio.Blast import NCBIXML
2 with open('../samples/sampleXblast.xml') as xmlfh:
3     for record in NCBIXML.parse(xmlfh):
4         for align in record.alignments:
5             print(align.title)
```

[Listing 9.7](#) (program BLASTparser1.py) produces an output like this:

```
gi|114816|sp|P04252.1|BAHG_VITST RecName: Full=Bacterial hemoglob<=
in; AltName: Full=Soluble cytochrome O
gi|52000645|sp|Q9RC40.1|HMP_BACHD RecName: Full=Flavohemoprotein;<=
```

¹⁴This listing is a reduced version of the actual output, some results were intentionally left out to avoid showing redundant data and facilitate the reader focusing on how the parser works.

```

AltName: Full=Flavohemoglobin; AltName: Full=Hemoglobin-like pro<=
tein; AltName: Full=Nitric oxide dioxygenase; Short=NO oxygenase;<=
Short=NOD
gi|52000637|sp|Q8ETH0.1|HMP_OCEIH RecName: Full=Flavohemoprotein;<=
AltName: Full=Flavohemoglobin; AltName: Full=Hemoglobin-like pro<=
tein; AltName: Full=Nitric oxide dioxygenase; Short=NO oxygenase;<=
Short=NOD

```

You can get more information from each alignment like the length of the target sequence, and other related information:

```

>>> align.length
406
>>> align.hit_id
'gi|52000637|sp|Q8ETH0.1|HMP_OCEIH'
>>> align.hit_def
'RecName: Full=Flavohemoprotein; AltName: Full=Flavohemoglobin; A<=
ltName: Full=Hemoglobin-like protein; AltName: Full=Nitric oxide <=
dioxygenase; Short=NO oxygenase; Short=NOD'
>>> align.hsps
[<Bio.Blast.Record.HSP object at 0x7fa444665eb8>, <Bio.Blast.Reco<=
rd.HSP object at 0x7fa444665ef0>]

```

hsps contain a list of *HSPs*. Each *HSP* instance, as already mentioned, has the information most users want to extract from a BLAST output. Look at an HSP:

```

>P04252.1 RecName: Full=Bacterial hemoglobin; AltName:
Full=Soluble cytochrome O
Length=146

Score = 93.6 bits (231), Expect(2) = 1e-34,
Method: Compositional matrix adjust.
Identities = 45/45 (100%), Positives = 45/45 (100%),
Gaps = 0/45 (0%)
Frame = +3

Query 123 VAAAHYPIVGQELLGAIKEVLGDAATDDILDAWGKAYGVIADVFI 257
          VAAAHYPIVGQELLGAIKEVLGDAATDDILDAWGKAYGVIADVFI
Sbjct 90  VAAAHYPIVGQELLGAIKEVLGDAATDDILDAWGKAYGVIADVFI 134

```

This is how this information can be retrieved with the BLAST parser:

```

>>> xmlfile = '../samples/sampleXblast.xml'
>>> blast_records = NCBIXML.parse(open(xmlfile))
>>> blast_record = next(blast_records)

```



```

>>> align = blast_record.alignments[0]
>>> align.hsps
[<Bio.Blast.Record.HSP object at 0x7fa444677e80>, <Bio.Blast.Record.HSP object at 0x7fa444677f28>]
>>> hsp = align.hsps[0]
>>> hsp.bits
93.5893
>>> hsp.score
231.0
>>> hsp.expect
1.06452e-34
>>> hsp.identities
45
>>> hsp.align_length
45
>>> hsp.frame
(3, 0)
>>> hsp.query_start
123
>>> hsp.query_end
257
>>> hsp.sbjct_start
90
>>> hsp.sbjct_end
134
>>> hsp.query
'VAAAHYPIVGQELLGAIKEVLGDAATDDILDAWGKAYGVIADVFI'
>>> hsp.match
'VAAAHYPIVGQELLGAIKEVLGDAATDDILDAWGKAYGVIADVFI'
>>> hsp.sbjct
'VAAAHYPIVGQELLGAIKEVLGDAATDDILDAWGKAYGVIADVFI'

```

Having this in mind, we can answer questions involving the accession numbers of the alignments with E value lesser than a threshold value? ([Listing 9.8](#)) and other questions involving any parameter in the BLAST output.

Listing 9.8: BLASTparser2.py: Extract accession numbers of sequences that have an E value less than a specific threshold

```

1 from Bio.Blast import NCBIXML
2 threshold = 0.0001
3 xmlfh = open('../samples/other.xml')
4 blast_record = next(NCBIXML.parse(open(xmlfh)))
5 for align in blast_record.alignments:

```

```

6     if align.hsps[0].expect < threshold:
7         print(align.accession)

```

Code explained: This program is very similar to [Listing 9.7](#). It retrieves the first BLAST record in the xml file (note the `next()` built-in function in line 4). This method is used because older Biopython version lacks the `NCBIXML.read()` method. If you are using Biopython 1.50, and there is only one BLAST record in the xml file, use `NCBIXML.read(open(xmlfh))`. The program walks through all *alignments* in `blast_record` (from line 5). For each alignment (`align` in line 5), it checks the expect value of the first *HSP* (line 6). If the E value is less than the threshold defined in line 2, the program prints the accession number of the alignment.

Note that when doing a BLAST search you can set an E value either from the command line or from `NcbiblastnCommandline` wrapper. Once the output is generated you can apply a filter like in [Listing 9.8](#).

Tip: BLAST Running and Processing without Biopython.

Although Biopython can be used to run and parse BLAST searches, we can get by without Biopython if necessary.

BLAST can be executed as any external program with `os.system` or with `subprocess.Popen`. Remember to set up the “m” option according to how you plan to process the output.

There are two ways to process the BLAST output. If the BLAST was set to produce the output in XML (with command line option “-m 7”), the result can be parsed with the tools shown in [Chapter 11](#). Another easier way to parse BLAST results is to use the CSV module (seen on page 90). To do this, the BLAST output should be formatted in a compatible way (with the command line option “-m 8”).

9.3.11 Biological Related Data

Biopython is not just a collection of tools. It has some biological related data. This data is included in Biopython for internal usage, like translation tables (`CodonTable.unambiguous_dna_by_name`) for the `Translate` function, and amino acid weights (`protein_weights`) for `molecular_weight` function.

Your code can also access these data. For example, the code in this interactive session accesses the dictionary that converts an “ambiguous dna value” into its possible values:

```

>>> from Bio.Data import IUPACData
>>> IUPACData.ambiguous_dna_values['M']
'AC'
>>> IUPACData.ambiguous_dna_values['H']

```

```
'ACT'
>>> IUPACData.ambiguous_dna_values['X']
'GATC'
```

Remember the protein weight calculator from [Listing 4.8](#) on page 77? With Biopython there is no need to define a dictionary with amino acid weights since such a dictionary is already included:

Listing 9.9: protwwbiopy.py: Protein weight calculator with Biopython

```
1 from Bio.Data.IUPACData import protein_weights as pw
2 protseq = raw_input('Enter your protein sequence: ')
3 total_w = 0
4 for aa in protseq:
5     total_w += pw.get(aa.upper(),0)
6 total_w -= 18*(len(protseq)-1)
7 print('The net weight is: {0}'.format(total_w))
```

The resulting program is shorter than the original version and there is no need to define a dictionary with values taken from a reference table; let Biopython developers handle this for you.

Most data available from **Bio.Data.IUPACData** and **Bio.Data.CodonTable** is presented in [Listings 9.10](#) and [9.11](#), respectively.

Listing 9.10: Data from Bio.Data.IUPACData

```
protein_letters
extended_protein_letters
ambiguous_dna_letters
unambiguous_dna_letters
ambiguous_rna_letters
unambiguous_rna_letters
ambiguous_dna_complement
ambiguous_dna_values
ambiguous_dna_weight_ranges
ambiguous_rna_complement
ambiguous_rna_values
ambiguous_rna_weight_ranges
avg_ambiguous_dna_weights
avg_ambiguous_rna_weights
avg_extended_protein_weights
extended_protein_values
extended_protein_weight_ranges
protein_weight_ranges
```

```

protein_weights
unambiguous_dna_weight_ranges
unambiguous_dna_weights
unambiguous_rna_weight_ranges
unambiguous_rna_weights

```

Listing 9.11: Data from Bio.Data.CodonTable

```

ambiguous_dna_by_id
ambiguous_dna_by_name
ambiguous_generic_by_id
ambiguous_generic_by_name
ambiguous_rna_by_id
ambiguous_rna_by_name
generic_by_id
generic_by_name
standard_dna_table
standard_rna_table
unambiguous_dna_by_id
unambiguous_dna_by_name
unambiguous_rna_by_id
unambiguous_rna_by_name

```

To get the bacterial DNA translation table:

```

>>> from Bio.Data.CodonTable import unambiguous_dna_by_id
>>> bact_trans=unambiguous_dna_by_id[11]
>>> bact_trans.forward_table['GTC']
'V'
>>> bact_trans.back_table['R']
'CGT'

```

To have a graphical representation of a translation table:

```

>>> from Bio.Data import CodonTable
>>> print CodonTable.generic_by_id[2]
Table 2 Vertebrate Mitochondrial, SGC1

```

	U	C	A	G	
U	UUU F	UCU S	UAU Y	UGU C	U
U	UUC F	UCC S	UAC Y	UGC C	C
U	UUA L	UCA S	UAA Stop	UGA W	A

U		UUG L		UCG S		UAG Stop		UGG W		G
-+-----+-----+-----+-----+--										
C		CUU L		CCU P		CAU H		CGU R		U
C		CUC L		CCC P		CAC H		CGC R		C
C		CUA L		CCA P		CAA Q		CGA R		A
C		CUG L		CCG P		CAG Q		CGG R		G
-+-----+-----+-----+-----+--										
A		AUU I(s)		ACU T		AAU N		AGU S		U
A		AUC I(s)		ACC T		AAC N		AGC S		C
A		AUA M(s)		ACA T		AAA K		AGA Stop		A
A		AUG M(s)		ACG T		AAG K		AGG Stop		G
-+-----+-----+-----+-----+--										
G		GUU V		GCU A		GAU D		GGU G		U
G		GUC V		GCC A		GAC D		GGC G		C
G		GUA V		GCA A		GAA E		GGA G		A
G		GUG V(s)		GCG A		GAG E		GGG G		G
-+-----+-----+-----+-----+--										

9.3.12 Entrez

Entrez is a search engine that integrates several health sciences databases at the National Center for Biotechnology Information (NCBI) website. From a single webpage you can search on diverse datasets such as “scientific literature, DNA and protein sequence databases, 3D protein structure and protein domain data, expression data, assemblies of complete genomes, and taxonomic information.”¹⁵

This search engine is available at <http://www.ncbi.nlm.nih.gov/sites/gquery>. You can use it online as any standard search engine, but using it from a browser is not useful for incorporating data in your scripts. That is why the NCBI created the “Entrez Programming Utilities” (eUtils). This is a server side set of tools for querying the Entrez database without a web browser and can be used for retrieving search results to include them in your own programs.

eUtils at a Glance

The user must construct a specially crafted URL. This URL should contain the name of the program to use in the NCBI web server and all required parameters (like database name and search terms). Once this URL is posted, the NCBI sends the resulting data back to the user. This data is sent, in most cases, in XML format.

The rationale behind this procedure is that the program must build the URL automatically, post it, retrieve and process the results. The URL is not supposed to be built manually, but by a program. The same is expected for the resulting XML file.

¹⁵<https://www.ncbi.nlm.nih.gov/books/NBK3807/>

It is possible to combine eUtils components to form customized data pipelines within these applications.

Biopython and eUtils

Python has tools to fetch a URL (`urllib2`) and to parse XML files (like `miniDOM`), so it could be used to access eUtils. Even using the relevant Python modules to interact with the eUtils involves a lot of work. For this reason, Biopython includes the **Entrez** module. The **Bio.Entrez** module provides functions to call every eUtils program without having to know how to build a URL or how to parse an XML file.

There are two ways to interact with the Entrez database: Query the database and retrieve the actual data. The first action can be performed with **esearch** and **egquery Bio.Entrez** functions, while the **efetch** and **esummary** functions are used for data retrieval. [Table 9.3](#) summarizes all functions available in the eUtils module.

TABLE 9.3 eUtils

Name	Description
<code>efetch</code>	Retrieves records in the requested format from a list of one or more primary IDs or from the user's environment.
<code>efetch</code>	Provides field index term counts, last update, and available links for each database.
<code>egquery</code>	Provides Entrez database counts in XML for a single search using Global Query.
<code>elink</code>	Checks for the existence of an external or Related Articles link from a list of one or more primary IDs.
<code>epost</code>	Posts a file containing a list of primary IDs for future use in the user's environment to use with subsequent search strategies.
<code>esearch</code>	Searches and retrieves primary IDs (for use in EFetch, ELink, and ESummary).
<code>espell</code>	Retrieves spelling suggestions.
<code>esummary</code>	Retrieves document summaries from a list of primary IDs or from the user's environment.
<code>read</code>	Parses the XML results returned by any of the above functions.

eUtils: Retrieving Bibliography

The following script queries PubMed through Entrez. PubMed is a search engine for MEDLINE, a literature database of life sciences and biomedical information.

Listing 9.12: `entrez1.py`: Retrieve and display data from PubMed

```

1 from Bio import Entrez
2 my_em = 'user@example.com'
3 db = "pubmed"
4 # Search de Entrez website using esearch from eUtils
5 # esearch returns a handle (called h_search)
6 h_search = Entrez.esearch(db=db, email=my_em,
7                           term='python and bioinformatics')
8 # Parse the result with Entrez.read()
9 record = Entrez.read(h_search)
10 # Get the list of Ids returned by previous search
11 res_ids = record["IdList"]
12 # For each id in the list
13 for r_id in res_ids:
14     # Get summary information for each id
15     h_summ = Entrez.esummary(db=db, id=r_id, email=my_em)
16     # Parse the result with Entrez.read()
17     summ = Entrez.read(h_summ)
18     print(summ[0]['Title'])
19     print(summ[0]['DOI'])
20     print('=====')
```

Provided that there is a working Internet connection when running [Listing 9.12](#), it outputs something like this:

```

Optimal spliced alignments of short sequence reads.
10.1093/bioinformatics/btn300
=====
Mixture models for protein structure ensembles.
10.1093/bioinformatics/btn396
=====
Contact replacement for NMR resonance assignment.
10.1093/bioinformatics/btn167
=====
```

eUtils: Retrieving Gene Information

Since eUtils is an interface for several databases, the same program that is used to retrieve bibliographic data ([Listing 9.12](#)) can be used to retrieve gene information. The key change in [Listing 9.13](#) is the database field (line 3).

Listing 9.13: `entrez2.py`: Retrieve and display data from PubMed

```

1 from Bio import Entrez
2 my_em = 'user@example.com'
3 db = "gene"
4 term = 'cobalamin synthase homo sapiens'
5 h_search = Entrez.esearch(db=db, email=my_em, term=term)
6 record = Entrez.read(h_search)
7 res_ids = record["IdList"]
8 for r_id in res_ids:
9     h_summ = Entrez.esummary(db=db, id=r_id, email=my_em)
10    summ = Entrez.read(h_summ)
11    print(r_id)
12    print(summ[0]['Description'])
13    print(summ[0]['Summary'])
14    print('=====')
```

Listing 9.13 (entrez2.py) produces a result like this:

```

326625
methylmalonic aciduria (cobalamin deficiency) cblB type
This gene encodes a protein that catalyzes the final step in <=
the conversion of vitamin B(12) into adenosylcobalamin (AdoCb<=
1), a vitamin B12-containing coenzyme for methylmalonyl-CoA m<=
utase. Mutations in the gene are the cause of vitamin B12-dep<=
endent methylmalonic aciduria linked to the cblB complementat<=
ion group. [provided by RefSeq]
=====
4524
5,10-methylenetetrahydrofolate reductase (NADPH)
Methylenetetrahydrofolate reductase (EC 1.5.1.20) catalyzes t<=
he conversion of 5,10-methylenetetrahydrofolate to 5-methylte<=
trahydrofolate, a cosubstrate for homocysteine remethylation <=
to methionine. [supplied by OMIM]
=====
(...)
```

Note that there is a number in this output that was not present in the result of [Listing 9.12](#). This number is the ID returned by the **esearch** function. This ID was used to retrieve the summary with the **esummary** function. The next code uses this ID to retrieve an actual DNA sequence:

```

>>> n = "nucleotide"
>>> handle = Entrez.efetch(db=n, id="326625", rettype='fasta')
>>> print handle.read()
>gi|326625|gb|M77599.1|HIVED82F0 Human immunodeficiency virus<=
```



```

type 1 gp120 (env) gene sequence
TTAATAGTACTTGGAATTCAACATGGGATTTAACACAACCTAATAGTACTCAGAATAAAGA
AGAAAATATCACACTCCCATGTAGAATAAAACAAATTATAAACATGTGGCAGGAAGTAGGA
AAAGCAATGTATGCCCTCCCATCAAAGGACAAATTAAATGTTTCATCAAATATTACAGGGC
TACTATTAACAAGAGATGGTGGTAATAGTGGTAACAAAAGCAACGACACCACCGAGACCTT
CAGACC

```

By changing the **rettype** parameter to “*genbank*” you can get the GenBank record instead of the plain sequence. Once the sequence is in GenBank format, it can parse it with the **SeqIO** module as seen on page 173. An alternative way to parse the results is to retrieve them in XML format and then parse them with the **Entrez.read()** function:

```

>>> handle = Entrez.efetch(db=n, id="326625", retmode='xml')
>>> record[0]['GBSeq_moltype']
'RNA'
>>> record[0]['GBSeq_sequence']
'ttaatagtacttggaattcaacatgggatttaacacaacttaatagtactcagaataaaga<=
agaaaatatcacactcccatgtagaataaaacaaattataaacatgtggcaggaagtaggaa<=
aagcaatgtatgccctcccatcaaaggacaaattaaatgttcatcaaataattacagggcta<=
ctattaacaagagatggtggtaatagtggtaacaaaagcaacgacaccaccgagaccttcag<=
acc'
>>> record[0]['GBSeq_organism']
'Human immunodeficiency virus 1'

```

9.3.13 PDB

PDB files store information regarding three-dimensional structures of molecules held at the Protein Data Bank.

This database, with more than fifty thousand records, is the reference repository of protein structural data. A PDB file stores spatial positions of atoms obtained by X-ray crystallography, NMR spectroscopy, and other experimental techniques.

This data is used by several programs, like molecule structure viewers like Deep View,¹⁶ Cn3D¹⁷ and PyMol,¹⁸ protein analysis, and structure prediction software such as MakeMultimer¹⁹ and Modeller.²⁰

Records of this database can be accessed through the RCSB webpage at <http://www.rcsb.org/pdb/home/home.do>.²¹ If you want to make your own application

¹⁶<http://spdbv.vital-it.ch>

¹⁷<http://www.ncbi.nlm.nih.gov/Structure/CN3D/cn3d.shtml>

¹⁸<http://www.pymol.org>

¹⁹<http://watcut.uwaterloo.ca/cgi-bin/makemultimer>

²⁰<http://www.salilab.org/modeller>

²¹RCSB is the **R**esearch **C**ollaboratory for **S**tructural **B**ioinformatics, the consortium in charge of the management of the PDB.

to analyze protein structure data, your program will have to be able to parse the data from PDB files. This is the role of the **Bio.pdb** module.

To effectively use the **Bio.PDB** module, you must first understand the PDB file structure. A protein structure is modeled with a top-down hierarchy. It begins with the **structure** class, down to the **atom** subclass. Intermediate orders are **model**, **chain** and **residue**. This hierarchy is also known as **SMCRA**. Some proteins don't follow this pattern, but PDB files do.²²

Bio.PDB Module

The PDB module provides the `PDBParser` class.²³ This class has the `get_structure` method. This method needs, as input, an id and a file name, and it returns a **structure** object. This **SMCRA** hierarchy can be accessed by an identifier as a key:

```
>>> from Bio.PDB.PDBParser import PDBParser
>>> pdbfn = '../..samples/1FAT.pdb'
>>> parser = PDBParser(PERMISSIVE=1)
>>> structure = parser.get_structure("1fat", pdbfn)
WARNING: Chain A is discontinuous at line 7808.
(... some warnings removed ...)
WARNING: Chain D is discontinuous at line 7870.
>>> structure.child_list
[<Model id=0>]
>>> model = structure[0]
>>> model.child_list
[<Chain id=A>, <Chain id=B>, <Chain id=C>, <Chain id=D>, <=
<Chain id= >]
>>> chain = model['B']
>>> chain.child_list[:5]
[<Residue SER het= resseq=1 icode= >, <Residue ASN het= <=
resseq=2 icode= >, <Residue ASP het= resseq=3 icode= >,<=
<Residue ILE het= resseq=4 icode= >, <Residue TYR het= <=
resseq=5 icode= >]
>>> residue = chain[4]
>>> residue.child_list
[<Atom N>, <Atom CA>, <Atom C>, <Atom O>, <Atom CB>, <=
<Atom CG1>, <Atom CG2>, <Atom CD1>]
>>> atom = residue['CB']
>>> atom.bfactor
```

²²There are some malformed PDBs out there. When the **Bio.PDB** module finds a problem it can generate an exception or a warning, depending on the *PERMISSIVE* argument (0 for no tolerance and 1 for the parser to issue warnings).

²³In some Linux installations you have to install the *python-numeric-ext* package for this module to run.

```
14.1300000000000001
>>> atom.coord
array([ 34.30699921, -1.57500005, 29.06800079], 'f')
```

The following program opens a PDB file that is compressed with **gzip**.²⁴ It scans through all chains of the protein, and in each chain it walks through all the atoms in each residue, to print the residue and atom name when there is a disordered atom:

Listing 9.14: pdb2.py: Parse a gzipped PDB file

```
1 import gzip
2 from Bio.PDB.PDBParser import PDBParser
3
4 def disorder(structure):
5     for chain in structure[0].get_list():
6         for residue in chain.get_list():
7             for atom in residue.get_list():
8                 if atom.is_disordered():
9                     print(residue, atom)
10    return None
11
12 pdbfn = '../samples/pdb1apk.ent.gz'
13 handle = gzip.GzipFile(pdbfn)
14 parser = PDBParser()
15 structure = parser.get_structure("test", handle)
16 disorder(structure)
```

9.3.14 PROSITE

PROSITE is a database of documentation entries describing protein domains, families, and functional sites as well as associated patterns and profiles used to identify them.

This database is accessed through the PROSITE site at <http://www.expasy.org/prosite> or distributed as a single plain-text file.²⁵ This file can be parsed with the *parse* function in the *Prosite* module:

```
>>> from Bio import Prosite
>>> handle = open("prosite.dat")
```

²⁴ gzip is the “standard” application used in *nix systems for file compression, but it is also available in most common platforms. It is shown in this example because most of the publicly accessible molecular data files are compressed in this format.

²⁵ Release 20.36, of 02-Sep-2008, is a 22-Mb file available at <ftp://ftp.expasy.org/databases/prosite/prosite.dat>.

```
>>> records = Prosite.parse(handle)
>>> for r in records:
    print(r.accession)
    print(r.name)
    print(r.description)
    print(r.pattern)
    print(r.created)
    print(r.pdoc)
    print("=====")
```

```
PS00001
ASN_GLYCOSYLATION
N-glycosylation site.
N-{P}-[ST]-{P}.
APR-1990
PDOC00001
=====
PS00004
CAMP_PHOSPHO_SITE
cAMP- and cGMP-dependent protein kinase phosphorylation site.
[RK] (2)-x-[ST] .
APR-1990
PDOC00004
=====
PS00005
PKC_PHOSPHO_SITE
Protein kinase C phosphorylation site.
[ST]-x-[RK] .
APR-1990
PDOC00005
=====
```

9.3.15 Restriction

Recombinant DNA technology is based on the possibility of combining DNA sequences (usually from different organisms) that would not normally occur together. This kind of biological cut and paste is accomplished by a restriction endonuclease, a special group of enzymes that works as specific molecular scissors.

The main characteristic of these enzymes is that they recognize a specific sequence of nucleotides and cut both DNA strands. When a researcher wants to introduce a cut in a known DNA sequence, he or she must first check which enzyme

has a specificity for a site inside the sequence. All available restriction enzymes are stored in a database called REBASE.²⁶

A well-known restriction enzyme is EcoRI. This enzyme recognizes the “GAATTC” sequence. So this enzyme cuts any double-stranded DNA having this sequence, like

```
CGCGAATTCGCG
GCGCTTAAGCGC
```

In this case, the restriction site is found in the middle of the top strand (marked with '-'): CGC-GAATTC-GCG. The separated pieces look like this:

```
CGC          GAATTCGCG
GCGCTTAA      GCGC
```

Bio.Restriction Module

Biopython provides tools for dealing with restriction enzymes, including enzyme information retrieved from REBASE. All restriction enzymes are available from **Restriction module**:

```
>>> from Bio import Restriction
>>> Restriction.EcoRI
EcoRI
```

Restriction enzyme objects have several methods, like **search**, that can be used to search for restriction sites in a DNA sequence:

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet.IUPAC import IUPACAmbiguousDNA
>>> alfa = IUPACAmbiguousDNA()
>>> gi1942535 = Seq('CGCGAATTCGCG', alfa)
>>> Restriction.EcoRI.search(gi1942535)
[5]
```

Note that the search function returns a list with all positions where the enzyme cuts. The position is the first nucleotide after the cut, beginning at 1 instead of 0 (as usual in other parts of Python). Another parameter in **search** is *linear*. It is defaulted to **False** and should be set as **True** when the sequence is circular.

Segments produced after a restriction can be seen with the **catalyze** function:

```
>>> Restriction.EcoRI.catalyze(gi1942535)
(Seq('CGCG', IUPACAmbiguousDNA()), Seq('AATTCGCG', <=
IUPACAmbiguousDNA()))
```

²⁶REBASE is available at <http://rebase.neb.com/rebase/rebase.html>.

To analyze several enzymes at the same time, there is the **RestrictionBatch** class:

```
>>> enz1 = Restriction.EcoRI
>>> enz2 = Restriction.HindIII
>>> batch1 = Restriction.RestrictionBatch([enz1, enz2])
>>> batch1.search(gi1942535)
{EcoRI: [5], HindIII: []}
```

The **search** function applied over a set of enzymes returns a dictionary:

```
>>> dd = batch1.search(gi1942535)
>>> dd.get(Restriction.EcoRI)
[5]
>>> dd.get(Restriction.HindIII)
[]
```

Enzymes can be added or removed as if the **RestrictionBatch** instance were a set:

```
>>> batch1.add(Restriction.EarI)
>>> batch1
RestrictionBatch(['EarI', 'EcoRI', 'HindIII'])
>>> batch1.remove(Restriction.EarI)
>>> batch1
RestrictionBatch(['EcoRI', 'HindIII'])
```

There are also some predefined sets in the *Restriction* module, like **AllEnzymes**, **CommOnly** and **NonComm**:

```
>>> batch2 = Restriction.CommOnly
```

Analysis Class: All in One

Analysis class simplifies dealing with multiple enzymes:

```
>>> an1 = Restriction.Analysis(batch1,gi1942535)
>>> an1.full()
{HindIII: [], EcoRI: [5]}
```

Up to this point, the result of the **full()** method in the **Analysis** object is the same as a **search** over a **RestrictionBatch**. **Analysis** provides:

```
>>> an1.print_that()
```

```
EcoRI      : 5.
```

Enzymes which do not cut the sequence.

```
HindIII
```

```
>>> an1.print_as('map')
```

```
>>> an1.print_that()
```

```

      5 EcoRI
      |
CGCGAATTCGCG
|||||||||||
GCGCTTAAGCGC
1                               12
```

Enzymes which do not cut the sequence.

```
HindIII
```

```
>>> an1.only_between(1,8)
```

```
{EcoRI: [5]}
```

This covers most of the functions available in the **Restriction** module. For more information please refer to the Biopython tutorial at <http://biopython.org/DIST/docs/cookbook/Restriction.html>.

9.3.16 SeqUtils

This module has several functions to deal with DNA and protein sequences, such as CG, GC skew, molecular weight, checksum algorithms, Codon Usage, Melting Temperature, and others. All functions are properly documented, so I will explain only a few functions to get the idea of how to use them.

DNA Utils

SeqUtils has plenty of functions that can be applied to DNA sequences. Let's see some of them:

GC content: The percentage of bases that are either guanine or cytosine is a parameter that affects some physical properties of the DNA molecule. It is calculated with the **GC** function:

```
>>> from Bio.SeqUtils import GC
```

```
>>> GC('gacgatcggtattcgtag')
50.0
```

DNA Melting Temperature: This can be calculated with the **MeltingTemp.Tm_staluc** function. This function implements the “nearest neighbor method”²⁷ and can be used for both DNA and RNA sequences:

```
>>> from Bio.SeqUtils import MeltingTemp
>>> MeltingTemp.Tm_staluc('tgcagtacgtatcgt')
42.211472744873447
>>> print('%.2f'%MeltingTemp.Tm_staluc('tgcagtacgtatcgt'))
42.21
```

Checksum functions: A checksum is usually a short alphanumeric string, based in an input file, mostly used to test data integrity. From any kind of data (like a text file, a DNA sequence), using an algorithm you can generate a small string (usually called a “signature”) that can represent the original data. Some programs attach checksum information to sequence information to ensure data integrity. A simple checksum is implemented by the GCG program.

This is a sequence in the gcg format:

```
ID   AB000263 standard; RNA; PRI; 368 BP.
XX
AC   AB000263;
XX
DE   Homo sapiens mRNA for prepro cortistatin like peptide.
XX
SQ   Sequence 37 BP;
AB000263 Length: 37 Check: 1149 ..
      1 acaagatgcc attgtccccc ggcctcctgc tgctgct
```

The **Check** number (1149 in this case) is derived from the sequence. If the sequence is changed, the number is (hopefully) changed. There is always a chance of a random collision, that is, when two different sequences generate the same signature. The “gcg checksum” is weak in the sense that it allows only 10000 different signatures. This is why there are some other stronger checksums like the crc32, crc64, and seguid.²⁸

All these checksums are available from the **Checksum** module. They are shown in order from the weaker to the strongest checksum algorithm.

²⁷For more information on nearest neighbor method, see the work of “Santalucia, et al. (1996) *Biochemistry* 35, 3555–3562.”

²⁸For more information on the checksums, refer to “Bassi, Sebastian and Gonzalez, Virginia. New checksum functions for Biopython.” Available from Nature Precedings <<http://dx.doi.org/10.1038/npre.2007.278.1>> (2007).


```
>>> from Bio.SeqUtils import CheckSum
>>> myseq = 'acaagatgccattgtcccccggcctcctgctgtgct'
>>> CheckSum.gcg(myseq)
1149
>>> CheckSum.crc32(myseq)
-2106438743
>>> CheckSum.crc64(myseq)
'CRC-A2CFDBE6AB3F7CFF'
>>> CheckSum.seguid(myseq)
'9V7Kf19tfPA5TntEP75YiZEm/9U'
```

Protein Utils

Protein-related functions are accessible from the **ProtParam** class. Available protein properties are *Molecular weight*, *aromaticity*, *instability index*, *flexibility*, *isoelectric point*, and *secondary structure fraction*. Function names are straightforward. See them in [Listing 9.15](#):

Listing 9.15: protparam.py: Apply PropParam functions to a group of proteins

```
1 from Bio.SeqUtils.ProtParam import ProteinAnalysis
2 from Bio.SeqUtils import ProtParamData
3 from Bio import SeqIO
4
5 with open('.././../samples/pdbaa') as fh:
6     for rec in SeqIO.parse(fh, 'fasta'):
7         myprot = ProteinAnalysis(str(rec.seq))
8         print(myprot.count_amino_acids())
9         print(myprot.get_amino_acids_percent())
10        print(myprot.molecular_weight())
11        print(myprot.aromaticity())
12        print(myprot.instability_index())
13        print(myprot.flexibility())
14        print(myprot.isoelectric_point())
15        print(myprot.secondary_structure_fraction())
16        print(myprot.protein_scale(ProtParamData.kd, 9, .4))
```

9.3.17 Sequencing

Sequencing projects usually generate *.ace* and *.phd.1* files.²⁹

²⁹This depends on sequencing technology; these files are generated by processing sequence trace chromatogram with popular sequencing processing software such as Phred, Phrap, CAP3, and Consed.

Phd Files

The DNA sequencer trace data is read by the Phred program. This program calls bases, assigns quality values to the bases, and writes the base calls and quality values to output files (with *.phd.1* extension).

The following code ([Listing 9.16](#)) shows how to extract the data from the *.phd.1* files:

Listing 9.16: phd1.py: Extract data from a .phd.1 file

```

1 import pprint
2 from Bio.Sequencing import Phd
3
4 fn = '../..samples/phd1'
5 fh = open(fn)
6 rp = Phd.RecordParser()
7 # Create an iterator
8 it = Phd.Iterator(fh, rp)
9 for r in it:
10     # All the comments are in a dictionary
11     pprint.pprint(r.comments)
12     # Sequence information
13     print('Sequence: %s' % r.seq)
14     # Quality information for each base
15     print('Quality: %s' % r.sites)
16 fh.close()

```

If you only want to extract the sequence, it is easier to use SeqIO:

```

>>> from Bio import SeqIO
>>> fn = '../..samples/phd1'
>>> fh = open(fn)
>>> seqs = SeqIO.parse(fh, 'phd')
>>> seqs = SeqIO.parse(fh, 'phd')
>>> for s in seqs:
>>>     print(s.seq)

```

```

ctccgtcggaacatcatcggatcctatcacagagtttttgaacgagttctcg
(...)

```

Ace Files

In a typical sequencing strategy, several overlapping sequences (or “reads”) are assembled electronically into one long contiguous sequence. This contiguous sequence

is called “contig” and is made with specialized programs like CAP3 and Phrap. Contig files are used for viewing or further analysis. Biopython has the **ACEParser** in the **Ace module**. For each .ace file you can get the number of contigs, number of reads, and some file information:

```
>>> from Bio.Sequencing import Ace
>>> fn='836CLEAN-100.fasta.cap.ace'
>>> acefilerecord=Ace.read(open(fn))
>>> acefilerecord.ncontigs
87
>>> acefilerecord.nreads
277
>>> acefilerecord.wa[0].info
['phrap 304_nuclsu.fasta.screen -new_ace -retain_duplicates', <=
'phrap version 0.990329']
>>> acefilerecord.wa[0].date
'040203:114710'
```

The **Ace.read** also retrieves relevant information of each contig as shown in [Listing 9.17](#).

Listing 9.17: ace.py: Retrieve data from an “ace” file

```
1 from Bio.Sequencing import Ace
2
3 fn = '../..samples/contig1.ace'
4 acefilerecord = Ace.read(open(fn))
5
6 # For each contig:
7 for ctg in acefilerecord.contigs:
8     print('=====')
9     print('Contig name: %s'%ctg.name)
10    print('Bases: %s'%ctg.nbases)
11    print('Reads: %s'%ctg.nreads)
12    print('Segments: %s'%ctg.nsegments)
13    print('Sequence: %s'%ctg.sequence)
14    print('Quality: %s'%ctg.quality)
15    # For each read in contig:
16    for read in ctg.reads:
17        print('Read name: %s'%read.rd.name)
18        print('Align start: %s'%read.qa.align_clipping_start)
19        print('Align end: %s'%read.qa.align_clipping_end)
20        print('Qual start: %s'%read.qa.qual_clipping_start)
21        print('Qual end: %s'%read.qa.qual_clipping_end)
```

```

22         print('Read sequence: %s'%read.rd.sequence)
23         print('=====')
```

9.3.18 SwissProt

SwissProt³⁰ is a hand-annotated protein sequence database. It is maintained collaboratively by the Swiss Institute for Bioinformatics (SIB) and the European Bioinformatics Institute (EBI), forming the UniProt consortium. It is known for its reliable protein sequences associated with a high level of annotation, and is the reference database for proteins. As of September 2008 it has almost 400,000 entries, while the whole UniProt database has more than 6,000,000 records. Its reduced size is due to its hand-curation process.

SwissProt files are text files structured so as to be usable by human readers as well as by computer programs. Specifications for this file format are available at <http://www.expasy.org/sprot/userman.html>, but there is no need to know its internals to parse it with Biopython.

A sample SwissProt file is shown below:³¹

```

ID  6PGL_ECOLC                      Reviewed;           331 AA.
AC  B1IXL9;
DT  20-MAY-2008, integrated into UniProtKB/Swiss-Prot.
DT  29-APR-2008, sequence version 1.
DT  02-SEP-2008, entry version 5.
DE  RecName: Full=6-phosphogluconolactonase;
DE      Short=6-P-gluconolactonase;
DE      EC=3.1.1.31;
GN  Name=pgl; OrderedLocusNames=EcolC_2895;
OS  Escherichia coli (strain ATCC 8739 / DSM 1576 / Crooks).
OC  Bacteria; Proteobacteria; Gammaproteobacteria; Enterobacteriales;
OC  Enterobacteriaceae; Escherichia.
OX  NCBI_TaxID=481805;
RN  [1]
RP  NUCLEOTIDE SEQUENCE [LARGE SCALE GENOMIC DNA].
RA  Copeland A., Lucas S., Lapidus A., Glavina del Rio T., Dalin E.,
RA  Tice H., Bruce D., Goodwin L., Pitluck S., Kiss H., Brettin T.;
RT  "Complete sequence of Escherichia coli C str. ATCC 8739.";
RL  Submitted (FEB-2008) to the EMBL/GenBank/DDBJ databases.
CC  -!- FUNCTION: Catalyzes the hydrolysis of 6-phosphogluconolactone
CC      to 6-phosphogluconate (By similarity).
CC  -!- CATALYTIC ACTIVITY: 6-phospho-D-glucono-1,5-lactone + H(2)O
```

³⁰<http://www.expasy.org/sprot>

³¹This file is slightly modified to fit in this page; the original file can be retrieved from <http://www.expasy.org/uniprot/B1IXL9.txt>.

```

CC      = 6-phospho-D-gluconate.
CC      -!- PATHWAY: Carbohydrate degradation; pentose phosphate pathway;
CC          D-ribulose 5-phosphate from D-glucose 6-phosphate (oxidative
CC          stage): step 2/3.
CC      -!- SIMILARITY: Belongs to the cycloisomerase 2 family.
CC      -----
CC      Copyrighted by the UniProt Consortium, see
CC      http://www.uniprot.org/terms Distributed under the Creative
CC      Commons Attribution-NoDerivs License
CC      -----
DR      EMBL; CP000946; ACA78522.1; -; Genomic_DNA.
DR      RefSeq; YP_001725849.1; -.
DR      GeneID; 6065358; -.
DR      GenomeReviews; CP000946_GR; EcolC_2895.
DR      KEGG; ecl:EcolC_2895; -.
DR      GO; GO:0017057; F:6-phosphogluconolactonase activity; IEA:HAMAP.
DR      GO; GO:0006006; P:glucose metabolic process; IEA:HAMAP.
DR      HAMAP; MF_01605; -; 1.
DR      InterPro; IPR015943; WD40/YVTN_repeat-like.
DR      Gene3D; G3DSA:2.130.10.10; WD40/YVTN_repeat-like; 1.
PE      3: Inferred from homology;
KW      Carbohydrate metabolism; Complete proteome; Glucose metabolism;
KW      Hydrolase.
FT      CHAIN           1       331           6-phosphogluconolactonase.
FT                                     /FTId=PRO_1000088029.
SQ      SEQUENCE   331 AA;  36308 MW;  D731044CFCF31A8F CRC64;
      MKQTVYIASP ESQQIHVWNL NHEGALTLTQ VVDVPGQVQP MVSPPDKRYL YVGVRPEFRV
      LAYRIAPDDG ALTFAAESAL PGSPTHISTD HQGQFVFGVS YNAGNVSVTR LEDGLPVGVV
      DVVEGLDGCH SANISPDNRT LWVPALKQDR ICLFTVSDDG HLVAQDPAEV TTVEGAGPRH
      MVFHPNEQYA YCVNELNSSV DVWELKDPHG NIECVQTLDM MPENFSDTRW AADIHITPDG
      RHLIACDRTA SLITVFSVSE DGSVLSKEGF QPTETQPRGF NVDHSGKYLI AAGQKSHHIS
      VYEIVGEQGL LHEKGRYAVG QGPMWVVVNA H
//

```

The [Listing 9.18](#) shows how to retrieve data from a SwissProt file with multiple records:

Listing 9.18: Retrieve data from a SwissProt file

```

1 from Bio import SwissProt
2 with open('../samples/spfile.txt') as fh:
3     records = SwissProt.parse(fh)
4     for record in records:
5         print('Entry name: %s' % record.entry_name)

```

```

6         print('Accession(s): %s' % ', '.join(record.accessions))
7         print('Keywords: %s' % ', '.join(record.keywords))
8         print('Sequence: %s' % record.sequence)

```

The [Listing 9.19](#) shows all attributes in records parsed by **SwissProt** module:

Listing 9.19: Attributes of a SwissProt record

```

1 from Bio import SwissProt
2 with open('../samples/spfile.txt') as fh:
3     record = next(SwissProt.parse(fh))
4     for att in dir(record):
5         if not att.startswith('__'):
6             print(att, getattr(record,att))

```

9.4 CONCLUSION

Most used Biopython features were covered in this chapter. Following the code samples presented here and the full programs in Section III should give you insight on how to use Biopython. You should also learn how to use the Python built-in help since online documentation tends to be more up to date than anything in print. Biopython development happens at a fast pace. So fast that this chapter was rewritten several times while I was working on it. The best way to keep updated with Biopython development is to subscribe to the Biopython development mailing list and receive the RSS feed from the code repository.

9.5 ADDITIONAL RESOURCES

- Chang J., Chapman B., Friedberg I., Hamelryck T., de Hoon M., Cock P., and Antão, T. Biopython tutorial and cookbook.
<http://www.biopython.org/DIST/docs/tutorial/Tutorial.html> or <http://www.biopython.org/DIST/docs/tutorial/Tutorial.pdf>.
- Hamelryck T, and Manderick B., PDB file parser and structure class implemented in Python. *Bioinformatics*. 2003 Nov 22;19(17):2308–10.
<https://www.ncbi.nlm.nih.gov/pubmed/14630660>
- Sohm, F., Manual in cookbook style on using the Restriction module.
<http://biopython.org/DIST/docs/cookbook/Restriction.html>
- Wu C.H., Apweiler R., Bairoch A., Natale D.A, Barker W.C., Boeckmann B., Ferro S., Gasteiger E., Huang H., Lopez R., Magrane M., Martin M.J., Mazumder R., O'Donovan C., Redaschi N. and Suzek B. (2006). The Universal Protein Resource (UniProt): An expanding universe of protein information. *Nucleic Acids Research* 34: D187–D191.

- Magrane M., and Apweiler R. (2002). Organisation and standardisation of information in Swiss-Prot and TrEMBL. *Data Science Journal* 1(1): 13–18.
<http://datascience.codata.org/articles/10.2481/dsj.1.13/galley/168/download/>
- Benson Dennis A., Karsch-Mizrachi Ilene, Lipman David J., Ostell James, and Wheeler David L.. GenBank. *Nucleic Acids Res.* 2008 January; 36(Database issue): D25–D30.
<http://dx.doi.org/10.1093/nar/gkm929>
- Larkin M.A., Blackshields G., Brown N.P., Chenna R., McGettigan P.A., McWilliam H., Valentin F., Wallace I.M., Wilm A., Lopez R., Thompson J.D., Gibson T.J., Higgins D.G.. Clustal W and Clustal X version 2.0. *Bioinformatics*. 2007 Nov 1;23(21):2947-8. Epub 2007 Sep 10.
- Wikipedia contributors. Restriction enzyme. Wikipedia, The Free Encyclopedia. February 13, 2009, 16:44 UTC.
http://en.wikipedia.org/wiki/Restriction_enzyme.
- EFetch for Sequence and other molecular biology databases.
<https://www.ncbi.nlm.nih.gov/books/NBK25499/#chapter4.EFetch>
- Cock P. Clever tricks with NCBI Entrez EInfo (& Biopython).
<https://news.open-bio.org/2009/06/21/ncbi-einfo-biopython/>

9.6 SELF-EVALUATION

1. What is an Alphabet in Biopython? Name at least four.
2. Describe Seq and SeqRecord objects.
3. What advantage provides a Seq object over a string?
4. Seq object provides some string operations. Why?
5. What is a MutableSeq object?
6. What is the relation between the Align and ClustalW modules?
7. Name the methods of the SeqIO module.
8. Why there is a comma near the end of line 7 in code 9.3?
9. Name five functions found in SeqUtils.
10. What kind of sequence files can be read with the **Sequencing** module?
11. What module would you use to retrieve data from the NCBI web server?
12. Make a program to count all ordered atoms in a PDB file. The PDB file must be passed to the program on the command line, in the form: `program.py file.pdb`.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

II

Advanced Topics



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Web Applications

CONTENTS

10.1	Introduction to Python on the Web	213
10.2	CGI in Python	214
10.2.1	Configuring a Web Server for CGI	215
10.2.2	Testing the Server with Our Script	215
	Sending Data to a CGI Program	216
10.2.3	Web Program to Calculate the Net Charge of a Protein (CGI version)	219
10.3	WSGI	221
10.3.1	Bottle: A Python Web Framework for WSGI	222
10.3.2	Installing Bottle	223
10.3.3	Minimal Bottle Application	223
10.3.4	Bottle Components	224
	Routes	224
	URL with Variable Parts	225
	Getting Data: request	225
	Templates	226
	Static Files	228
10.3.5	Web Program to Calculate the Net Charge of a Protein (Bottle Version)	229
10.3.6	Installing a WSGI Program in Apache	232
10.4	Alternative Options for Making Python-Based Dynamic Web Sites ...	232
10.5	Some Words about Script Security	232
10.6	Where to Host Python Programs	234
10.7	Additional Resources	235
10.8	Self-Evaluation	236

10.1 INTRODUCTION TO PYTHON ON THE WEB

We have just seen how to run programs locally. This chapter shows how to port them to the web.

The main advantage of making a program available on the web is that it can reach more users without the need for them to install a copy of the program and to have a Python installation. Sometimes the program accesses demanding resources like huge databases that can't be installed on the end user's hard drive.

To make web applications you need tools other than Python, like HTML, CSS, JS, web server management, and more. Those topics are beyond the scope of this book, for which reason I recommend that you read up on them if you have never designed a web page before. Knowing the basics of HTML has special importance as most IT Labs have staff dedicated to the setup and maintenance of the web servers, but the HTML design is something that they will rarely do for you. For more information on HTML, please see the “Additional Resources” section. Regarding the web server, Python provides one that is useful for development and testing, but not for use in production. In this case, you should use a stand-alone server program like **Apache** or **Nginx**; since **Apache** is by far the most popular, this book will cover how to set it up. It is common that the web server is provided by your institution, but it is also common now that the IT department provides a virtual machine where you should install all required software instead of only your app.

There are several ways to use Python on a web server, **CGI** (**C**ommon **G**ateway **I**nterface), **mod_python**, and **WSGI** (**W**eb **S**erver **G**ateway **I**nterface). CGI is the oldest method to run dynamic content in a web page. The first web servers only showed static HTML, until the **CGI** protocol was defined in 1993. It is still in use today and some hosting companies even offer **CGI** as the only option to make interactive web servers. As an advantage, it is the easiest to configure and is available on almost all web servers without having to install additional software. It is essentially a protocol to connect an application, written in any language, with a web server. **mod_python** in particular consists of an Apache Module that integrates Python with the web server. The advantage of this approach is the fast script execution, since the Python interpreter is loaded with the Web server. WSGI, in turn, is a “specification for web servers and application servers to communicate with web applications.” Since it is a specification, there are several implementations. The main advantage of WSGI is that once you have made a WSGI application, it can be deployed in any WSGI-compatible server¹ (or even using the Python provided web server). As in **mod_python**, the execution speed of WSGI-based programs is better than CGI, because there is no overhead for starting the Python interpreter on each request.

10.2 CGI IN PYTHON

For this section, I assume you already have an Apache web server running. If not, it can be installed in Debian/Ubuntu-based Linux distributions with:

```
$ sudo apt-get install apache2
```

As an alternative, you can hire any web hosting plan; most of them have Apache and CGI pre-installed. For more information on web hosting please see page 234.

¹For a comparison on WSGI web servers check out this article: <https://www.digitalocean.com/community/tutorials/a-comparison-of-web-servers-for-python-based-web-applications>.

10.2.1 Configuring a Web Server for CGI

In the server configuration file² there should be specifications that scripts can be executed via CGI, in which directories, and how they will be named.

If the scripts will be located at `/var/www/apache2-default/cgi-bin`, we have to include the following lines in the server's configuration file.

```
<Directory /var/www/apache2-default/cgi-bin>
    Options +ExecCGI
</Directory>
```

Add the following line in the config file to specify that the executable scripts are those that have the file extension `.py`

```
AddHandler cgi-script .py
```

If the configuration file already has a line with the file extensions registered, you only need to add `.py` to it.

```
AddHandler cgi-script .cgi .pl .py
```

Finally we have to configure the `ScriptAlias` variable. It requires the path that the user will see in the URL and the path where the scripts are stored.

```
ScriptAlias /cgi-bin/ /var/www/apache2-default/cgi-bin/
```

This is all there is to the server configuration file. The only thing that is left to do is make sure that the script has the Apache user permissions. From the server's terminal, enter:

```
chmod a+x MyScript.py
```

If you only have FTP access, use an FTP client to set the permissions.

10.2.2 Testing the Server with Our Script

The following code can be used to confirm that the server is ready to execute CGI programs:

Listing 10.1: `firstcgi.py`: First CGI script

```
1 #!/usr/bin/env python
2 print("Content-Type: text/html\n")
3 print("<html><head><title>Test page</title></head><body>")
4 print("<h1>HELLO WORLD!</h1>")
5 print("</body></html>")
```

²In Apache web server, in most cases the configuration file is [httpd.conf](#) or [apache2.conf](#) and it is located at `/etc/apache2` directory. This can change on each installation.

Code explanation: The first line indicates the location of the Python interpreter. Usually, this line is optional and it is added only when we want to run the script directly without first having to call the Python interpreter. For CGI programs, this line is mandatory.³ The second line is important for the web server to know that it is going to be sent an HTML page. We have to send the string `Content-Type/html` followed by two carriage returns. Although on line 2 there is only one implicit carriage return (`\n`), the other one is added by the `print` command. The rest of the program is similar to the others that we've done up to this point, the difference being that we print HTML code to be read by the browser.

If we upload this program to a web server and then access the page with our browser, the results we will see will be similar to [Figure 10.2.2](#). If everything goes well, we won't see the content of the file but the product of its execution on the server. This product (an HTML page) will be rendered by the web browser (see [Figure 10.1](#)).



Figure 10.1 Our first CGI.

Take into account that in order to test our pages we need them to be processed by a web server, and not open them directly from our hard drive. In this case we will have as a result what you see in [Figure 10.2](#), instead of the page rendered by the web browser.

Sending Data to a CGI Program

The previous program is not very useful, it is just a static page that doesn't accept any parameters from the user. Let's see an example of a minimalist HTML form that sends data to a Python program that will use this data.

The first step is to design the form. In this case we will create a simple form with one field and it will be saved as `greeting.html`:

³If you don't know the path to the Python interpreter, ask the system administrator to install your script. Another option, if you have access to the server command line, is to execute `whereis python`.

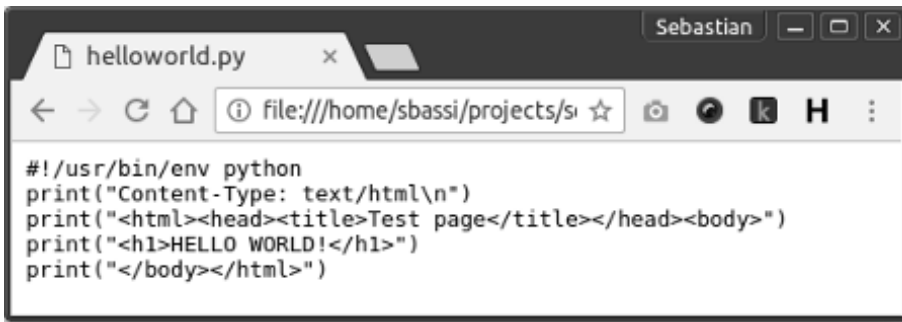


Figure 10.2 CGI accessed from local disk instead from a web server.

Listing 10.2: greeting.html: HTML front end to send data to a CGI program

```

1 <html><head><title>Very Simple Form</title></head>
2 <body>
3 <form action='cgi-bin/greeting.py' method='post'>
4 Your name: <input type='text' name='username'> <p>
5 <input type='submit' value='Send'>
6 </form></body></html>

```

Code explained: There are two important features to note on this small form. Line 3 is specified where the program that is going to process the data is located (`cgi-bin/greeting.py`). On line 4 there is the field that the user has to fill (“text” type), with an associated variable (`username`). This variable name is important because the information entered by the user will be bound to this name. The form looks like the one in [Figure 10.2.2](#).

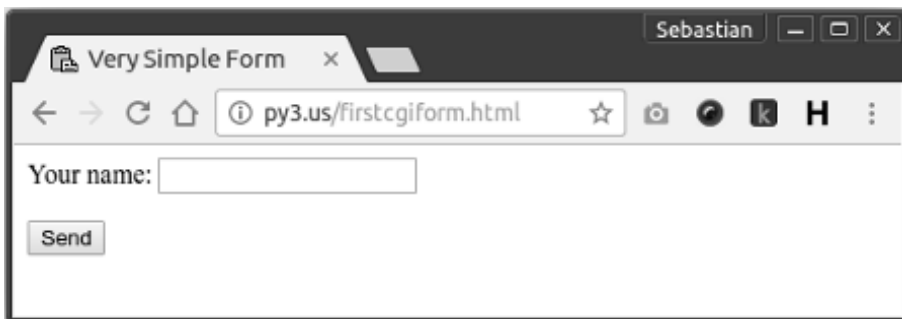


Figure 10.3 greeting.html: A very simple form.

Let’s see how to write the code that will accept data sent by the form and from it will build a Web page “on the fly.”

Listing 10.3: `greeting.py`: CGI program that processes the form in `greeting.html`.

```

1 #!/usr/bin/env python
2 import cgi
3 print("Content-Type: text/html\n")
4 form = cgi.FieldStorage()
5 name = form.getvalue("username","NN")[:10]
6 print("<html><head><title>A CGI script</title></head>")
7 print("<body><h2>Hello {0}</h2></body></html>".format(name))

```

Code explained: On line 4 we create an instance (`form`) from the class `cgi.FieldStorage`. This class takes the values sent by the form and make them accessible in a dictionary-like fashion. On the next line (5), we access the data sent by the form, and also trim the number of characters we are going to pass to the print function; this is done to mitigate a potential security problem.⁴ The `getvalue` method takes as a necessary argument, the name of the field whose content we want to access. The second argument is optional and indicates which value will be returned in case the wanted field is blank. Take note that this is similar to the `get` dictionary function. From line 6 forward, the program prints the HTML code using the content of the variable. This is the code that will be rendered in the browser.

In summary, we used the web form in [Listing 10.2](#) to enter a name and press “Send.” This sends the data. It is then read by the program thanks to the `cgi.FieldStorage` class and referenced as a variable name that is used in the program to generate a web page. See the output in [Figure 10.5](#).

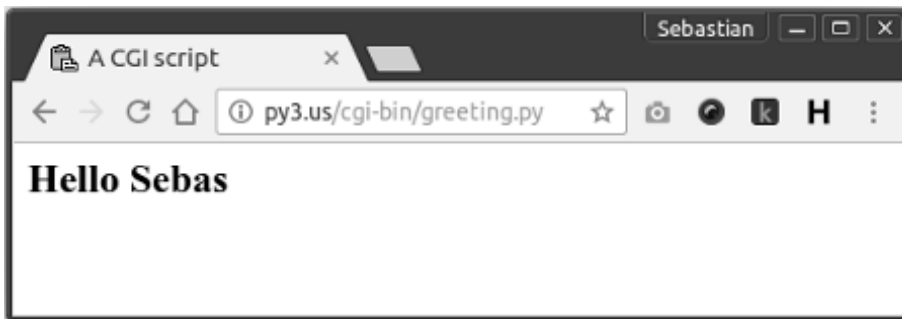


Figure 10.4 Output of CGI program that processes `greeting.html`.

⁴For more information on securing web sites please see page 232.

10.2.3 Web Program to Calculate the Net Charge of a Protein (CGI version)

Using the code from [Listing 4.14](#), we can easily adapt it to use from a web page. As a first step we need to design a form where a user can enter the data. This is a proposed form:

Listing 10.4: protcharge.html: HTML front end to send data to a CGI program

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head><meta charset="utf-8">
4     <title>Protein Charge Calculator</title>
5     <link href="css/bootstrap.min.css" rel="stylesheet">
6   </head>
7   <body style="background-color:#e7f5f5;">
8     <div class="container"><h2>Protein Charge Calculator</h2>
9     <form action="/cgi-bin/protcharge.py" method="post">
10       <div class="row">
11         <div class="col-sm-8">
12           <div class="form-group">
13             <label for="aaseq">Enter the amino-acid sequence:</label>
14             <textarea name="aaseq" rows="5" cols="40"></textarea>
15           </div>
16         </div>
17       </div>
18       <div class="row">
19         <div class="col-sm-8">
20           <div class="form-group">
21             <label for="prop">Do you want to see the proportion of
22               charged amino-acid?</label>
23             <div class="radio">
24               <label>
25                 <input type="radio" name="prop" value="y">Yes
26               </label>
27             </div>
28             <div class="radio">
29               <label>
30                 <input type="radio" name="prop" value="n">No
31               </label>
32             </div>
33             <label for="title">Job title (optional):</label>
34             <input type="text" size="30" name="title" value="">
35             <br>
36             <button type="submit" class="btn btn-primary">Send

```

```

37     </button>
38 </div>
39 </div>
40 </div>
41 </form>
42 </div>
43 </body>
44 </html>

```

Figure 10.5 shows how the form in Listing 10.4 is rendered in a web browser.

Figure 10.5 Form protcharge.html ready to be submitted.

Below is the code (`protcharge.py`) that will be called when the form is used:

Listing 10.5: `protcharge.py`: Back-end code to calculate the net charge of a protein and proportion of charged amino acid

```

1 #!/usr/bin/env python
2 import cgi, cgiib

```

```

3
4 def chargeandprop(aa_seq):
5     protseq = aa_seq.upper()
6     charge = -0.002
7     cp = 0
8     aa_charge = {'C':-.045,'D':-.999,'E':-.998,'H':.091,
9                  'K':1,'R':1,'Y':-.001}
10    for aa in protseq:
11        charge += aa_charge.get(aa, 0)
12        if aa in aa_charge:
13            cp += 1
14    prop = float(cp)/len(aa_seq)*100
15    return (charge, prop)
16
17 cgitb.enable()
18 print('Content-Type: text/html\n')
19 form = cgi.FieldStorage()
20 seq = form.getvalue('aaseq', 'QWERTYYTREWQRTYEYTRQWE')
21 prop = form.getvalue('prop', 'n')
22 jobtitle = form.getvalue('title','No title')
23 charge, propvalue = chargeandprop(seq)
24 print('<html><body>Job title:{0}<br/>'.format(jobtitle))
25 print('Your sequence is:<br/>{0}<br/>'.format(seq))
26 print('Net charge: {0}<br/>'.format(charge))
27 if prop == 'y':
28     print('Proportion of charged AA: {0:.2f}<br/>'
29           .format(propvalue))
30 print('</body></html>')

```

Figure 10.2.3 shows the resulting HTML page after code in Listing 10.5 is executed.

Code explanation: The code to calculate the charge and the proportion of charged amino acid, are in the function that starts at line 4. On line 19 we create an instance (**form**) of the class **cgi.FieldStorage**. The form object is responsible for taking the values sent by the form and making them available in a dictionary-like fashion. From line 20 to 22 we retrieve values entered by the user. In line 24, the “net charge” and “proportion of charged amino acids” are evaluated. Line 25 up to the end generates the HTML that will be sent to the browser.

10.3 WSGI

Before WSGI there was a lot of incompatible choices for web programming in Python. Some of them were *web frameworks*, that is, a set of programs for development of dynamic web sites. The problem with some of these frameworks was that

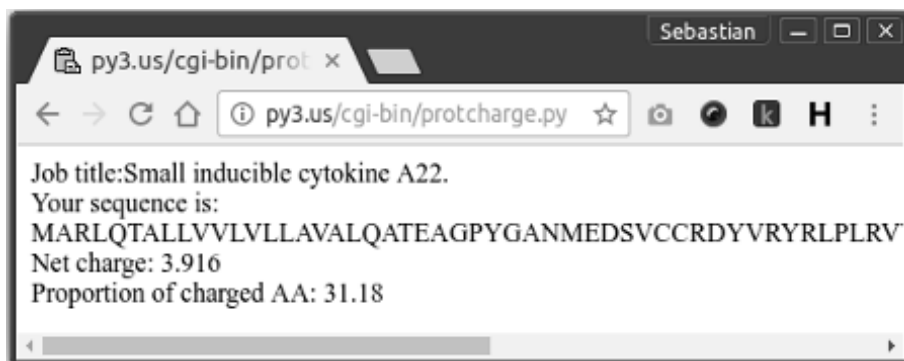


Figure 10.6 Net charge CGI result.

each one operated in a different way and most of them were tied to a web server, limiting the choice of web server/application pair.

WSGI was made to fill this gap, and it is defined as a “simple and universal interface between web servers and web applications or frameworks.” A lot of components (or middleware) are now WSGI compatible, so the programmer doesn’t need to deal directly with WSGI. Once an application works with a middleware, it can be deployed in any WSGI compliant server. WSGI is now standardized and part of the Python language^{footnote}As described in PEP-3333 at <https://www.python.org/dev/peps/pep-3333/>. For these reasons, WSGI is the recommended choice for web developing in Python.

10.3.1 Bottle: A Python Web Framework for WSGI

Bottle is a micro web-framework for Python. It is distributed as a single file and has no dependencies, that is, it only needs Python to run. **Bottle** provides 4 basic components:

- Routing: A way to translate (or map) a URL to a Python function. So each time a user requests a URL (that can have variable parts), a specific function is executed.
- Templates: A built-in template engine and support out of the box for three third-party templates (mako, jinja2, and cheetah).
- Utilities: A dictionary-like object to access to form data, file uploads, cookies, headers, and other metadata.
- Server: Development server and support for an external HTTP server, included any WSGI compatible HTTP server.

There are other alternatives to **Bottle**. The most prominent one is **Flask** (see Table 10.1 at page 233 at the end of this chapter for more options), but at this

moment it supports a template engine that is not Python 3 compatible, so **Bottle** is the best choice at this moment.

10.3.2 Installing Bottle

Bottle is available from its web page at <https://bottlepy.org>, but as most external packages, it can be installed with `pip install` under a virtual environment. The following snippet shows how to create the virtual environment (named `bottleproj`), how to activate it, and how to install Bottle in the `bottleproj` virtual environment:

```
$ virtualenv bottleproj
Using base prefix '/usr'
New python executable in /home/sb/bottleproj/bin/python3
Also creating executable in /home/sb/bottleproj/bin/python
Installing setuptools, pip, wheel...done.
$ . bottleproj/bin/activate
(bottleproj) $ pip install bottle
Collecting bottle
  Downloading bottle-0.12.13.tar.gz (70kB)
  (...)
Successfully built bottle
Installing collected packages: bottle
Successfully installed bottle-0.12.13
```

The equivalent command for Anaconda distribution is:

```
$ conda create -n bottleproj bottle
```

Since **Bottle** is contained in a file, an alternative installation method is to download the file from <https://raw.githubusercontent.com/bottlepy/bottle/master/bottle.py> and copy it to the same directory where your script resides.

10.3.3 Minimal Bottle Application

Here is a simple “Hello World” application in Bottle:

Listing 10.6: `helloworld.py`: Hello World in Bottle

```
1 from bottle import route, run
2
3 @route('/')
4 def index():
5     return '<h2>Hello World!</h2>'
6
7 run(host='localhost', port=8000)
```

In the first line we import two components of **Bottle** (`route` and `run`). In line 3 we assign a route or path to a function that starts in the next line. This is the URL (web address) that the user should enter after the domain to get this page. When an user types this path (in this case, the root level) in his/her browser, the function `index` will run. This function (in line 4) just returns “<h2>Hello World!</h2>.” Lines 7 starts the server.

Here is the output of this program to the terminal:

```
(bottleproj) $ python helloworldbottle.py
Bottle v0.13-dev server starting up (using WSGIRefServer())...
Listening on http://localhost:8000/
Hit Ctrl-C to quit.
```



Figure 10.7 Hello World program made in Bottle, as seen in a browser.

10.3.4 Bottle Components

Routes

Using the `@route` decorator we define how the URL will look for each page. The following snippet shows 2 pages, a root page without any path and an *about page* with `/about` path in the URL:

```
@route('/')
def index():
    return 'Top level, or Index Page'

@route('/about')
```

```
def about():
    return 'The about page'
```

URL with Variable Parts

In some sites, part of the URL are one or more variables that are passed to the server to build the web page; for example, in <https://stackoverflow.com/questions/6224052>, the part with the number 6224052 is a variable part. This number is passed to the program and used as a key to search the article content in a database.

The following code shows a URL with a fixed and a variable part. The fixed part is `/greet/` while the variable part is called `name`. Any string that is in its place, will be passed to the associated function (`shows_greeting`) as a parameter.

```
@route('/greet/<name>')
def shows_greeting(name):
    return 'Hello {0}'.format(name)
```

If you hit the URL <http://127.0.0.1:5000/greet/Adele>, you will see a page with the text Hello Adele.

Getting Data: request

request is a dictionary-like object with some useful properties. It stores cookies, values sent in a form, HTTP headers, files, and more. Let's see some useful properties:

- **request.form:** All variables from a web form are available from this dictionary-like object. If there is data in a form field called `username`, the way to access the field value is with `request.form.get('username')`.
- **request.method:** The HTTP method used when requesting the page. When a browser retrieves a web page, it sends a 'GET' type of request. When a URL is hit because a form is being submitted, it is a 'POST' request. There are other type of requests ('PUT', 'PATCH', and 'DELETE') but they won't be covered here.⁵
- **request.args:** To access parameters submitted in the URL. Used when the URL has the form `?key=value`. It is also a dictionary-like object. If you have a site with a URL like <http://example.com/position?lat=37.51&long=115.71>, there are two keys, `lat` and `long`, whose values are 37.51 and 115.71 respectively. To retrieve `lat` you can use `request.args['lat']` or

⁵For more information on request methods see http://www.w3schools.com/tags/ref_httpmethods.asp.

`request.args.get('lat')`. These kinds of URLs are discouraged, and user-friendly URLs are the norm, in this case it could be <http://example.com/position/37.51/115.71>⁶.

- **request.files**: When a file is uploaded, it is passed to the program as `request.files['filename']`.

Templates

In previous examples our methods were returning strings (plain-text string or HTML string). The preferred way to build an HTML file to be sent to the user, is to have a template and variable data, and with both components, make a final (or rendered) HTML. To do this, use the **template** method provided by **Bottle**. The general form of the **template** method is: `template(template_name, **dictionary)`. A template is usually an HTML file with variables as placeholders for final values. The following text is a template with one variable:

Listing 10.7: index.tpl: Template for Bottle with variables

```
1 <html lang="en">
2   <body>
3     <h1>Hello {{ name }}!</h1>
4   </body>
5 </html>
```

If this file is called `index.tpl` and it is stored in the folder **views**, it can be rendered with this code:

Listing 10.8: indextemplate.py: Bottle code for template with variables

```
1 from bottle import route, run, template
2
3 @route('/greet/<username>')
4 def shows_greeting(username):
5     return template('index', **{'name':username})
6
7 run(host='localhost', port=8000)
```

Templates can also have flow control commands so you can control which part of the template is rendered. For example, consider the following template (`index2.tpl`):

⁶For more advice on URLs please see <https://support.google.com/webmasters/answer/76329>.

Listing 10.9: index2.tpl: Template for Bottle with variables and flow control

```

1 <html lang="en">
2   <body>
3   %if name[0].isalpha():
4       <h1>Hello {{ name }}!</h1>
5   %else:
6       <h1>Your user name must can't start with a number</h1>
7   %end
8   </body>
9 </html>

```

This template has Python-like code in lines 3, 5 and 7. It looks like Python, but has an `%end` (line 7) which is not part of the normal Python syntax. This is because in Python code-blocks are marked with indentation and in the templates, indentation are not taken into account, so the `%end` mark must be present. To use this template, change line 5 in [Listing 10.8](#) to point to the `index2.tpl` file. The new [Listing \(10.10\)](#) is called `indextemplate2.py`:

Listing 10.10: index2.py: Bottle code for template with variables

```

1 from bottle import route, run, template
2
3 @route('/greet/<username>')
4 def shows_greeting(username):
5     return template('index2', **{'name':username})
6
7 run(host='localhost', port=8000)

```

This listing uses the template `index2.tpl` ([Listing 10.9](#)) that, in line 3, checks for the first character in the variable `name`; if it is true, it prints the same message as in the first template, if not, it will print the message you can see in line 6 of the template.

Note that we can get the same result by making the decision (the if clause) in the code instead of doing it in the template, see the following code and template:

Listing 10.11: indextemplate3.py: Bottle code with logic in code instead of in templates

```

1 from bottle import route, run, template
2
3 @route('/greet/<username>')
4 def shows_greeting(username):

```

```

5     if username[0].isalpha():
6         msg = 'Hello {0}!'.format(username)
7     else:
8         msg = "Your username must can't start with a number"
9     return template('index3', **{'msg':msg})
10
11 run(host='localhost', port=8000)

```

Template for [Listing 10.11](#) (index3.tpl):

Listing 10.12: index3.tpl: template for indextemplate3.py

```

1 <html lang="en">
2   <body>
3     <h1>{{ msg }}</h1>
4   </body>
5 </html>

```

In [Listing 10.11](#) (file `indextemplate3.py`) we check for the first letter of the user name in line 6, so we move the logic away from the template, resulting in an easier to read template. Since both [Listing 10.10](#) and [10.11](#) produce the same output, seems like both strategies are equivalent. They are not. Templates support logic, but it is better to have complex logic in your code (where you have better tools to debug it) rather than in the HTML (where usually a web designer with no Python knowledge will edit it). In this case, [Listing 10.11](#) is preferred over 10.10. This does not mean that you should avoid using logic in any template. Sometimes it makes a lot of sense, like in this situation:

```

<ul>
% for item in items:
  <li>{{item}}</li>
% end
</ul>

```

The takeaway from this is to use logic in the templates where you estimate that it will not make the site harder to maintain.

Static Files

Some files are served in a static manner, which means that they are not generated on the fly by a backend process. The most common cases are css, JavaScript, and images files. You can return a static file by returning a template without any variables, but **Bottle** has the method `static_file` to handle these files. `static_file`

provides the extra functionality needed in this case.⁷ You need to pass the filename and path where this file resides:

```
@route('/static/rss.xml')
def rss_static():
    return static_file('rss.xml', root='static/')
```

The path can be passed with variable parts, by enclosing the variable part between < and >:

```
@route('/static/js/<filename>')
def js_static(filename):
    return static_file(filename, root='static/js/')
```

10.3.5 Web Program to Calculate the Net Charge of a Protein (Bottle Version)

Figure 10.8 Form for the web app to calculate the net charge of a protein.

Here is the **Bottle** version the web program to calculate the net charge of a

⁷See <https://bottlepy.org/docs/dev/tutorial.html#tutorial-static-files> for more information on this method.

protein. We need an HTML template for the web form. In this case we can use the same HTML file as in `protchargeformcgi.html` (page 219) with a modification in line 9. The “action attribute” in the “form element” should point to a new URL. Now it reads:

```
<form action='/protcharge' method='post'>
```

The complete file is called `protchargeformbottle.html` and can be found the book repository at <https://github.com/Serulab/Py4Bio/tree/master/code>. When the form is used and the user presses SEND, the browser will make a POST request to `/protcharge` URL. This will execute the following code:

Listing 10.13: `protchargebottle.py`: Back-end of the program to calculate the net charge of a protein using **Bottle**

```
1 from bottle import route, run, static_file, view, post, request
2
3 def chargeandprop(aa_seq):
4     """ Calculates protein net charge and charged AA proportion
5     """
6     protseq = aa_seq.upper()
7     charge = -0.002
8     cp = 0
9     aa_charge = {'C':-.045,'D':-.999,'E':-.998,'H':.091,
10                  'K':1,'R':1,'Y':-.001}
11     for aa in protseq:
12         charge += aa_charge.get(aa, 0)
13         if aa in aa_charge:
14             cp += 1
15     prop = float(cp)/len(aa_seq)*100
16     return (charge, prop)
17
18 @route('/')
19 def index():
20     return static_file('protchargeformbottle.html', root='views/')
21
22 @route('/css/<filename>')
23 def css_static(filename):
24     return static_file(filename, root='css/')
25
26 @post('/protcharge')
27 @view('result')
28 def protcharge():
29     seq = request.forms.get('aaseq', 'QWERTYYTREWQRTYEYTRQWE')
```

```

30     prop = request.forms.get('prop','n')
31     title = request.forms.get('title', 'No title')
32     charge, propvalue = chargeandprop(seq)
33     return {'seq': seq, 'prop': prop, 'title': title,
34            'charge': round(charge, 3), 'propvalue': propvalue}
35
36 run(host='localhost', port=8000)

```

In [Listing 10.13](#) (file `protchargebottle.py`) there are 4 functions. One function that handles the actual net charge calculation (`chargeandprop`) and three that handle the mapping of different URLs. The `index` function is run when the user hits the home page and returns the HTML with the form (the `protchargeformbottle.html` file), which can be seen in [Figure 10.8](#). The `css_static` function returns the css needed for proper visualization of the form and the result page. The `protcharge` function is executed when the url `domain/protcharge` receives a post request, and this is sent when the user press the “SEND” button in the form in the `protchargeformbottle.html` file. This function returns a dictionary with all the values needed for rendering the result page. The template used in this case is the file `result.html` as is shown in the view decorator at line 27. Remember that this file must be in the directory `view` in order for this decorator to work.

Listing 10.14: `result.html`: Template for showing the result of method `protcharge`

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head><meta charset="utf-8">
4     <title>Protein Charge Calculator: Result</title>
5     <link href="css/bootstrap.min.css" rel="stylesheet">
6   </head>
7   <body style="background-color:#e7f5f5;">
8     <div class="container"><h2>Result</h2>
9     <p>Job title: {{title}}</p>
10    <p>Your sequence is: {{seq}}</p>
11    <p>Net charge: {{charge}}</p>
12    % if prop == 'y':
13    <p>Proportion of charged AA: {{propvalue}}<p>
14    % end
15  </div>
16 </body>
17 </html>

```

10.3.6 Installing a WSGI Program in Apache

There are several ways to run a WSGI in the Apache web server. In this book we will use `mod_wsgi`, an Apache module made to host Python applications which supports the Python WSGI interface.

The module can be downloaded from the project website⁸ or installed with the operating system package manager.⁹

Once `mod_wsgi` is installed, you have to modify the `apache.conf` file by adding a line like this:

```
WSGIScriptAlias webpath path_in_server
```

where `webpath` is the path seen by the user and `path_in_server` is the path to the file that will receive all the requests in this directory. For example,

```
WSGIScriptAlias / /var/www/sitepath/htdocs/test.wsgi
```

That means that every request pointing to any page in the root directory of the web server will be handled by a script located in `/var/www/sitepath/htdocs/-test.wsgi`.

10.4 ALTERNATIVE OPTIONS FOR MAKING PYTHON-BASED DYNAMIC WEB SITES

Solutions presented up to this point are useful enough to build small and medium-sized sites from the ground up. But if your website uses advanced features like database support, user and session management, administrative interface, internationalization, caching, and others, it would be better to use a full feature web framework where most of these features are already covered. Since these types of applications are beyond the scope of this book, I will show a table that summarizes the most important frameworks in Table 10.1. The table is sorted roughly on abstraction level. The first entries are systems with fewer features and require more tweaking to achieve the same result than a higher-level framework.

No framework has received the status of “Python official web framework,” so there is some usage and developer dispersion, but **Django** is by far the most popular Python web framework. If you want to learn the most used and supported web framework, **Django** is the way to go.

10.5 SOME WORDS ABOUT SCRIPT SECURITY

If your scripts will run on trusted environments (that is, not on the Internet), you can skip this section and jump to the next section on page 234.

Something to have in mind when designing web applications is that the user

⁸<http://code.google.com/p/modwsgi/>

⁹It is called `libapache2-mod-wsgi` in Debian-based systems.

TABLE 10.1 Frameworks for Web Development

Name	URL	Description
Flask	tornadoweb.org	Simple web framework very similar to Bottle
Tornado	tornadoweb.org	Web framework and asynchronous network library used for long polling and websockets
Plone	plone.net	Ready to use Python-based customizable content management system
Django	djangoproject.com	High-level Python web framework that encourages rapid development
TurboGears	turbogears.org	Full-stack solution with AJAX and multi-database support.
Web2py	www.web2py.com	Free open source full-stack framework for rapid development of fast, scalable, secure, and portable database-driven web-based applications

may (and will) enter data in an unexpected format. When the form is publicly accessible on the Internet, this threat shouldn't be underestimated.

There will be people who will not know how to complete the online form and try whatever they think is best. There will be attackers who will test your site looking for any exploitable vulnerability.

A first barrier that can be used to avoid misuse of your scripts is to use JavaScript (JS) for form validation. It is not the purpose of this book to teach JS, so there are links in the “Additional Resources” section.

JS can be used to avoid end-user related issues, but it is rather useless as a deterrent for anyone who is determined to attack your server. If anyone wants to interact with your script, they could do it without using the web browser, bypassing completely your carefully created JS code. This is why all data validation must also be done “server side.”

Another critical point to watch out for is when a script accesses a database engine. There is a chance that an attacker could inject SQL commands to produce unwanted results (like listing the full contents of a table with sensitive information like usernames and passwords¹⁰). This kind of attack is called “SQL injection” and it will be covered in the “Python and Databases” chapter ([Chapter 12](#)).

There is no rule of thumb regarding how to sanitize every kind of input, but it

¹⁰You should not store passwords in plain text in a database. The best practice is to store a hash of the password instead, using a hash function like PBKDF2 or bcrypt. Apart from the hash, you should also add some “salt,” that is, a random string to avoid an attacker using pre-hashed keys to find a match. Do not use non-tested custom made algorithms or fast cryptographic hash functions such as MD5, SHA1, SHA512, etc.

depends on the particular application. Following there are some outlines of what to take into consideration at the moment of designing the security of your application.

1. Identify where the data can access the application. Clearly the most evident point of entry are the forms you set up for data input. But you should not overlook other points of entry like URLs, files stored on the server, and other web sites if your scripts read external sources like RSS feeds.
2. Watch for escape characters used by the program your application interacts with. These should always be filtered. If your program accesses a Unix shell, filter the “;” character (semicolon) since it can be used to issue arbitrary commands. This depends on the type of shell your system is using. Some characters you should consider watching are: `;`, `&&`, `||`, `\` and `"`.
3. Consider making a list of valid accepted characters (a “white list”) to make sure that your strings have only the required characters.
4. The running privileges of the web server program must be the lowest possible. Most Unix systems use an adhoc user for the web server process. This is called the “Principle of Least Privilege.” The program is given the smallest amount of privilege required to do its job. This limits the abuse that can be done to a system if the web server process is hijacked by an attacker.

10.6 WHERE TO HOST PYTHON PROGRAMS

If you’ve satisfactorily tested your scripts on your local server, it’s time to put them on the Internet so that the rest of the world can enjoy them. Usually the institution that you work for has a web server where you can store your scripts, for which the first step would be to ask for support from your IT department. In the event that you don’t get a satisfactory response, you would have to consider resolving the problem by yourself. It is not too difficult. There are thousands of web hosting businesses. Look for one that explicitly supports Python.

Among the diverse plans that are offered by the web hosting businesses, choose the “shared” plan type if your script is very simple and does not involve installation of programs or additional modules. If your script executes programs that aren’t installed on the server, as is the case with Biopython, you can ask for it to be installed. Ask before contracting the service if they install modules on demand. Another problem that can surface is with the web frameworks. Some work as a long running process, which is not permitted by the hosting agreement.

Make sure that the version of Python installed on the hosting server is compatible with your scripts. This is not a minor topic considering that operating systems used for servers tend to use a “stable” version of every software instead of the latest.

If the web hosting service does not allow program installation, you will have to consider a dedicated hosting solution, where you have root access to a computer

where there are not limits with regard to what you can install. Thanks to virtualization technologies, it is possible to contract a dedicated virtual hosting plan at a more than affordable price (also known as **Virtual Private Server** or **VPS**). This is because the computer is shared between various users, but it differs from the shared hosting plan type in that each user has total access to the server. For very demanding applications, this may not be the best solution and you may have to resort to the use of dedicated hosting (not virtual).

An alternative to servers is the Google App Engine. This system enables you to build web applications on the same scalable systems that power Google applications. Let Google take care of Apache web server configuration files, startup scripts, databases, server monitoring and software upgrades. Just write your Python code. Applications designed for this engine are implemented using the Python programming language. The App Engine Python runtime environment includes a specialized version of the Python interpreter. For more information on “Google App Engine,” see <https://cloud.google.com>.

Amazon Web Services (AWS) has also a “serverless” solution called **Lambda**. With AWS Lambda, you focus only on your code and Lambda manages the compute fleet that offers a balance of memory, CPU, network, and other resources. The downside is that you pay per program execution and can’t run the programs as you already have them written; you need to make some adjustments to make it compatible with this particular environment. It runs only a handful of languages, but Python is included. For more information see <https://aws.amazon.com/lambda/> and <http://docs.aws.amazon.com/lambda>.

10.7 ADDITIONAL RESOURCES

- W3Schools: JavaScript form validation.
https://www.w3schools.com/js/js_validation.asp
- Data validation.
https://www.owasp.org/index.php/Data_Validation
- JavaScript-Coder.com: JavaScript form validation : Quick and Easy!
<http://www.javascript-coder.com/html-form/javascript-form-validation.phtml>
- HTML reference: A free guide to HTML.
<http://htmlreference.io>
- Bootstrap: The most popular HTML, CSS, and JS framework for developing responsive, mobile first projects on the web.
<http://getbootstrap.com>
- PureCSS: A set of small, responsive CSS modules that you can use in every web project.
<http://purecss.io>

- BottlePlate: A bottle template for python 3.3+ web applications or API servers.
<https://github.com/Rolinh/bottleplate>
- Bottle + uWSGI: simple web app configuration and fun hidden features.
<https://goo.gl/X8Up6S>
- Decanter: Creates a Bottle based directory structure with an example view and controller.
<http://gengo.github.io/decanter/>
- Learn web development.
<https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django>
- Web programming in Python.
<http://wiki.python.org/moin/WebProgramming>
- wuzz: An interactive command line tool for HTTP inspection.
<https://github.com/asciimoo/wuzz>

10.8 SELF-EVALUATION

1. What is CGI?
2. What is WSGI? Why is it the recommended choice for web programming?
3. What is the rationale for using Bottle or any other “web framework”?
4. What is a template language?
5. What is a static file and why should you serve it in a different way?
6. Python includes a limited web server. Why would you use such a web server if there are free full-featured web servers like Apache?
7. Name security considerations to take into account when running a Web server on the Internet.
8. Why is client-side data validation not useful as server-side data validation?
9. What is the difference between shared, dedicated, and virtual dedicated hosting? When would you use dedicated hosting over a shared plan?

XML

CONTENTS

11.1	Introduction to XML	237
	What Is XML?	237
	XML in 10 Points	238
11.2	Structure of an XML Document	241
	Prologue	242
	Body	243
11.3	Methods to Access Data inside an XML Document	246
	cElementTree	246
	11.3.1 SAX: cElementTree Iterparse	246
	BeautifulSoup	248
11.4	Summary	251
11.5	Additional Resources	252
11.6	Self-Evaluation	252

11.1 INTRODUCTION TO XML

What Is XML?

A widespread problem in all branches of information technology is the storage and interchange of data. Each application has its own particular way of storing the generated information, which is often a problem, especially when we don't have the application that generated the data.

For example, Sanger DNA sequencers made by Applied Biosystems store data in files with the extension .ab1. If we want to access data stored in such a file, we need to know how it is structured internally. In this case, the creator of the format has released the specification of the file,¹ and it would be possible, though not easy, to write code to extract our data from these files. Usually we do not have such good luck, and it is very common to find data file formats poorly documented, or not documented at all. In many cases those who have wanted to open these files have had to resort to “reverse engineering,” with mixed results. To avoid this type of problem and to make more fluid exchange of data between applications from

¹File format specification for ABI files are available at http://www6.appliedbiosystems.com/support/software_community/ABIF_File_Format.pdf.

different manufacturers, the W3C² developed the eXtensible Markup Language, better known as XML.

XML is a way of representing data. Practically any type can be represented using XML. Configuration files, databases, web pages, spreadsheets, and even drawings can be represented and stored in XML.

For some specific applications, there are subsets of XML, prepared for representing a particular type of data. So, mathematical formulas can be stored in an XML dialect called MathML,³ vector graphics in SVG,⁴ and chemical formulas in CML.⁵ All major bioinformatics databases have their data available in XML. This means that, by learning to read XML, we can access a multitude of files from the most diverse origins.

Before going into details on how to process this type of file, review this W3C document called “XML in 10 points”⁶ that shows the big picture:

XML in 10 Points

1. XML is for structuring data: Structured data includes things like spreadsheets, address books, configuration parameters, financial transactions, and technical drawings. XML is a set of rules (you may also think of them as guidelines or conventions) for designing text formats that let you structure your data. XML is not a programming language, and you don’t have to be a programmer to use it or learn it. XML makes it easy for a computer to generate data, read data, and ensure that the data structure is unambiguous. XML avoids common pitfalls in language design: it is extensible, platform-independent, and it supports internationalization and localization. XML is fully Unicode-compliant.
2. XML looks a bit like HTML: Like HTML, XML makes use of tags (words bracketed by ‘<’ and ‘>’) and attributes (of the form name="value"). While HTML specifies what each tag and attribute means, and often how the text between them will look in a browser, XML uses the tags only to delimit pieces of data, and leaves the interpretation of the data completely to the application that reads it. In other words, if you see “<p>” in an XML file, do not assume it is a paragraph. Depending on the context, it may be a price, a parameter, a person, a p... (and who says it has to be a word with a “p?”).
3. XML is text, but isn’t meant to be read: Programs that produce spreadsheets, address books, and other structured data often store that data on disk, using

²The World Wide Web Consortium, abbreviated W3C, is an international consortium that produces standards for the World Wide Web.

³<http://www.w3.org/Math>

⁴<http://www.w3.org/Graphics/SVG>

⁵<http://www.xml-cml.org>

⁶Taken from <http://www.w3.org/XML/1999/XML-in-10-points>. Authorized by “©[1999] World Wide Web Consortium, (Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University). All Rights Reserved.”

either a binary or text format. One advantage of a text format is that it allows people, if necessary, to look at the data without the program that produced it; in a pinch, you can read a text format with your favorite text editor. Text formats also allow developers to more easily debug applications. Like HTML, XML files are text files that people shouldn't have to read, but may when the need arises. Compared to HTML, the rules for XML files allow fewer variations. A forgotten tag, or an attribute without quotes makes an XML file unusable, while in HTML such practice is often explicitly allowed. The official XML specification forbids applications from trying to second-guess the creator of a broken XML file; if the file is broken, an application has to stop right there and report an error.

4. XML is verbose by design: Since XML is a text format and it uses tags to delimit the data, XML files are nearly always larger than comparable binary formats. That was a conscious decision by the designers of XML. The advantages of a text format are evident (see point 3), and the disadvantages can usually be compensated at a different level. Disk space is less expensive than it used to be, and compression programs like zip and gzip can compress files very well and very fast. In addition, communication protocols such as modem protocols and HTTP/1.1, the core protocol of the Web, can compress data on the fly, saving bandwidth as effectively as a binary format.
5. XML is a family of technologies: XML 1.0 is the specification that defines what “tags” and “attributes” are. Beyond XML 1.0, “the XML family” is a growing set of modules that offer useful services to accomplish important and frequently demanded tasks. XLink describes a standard way to add hyperlinks to an XML file. XPointer is a syntax in development for pointing to parts of an XML document. An XPointer is a bit like a URL, but instead of pointing to documents on the Web, it points to pieces of data inside an XML file. CSS, the style sheet language, is applicable to XML as it is to HTML. XSL is the advanced language for expressing style sheets. It is based on XSLT, a transformation language used for rearranging, adding and deleting tags and attributes. The DOM is a standard set of function calls for manipulating XML (and HTML) files from a programming language. XML Schemas 1 and 2 help developers to precisely define the structures of their own XML-based formats. There are several more modules and tools available or under development. Keep an eye on W3C's technical reports page.
6. XML is new, but not that new: Development of XML started in 1996 and it has been a W3C Recommendation since February 1998, which may make you suspect that this is rather immature technology. In fact, the technology isn't very new. Before XML there was SGML, developed in the early '80s, an ISO standard since 1986, and widely used for large documentation projects. The development of HTML started in 1990. The designers of XML simply took the best parts of SGML, guided by the experience with HTML, and

produced something that is no less powerful than SGML, and vastly more regular and simple to use. Some evolutions, however, are hard to distinguish from revolutions... And it must be said that while SGML is mostly used for technical documentation and much less for other kinds of data, with XML it is exactly the opposite.

7. XML leads HTML to XHTML: There is an important XML application that is a document format: W3C's XHTML, the successor to HTML. XHTML has many of the same elements as HTML. The syntax has been changed slightly to conform to the rules of XML. A format that is "XML-based" inherits the syntax from XML and restricts it in certain ways (e.g., XHTML allows "<p>", but not "<r>"); it also adds meaning to that syntax (XHTML says that "<p>" stands for "paragraph", and not for "price", "person", or anything else).
8. XML is modular: XML allows you to define a new document format by combining and reusing other formats. Since two formats developed independently may have elements or attributes with the same name, care must be taken when combining those formats (does "<p>" mean "paragraph" from this format or "person" from that one?). To eliminate name confusion when combining formats, XML provides a namespace mechanism. XSL and RDF are good examples of XML-based formats that use namespaces. XML Schema is designed to mirror this support for modularity at the level of defining XML document structures, by making it easy to combine two schemas to produce a third which covers a merged document structure.
9. XML is the basis for RDF and the Semantic Web: W3C's Resource Description Framework (RDF) is an XML text format that supports resource description and metadata applications, such as music play-lists, photo collections, and bibliographies. For example, RDF might let you identify people in a Web photo album using information from a personal contact list; then your mail client could automatically start a message to those people stating that their photos are on the Web. Just as HTML integrated documents, images, menu systems, and forms applications to launch the original Web, RDF provides tools to integrate even more, to make the Web a little bit more into a Semantic Web. Just like people need to have agreement on the meanings of the words they employ in their communication, computers need mechanisms for agreeing on the meanings of terms in order to communicate effectively. Formal descriptions of terms in a certain area (shopping or manufacturing, for example) are called ontologies and are a necessary part of the Semantic Web. RDF, ontologies, and the representation of meaning so that computers can help people do work are all topics of the Semantic Web Activity.
10. XML is license-free, platform-independent and well-supported: By choosing XML as the basis for a project, you gain access to a large and growing community of tools (one of which may already do what you need!) and engineers

experienced in the technology. Opting for XML is a bit like choosing SQL for databases: you still have to build your own database and your own programs and procedures that manipulate it, but there are many tools available and many people who can help you. And since XML is license-free, you can build your own software around it without paying anybody anything. The large and growing support means that you are also not tied to a single vendor. XML isn't always the best solution, but it is always worth considering.

11.2 STRUCTURE OF AN XML DOCUMENT

We do not need to know the details of the internal structure of an XML document. This is because Python has its own tools for accessing this type of file. The developers of Python had to deal with the internals of XML in order to build these tools; however I think that is necessary to have a slight notion of the structure of XML files in order to make better use of the tools provided by Python.

Let's see a sample XML document, in this case an UniProt record:⁷

Listing 11.1: Q9JJE1.xml: UniProt record in XML

```
<?xml version="1.0" encoding="UTF-8"?>
<uniprot xmlns="http://uniprot.org/uniprot"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://uniprot.org/uniprot
    http://www.uniprot.org/support/docs/uniprot.xsd">
<entry dataset="TrEMBL" created="2000-10-01" version="35">
  <accession>Q9JJE1</accession>
  <organism key="1">
    <name type="scientific">Mus musculus</name>
    <lineage>
      <taxon>Eukaryota</taxon>
      <taxon>Metazoa</taxon>
      <taxon>Chordata</taxon>
      <taxon>Craniata</taxon>
      <taxon>Vertebrata</taxon>
      <taxon>Euteleostomi</taxon>
      <taxon>Mammalia</taxon>
      <taxon>Eutheria</taxon>
      <taxon>Euarchontoglires</taxon>
      <taxon>Glires</taxon>
      <taxon>Rodentia</taxon>
      <taxon>Sciurognathi</taxon>
```

⁷This record was altered to fit the page. This file can be found in the book Github repository with the name `uniprotrecord.xml` under the `samples` directory.


```

    <taxon>Muroidea</taxon>
    <taxon>Muridae</taxon>
    <taxon>Murinae</taxon>
    <taxon>Mus</taxon>
  </lineage>
</organism>
<dbReference type="UniGene" id="Mm.248907" key="5"/>
<sequence length="393" checksum="E0C0CC2E1F189B8A">
MPKKKPTPIQLNPAPDGSVNGTSSAETNLEALQKKLEELDEQQRKRL
EAFLTQKQKVGELKDDDFEKISELGAGNGGVVFKVSHKPSGLVMARKLIH
LEIKPAIRNQIIRELQVLHECNSPYIVGFYGAFYSDGEISICMEHMDGGS
LDQVLKKAGRIPEQILGKVSIAVIKGLTYLREKHKIMHRDVKPSNILVNS
RGEIKLCDFGVSGQLIDSMANSFVGTRSYMSPERLQGTHYSVQSDIWSMG
LSLVEMAVGRYPPIPPDAKELELLFGCHVEGDAAETPPRPRTPGGPLSSY
GMDSRPPMAIFELLDYIVNEPPPKLPSGVFSLEFQDFVNKCLIKNPAERA
DLKQLMVHAFIKRSDAEEVDFAGWLCSTIGLNQPSTPTHAASI
</sequence>
</entry>
</uniprot>

```

In broad outlines, the structure of an XML document is simple. It generally consists of a prologue, a body, and an epilogue.⁸

Prologue

The prologue is an optional section that marks the beginning of the XML data and gives important information to the parser. A prologue might have only one line, like this one,

```
<?xml version="1.0" encoding="UTF-8"?>
```

Or several lines:

```

<?xml version="1.0"?>
<!DOCTYPE BlastOutput PUBLIC "-//NCBI//NCBI BlastOutput/EN"
"http://www.ncbi.nlm.nih.gov/dtd/NCBI_BlastOutput.dtd">
<!-- edited with XMLSPY (http://www.xmlspy.com) by Andy -->

```

The first line is the **XML declaration** where the XML version and character code are specified. Character code information is optional only if the document is encoded in UTF-8 or UTF-16.

The second line is the **DOCTYPE declaration**, whose purpose is to relate the XML document with a document type definition (DTD). This DTD file contains

⁸The epilogue is seldom used, so the prologue and the body are the most important parts of an XML file.

information about the particular structure of the XML file: it says which tags and attributes are permitted, as well as where they can be found. In some cases, in place of a DTD reference, there are references to an alternate method to DTD called XML Schema which serves the same function but with better performance, and with a syntax based on XML. The structure of a DTD or XML Schema file is beyond the scope of this book; however, there are several, quite complete, references on the Internet (see **Additional Resources** at the end of this chapter).

The third line, in this case, is a comment. It is equivalent to `#` in Python. It begins with “`<!--`” and ends with “`-->`”, and can be in the prologue as well as in the body of an XML document. It is the same type of comment that is used in HTML and it can span multiple lines.

Body

The body is where the **elements** reside, the true protagonists of XML files. An element is the information from the beginning of the start tag to the end of the end tag, including all that lies in between.

Here is an example of an element that can be found in the body of an XML document:

```
<taxon>Eukaryota</taxon>
```

where `<taxon>` is the start tag, `</taxon>` is the end tag, and the contents (**Eukaryota**), is that which is between the two tags.

Elements may show up empty. It is valid to write, for example:

```
<accession></accession>
```

While in this case it doesn’t make much sense to have nothing contained in the “accession” element (an UniProt record should always have a number of accessions), it is possible that for other types of data the contents of an element will be optional.

There is an abbreviated way to represent empty elements, called an “empty element tag,” which consists of the name of the element followed by a forward slash (/), all enclosed by angle brackets, for example:

```
<accession/>
```

The elements can be “nested” inside one another. In [Listing 11.1](#) we can see how the element “taxon” is nested within “lineage.” This gives an idea of a hierarchical structure: there are elements that are subordinate to others. We see that “taxon” is an element of “lineage,” which is an element of “organism.” Normally this type of structure is compared to a tree. The first element is called the “Document Element” (in this case, “uniprot”), from which hangs all the rest, which are its “children.” To obtain a graphical representation of this tree, one can use a program like XML Viewer,⁹ (see [Figure 11.2](#)) or a website like <http://codebeautify.org/xmlviewer> that shows both the data and the document structure as seen in [Figure 11.2](#).

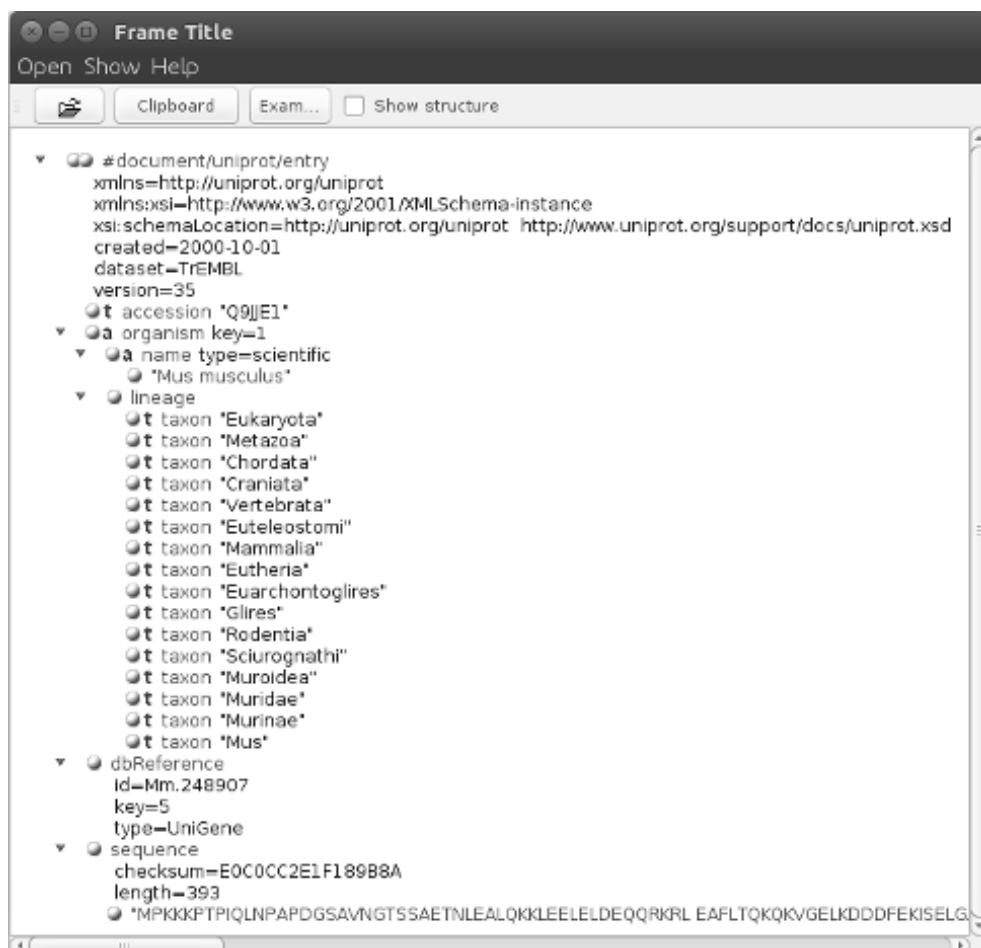


Figure 11.1 Screenshot of XML viewer: Tree viewer shows the structure of document Q9JJE1.xml (Listing 11.1).

Some elements have “attributes,” that is, additional information about the element. The general syntax of an element with an attribute is

```
<element attributeName="value">
```

Continuing with the example of Listing 11.1, we come across other elements with attributes as, for example,

```
<name type="scientific">
```

In this case the element called “name” has the attribute “type,” which has a

⁹XML Viewer is available at <http://sourceforge.net/projects/ulmxmlview>.

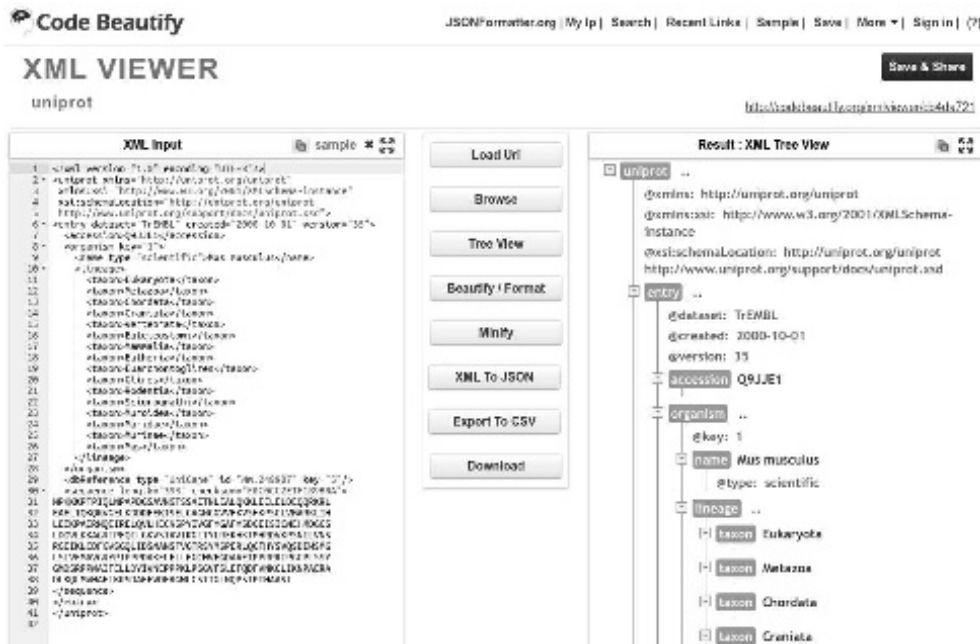


Figure 11.2 Codebeautify XML viewer, shows structure and data from document Q9JJE1.xml (Listing 11.1).

value of “scientific.” Additionally it can have more than one attribute, as in the element “sequence”:

```
<sequence length="393" checksum="E0C0CC2E1F189B8A">
```

Here the attributes are “length” and “checksum”, whose values are “393” and “E0C0CC2E1F189B8A,” respectively.

At this stage we already have elements that give us an idea of the data contained in an XML file. Of the record of file Q9JJE1.xml (Listing 11.1) we can say that the element “sequence” contains a nucleotide sequence of a length of 393 bp, a known signature, and an ID “Q9JJE1” from the UniProt base. All this without prior knowledge of the data structure and without the use of a special program. Try to open an **.ab1** file to see if you can find any recognizable element.

Despite having a general overview of the structure of XML files, you will find the format has other particularities that go beyond the scope of this book. If you are interested in knowing more about XML, see the list of resources at the end of the chapter.

The following section shows how to access the contents of XML documents using Python.

11.3 METHODS TO ACCESS DATA INSIDE AN XML DOCUMENT

Regardless of the programming language you use, there are two strategies that you can use to gain access to the information contained in an XML file.

On one hand, you can read the file in its entirety, analyze the relationships between the elements, and build a tree-type structure, by which the application can navigate the data. This is called the Document Object Model (DOM) and is the manner recommended by the W3C in parsing XML documents.

Another possibility is that the application detects and reports events such as the start and the end of an element, without the necessity of constructing a tree-type representation. In the case that a tree representation is needed, this task is left to the programmer. This is the method used by the **Simple API for XML (SAX)**. Generally these types of parsers are called “event-driven parsers.” In this chapter we will see, as an example of an event-based parser, **Iterparse** from **cElementTree**.

In some cases it is convenient to use DOM, while in other cases SAX is the preferred option. DOM usually implies saving the whole tree in memory for later traversal. This can present a problem at the time of parsing large documents, especially when what you want to do is simply detect the presence of a single element’s value. In these cases a SAX is the most efficient parser. Nevertheless, many applications require operating on all the elements within the tree, for which we must turn to DOM. From the perspective of the programmer, the DOM interface is easier to use than SAX as it doesn’t require event-driven programming.

cElementTree

cElementTree is SAX parser that is optimized to parse quickly and with less use of memory. Another advantage of **cElementTree** is a function called *Iterparse*. This function provides us the use of an event based parser, which will be explained in the next section.

11.3.1 SAX: cElementTree Iterparse

cElementTree Iterparse isn’t SAX, but it is included here because, unlike the other parsers, it is based on events.

Iterparse returns a flow iterable by tuples in the form (event, element). It is used to iterate over the elements and processing them on the fly.

Get the protein sequence and its attributes:

```
>>> import xml.etree.cElementTree as cET
>>> for event, elem in cET.iterparse('uniprotrecord.xml',
    events=('start', 'end')):
    if event=='end' and 'sequence' in elem.tag:
        print('Sequence: {0}'.format(elem.text))
        print('Checksum: {0}'.format(elem.attrib["checksum"]))
        print('Length: {0}'.format(elem.attrib["length"]))
```

```
elem.clear()
```

Sequence:

```
MPKKKPTPIQLNPAPDGSVNGTSSAETNLEALQKKLEELDEQQRKRL
EAFLTQKQKVGELKDDDFEKISELGAGNGGVVFKVSHKPSGLVMARKLIH
LEIKPAIRNQIIRELQVLHECNSPYIVGFYGFYSDGEISICMEHMDGGS
LDQVLKKAGRIPEQILGKVSIAVIKGLTYLREKHKIMHRDVKPSNILVNS
RGEIKLCDFGVSGQLIDSMANSFVGTRSYMSPERLQGTHYSVQSDIWSMG
LSLVEMAVGRYPIPPDAKELELLFGCHVEGDAAETPPRPRTPGGPLSSY
GMDSRPPMAIFELLDYIVNEPPPKLPSGVFSLEFQDFVNKCLIKNPAERA
DLKQLMVHAFIKRSDAEEVDFAGWLCSTIGLNQPSTPTHAASI
```

Checksum: E0C0CC2E1F189B8A

Length: 393

iterparse returns a tuple. The first element of the tuple is *event* and can be one of two values: 'start' or 'end'. If the event we received is 'start', it means that we can access the name of the element and its attributes, but not necessarily its text. When we receive 'end', we can be assured that we've processed all the components of that element. For this reason the previous code checked not only that we had reached the chosen element, but that we had also found the 'end' event.¹⁰ If the parser were to return only 'end', there would be no need for this check:

```
>>> for event, elem in cET.iterparse('uniprotrecord.xml'):
    if 'sequence' in elem.tag:
        print('Sequence: {0}'.format(elem.text))
        print('Checksum: {0}'.format(elem.attrib["checksum"]))
        print('Length: {0}'.format(elem.attrib["length"]))
        elem.clear()
```

Sequence:

```
MPKKKPTPIQLNPAPDGSVNGTSSAETNLEALQKKLEELDEQQRKRL
EAFLTQKQKVGELKDDDFEKISELGAGNGGVVFKVSHKPSGLVMARKLIH
LEIKPAIRNQIIRELQVLHECNSPYIVGFYGFYSDGEISICMEHMDGGS
LDQVLKKAGRIPEQILGKVSIAVIKGLTYLREKHKIMHRDVKPSNILVNS
RGEIKLCDFGVSGQLIDSMANSFVGTRSYMSPERLQGTHYSVQSDIWSMG
LSLVEMAVGRYPIPPDAKELELLFGCHVEGDAAETPPRPRTPGGPLSSY
GMDSRPPMAIFELLDYIVNEPPPKLPSGVFSLEFQDFVNKCLIKNPAERA
DLKQLMVHAFIKRSDAEEVDFAGWLCSTIGLNQPSTPTHAASI
```

¹⁰In the current implementation, the parser goes along reading 16 Kb chunks, so in this case the whole sequence could be read from the 'start' element. To make sure that you pick up all the elements you should read it after an 'end' element.

Checksum: E0C0CC2E1F189B8A

Length: 393

As for the **clean** method, it is used to “clean up” the node after it’s used, because unlike a classic SAX parser like `ElementTree`, **iterparse** constructs a complete tree. The problem with this code is that the primary element remains with all its (now empty) children, and that uses memory. In this simple example, this behavior is not problematic, but it could be when processing large files. The ideal would be to access the parent node in order to clean it up.

A way to do this is to save a reference to the first variable; for this we create an iterator and obtain from it the first element, calling it “root”:

```
>>> allelements = iterparse('uniprotrecord.xml', events=('start',<=
'end'))
>>> allelements = iter(allelements)
>>> event, root = next(allelements)
```

Now we process it the same as before, only this time we can delete the parent element specifically:

```
>>> for event, elem in allelements:
    if event=='end' and 'sequence' in elem.tag:
        print(elem.text)
        root.clear()
```

BeautifulSoup

BeautifulSoup¹¹ is an external module that is used to parse XML and HTML files. Its main advantage over Python built-in parsers, is that it can parse malformed (broken) HTML files.

This module calls another module that does the parsing job in background. In this case we use **lxml**; there are others, but I use it because it is the most feature-rich and easy-to-use library for processing XML and HTML in the Python language.

Since it is an external module you have to install it:

```
$ pip install beautifulsoup4
```

or if you are using **Anaconda**:

```
$ conda install beautifulsoup4
```

Once installed, the first step is to create a **BeautifulSoup** object by calling its class with two parameters, a file object and the parser. Here is the general form:

¹¹ Available at <https://www.crummy.com/software/BeautifulSoup>.

BeautifulSoup(FILE_OBJECT or STRING, PARSEr)

Let's see it in action:

```
>>> from bs4 import BeautifulSoup as bs
>>> soup = bs(open('uniprotrecord.xml'), 'lxml')
```

If the xml file is not a local file, but an Internet resource, you can use the **requests** library.¹² If this is the case, first install and import requests:

```
(py4bio) $ pip install requests
Collecting requests
  Downloading requests-2.13.0-py2.py3-none-any.whl (584kB)
    100% |*****| 593kB 968kB/s
Installing collected packages: requests
Successfully installed requests-2.13.0
(py4bio) $ python
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import requests
```

Once imported, use **.get()** to get the URL and **content** to retrieve the content:

```
>>> url = 'https://s3.amazonaws.com/py4bio/uniprotrecord.xml'
>>> req = requests.get(url)
>>> c = req.content
```

The content of the XML file (c) can be used as parameter for **BeautifulSoup**:

```
>>> from bs4 import BeautifulSoup as bs
>>> soup = bs(c, 'lxml')
```

We have now a **BeautifulSoup** object that is called **soup**. To access an element, just use the element name as a property of this object. To get the sequence, use **soup.sequence**:

```
>>> soup.sequence
<sequence checksum="EOC0CC2E1F189B8A" length="393">
MPKKKPTPIQLNPAPDGSVNGTSSAETNLEALQKKLEELDEQQRKRL
EAFLTQKQKVGELKDDDFEKISELGAGNGGVVFKVSHKPSGLVMARKLIH
LEIKPAIRNQIIRELQVLHECNSPIYVGFYGAIFYSDGEISICMEHMDGGS
LDQVLKKAGRIPEQILGKVSIAVIKGLTYLREKHKIMHRDVKPSNILVNS
```

¹²There are built-in libraries to retrieve files from Internet (like **urllib2**), but **requests** is less complex and has more features than any built-in library.


```
RGEIKLCDFGVSGQLIDSMANSFVGTRSYMSPERLQGTHYSVQSDIWSMG
LSLVEMAVGRYPIPPDAKELELLFGCHVEGDAAETPPRPRTPGGPLSSY
GMDSRPPMAIFELLDYIVNEPPPKLPSGVFSLEFQDFVNKCLIKNPAERA
DLKQLMVHAFIKRSDAEVDFAGWLCSTIGLNQPSTPTHAASI
</sequence>
```

If you want the content of this element, use **string**:

```
>>> soup.sequence.string
'\nMPKKKPTPIQLNPAPDGSVNGTSSAETNLEALQKKLEELDEQQRKRL\nEAFLTQKQKV<=
GELKDDDFEKISELGAGNGGVVFKVSHKPSGLVMARKLIH\nLEIKPAIRNQIIRELQVLHECNS<=
PYIVGFYGFYSDGEISICMEHMDGGS\nLDQVLKKAGRIPEQILGKVSIAVIKGLTYLREKHKI<=
MHRDVKPSNILVNS\nRGEIKLCDFGVSGQLIDSMANSFVGTRSYMSPERLQGTHYSVQSDIWSM<=
G\nLSLVEMAVGRYPIPPDAKELELLFGCHVEGDAAETPPRPRTPGGPLSSY\nGMDSRPPMAI<=
FELLDYIVNEPPPKLPSGVFSLEFQDFVNKCLIKNPAERA\nDLKQLMVHAFIKRSDAEVDFAG<=
WLCSTIGLNQPSTPTHAASI\n'
```

To get the “checksum” and “length” properties of this element, use `.get()`:

```
>>> soup.sequence.get('checksum')
'E0C0CC2E1F189B8A'
>>> soup.sequence.get('length')
'393'
```

If an element has multiple children, you can iterate through it. For example, the element “lineage” has multiple elements of type “taxon.”

```
>>> for taxon in soup.lineage.children:
    if taxon.string != '\n':
        print(taxon.string)
```

```
Eukaryota
Metazoa
Chordata
Craniata
Vertebrata
Euteleostomi
Mammalia
Eutheria
Euarchontoglires
Glires
Rodentia
Sciurognathi
Muroidea
Muridae
```

Murinae
Mus

Here is the way to get the same data (sequence, Checksum and Length) we where getting with the **cElementTree** parser in the same format:

```
>>> print('Sequence: {0}'.format(soup.sequence.string))
Sequence:
MPKKKPTPIQLNPAPDGSAVNGTSSAETNLEALQKKLEELDEQQRKRL
EAFLTQKQKVGELKDDDFEKISELGAGNGGVVFKVSHKPSGLVMARKLIH
LEIKPAIRNQIIRELQVLHECNSPYIVGFYGAFYSDGEISICMEHMDGGS
LDQVLKKAGRIPEQILGKVSIAVIKGLTYLREKHKIMHRDVKPSNILVNS
RGEIKLCDFGVSGQLIDSMANSFVGTRSYMSPERLQGTHYSVQSDIWSMG
LSLVEMAVGRYPPIPPDAKELELLFGCHVEGDAAETPPRPRTPGGPLSSY
GMDSRPPMAIFELLDYIVNEPPPKLP SGVFSLEFQDFVNKCLIKNPAERA
DLKQLMVHAFIKRSDAEEVDFAGWLCSTIGLNQPSTPTHAASI
>>> print('Checksum: {0}'.format(soup.sequence.get('checksum')))
Checksum: EOC0CC2E1F189B8A
>>> print('Length: {0}'.format(soup.sequence.get('length')))
Length: 393
```

11.4 SUMMARY

XML means **eXtensible Markup Language** and was created to enable a standard way of storing and exchanging data. One of the advantages of XML is that it is supported by various programming languages, among which is Python. XML documents consist of a prologue, a body, and an epilogue. The prologue contains information on the version, the encoding, and the structure of that document. The body contains all the information of the document, divided into hierarchically ordered elements. Each element consists of a tag with its text. Optionally, an element can have attributes. There also exist elements without text at all, called “empty elements.”

Without regard to the programming language used, there are two major strategies used when accessing these types of files. On one hand, it can analyze the relationships between all the elements, and construct the corresponding tree. This implies having the whole file structure in memory, and is called the **Document Object Model (DOM)**. The other option is to recurse over the file and generate events by which we can then travel, recursing on each distinct element. At each event we can process our data. These are called “event-driven parsers” and the most well known is **Simple API for XML (SAX)**.

In this chapter we presented as an example of a parser based on events, and we saw the use of `Iterparse`, provided by `cElementTree`. DOM is often easier to use because it does not involve event handling; however, on some occasions it’s more convenient to use a parser based on events, especially for large files. An alternative

external module that can also be used to parse broken HTML is BeautifulSoup. This module relies on another parser (typically **lxml**) but is easy to use.

11.5 ADDITIONAL RESOURCES

- Extensible Markup Language (XML). Links to W3C recommendations, proposed recommendations and working drafts.
<http://www.w3.org/XML>
- BeautifulSoup documentation.
<https://www.crummy.com/software/BeautifulSoup/bs4/doc/>
- Web scraping workshop. Using requests and BeautifulSoup, with the most recent BeautifulSoup 4 docs.
<https://gist.github.com/bradmontgomery/1872970>
- Mark Pilgrim. *Dive into Python 3*. Chapter 12; XML processing.
<http://www.diveintopython3.net/xml.html>
- Python and XML: An introduction.
http://www.boddie.org.uk/python/XML_intro.html
- Resources on DTD.
<http://www.w3schools.com/dtd/>, <http://www.xmlfiles.com/dtd>, and <http://www.w3.org/TR/REC-xml/#dt-doctype>.
- Resources on XML schema:
https://www.w3schools.com/xml/schema_intro.asp.

11.6 SELF-EVALUATION

1. What does the OpenOffice format have in common with RSS feeds and Google Earth's geographic coordinates?
2. What are the benefits of using XML for data storage and information interchange?
3. When will you not use XML?
4. Why should an XML parser not read a malformed XML document?
5. Distinguish between the terms tag, element, attribute, value, DTD, and Schema.
6. In the example XML file ([Listing 11.1](#)) there is one empty-element tag. Which one is it?

7. What is the difference between the SAX and DOM models of XML file processing?
8. If you have to parse an XML file that has a size approaching or exceeding available RAM, what is the recommended parser?
9. In **cElementTree.iterparse** there are both **start** and **end** event types. By default it returns only the **end** event. When would you use the information in a **start** event?
10. Make two programs to parse all hit names in an XML BLAST output. One program should use Python XML tools and the other should read the input file as text.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Python and Databases

CONTENTS

12.1	Introduction to Databases	256
	What Is a Database?	256
	Database Types	256
	12.1.1 Database Management: RDBMS	257
	12.1.2 Components of a Relational Database	258
	A Key Concept: Primary Key	259
	12.1.3 Database Data Types	259
12.2	Connecting to a Database	261
12.3	Creating a MySQL Database	262
	12.3.1 Creating Tables	263
	12.3.2 Loading a Table	264
12.4	Planning Ahead	266
	12.4.1 PythonU: Sample Database	266
	Grades Table	266
	Courses Table	268
12.5	SELECT: Querying a Database	269
	Simple Query	269
	Combining Two Queries	269
	Querying Several Tables	270
	12.5.1 Building a Query	271
	12.5.2 Updating a Database	273
	12.5.3 Deleting a Record from a Database	273
12.6	Accessing a Database from Python	274
	12.6.1 PyMySQL Module	274
	12.6.2 Establishing the Connection	274
	12.6.3 Executing the Query from Python	275
12.7	SQLite	276
12.8	NoSQL Databases: MongoDB	278
	12.8.1 Using MongoDB with PyMongo	278
12.9	Additional Resources	282
12.10	Self-Evaluation	284

12.1 INTRODUCTION TO DATABASES

The amount of data that is handled in a typical bioinformatics project forces us to use something more versatile than the data structures bundled with Python. Lists, tuples, and dictionaries are very flexible, but they are not suitable to model all the complexity associated with real-world data. Sometimes it is necessary to have a permanent data repository (in computer terms this is called **data persistence**), since data structures are available only while the program is running. While it is possible to write out the data to a file using **pickle**, this is not as efficient as using a database engine designed for that purpose.

What Is a Database?

A database is an ordered collection of related data. Generally, they are constructed to model real-world situations: a person's video collection, the students of a university, a firm's inventory, etc. The database stores relevant data for the users of our program. In modeling the students of a university, we have to take into account the first and last names, the year of entry, and the subjects studied; we wouldn't care about the hair color or height of the student. Designing a database is like modeling a natural process. The first step is to determine what are the relevant variables.

One advantage of databases is that, in addition to data storage, they provide search tools. Some of these searches have immediate replies, such as "how many students are there?" Others are trickier, involving combining different information sources to enable a response, as for example "How many different subjects, on average, did each 2017 freshman take?" In a biological database a typical question might be "What are the proteins with a weight of less than 134 kDa that have been crystallized?" It's interesting to note that there is no need to anticipate all the questions that could be asked; but having an idea of the most common questions will help the design process.

In any case, the advantage of having a database is that we can ask these questions and receive these answers without having to program the search mechanism. That is the job of the **database engine**, which is optimized to quickly handle large amounts of data. Using Python, we communicate with the database engine and process its responses, without having to worry about the internal processes. This doesn't mean we have to totally disengage from the functioning of the database, as the more we understand the internals, the better results we can achieve.

Database Types

Not all databases are the same. There are different theoretical models for describing both the structure of the database and the interrelationships of the data. Some of the most popular models are: hierarchical, network, relational, entity-relationship and document based (or NoSQL). Choosing between the different models is more a job for IT professionals than for bioinformatics researchers. In this chapter we will spend most of our time with the **relational model** due to the flexibility it offers,

the many implementations available, and (why not?) its popularity. There is also an overview NoSQL databases.

A relational database is a database that groups data using common attributes. The resulting sets of organized data can be handled in a logical way.

For example, a data set containing all the real estate transactions in a town can be grouped by the year the transaction occurred; or it can be grouped by the sale price of the transaction; or it can be grouped by the buyer's last name; and so on.

Such a grouping uses the relational model. Hence such a database is called a "relational database." To manage a relational database, you must use a specific computer language called SQL (Structured Query Language). It allows a programmer to create, query, update, and delete data from a database. Although SQL is an ANSI standard, there are multiple non-compatible implementations. Even if they are different, since all versions are based in the same published standard, it is not hard to transfer your knowledge from one SQL dialect to another.

Among the different implementations of relational databases and query languages, this book focuses on two of them: MySQL and SQLite. MySQL (it is pronounced "My Ess Cue Ell") is the most popular database used in web applications, with more than 10 million installations. The great majority of small and medium websites use MySQL. While many system administrators would not consider using MySQL for very demanding applications, there are many high-traffic sites successfully using it. One example of this is YouTube.com. Other popular MySQL-based sites are Wikipedia, Flickr, Facebook, and Slashdot.org.¹ SQLite's target is much more narrowly defined: it is made for small embedded systems, both hardware and software. The Firefox browser uses SQLite internally, as does macOS and others. This versatility is due to its small size (about 250 KB), its lack of external dependencies, and its storage of a database in a single file. These advantages of small size and simplicity are offset by a lack of features, but for its unique niche this is not a problem.

In both cases, the fundamentals are similar and the concepts explained in this chapter are applicable to all relational databases. When a characteristic is exclusive to a database in particular, this will be pointed out.

12.1.1 Database Management: RDBMS

RDBMS stands for **Relational DataBase Management System**. It is software designed to act as an interface between the database engine, the user, and the applications. The just mentioned MySQL and SQLite are examples of RDBMS.²

In the case of MySQL, the RDBMS is separated into two components: A server and a client. The server is the program that accomplishes the hard work associated with the database engine; it can work on our own computer or on a remotely accessible server. The client is the program that gives us an interface to the server.

¹Granted, they are not default installations running on commodity hardware, but highly optimized installations running on branded hardware.

²Other-well known RDBMS are Oracle, DB2, and PostgreSQL.

MySQL provides its own client (`mysql`), which is a command line program, but there are some alternatives. A popular client is `PhpMyAdmin`,³ which requires a web server to run, but provides to the final user a nice Web-based front-end to the MySQL server (see Figure 12.1). There are also desktop clients with the same function, like `MySQL Workbench`⁴ (the free and multi-platform official version from Oracle), `SQLyog`,⁵ and `Navicat`⁶ among others.

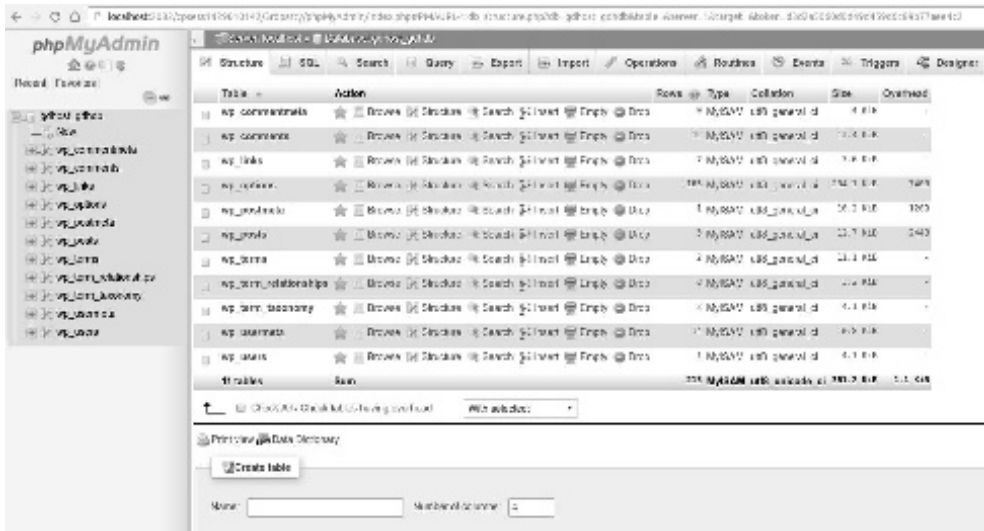


Figure 12.1 Screenshot of PhpMyAdmin: Easy-to-use HTML front end to administer a MySQL database.

SQLite, on the other hand, is available as a library to include in your programs or as a stand-alone executable. Python has a built-in module (`sqlite3`) to interface with SQLite and it works “out of the box” if Python was compiled with SQLite present (the most likely option). It can also be linked to an external executable file with the module `pysqlite2.dbapi2`.

12.1.2 Components of a Relational Database

The first concept of databases we need to understand is that of **entities**. Formally, an entity is defined as every significant element that should be stored. We should distinguish between an **entity type** and the **occurrence** of an entity. In an administration database, `Students` is an entity type, while each student in particular is an occurrence of this entity.

Each **entity** has its own **attributes**. The **attributes** are the data associated to an **entity**. Let’s go back to the college administration database we have just

³<http://www.phpmyadmin.net>

⁴<https://dev.mysql.com/downloads/workbench/>

⁵<http://webyog.com/en/>

⁶<http://www.navicat.com>

schemed. *name*, *lastname*, *DateJoined*, and *OutstandingBalance* are attributes of the **entity** *Students*.

In turn, each entity has its own attributes. The attributes are the data associated with an entity. Let's create a college administration database, where **Name**, **Lastname**, **DateJoined**, and **OutstandingBalance** are attributes of the entity *Students*.

The data in a relational database are not isolated, but as the name implies, they are represented by **relations**. A relation maps a key, or a grouping of keys, with a grouping of rows. Each key corresponds to an occurrence of one entity, which relates to the group of attributes associated with that occurrence. These relationships are displayed as tables, independently of how they are stored physically. A database can have multiple tables. Continuing the example of the university administration database, we might have a table with information on the students and another on the professors, as each entity has its own attributes.

In [Table 12.1](#) we can see an example of the **students** relation.

TABLE 12.1 Students in Python University

Name	LastName	DateJoined	OutstandingBalance
Harry	Wilkinson	2006-02-10	No
Jonathan	Hunt	2004-02-16	No
Harry	Hughes	2005-03-20	No
Kayla	Allen	2001-03-15	Yes
Virginia	Gonzalez	2003-04-02	No

A Key Concept: Primary Key

Every table has to have a means of identifying a row of data; it must have an attribute, or group of attributes, that serves as a unique identifier. This attribute is called a **primary key**. In the case that no single attribute can be used as a primary key, several can be taken simultaneously to make a **composite key**. Returning to [Table 12.1](#), we can see that the attribute *Name* cannot be used as a primary key, as there is more than one occurrence of an entity with the same attribute (Joe Campbell and Joe Doe share the same first name). One solution to this problem would be to use *Name* and *LastName* as a composite key; but this would not be the best solution, because it's still possible to have more than one occurrence of an entity sharing this particular composite key, such as another Joe Doe. For this reason, normally we add to the table an ID field—a unique identifier—instead of depending on the data to have a primary key. In most databases there are mechanisms for automatically generating such a primary key when we insert data. Let us look at a version of [Table 12.1](#) with a new attribute that can be used as the primary key:

TABLE 12.2 Table with primary key

ID	Name	LastName	DateJoined	OutstandingBalance
1	Harry	Wilkinson	2006-02-10	No
2	Jonathan	Hunt	2004-02-16	No
3	Harry	Hughes	2005-03-20	No
4	Kayla	Allen	2001-03-15	Yes
5	Virginia	Gonzalez	2003-04-02	No

12.1.3 Database Data Types

As in programming languages, databases have their own data types. For example, in Python we have *int*, *float*, and *string* (among others); databases have their own data types such as *tinyint*, *smallint*, *mediumint*, *int*, *bigint*, *float*, *char*, *varchar*, *text*, and others. You may be wondering why there are so many data types (such as five different data types for integers). The main reason is that with so many options it is possible to make the best use of available resources. If we need a field where we wish to store the age of the students, we can achieve that with a field of type *tinyint*, as it supports a range of values between -128 and 127 (which can be stored in one byte). Of course, we can just as well store it in a field of type *int*, which supports a range between -2147483648 to 2147483647 (that is, 4 bytes); but that would be a waste of memory, as the system must unnecessarily reserve space. Because of the difference in the number of bytes, a number stored as *int* occupies 4 times as much RAM and disk space as one stored as *tinyint*. The difference between one and four bytes may seem insignificant and not worth mentioning, but then multiply it by the number of data entries you have; when the dataset is large enough, disk space and access time could be an issue. That is why you should be aware of the data type storage requirements.⁷

Table 12.3 summarizes the characteristics of the main data types in MySQL. Note that some of the minor characteristics may vary depending on the version of MySQL used, which is why it is advisable to consult the documentation for your particular version.⁸ In the case of SQLite, there are only 5 data types: *INTEGER*, *REAL*, *TEXT*, *BLOB*, and *NULL*. However, one must realize that SQLite is typeless, and that any data can be inserted into any column. For this reason, SQLite has the idea of “type affinity”: it treats the data types as a recommendation, not a requirement.⁹

⁷Estimating what data types are adequate for the situation is no minor issue. In the online multi-player game *World of Warcraft*, some players found they could not receive more gold when they had reached the limit of the variable in which money was stored, a signed 32-bit integer. Much more serious was the case of the software in the Ariane 5 rocket when a 64-bit real was converted to a 16-bit signed integer. This led to a cascade of problems culminating in destruction of the entire flight, costing US\$ 370 million.

⁸MySQL has a complete online reference manual. Data Type documentation for MySQL 5.7 is available at <https://dev.mysql.com/doc/refman/5.7/en/data-types.html>.

⁹For more information about the idea of “type affinity” I recommend the section “Datatypes in SQLite Version 3” (<http://www.sqlite.org/datatype3.html>) of the SQLite online documentation.

TABLE 12.3 Most Used MySQL Data Types

Data type	Comment
TINYINT	±127 (0-255 UNSIGNED.)
SMALLINT	±32767 (0-65535 UNSIGNED.)
MEDIUMINT	±8388607 (0-16777215 UNSIGNED.)
INT	±2147483647 (0-4294967295 UNSIGNED.)
BIGINT	±9223372036854775807 (0-18446744073709551615 UNSIGNED.)
FLOAT	A small number with a floating decimal point.
DOUBLE	A large number with a floating decimal point.
DATETIME	From '1000-01-01 00:00:00' to '9999-12-31 23:59:59'
DATE	From '1000-01-01' to '9999-12-31'
CHAR(n)	A fixed section with n characters long (up to 255).
VARCHAR(n)	A variable section with n characters long (up to 255).
TEXT	A string with a maximum length of 65535 characters.
BLOB	A binary string version of TEXT.
MEDIUMTEXT	A string with a maximum length of 16777215 characters.
MEDIUMBLOB	Binary string equivalent to MEDIUMTEXT.
LONGTEXT	A string with a maximum length of 4294967295 characters.
LOBLOB	Binary string equivalent to LONGTEXT.
ENUM	String value taken from a list of allowed values.

12.2 CONNECTING TO A DATABASE

To connect to the MySQL database server, you need a valid user, and to set up a user, you need to connect to the database. This catch-22 is solved by accessing the server with the default credentials (user: “root”, and no password). From the command line, if the server is in the same computer, it is possible to access with this command:

```
$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 234787469
Server version: 5.5.51-38.2 Percona Server (GPL), Release 38.2
```

Copyright (c) 2000, 2016, Oracle. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

```
mysql>
```

From now on, interaction with MySQL server will be shown by using the php-MyAdmin front-end.

12.3 CREATING A MYSQL DATABASE

Before working with a database, we should create one. You could skip this step if you plan to access a database previously created. But it is likely that sooner or later you will need to create your own database. Creating a database is a simple task and will help you to understand the data you are going to handle, and create more effective queries.

Since database creation is something that is done only once for each database, there is not much need to automate this task with a program. This step is usually done manually. My recommendation is to use a graphical tool to design the database. phpMyAdmin or Navicat will do the job.

To create a database from the MySQL console:

```
mysql> CREATE DATABASE PythonU;
Query OK, 1 row affected (0.01 sec)
```

This will create the **PythonU** MySQL database. To create a database from phpMyAdmin, press “New” in the left panel and fill a form field with the proposed name of the database in “**Create new database**” (see [Figure 12.2](#)).

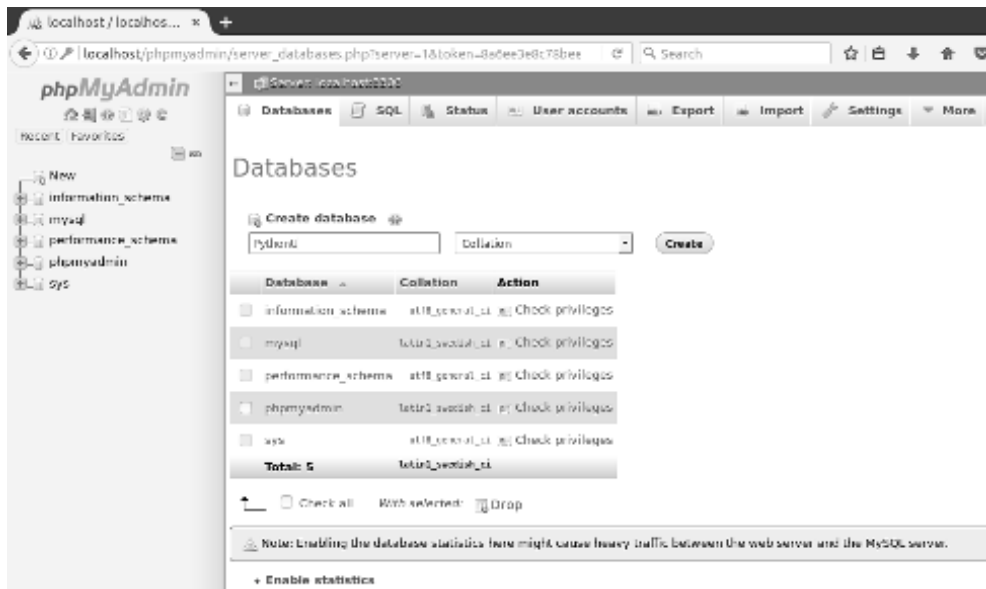


Figure 12.2 Creating a new database using phpMyAdmin.

12.3.1 Creating Tables

Once we have a newly created database, the next step is to create the tables where the data will be stored. Creating the tables using this kind of software doesn't seem a problem worth mentioning in this book, so we will focus more on the table structure rather than on the procedure for dealing with a GUI tool.

We must keep in mind that a table represents a relationship between the data; it makes no sense to create a table for one entity and then populate it with data of another entity. Continuing with the example of our “Python University,” we can think about what information related to students we need to store in the `Students` table.

As we saw earlier, in the table `Students` we assigned the following fields: `ID`, `Name`, `LastName`, `DateJoined`, and `OutstandingBalance`.

There are “good practices,” for database design. It is certainly not easy to convey the necessary knowledge to achieve an efficient design for every situation in this space; in any case, good database design is something that one learns with practice.

Let's see how we define each field in this case:

ID: Is a unique id for each registrant. Since Python University is expected to have several students, an unsigned INT data type is used (up to 4294967295). There is no need to use negative numbers in an ID, so this field should be set as unsigned.

Name: Since the size of a name is variable with less than 255 characters, VARCHAR is used. The maximum size for names in characters, according to my arbitrary criteria, is 150.

LastName: This field was set with the same criteria as the former field. The only difference is in the maximum size for a last name; which is set to 200 characters.

DateJoined: There is not much choice here. A simple DATE field would do it best.

OutstandingBalance: This field represents whether the student has paid the tuition in full or not. Since there are only two possible values (paid or not paid), a BOOL data type is chosen. This data type stores a 0 or a 1. It is up to the programmer to assign a meaning to this value, but in mathematical notation, 0 stands for FALSE and 1 for TRUE, so this convention is generally used.

The last choice is the table type (**InnoDB** or **MyISAM**). In this case it is OK to leave the default option (MyISAM), which will be appropriate for most uses. Please see Advanced Tip: MyISAM vs InnoDB on page 265 for a brief discussion on both table types.

If you want to manually create the table, first you must select the database to use:

```
mysql> use PythonU;
Database changed
```

And after the database is selected, type these commands into the MySQL prompt (also available at the book GitHub repository as `db/studentstbl.sql`):

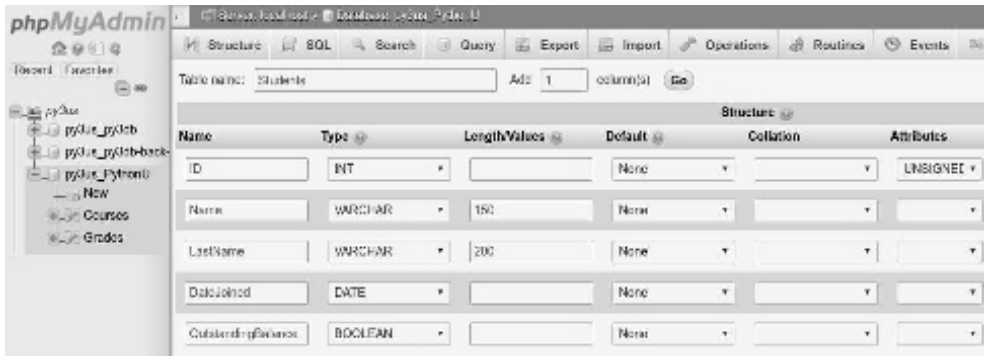


Figure 12.3 Creating a new table using phpMyAdmin.

```
CREATE TABLE 'Students' (
  'ID' INT UNSIGNED NOT NULL AUTO_INCREMENT,
  'Name' VARCHAR(150) NOT NULL,
  'LastName' VARCHAR(200) NOT NULL,
  'DateJoined' DATE NOT NULL,
  'OutstandingBalance' BOOLEAN NOT NULL,
  PRIMARY KEY ('ID') ) ENGINE = MyISAM;
```

No wonder I recommended the use of a GUI to design the table!

Tip: Creating a Database Using Another Database as a Template.

Instead of manually defining each field on each table, you could import a “MySQL dump” from another database and create a database in one step. There are two different kinds of dump files: “Structure only” and “structure and data” dump files. Both files are imported the same way into a database:

```
$ mysql -p database_name < dbname.sql
```

Where do you get the dump file from?. You can get a dump file from the backup of another database or from the installation files of a program that requires a database.

12.3.2 Loading a Table

Once we have the table created, it is time to load the data into it. This operation can be done from any MySQL front-end, either row by row or in batch. Since there are several data to load at the beginning, and the manual data load is intuitive, let’s see how to load data in batch mode.

The most common way to upload data is by using csv files. This kind of file was

reviewed in Section 5.3 (page 90). To upload the data that is seen in [Table 12.2](#), we can prepare a csv file (`dbdata.csv`) with the following format:

```
1,Harry,Wilkinson,2016-02-10,0
2,Jonathan,Hunt,2014-02-16,0
3,Harry,Hughes,2015-03-20,0
4,Kayla,Allen,2016-03-15,1
5,Virginia,Gonzalez,2003-04-02,0
```

To load the csv file into the MySQL database, use the `LOAD DATA INFILE` command¹⁰ at the MySQL prompt:

```
mysql> LOAD DATA LOCAL INFILE 'dbdata.csv' INTO TABLE Students
        FIELDS TERMINATED BY ',';
```

If you want to avoid doing it by yourself, there is also a web service that convert a CSV file into a MySQL table at <https://sqlizer.io>. It is free to convert a file to SQL for up to 5000 rows of data and for personal use. My advice is to try to do it by yourself, it is not so hard.

An alternative way, using **INSERT** statements:

```
INSERT INTO 'Students' ('ID', 'Name', 'LastName', 'DateJoined',
    'OutstandingBalance') VALUES
(1, 'Harry', 'Wilkinson', '2016-02-10', 0),
(2, 'Jonathan', 'Hunt', '2014-02-16', 0),
(3, 'Harry', 'Hughes', '2015-03-20', 0),
(4, 'Kayla', 'Allen', '2016-03-15', 1),
(5, 'Virginia', 'Gonzalez', '2017-04-02', 0);
```

Once the data is loaded into the database, the table looks like the one in [Figure 12.4](#).

Advanced Tip: MyISAM versus InnoDB

There are several formats for the internal data structures in MySQL tables. The most commonly used formats are InnoDB and MyISAM. MyISAM is used by default and is characterized by its higher reading speed (using `SELECT` operations), and it uses less disk space. It is slower than InnoDB at writing, since when data is being recorded, the table is momentarily blocked until finished, so all other operations must wait to complete. This limitation doesn't exist in an InnoDB table. The main advantage of this format is that it allows secure transactions and has a better

¹⁰For a complete reference of this command, see the MySQL online manual at <https://dev.mysql.com/doc/refman/5.7/en/load-data.html>.

ID	Name	LastName	DateJoined	OutstandingBalance
1	Harry	Wilkinson	2016-02-10	0
2	Jonathan	Hunt	2014-02-16	0
3	Harry	Hughes	2015-03-20	0
4	Kayla	Allen	2016-03-15	1
5	Virginia	Gonzalez	2017-04-02	0

Figure 12.4 View of the Student table.

crash recovery. InnoDB is thus recommended for intensively updated tables and for storing sensitive information. To sum up, since you can use different table types in the same database, choose the appropriate table type according to the operation that will be performed most on each table.

12.4 PLANNING AHEAD

Making a database requires some planning. This is specially true when there is a large amount of data and we want to optimize the time it takes to answer our queries. A bad design can make a database unusable. In this section I present a sample database to show the basis of database design. To have a better idea of designing relational databases, you could read about *Database normalization* in “Additional resources.”, but this section should give you a brief glimpse of this subject.

12.4.1 PythonU: Sample Database

Let’s keep the example of a student database from the fictional Python University (whose database is called **PythonU**), to store student data and subjects taken. To store student data, we already have the table **Students**. We have to make a table to store the grades associated with each subject (a “Grades” table). As in many other aspects of programming, there is more than one way to accomplish this. We will start by showing some non-optimal ways, to better understand why there is a recommended way.

Grades Table

In the table **Grades** we want to store for each student: subjects studied, grade, and date when each course was taken.

A proposed **Grades** table for two courses is depicted in [Figure 12.5](#).

This design (schema) has several flaws. The first design error in the table is its

StudentID	Python_101	Python_101-TERM	Mathematics_for_CS	Mathematics_for_CS-TERM
1	7	2016/1	8	2016/1
2	6	2015/1	9	2015/2
3	5	2016/1	7	2016/1
4	9	2016/1	8	2016/2

Figure 12.5 An intentionally faulty “Grades” table.

inflexibility. If we want to add a new course, we have to modify the table. This is not considered good programming practice; a change in the structure of an already populated table is an expensive operation that must be avoided whenever possible. The other problem that arises from this design, as seen in the diagram, is that there is no place to store the grade of a student who has taken a course more than once. How do we solve this? With a more intelligent design. An almost optimal solution can be seen in [Figure 12.6](#).

StudentID	Course	Grade	Term
1	Python 101	7	2016/1
1	Mathematics for CS	8	2016/1
2	Python 101	6	2015/1
2	Mathematics for CS	9	2015/2
3	Python 101	5	2016/1
3	Mathematics for CS	7	2016/2
4	Python 101	9	2016/1
4	Mathematics for CS	8	2016/2

Figure 12.6 A better “Grades” table.

The first problem, the need to redesign the table for entering new subjects, we solve by entering the course name as a new field: **Course**. This field can be of type TEXT or VARCHAR. And the problem of being able to keep track of when a student took a course more than once, we solved with the **Term** field. While this is a decidedly better design than the previous one, it is far from being optimal. It is evident that storing the name of each subject for each student is an unnecessary waste of resources. A way to save this space is to use the data type ENUM in the field **Course**; in this way we can save a substantial amount of space, because MySQL internally uses one or two bytes for each entry of this type. The table remains the same as seen before ([Figure 12.6](#)), and only changes the way the field **Course** is defined, saving disk space as mentioned.

Is this the best way? The problem with using ENUM with the field **Course** is that when we wish to add a new subject, we still have to alter the table structure. This modification, to add a new option to the ENUM, is not as costly as adding a new column, but conceptually it is not a good idea to modify the definition of

a new table in order to accommodate a new type of data. In cases like these, we resort to “lookup tables.”

Courses Table

CourseID	Course_Name
1	Python 101
2	Mathematics for CS

Figure 12.7 Courses table: A lookup table.

A lookup table is a reference table that is used to store values that are used as content of a column located in another table. Continuing with the example of Python University, we can make a lookup table for the subjects (see [Figure 12.7](#)).

This **Courses** table contains a field for storing the ID of the course (**CourseID**) and another for the name of the course (**Course_Name**). For this scheme to work, we must change the field **Course** of the table **Grades**; in place of an ENUM field, we now use an INT field (see [Figure 12.8](#)).

StudentID	Course	Grade	Term
1	1	7	2016/1
1	2	8	2016/1
2	1	6	2015/1
2	2	9	2015/2
3	1	5	2016/1
3	2	7	2016/1
4	1	9	2016/1
4	2	8	2016/2

Figure 12.8 Modified “Grades” table.

The data in **CourseID** now correspond to that of the field **Course** in the table **Grades**. Using a single lookup, we can then link the ID with the corresponding course name. This way we save the same amount of space in the **Students** table as when we used an ENUM for the **Course** field, with the additional advantage that we can expand the list of subjects simply by adding one element to the **Courses** table.

Tip: ENUM field type versus Lookup Table

We have seen how convenient it is to use a lookup table in place of an ENUM

field. You are probably wondering how to decide when to use one strategy or the other when designing your database. ENUM is better than TEXT or VARCHAR in the cases where the number of possibilities is limited and not expected to vary: for example, a list of colors, the months of the year, and other options that by their very nature have a set range. One disadvantage that should be taken into account with regard to ENUM, is that it is a data type specific to MySQL, which may not be available on other DB engines; this limits the potential portability of the database.

Now we have the PythonU database with 3 tables: **Students**, **Grades**, and **Courses**. It's time to learn how to construct queries.

12.5 SELECT: QUERYING A DATABASE

The most useful operation in a database, once it is created and populated, is querying its contents. We can extract information from one table or many tables simultaneously. For example, to have a list of students, the table **Students** must be queried. On the other hand, if we want to know a student's average grades, we need to query the **Students** and **Grades** tables. In addition, there are cases where one must query 3 tables simultaneously, as when finding out a student's grade in one particular subject.

Let's look at each case:

Simple Query

To obtain a listing of students (first and last names) from the **Students** table, we would use the following command at the MySQL prompt:

```
mysql> SELECT Name, LastName FROM Students;
+-----+-----+
| Name      | LastName |
+-----+-----+
| Harry     | Wilkinson|
| Jonathan  | Hunt     |
| Harry     | Hughes   |
| Kayla     | Allen    |
| Virginia  | Gonzalez |
+-----+-----+
5 rows in set (0.00 sec)
```

Combining Two Queries

To obtain the average grade of a given student, we need to extract all the grades corresponding to that student. As the grades are in the **Grades** table and the names

in the `Students` table, we need to query both tables in order to receive a reply to our question. First we need to query `Students` for the ID of the student; then with this ID we must search for all corresponding records.

To get the grade average of Harry Wilkinson:

```
SELECT AVG(Grade) FROM Grades
WHERE StudentID = (SELECT ID FROM Students
WHERE Name='Harry' AND LastName='Wilkinson');
```

We can also accomplish it with a single query, without using the nested `SELECT`:

```
SELECT AVG(Grade) FROM Grades, Students
WHERE Grades.StudentID=Students.ID
AND Students.Name='Harry' AND Students.LastName='Wilkinson';
```

There are two new things to understand in this example: When we use fields from more than one table, we should prepend the table name to avoid ambiguities in the field names. Thus, `StudentID` becomes `Grades.StudentID`. The following statement is equivalent to the above:

```
SELECT AVG(Grade) FROM Grades, Students
WHERE StudentID=ID AND Name='Harry' AND LastName='Wilkinson';
```

If the field name is present only in one table, there is no need to add the table name, but it makes the query easier to parse for the programmer.

The other feature worth pointing out in this example is that instead of looking only at the student ID, there is a condition that matches the IDs of both tables (`Grades.StudentID = Students.ID`).

In either case, the result is 7.5.

Querying Several Tables

To retrieve the grade average of one student (Harry Hughes) in one particular course (Python 101), there is a need to build a query using more than one table:

```
SELECT Grades.Grade FROM Grades, Courses, Students
WHERE Courses.CourseID = Grades.Course
AND Courses.Course_Name = 'Python 101'
AND Students.ID = Grades.StudentID
AND Students.Name = 'Harry' AND Students.LastName = 'Hughes';
```

12.5.1 Building a Query

The general syntax of `SELECT` statements is

```
SELECT field(s)_to_retrieve FROM table(s)_where_to_look_for
WHERE condition(s)_to_met] [ORDER BY ordering_criteria]
[LIMIT limit_the_records_returned];
```

To use grouping functions, include at the end of your query:

```
GROUP BY variable_to_be_grouped HAVING condition(s)_to_met
```

The aggregating functions are `AVG()`, `COUNT()`, `MAX()`, `MIN()` and `SUM()`.

Note that `HAVING` works like `WHERE`. The difference is that `HAVING` is used only with `GROUP BY` since it restricts the records after they have been grouped.

These constructs can be understood better with actual examples. The following cases show how to execute the queries from the MySQL command line.

To get all the elements of a table, use wildcards:

```
mysql> select * from Students;
Connection id:      234793415
Current database: PythonU
```

```
+----+-----+-----+-----+-----+
| ID | Name      | LastName | DateJoined | OutstandingBalance |
+----+-----+-----+-----+-----+
|  1 | Harry     | Wilkinson | 2016-02-10 | 0 |
|  2 | Jonathan  | Hunt      | 2014-02-16 | 0 |
|  3 | Harry     | Hughes    | 2015-03-20 | 0 |
|  4 | Kayla     | Allen     | 2016-03-15 | 1 |
|  5 | Virginia  | Gonzalez  | 2017-04-02 | 0 |
+----+-----+-----+-----+-----+
5 rows in set (0.08 sec)
```

To obtain a count of all elements in a table:

```
mysql> select COUNT(*) from Students;
+-----+
| COUNT(*) |
+-----+
|         5 |
+-----+
1 row in set (0.00 sec)
```

To see the grade average of all students:

```
mysql> select avg(Grade) from Grades GROUP BY StudentID;
+-----+
| avg(Grade) |
+-----+
|      7.5000 |
|      7.5000 |
|      6.0000 |
|      8.5000 |
+-----+
4 rows in set (0.17 sec)
```

To retrieve the best grade of one particular student (Harry Wilkinson):

```
mysql> select max(Grades.Grade) from Grades,Students
WHERE studentID=ID AND Students.Name='Harry'
AND Students.Lastname='Wilkinson';
+-----+
| max(Grades.Grade) |
+-----+
|                  8 |
+-----+
1 row in set (0.00 sec)
```

Which courses have the string “101” in their names?

```
mysql> SELECT Course_Name FROM Courses
WHERE Course_Name LIKE '%101%';
+-----+
| Course_Name |
+-----+
| Python 101  |
+-----+
1 row in set (0.00 sec)
```

Note that % is used as a wildcard character when working with strings.

How many students have flunked a class? Supposing that the passing grade is 7, this query is equivalent to asking how many grades are below 7.

```
mysql> SELECT Name,LastName,Grade FROM Students,Grades
WHERE Grades.Grade<7 and Grades.StudentID=Students.id;
+-----+-----+-----+
| Name      | LastName | Grade |
+-----+-----+-----+
| Jonathan  | Hunt     | 6     |
```

```
| Harry      | Hughes      |      5 |
+-----+-----+-----+
2 rows in set (0.00 sec)
```

The above was simply an example of the possibilities of the `SELECT` command. For more complex queries I recommend the resources indicated in “Additional Resources.”

12.5.2 Updating a Database

While values can be changed using any of the aforementioned GUI tools, it’s good to know the syntax for updating data, to enable implementing it from Python when necessary.

The general syntax is:¹¹

```
UPDATE table_name(s) SET variable1=expr1 [,variable2=expr2 ...]
[WHERE condition(s)];
```

Suppose you want the database to reflect the fact that Joe Campbell didn’t pay his tuition, therefore we must make sure the `OutstandingBalance` field in the `Students` table is set to Y. Here is the SQL command with the server’s response:

```
mysql> UPDATE Students SET OutstandingBalance='Y'
WHERE Name='Harry' and LastName='Wilkinson';
Query OK, 1 row affected (0.67 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

It is also possible, instead of changing a specific value, to apply a function ¹² to all values in a column. For example, to subtract one point from all grades:

```
mysql> UPDATE Grades SET Grade = Grade-1;
Query OK, 8 rows affected (0.00 sec)
Rows matched: 8  Changed: 8  Warnings: 0
```

12.5.3 Deleting a Record from a Database

To delete a record use the `DELETE` command:

```
mysql> DELETE from Students WHERE ID = "5";
Query OK, 1 row affected (0.02 sec)
```

¹¹For more information on this command, see the MySQL manual at <https://dev.mysql.com/doc/refman/5.7/en/update.html>.

¹²Any valid MySQL function can be used. To see a list with available functions, check the MySQL manual at <https://dev.mysql.com/doc/refman/5.7/en/functions.html>.

As in `SELECT`, the `WHERE` clause specifies the conditions that identify which rows to delete. Without the `WHERE` clause, all rows are deleted. But this is not the best way to delete a whole table. Instead of deleting all records row by row, you can use the `TRUNCATE` command, which drops and re-creates the table. This is faster for large tables.

12.6 ACCESSING A DATABASE FROM PYTHON

Now that we know how to access our data using SQL, we can take advantage of Python's tools for interfacing with databases.

12.6.1 PyMySQL Module

This module allows accessing MySQL databases from Python.¹³ It's not installed by default; in a shared web hosting environment you may have to request the installation of the PyMySQL Python module. To know if the module is installed, try importing it. If you get an import error, it's not installed:

```
>>> import pymysql
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'pymysql'
>>>
```

To install it, use `pip` or `conda`:

```
(py4bio) $ pip install PyMySQL
Collecting PyMySQL
  Downloading PyMySQL-0.7.10-py2.py3-none-any.whl (78kB)
    100% |*****| 81kB 934kB/s
Installing collected packages: PyMySQL
Successfully installed PyMySQL-0.7.10
```

12.6.2 Establishing the Connection

There is the `connect` method in the `MySQLdb` module. This method returns a connection object that we'll need to act upon later, so we should give it a name (in the same way we would give a name to the object resulting from opening a file):

```
>>> import pymysql
>>> db = pymysql.connect(host="localhost", user="root",
... passwd="mypassword", db="PythonU")
```

¹³There is another module to access MySQL database called **MySQLdb**. This module was featured in a previous edition of this book, but is no longer used in this edition because at this time is not Python 3 compatible.

12.6.3 Executing the Query from Python

Once the connection to the database is established, we have to create a **cursor**. A cursor is a structure used to walk through the records of the result set.

The method used to create the cursor has a clever name, **cursor()** :

```
>>> cursor = db.cursor()
```

The connection is established, and the cursor has been created. It is time to execute some SQL commands:

```
>>> cursor.execute("SELECT * FROM Students")
5
```

The **execute** method is used to execute SQL commands. Note that there is no need to add a semicolon (;) at the end of the command. Now the question is how to retrieve data from the cursor object. To get one element, use **fetchone()**:

```
>>> cursor.fetchone()
(1, 'Harry', 'Wilkinson', datetime.date(2016, 2, 10), 0)
```

fetchone() returns a row with the elements of the first record of the table. Remaining records can be extracted one by one in the same way:

```
>>> cursor.fetchone()
(2, 'Jonathan', 'Hunt', datetime.date(2014, 2, 16), 0)
>>> cursor.fetchone()
(3, 'Harry', 'Hughes', datetime.date(2015, 3, 20), 0)
```

In contrast, **fetchall()** extracts all the elements at once:

```
>>> cursor.fetchall()
((1, 'Harry', 'Wilkinson', datetime.date(2016, 2, 10), 0),
 (2, 'Jonathan', 'Hunt', datetime.date(2014, 2, 16), 0),
 (3, 'Harry', 'Hughes', datetime.date(2015, 3, 20), 0),
 (4, 'Kayla', 'Allen', datetime.date(2016, 3, 15), 1),
 (5, 'Virginia', 'Gonzalez', datetime.date(2017, 4, 2), 0))
```

Which method to use depends on the amount of data returned, the available memory, and above all, what we're trying to accomplish. When working with limited datasets, there's no problem using **fetchall()**; but if the result is too large to fit in memory, one must implement a strategy like the one found in [Listing 12.1](#).

Listing 12.1: `pymysql1.py`: Reading results once at a time

```

1 import pymysql
2 db = pymysql.connect(host='localhost',
3                       user='root', passwd='secret', db='PythonU')
4 cursor = db.cursor()
5 recs = cursor.execute('SELECT * FROM Students')
6 for x in range(recs):
7     print(cursor.fetchone())

```

While the code in [Listing 12.1](#) works flawlessly, it was shown as an example of using *fetchone()*. It is possible to iterate directly over the cursor object:

Listing 12.2: pymysql12.py: Iterating directly over the DB cursor

```

1 import pymysql
2 db = pymysql.connect(host='localhost',
3                       user='root', passwd='secret', db='PythonU')
4 cursor = db.cursor()
5 cursor.execute('SELECT * FROM Students')
6 for row in cursor:
7     print(row)

```

12.7 SQLITE

In **SQLite** a new database is created when passing a file name in the command line, as in:

```

$ sqlite3 PythonU.db
SQLite version 3.3.5
Enter ".help" for instructions
sqlite>

```

Basic syntax to create a table in **SQLite**:

```

CREATE TABLE table_name(
    column1 datatype PRIMARY KEY(one or more columns),
    column2 datatype,
    .....
    columnN datatype);

```

For example, here is the command to create the Students table:

```

sqlite> create table Students(
ID int,
Name text,

```

```

LastName char,
DateJoined datetext,
OutstandingBalance Boolean);

```

To import the data from a CSV file, set the separator and do the import:

```

sqlite> .separator ,
sqlite> .import dbdata.csv Students

```

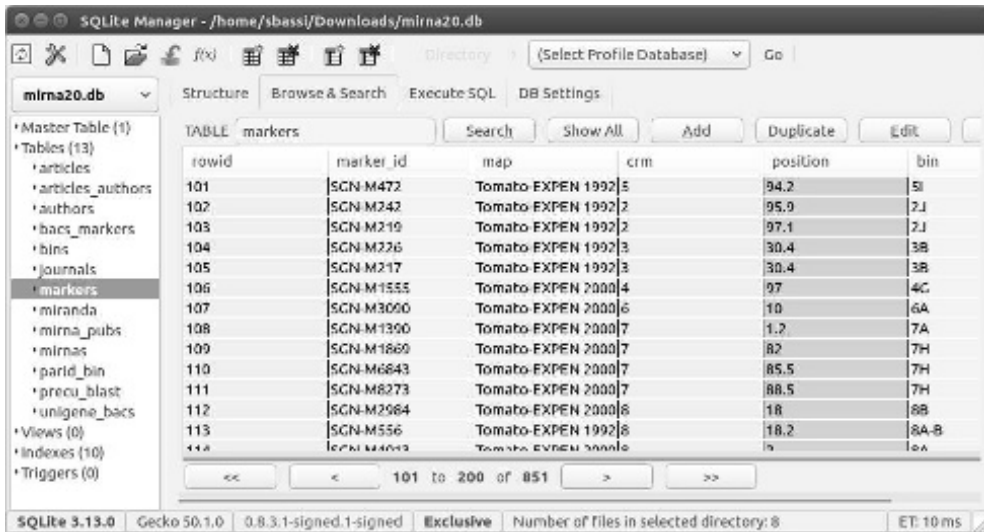


Figure 12.9 Screenshot of SQLite manager: A SQLite GUI as a Firefox add-on.

The following example shows that, practically speaking, there is no difference in working with one database type or another:

Listing 12.3: `sqlite1.py`: Same script as 12.2, but with SQLite

```

1 import sqlite3
2 db = sqlite3.connect('../samples/PythonU.db')
3 cursor = db.cursor()
4 cursor.execute('Select * from Students')
5 for row in cursor:
6     print(row)

```

The only thing that changed in Listing 12.3 with respect to Listing 12.2 was the first two lines. In line 1, module `sqlite3` was imported instead of `MySQLdb`. Meanwhile, in line 2 the connection code is far simpler, as it does not require a password or a username to connect to an SQLite database.¹⁴

¹⁴Access permissions can be applied by using the normal file access permissions of the underlying operating system.

This is the output of `sqlite1.py` (Listing 12.3):

```
(1, 'Harry', 'Wilkinson', '2016-02-10', 0)
(2, 'Jonathan', 'Hunt', '2014-02-16', 0)
(3, 'Harry', 'Hughes', '2015-03-20', 0)
(4, 'Kayla', 'Allen', '2016-03-15', 1)
(5, 'Virginia', 'Gonzalez', '2003-04-02', 0)
```

As with MySQL, there are some GUIs for SQLite. SQLite Administrator¹⁵ is a Windows application¹⁶ that allows the user to create new databases or modify existing ones. SQLite Manager¹⁷ has similar capacities but is available both for Windows and macOS. A multi-platform SQLite front-end is the SQLite Manager Firefox add-on,¹⁸ it works on any platform on which Firefox browser runs. See Figure 12.9 for a screen-shot of SQLite Manager.

12.8 NOSQL DATABASES: MONGODB

There are several NoSQL type of databases, such as Cassandra, CouchDB, and MongoDB. This book covers the latest because it is a good product, it is open source and has extensive Python support.

Why would I use non-relational database? This type of database has several advantages over the classic SQL databases: It has no schema (AKA schemaless), so it is better for semi-structured, complex, or nested data. This also means that new properties can be added “on the fly” without the need to restructure a table and change current stored data. Performance gains are obtained because there are usually fewer requests and table lookups to get the same data as in SQL. Also, by reducing consistency you can get more performance (this is mostly used in heavily demand scenario and for data that is not critical). Another advantage is the scalability; instead of upgrading the server as in SQL databases, in most cases you can upgrade DB capability buy adding more servers.

If you are not planning to be in any described scenario, you may not need to a NoSQL database. This type of database is not a silver bullet. It is not recommended for every situation but is worth learning about it so you can recognize the right moment to use it and be ready when the time comes.

12.8.1 Using MongoDB with PyMongo

In order to follow the rest of this chapter, you need to either install MongoDB or have access to a MongoDB installation. A local install is fairly easy to accomplish. Download the last version from the MongoDB download center.¹⁹ If you choose to

¹⁵ Available at <http://sqliteadmin.orbmu2k.de>.

¹⁶ It also works on Linux with Wine.

¹⁷ Available from <http://www.sqlabs.net/sqlitemanager.php>.

¹⁸ Available at <http://code.google.com/p/sqlite-manager>.

¹⁹ <https://www.mongodb.com/download-center>

use a service that provides MongoDB on the cloud (like Mlab²⁰), you don't need to install a local server. In any case, you will need to install **PyMongo**:

```
(py4bio) $ pip install pymongo
Collecting pymongo
  Downloading pymongo-3.4.0-cp35-cp35m-manylinux1_x86_64.whl (359kB)
    100% |*****| 368kB 3.0MB/s
Installing collected packages: pymongo
Successfully installed pymongo-3.4.0
```

Its use is similar to PyMySQL. Import the Mongo Client:

```
>>> from pymongo import MongoClient
```

The general syntax to instantiate the Mongo Client is:

```
MongoClient(CONNECTION_STRING)
```

where the connection string takes this form:

```
'mongodb://USER:PASSWORD@DOMAIN:PORT/DB'
```

If the MongoDB server is in the local machine:

```
>>> from pymongo import MongoClient
>>> client = MongoClient('localhost:27017')
```

To see databases available in this MongoDB server:

```
>>> client.database_names()
['Employee', 'admin', 'local']
```

The command to create a DB is the same as to connect to existing one:

```
>>> db = client.PythonU
```

Verify that it is already created:

```
>>> client.database_names()
['Employee', 'PythonU', 'admin', 'local']
```

You can delete a database with **drop_database** method:

```
>>> client.drop_database('Employee')
```

²⁰<https://mlab.com>

To create a collection (the equivalent of a SQL table) inside a DB, use the same method as used when creating a DB:

```
>>> students = db.Students
```

This “students” collection is ready to accept documents. A document is the equivalent of a record in SQL. Unlike SQL records, they are stored as JSON documents and can have any structure (that is why it is also called schemaless). With **pymongo** we can insert Python dictionaries instead of JSON documents. Here are some Python dictionaries with all information related to each student (shown here only first two records):

```
{
    'Name': 'Harry',
    'LastName': 'Wilkinson',
    'DateJoined': '2016-02-10',
    'OutstandingBalance': False,
    'Courses': [('Python 101', 7, '2016/1'),
                ('Mathematics for CS', 8, '2016/1')]
}

{
    'Name': 'Jonathan',
    'LastName': 'Hunt',
    'DateJoined': '2014-02-16',
    'OutstandingBalance': False,
    'Courses': [('Python 101', 6, '2016/1'),
                ('Mathematics for CS', 9, '2015/2')]
}
```

This is a difference with SQL databases. Instead of having this information distributed into 3 tables, and having to resort to relationships, it is all in one document.

To insert these dictionaries into the collection:

```
>>> student_1 = {'Name':'Harry', 'LastName':'Wilkinson',
                  'DateJoined':'2016-02-10', 'OutstandingBalance':False,
                  'Courses':[('Python 101', 7, '2016/1'), ('Mathematics for CS',
                  8, '2016/1')]}
>>> student_2 = {'Name':'Jonathan', 'LastName':'Hunt',
                  'DateJoined':'2014-02-16', 'OutstandingBalance':False,
                  'Courses':[('Python 101', 6, '2016/1'), ('Mathematics for CS',
                  9, '2015/2')]}
>>> students.count()
0
```

```
>>> students.insert(student_1)
ObjectId('58b64f201d41c8066755035e')
>>> students.insert(student_2)
ObjectId('58b64f251d41c8066755035f')
>>> students.count()
2
```

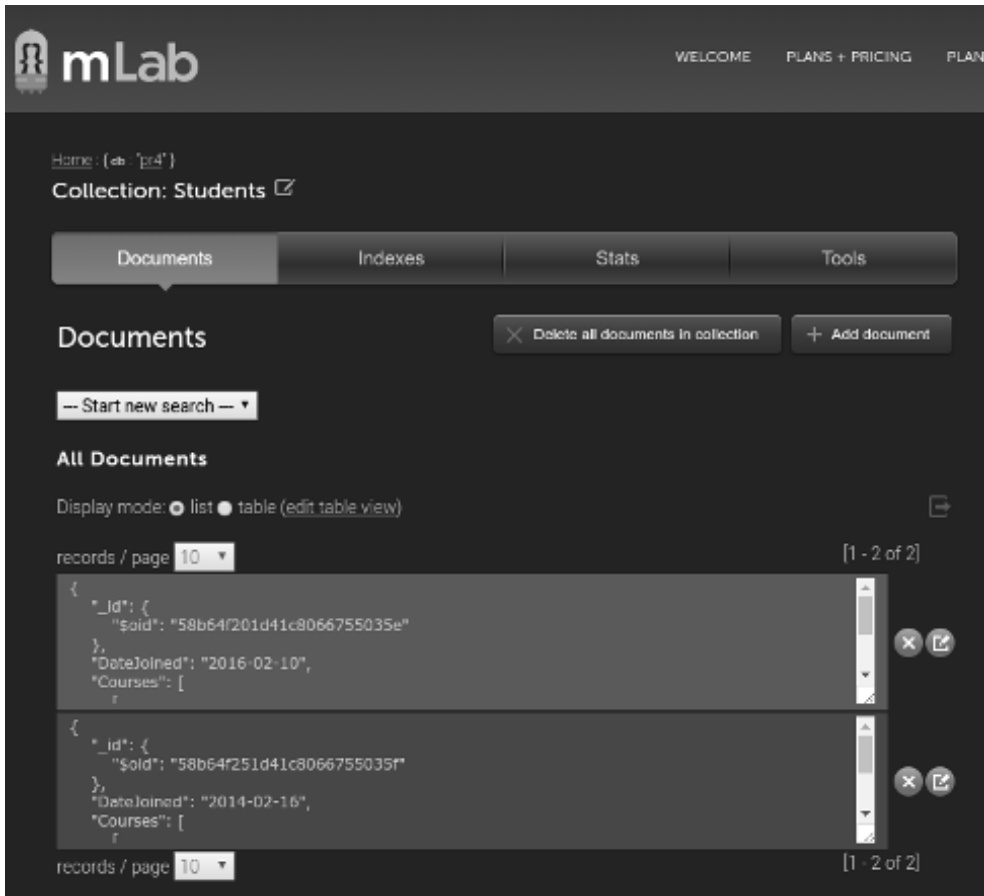


Figure 12.10 Documents in a collection, from MLab, a MongoDB cloud provider.

After each insertion, the `insert()` method returns the **ObjectId**, which is a key for each document. This ID is a 12-byte number that contains the date when the document was entered and an unique number for the collection. You can use this ID to retrieve a record, using the `find_one()` method:

```
>>> from bson.objectid import ObjectId
>>> search_id = {'_id': ObjectId('58b64f251d41c8066755035f')}
>>> my_student = students.find_one(search_id)
>>> my_student['LastName']
```


'Hunt'

You can get the insertion date with the `generation_time` property:

```
>>> my_student['_id'].generation_time
datetime.datetime(2017, 3, 1, 4, 33, 41, tzinfo=<bson.tz_util.<=
.FixedOffset object at 0x7f3eb8c3fd68>)
```

With `find()` you get a generator object that can be iterated over:

```
>>> for student in students.find():
...     print(student['Name'], student['LastName'])
...
Harry Wilkinson
Jonathan Hunt
```

If you want the whole list, use the built-in method `list()`:

```
>>> list(students.find())
[{'_id': ObjectId('58b64f201d41c8066755035e'), 'DateJoined': '<=
'2016-02-10', 'Courses': [['Python 101', 7, '2016/1'], ['Math<=
ematics for CS', 8, '2016/1']], 'OutstandingBalance': False, '<=
'LastName': 'Wilkinson', 'Name': 'Harry'}, {'_id': ObjectId('<=
58b64f251d41c8066755035f'), 'DateJoined': '2014-02-16', 'Cour<=
ses': [['Python 101', 6, '2016/1'], ['Mathematics for CS', 9,<=
'2015/2']], 'OutstandingBalance': False, 'LastName': 'Hunt',<=
'Name': 'Jonathan'}]
```

12.9 ADDITIONAL RESOURCES

- Database interfaces in Python.
<https://wiki.python.org/moin/DatabaseInterfaces>
- MySQL queries examples.
<http://www.pantz.org/software/mysql/mysqlcommands.html>
- Richard Hipp. SQLite lecture.
<https://youtu.be/gpxnbly9bz4>
- SQLite FAQ.
<https://sqlite.org/faq.html>
- The Unofficial MySQL 8.0 Optimizer Guide.
<http://www.unofficialmysqlguide.com>
- Why schemaless?
<https://www.mongodb.com/blog/post/why-schemaless>

- Installing PyMongo: The Python MongoDB Connector.
<http://codehandbook.org/pymongo-tutorial-crud-operation-mongodb>
- NOSQL data modeling techniques.
<https://goo.gl/iZcF0y>
- Database normalization basics. Normalizing your database.
<https://goo.gl/x7tbX4>
- Software:
 - MySQL homepage.
<http://www.mysql.com>
 - Squirrel SQL Client - JDBC SQL GUI Client
<http://www.squirrelsql.org/>
 - SQLite homepage
<http://www.sqlite.org/>
 - SQLite Administrator
<http://sqliteadmin.orbmu2k.de/>
 - PostgreSQL home page
<http://www.postgresql.org>
- Alternative Solutions:
 - Choosing a non-relational database; why we migrated from MySQL to MongoDB.
<https://goo.gl/8f7KeB>
 - The CouchDB Project
<http://couchdb.apache.org>
 - HyperTable: Performance and scalability.
<http://www.hypertable.org>
 - Apache Libcloud: An unified interface to the cloud in Python.
<http://libcloud.org>

12.10 SELF-EVALUATION

1. What is a database?
2. Give some examples of databases.
3. What is a relational database?
4. Define the following terms: entity, attributes, and relationships.
5. What is non-relational database?
6. What is a query?
7. Translate this query into English:
`SELECT LastName,Grade FROM Student,Grades WHERE Grades.Grade>3;`
8. What is the difference between MySQL and SQLite?
9. When is it appropriate to use SQLite?
10. What are the limitations of SQLite with regard to MySQL?
11. Name the advantages and disadvantages of NoSQL over SQL databases.

Regular Expressions

CONTENTS

13.1	Introduction to Regular Expressions (REGEX)	285
13.1.1	REGEX Syntax	286
13.2	The re Module	287
	re.search	288
	re.findall	288
	re.finditer	289
	re.match	289
13.2.1	Compiling a Pattern	290
	Groups	291
13.2.2	REGEX Examples	292
13.2.3	Pattern Replace	294
	re.sub	294
	re.subn	294
13.3	REGEX in Bioinformatics	294
13.3.1	Cleaning Up a Sequence	296
13.4	Additional Resources	297
13.5	Self-Evaluation	298

13.1 INTRODUCTION TO REGULAR EXPRESSIONS (REGEX)

A common feature of every scripting language is support of regular expressions (REGEX in programming jargon). What are regular expressions? They are expressions that summarize a text pattern. A known case of regular expression is the abbreviations used in most operating systems, like using “`ls *.py`” (or “`dir *.py`”) to list all files ended in “.py.” These are known as wildchars.

When doing text processing it is often necessary to give special treatment to strings containing a specific condition. For example, you may want to extract everything that is between `<pre>` and `</pre>` in an HTML file, or remove from a file any character that is not A, T, C, or G.

Biological applications of this feature are straightforward. Regular expressions can be used to locate domains in proteins, sequence patterns in DNA like CpG

islands, repeats, restriction enzyme, nuclease recognition sites, and so on. There are even biological databases devoted to protein domains, like PROSITE.¹

Nevertheless, your programming needs may not include the use of regular expressions. In this case, you can skip this chapter and read it when you need it. The rest of this book can be read without knowledge of regular expressions.

Each language has its own REGEX syntax. In Python, this syntax is close to the one used in Perl. So if you know Perl, learning Python REGEX is easy. If you have never heard of REGEX before, don't worry, basic REGEX syntax is not so hard to learn. Some REGEX can turn into obscure and complex expressions in specific cases. Due to this potential complexity, there are even whole books on this subject.²

13.1.1 REGEX Syntax

In general, the letters and characters match with themselves. “Python” is going to match with “Python” (but not with “python”). The exceptions to this rule are meta-characters, which are characters that have a special meaning in the context of the REGEX:

. ^ \$ * + ? { [] \ | ()

Let's see the meaning of the most commonly used special characters:

. (**dot**): Matches any character, except the new line: “ATT.T” will match “ATTCT”, “ATTFT” but not “ATTTCT”.

^ (**carat**): Matches the beginning of the chain: “^AUG” will match “AUGAGC” but not “AAUGC”. Using inside a group means “opposite”.

\$ (**dollar**): Matches the end of the chain or just before a new line at the end of the chain: “UAA\$” will match “AGCUAA” but not “ACUAAG”.

***** (**star**): Matches 0 or more repetitions of the preceding token: “AT*” will match “AAT”, “A”, but not “TT”.

+ (**plus**): The resulting REGEX will match 1 or more repetitions of the preceding REGEX: “AT+” will match “ATT”, but not “A”.

? (**question mark**): The resulting REGEX matches 0 or 1 repetitions of the preceding RE. “AT?” will match either “A” or “AT”.

(...): Matches whatever regular expression is inside the parentheses, and indicates the start and end of a group. To match the literals “(” or “)”, use “\ (” or “\)”, or enclose them inside a character class: “[()]”.

(?:...): A non-grouping version of regular parentheses. The substring matched by the group cannot be retrieved after performing a match.

{n}: Exactly n copies of the previous REGEX will match: “(ATTG){3}” will match “ATTGATTGATTG” but not “ATTGATTG”.

¹<http://prosite.expasy.org/>

²Please see **Additional Resources** for book recommendations.

TABLE 13.1 REGEX Special Sequences

Name	Description
<code>\number</code>	The contents of the group of the same number, starting from 1
<code>\A</code>	Only at the start of the string
<code>\b</code>	The empty string, only at the beginning or end of a word
<code>\B</code>	The empty string, only when it is not at the beginning or end of a word
<code>\d</code>	Any decimal digit (as <code>[0-9]</code>)
<code>\D</code>	Any non-digit (as <code>[^0-9]</code>)
<code>\s</code>	Any whitespace character (as <code>[\t\n\r\f\v]</code>)
<code>\S</code>	Any non-whitespace character (as <code>[^\t\n\r\f\v]</code>)
<code>\w</code>	Any alphanumeric character (as <code>[a-zA-Z0-9_]</code>)
<code>\W</code>	Any non-alphanumeric character (as <code>[^a-zA-Z0-9_]</code>)
<code>\Z</code>	Only the end of the string

{m,n}: The resulting REGEX will match from *m* to *n* repetitions of the preceding REGEX: “(AT){3,5}” will match “ATATTATATAT” but not “ATATTATAT”. Without *m*, it will match from 0 repetitions. Without *n*, it will match all repetitions.

[] (square brackets): Indicates a set of characters. “[A-Z]” will match any uppercase letter and “[a-z0-9]” will match any lowercase letter or digit. Meta characters are not active inside REGEX sets. “[AT*]” will match “A”, “T” or “*”. The `^` inside a set will match the complement of a set. “[^R]” will match any character but “R”.

"\" (backslash): Used to escape reserved characters (to match characters like “?”, “*”). Since Python also uses backslash as the escape character, you should pass a raw string to express the pattern.

| (vertical bar): As in logic, it reads as “or”. Any number of REGEX can be separated by “|”. “A|T” will match “A”, “T” or “AT”.

There are also special sequences with “\” and a character. They are listed in [Table 13.1](#).

13.2 THE RE MODULE

The **re** module provides methods like `compile`, `search`, `findall`, `match`, and other. These functions are used to process a text using a pattern built with the REGEX syntax.

A basic search works like this:

```
>>> import re
>>> mo = re.search('hello', 'Hello world, hello Python!')
```

re.search

The **search** from the **re** method requires a pattern as a first argument and as a second argument, a string where the pattern will be searched. In this case the pattern can be translated as “H or h, followed by ello.” When a match is found, this function returns a **match object** (called **mo** in this case) with information about the first match. If there is no match, it returns **None**. A **match object** can be queried with the methods shown here:

```
>>> mo.group()
'hello'
>>> mo.span()
(13, 18)
```

group() returns the string matched by the REGEX, while **span()** returns a tuple containing the (start, end) positions of the match (that is the (0, 5) returned by **mo.span()**).

This result is very similar to what the index method returns:

```
>>> 'Hello world, hello Python!'.index('hello')
13
```

The difference lies in the chance of using REGEX instead of plain strings. For example, we would like to match “Hello” and “hello”:

```
>>> import re
>>> mo = re.search('[Hh]ello', 'Hello world, hello Python!')
```

The first match now is,

```
>>> mo.group()
'Hello'
```

re.findall

To find all the matches, and not just the first one, use **findall**:

```
>>> re.findall("[Hh]ello", "Hello world, hello Python,!")
['Hello', 'hello']
```

Note that **findall** returns a list with the actual matches instead of match objects.

re.finditer

If we want to have a match object for each match, there is the **finditer** method. As an additional bonus, it doesn't return a list, but an iterator. This means that each time **finditer** is invoked it returns the next element without having to calculate them all at once. As with any iterator, this optimizes memory usage:

```
>>> re.finditer("[Hh]ello", "Hello world, hello Python,!")
<callable-iterator object at 0xb6f43d8c>
```

Walking on the results:

```
>>> mos = re.finditer("[Hh]ello", "Hello world, hello Python,!")
>>> for x in mos:
    print(x.group())
    print(x.span())

Hello
(0, 5)
hello
(13, 18)
```

re.match

This **match** method works like **search** but it looks only at the start of a string. When the pattern is not found, it returns **None**:

```
>>> mo = re.match("hello", "Hello world, hello Python!")
>>> print mo
None
```

As **search**, when the pattern is found, it returns a match object:

```
>>> mo = re.match("Hello", "Hello world, hello Python!")
>>> mo
<_sre.SRE_Match object at 0xb7b5eb80>
```

This match object can be queried as before:

```
>>> mo.group()
'Hello'
>>> mo.span()
(0, 5)
```


13.2.1 Compiling a Pattern

A pattern can be compiled (converted to an internal representation) to speed up the search. This step is not mandatory but recommended for large amounts of text. Let's see **findall** with a regular pattern and then with a “compiled” pattern (**rgx**):

```
>>> re.findall("[Hh]ello","Hello world, hello Python,!")
['Hello', 'hello']
>>> rgx = re.compile("[Hh]ello")
>>> rgx.findall("Hello world, hello Python,!")
['Hello', 'hello']
```

Compiled patterns have all methods available in the **re** module:

```
>>> rgx = re.compile("[Hh]ello")
>>> rgx.search("Hello world, hello Python,!")
<_sre.SRE_Match object at 0xb6f494f0>
>>> rgx.match("Hello world, hello Python,!")
<_sre.SRE_Match object at 0xb6f493d8>
>>> rgx.findall("Hello world, hello Python,!")
['Hello', 'hello']
```

Program 13.1 shows how to compile a pattern in the context of a search:

Listing 13.1: findTAT.py: Find the first “TAT” repeat

```
1 import re
2 seq = "ATATAAGATGCGCGCGCTTATGCGCGCA"
3 rgx = re.compile("TAT")
4 i = 1
5 for mo in rgx.finditer(seq):
6     print('Occurrence {0}: {1}'.format(i, mo.group()))
7     print('Position: From {0} to {1}'.format(mo.start(),
8                                             mo.end()))
9     i += 1
```

Code explanation: In line 3 the pattern (TAT) is compiled. The compiled object returned in line 3 (**rgx**) has the methods found in the **re** module, like **finditer**. This operation returns a “match” type object (**mo**). From this object, in lines 6 and 7, the **group** and **span** methods are invoked. Note that **mo.start()** and **mo.end()** are equivalent to **mo.span()[0]** and **mo.span()[1]**.

This is the result of running the program:

```
Occurrence 1: TAT
Position: From 1 to 4
Occurrence 2: TAT
Position: From 18 to 21
```

Groups

Sometimes you need to match more than one pattern; this can be done by grouping. Groups are marked by a set of parentheses (“()”). Groups can be “capturing” (“named” or “unnamed”) and “non-capturing.” The difference between them will be clear later.

A “capturing” group is used when you need to retrieve the contents of a group. Groups are captured with **groups**. Don’t confuse **group** with **groups**. As seen on page 288, **group** returns the string matched by the REGEX.

```
>>> import re
>>> seq = "ATATAAGATGCGCGCGCTTATGCGCGCA"
>>> rgx = re.compile("(GC){3,}")
>>> result = rgx.search(seq)
>>> result.group()
'GCGCGCGC'
```

This case is just like code snippet shown in page 288. Instead, **groups** return a tuple with all the subgroups of the match. In this case, since **search** returns one match and there is one group in the pattern, the result is a tuple with one group:

```
>>> result.groups()
('GC',)
```

There is a “CG” group, like in the pattern. If you want the whole pattern returned by groups, you need to declare another group like in this example:

```
>>> rgx = re.compile("((GC){3,})")
>>> result = rgx.search(seq)
>>> result.groups()
('GCGCGCGC', 'GC')
```

Both groups present in the pattern are retrieved (counting from left to right). This is true because by default every group is “capturing.” If you don’t need the internal subgroup (the “CG” group), you can label as “non-capturing.” This is done by adding “?” at the beginning of the group:

```
>>> # Only the inner group is non-capturing
>>> rgx = re.compile("(?:GC){3,}")
>>> result = rgx.search(seq)
>>> result.groups()
('GCGCGCGC',)
```

findall also behaves differently if there is a group in the pattern. Without a group it returns a list of matching strings (as seen on page 288). If there is one group in the pattern, it returns a list with the group. If there is more than one group, it return a list of tuples:

```
>>> rgx = re.compile("TAT") # No group at all.
>>> rgx.findall(seq) # This returns a list of matching strings.
['TAT', 'TAT']
>>> rgx = re.compile("(GC){3,}") # One group. Return a list
>>> rgx.findall(seq) # with the group for each match.
['GC', 'GC']
>>> rgx = re.compile("((GC){3,})") # Two groups. Return a
>>> rgx.findall(seq) # list with tuples for each match.
[('GCGCGCGC', 'GC'), ('GCGCGC', 'GC')]
>>> rgx = re.compile("(?:GC){3,}") # Using a non-capturing
>>> rgx.findall(seq) # group to get only the matches.
['GCGCGCGC', 'GCGCGC']
```

Groups can be labeled to refer to them later. To give a name to a group, use: **?P<name>**. [Listing 13.2](#) shows how to use this feature:

Listing 13.2: subgroups.py: Find multiple sub-patterns

```
1 import re
2 rgx = re.compile("(?P<TBX>TATA..)*(?P<CGislands>(?:GC){3,})")
3 seq = "ATATAAGATGCGCGCGCTTATGCGCGCA"
4 result = rgx.search(seq)
5 print(result.group('CGislands'))
6 print(result.group('TBX'))
```

This program returns:

```
GCGCGC
TATAGA
```

13.2.2 REGEX Examples

As a REGEX example, [Listing 13.3](#) shows how many lines in a given file have a pattern entered from the command line.³ The program is executed like this:

`program_name.py file_name pattern` Where `file_name` is the name of the file where `pattern` is searched.

Listing 13.3: regexsys1.py: Count lines with a user-supplied pattern on it

```
1 import re, sys
2 myregex = re.compile(sys.argv[2])
```

³There are more efficient ways to accomplish this, like using the Unix **grep** command, but it is shown here for a didactic purpose.

```

3 counter = 0
4 with open(sys.argv[1]) as fh:
5     for line in fh:
6         if myregex.search(line):
7             counter += 1
8 print(counter)

```

Code explained: The `re` module is imported and the expression to search is “compiled” (line 1 and 2). This “compilation” is optional but recommended. It accelerates the search by compiling the REGEX into an internal structure that is later used by the interpreter. `sys.argv` is a list of strings. Each string is an argument taken from the command line. If the command line is `program.py word myfile.txt`, the content of `sys.argv` is `['program.py', 'word', 'myfile.txt']` (a list with 3 strings). In line 4 the program opens the file entered as the first argument. In line 5 it parses the open file and in 5 and 6 it does the regular expression search “Python” within each line. If the expression is found, the counter variable (`counter`) is incremented by one (line 7).

This script doesn’t count how many occurrences of your word are in the file. If a word is repeated more than once in the same line, it is counted as one. The following script counts all the occurrences of a given pattern:

Listing 13.4: `countinfile.py`: Count the occurrences of a pattern in a file

```

1 import re, sys
2 myregex = re.compile(sys.argv[2])
3 i = 0
4 with open(sys.argv[1]) as fh:
5     for line in fh:
6         i += len(myregex.findall(line))
7 print(i)

```

Tip: Testing a REGEX with Kodos

Kodos is a nice GUI utility (made in Python) that allows you to test and debug your regular expressions. It has a window where you enter your REGEX pattern and another window where you enter a string to test your REGEX pattern against. As a result you will have the matching group information (if applicable), the match of the REGEX pattern in relation to the text string by using colors, and several variations of using the REGEX pattern in a Python application.

The program is released under the GNU Public License (GPL) and it is available at <http://kodos.sourceforge.net>. Since it is made in Python, it runs in all major operating systems (Windows, macOS and Linux).

13.2.3 Pattern Replace

The `re` module can be used to replace patterns with the `sub` function:

`re.sub`

`sub(rpl,str[,count=0])`: Replace *rpl* with the portion of the string (*str*) that coincides with the REGEX to which it applies. The third parameter, which is optional, indicates how many replacements we want made. By default the value is zero and means that it replaces all of the occurrences. It is very similar to the string method called `replace`, just that instead of replacing one text for another, the replaced text is located by a REGEX.

Listing 13.5: `deletgc.py`: Delete GC repeats (more than 3 GC in a row)

```
1 import re
2 regex = re.compile("(?:GC){3,}")
3 seq="ATGATCGTACTGCGCGCTTCATGTGATGCGCGCGCGCAGACTATAAG"
4 print "Before:",seq
5 print "After:",regex.sub("",seq)
```

The product of this program is

```
Before: ATGATCGTACTGCGCGCTTCATGTGATGCGCGCGCGCAGACTATAAG
After:  ATGATCGTACTTTCATGTGATAGACTATAAG
```

`re.subn`

`subn(rpl,str[,count=0])`: This has the same function as `sub`, differing in that instead of returning the new string, it returns a tuple with two elements: the new string and the number of replacements made. This function is used when, in addition to replacing a pattern in a string, it's required to know how many replacements have been made.

With this we have a very general vision of the possibilities that Regular Expressions open for us. The idea was to give an introduction to the subject and tools to start making our own REGEX. Next, we will see an example use of what has been learned so far.

13.3 REGEX IN BIOINFORMATICS

As I mentioned at the beginning of the chapter, the REGEX can be used to search PROSITE style patterns.⁴ The patterns are sequences of characters that describe a

⁴If you are not familiar with protein patterns, please take a look at the PROSITE user manual, located at: <http://prosite.expasy.org/prosuser.html>.

group of sequences in a condensed form. For example, the following is the pattern for the active site of the enzyme isocitrate lyase:

K- [KR] -C-G-H- [LMQR]

This pattern is interpreted as: a K in the first position, a K or R in the second, then the sequence CGH, and finally, one of the following amino acids: L, M, Q or R. If we want to search for this pattern in this sequence, as a first measure one must convert the pattern from PROSITE to a Python REGEX. The conversion in this case is immediate:

"K [KR] CGH [LMQR] "

To change a PROSITE profile to REGEX basically consists of removing the hyphens (-), replacing the numbers between parentheses with numbers between braces, and replacing the "x" with a period. Let's see an example of the adenyl cyclase associated protein 2:

PROSITE version:

[LIVM] (2) -x-R-L- [DE] -x(4) -R-L-E

REGEX version:

" [LIVM] {2} .RL [DE] . {4}RLE"

Let's suppose that we want to find a pattern of this type in a sequence in FASTA format. Besides finding the pattern, we may need to retrieve it in a context, that is, 10 amino acids before and after the pattern. Here is a sample FASTA file:

```
>Q5R5X8|CAP2_PONPY CAP 2 - Pongo pygmaeus (Orangutan).
MANMQGLVERLERAVSRLESLSAESHRRPPGNCGEVNGVIGGVAPSVEAFDKLMDSMVAEF
LKNSRILAGDVETHAEMVHSAFQAQRAFLLMASQYQQPHENDVAALLKPISEKIQEIQTF
RERNRGSNMFNHL SAVSESIPALGWIAVSPKPGPYVKEMNDAATFYTNRVLKDYKHSDLR
HVDWVKSYLNIWSELQAYIKEHHTTGLTWSKTGPVASTVSAFVLSGPGLP P P P P P P P
PGPPPLENEGKKEESSPSRSALFAQLNQGEAITKGLRHVTDDQKTYKNPSLRAQGGQTR
SPTKSHTPSPTSPKSYPSQKHAPVLELEGKKWRVEYQEDRNDLVISETELKQVAYIFKCE
KSTLQIKGKVN S I I D N C K K L G L V F D N V V G I V E V I N S Q D I Q I Q V M G R V P T I S I N K T E G C H
IYLS DALDCEIVSAK S SEMNILIPQDGDYREFPIPEQFKTAWD GSKLITEPAEIMA
```

The program in [Listing 13.6](#) (`searchinfasta.py`) reads the FASTA file.

Listing 13.6: `searchinfasta.py`: Search a pattern in a FASTA file

```

1 import re
2 pattern = "[LIVM]{2}.RL[DE].{4}RLE"
3 with open('/home/sb/bioinfo/prot.fas') as fh:
4     fh.readline() # Discard the first line.
5     seq = ""
6     for line in fh:
7         seq += line.strip()
8     rgx = re.compile(pattern)
9     result = rgx.search(seq)
10    patternfound = result.group()
11    span = result.span()
12    leftpos = span[0]-10
13    if leftpos<0:
14        leftpos = 0
15    print(seq[leftpos:span[0]].lower() + patternfound +
16          seq[span[1]:span[1]+10].lower())

```

The result of this program is

```
lrsyrrdewaLLTRLDAQWERLElwmdrfatki
```

Code explanation: Up to line 7, the program reads the FASTA file and stores the protein sequence (`seq`). In line 8 the pattern defined in line 2 is compiled. The search is done at line 9. From line 10 onward the program works on displaying the result. As requested, the resulting pattern is shown in a context of 10 amino acids on each side.

13.3.1 Cleaning Up a Sequence

It's more than common to find a file with sequences in a non-standard format, such as the following sequence:

```

1  ATGACCATGA TTACGCCAAG CTCTAATACG ACTCACTATA GGGAAAGCTT GCATGCCTGC

61  AGGTCGACTC TAGAGGATCT ACTAGTCATA TGGATATCGG ATCCCCGGGT ACCGAGCTCG

121 AATTCACTGG CCGTCGTTTT

```

The following code reads a text file with the sequence in this format and returns only the sequence, without any strange (number or whitespace) character:

Listing 13.7: `cleanseq.py`: Cleans a DNA sequence

```

1 import re
2 regex = re.compile(' |\d|\n|\t')

```

```

3 seq = ''
4 for line in open('pMOSBlue.txt'):
5     seq += regex.sub('',line)
6 print seq

```

This program prints:

```

ATGACCATGATTACGCCAAGCTCTAATACGACTCACTATAGGGAAAGCTTGCATGCCTGCAGGTC<=
GACTCTAGAGGATCTACTAGTCATATGGATATCGGATCCCCGGGTACCGAGCTCGAATTCAGTGG<=
CCGTCGTTTT

```

Code explained: Line 2 defines the characters we are going to search for removal. In this case the characters are white spaces, numbers, carriage return, and tabs. In lines 4 and 5 the program parses all the lines of the file (`pMOSBlue.txt`) and removes the pattern each time it's found.

13.4 ADDITIONAL RESOURCES

- Jeffrey EF Friedl, *Mastering Regular Expressions*, Third Edition, 2006, O'Reilly Media.
<http://shop.oreilly.com/product/9780596528126.do>
- Tony Stubblebine, *Regular Expression Pocket Reference*, Second Edition, 2007. O'Reilly Media.
<http://www.oreilly.com/catalog/9780596514273/>
- The premier web site about regular expressions.
<http://www.regular-expressions.info>.
- Regular expressions in Java. Test your regular expressions online.
<http://www.javaregex.com/test.html>
- Python regular expression builder. Pyreb is a wxPython GUI to the re python module; it will speed up the development of Python regular expression.
<http://savannah.nongnu.org/projects/pyreb>
- Python's hidden regular expression gems, by Armin Ronacher.
<http://lucumr.pocoo.org/2015/11/18/pythons-hidden-re-gems/>
- Harry J Mangalam. tacg: a grep for DNA. *BMC Bioinformatics* 2002, 3:8.
<http://www.biomedcentral.com/1471-2105/3/8>

13.5 SELF-EVALUATION

1. What is a REGEX?
2. What is the difference between a “capturing” and a “non-capturing” group?
3. How can text pattern search be applied to biology?
4. Line 13 of [Listing 13.6](#) (page 295) is checked if the value `leftpos` is less than 0. Why?
5. In List 13.7, the pattern used was “`|\d|\n|\t`”. What other alternative could have been employed?
6. Make a program that retrieves all phone number found in a file. The numbers must be in the format `nnn-nnn-nnnn`, where `n` is a number.
7. Make a program to retrieve every e-mail ending in `.com` present in every file in a given directory.
8. Make a program to sort if a sequence is made of DNA or amino acids. Hint: DNA sequences can only have these characters: “ATCGN.”
9. Write a REGEX pattern to detect a HindII restriction site. This enzyme recognizes the DNA sequence `GTyrAC` (where “Y” means “C” or “T” and “R” means “G” or “A”).
10. What is the meaning of the following REGEX? Write a string that matches it.

```
"[0-9]{1,4}/[0-9]{1,2}/[0-9]{1,2}"
```

Graphics in Python

CONTENTS

14.1	Introduction to Bokeh	299
14.2	Installing Bokeh	299
14.3	Using Bokeh	301
14.3.1	A Simple X-Y Plot	303
14.3.2	Two Data Series Plot	304
14.3.3	A Scatter Plot	306
14.3.4	A Heatmap	308
14.3.5	A Chord Diagram	309

14.1 INTRODUCTION TO BOKEH

Bokeh is an interactive visualization library that runs in a web browser. It mimics the style of **D3.js**, a popular library for web graphics made in JavaScript. It also has taken ideas from **MATLAB**¹. Users from both system will find the similarities and learn fast. If you have never used a graphical library before, some concepts will look strange or out of place, so you will have to follow this chapter closely, without skipping any section. Once you know how to use this library, it will help you make interactive plots, dashboards, and data applications.

How does it work? It has two main components; first a JavaScript library called **BokehJS** that runs in the browser. This library is used to render the graphics and also to provide interaction such as zoom, pan, and save. As input, this library uses JSON objects with all the information needed to draw the plot. The other component is a Python library that produces these JSON objects.

14.2 INSTALLING BOKEH

Bokeh can be installed with **pip** or with **conda** if you use the Anaconda distribution. The former method does not install the examples, so the latter (conda) is preferred. If you install Bokeh with pip and want the examples, download them from the GitHub page.²

¹MATLAB[®] is a registered trademark of The MathWorks, Inc. For product information please contact: The MathWorks, Inc. 3 Apple Hill Drive Natick, MA, 01760-2098 USA. Tel: 508-647-7000. Fax: 508-647-7001. E-mail: info@mathworks.com. Web: www.mathworks.com.

²<https://github.com/bokeh/bokeh>

Installing Bokeh with pip:

```
(py4bio) $ pip install bokeh
Collecting bokeh
  Downloading bokeh-0.12.4.tar.gz (5.6MB)
    100% |*****| 5.6MB 213kB/s
Requirement already satisfied: six>=1.5.2 in ./py4bio/lib/python2.7<=
/site-packages (from bokeh)
Collecting PyYAML>=3.10 (from bokeh)
(...)
Successfully built bokeh PyYAML tornado MarkupSafe
Installing collected packages: PyYAML, python-dateutil, MarkupSafe,<=
Jinja2, singledispatch, certifi, backports-abc, tornado, futures, <=
bokeh
Successfully installed Jinja2-2.9.5 MarkupSafe-1.0 PyYAML-3.12 back<=
ports-abc-0.5 bokeh-0.12.4 certifi-2017.1.23 futures-3.0.5 python-d<=
ateutil-2.6.0 singledispatch-3.4.0.3 tornado-4.4.2
```

With Conda, first create an environment:

```
$ conda create --name p4b
Fetching package metadata .....
.Solving package specifications: .
Package plan for installation in environment /sb/anaconda3/envs/p4b:
```

The following empty environments will be CREATED:

```
/sb/anaconda3/envs/p4b
```

```
Proceed ([y]/n)?
```

```
#
# To activate this environment, use:
# > source activate p4b
#
# To deactivate this environment, use:
# > source deactivate p4b
#
```

Activate this environment and install Bokeh:

```
$ source activate p4b
(p4b) $ conda install bokeh
Fetching package metadata .....
Solving package specifications: .....
```

Package plan for installation in environment /sb/anaconda3/envs/p4b:

The following packages will be downloaded:

(...)

The following NEW packages will be INSTALLED:

bokeh: 0.12.4-py36_0

jinja2: 2.9.5-py36_0

(...)

Proceed ([y]/n)?

Fetching packages ...

bokeh-0.12.4-p 100% |#####| Time: 0:00:00 10.06 MB/s

Extracting packages ...

[COMPLETE] |#####| 100%

Linking packages ...

[COMPLETE] |#####| 100%

(bokeh) sbassi@sbassi-MS-7641:~\$

14.3 USING BOKEH

The main class is called **Figure** and it includes methods for adding elements to a plot. It also composes default axes, grids, and tools in a sensible way. To instantiate this class we use the **figure** method that accepts basic parameters such as the title of the graphic and the labels of the axis. The generated figure object has methods to add elements to the figure. A typical method to add elements is **circle**. Let's see it in action:

Listing 14.1: basiccircle.py: A circle made with Bokeh

```
1 from bokeh.plotting import figure, output_file, show
2
3 p = figure(width=400, height=400)
4 p.circle(2, 3, radius=.5, alpha=0.5, color='red')
5 output_file("out.html")
6 show(p)
```

Code explanation: In line 1 we import all required elements: **figure**, **output_file**, and **show**. In line 3 we call the **figure** method to instantiate the **Figure** class. The parameters we use are *width* and *height*, which are the width and height of the image. This object is called **p**. In line 4, we call the **circle** method in the **p Figure** instance. In this case the parameters are *X* axis value, *Y* axis value,

the radius of the circle (**radius**), a transparency value (**alpha**), and a color (**color**). The general form for this method is `circle(x, y, **kwargs)` (where ****kwargs** are an undetermined number of keyword arguments). The method **output_file** in line 5 sets a name of an HTML file that the browser will load with the graphic. The command **show()** in the last line (6) calls the web browser with the **output_file**. This will fire a web browser as can be seen in [figure 14.1](#). The resulting graphic is interactive, you can pan and zoom using the controls on the right. If you just want a png image, use the save icon (the one with the floppy disk logo).

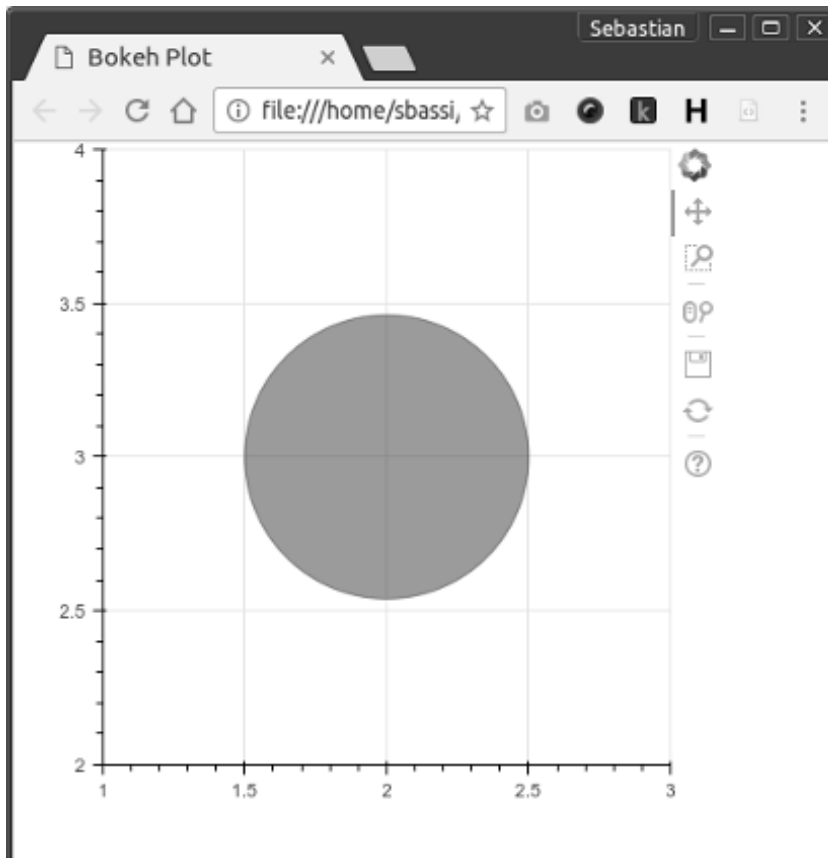


Figure 14.1 A circle with Bokeh.

To make multiple circles, just enter their coordinates in a list for each axis. For example the following code displays four circles:

Listing 14.2: `fourcircles.py`: 4 circles made with Bokeh

```
1 from bokeh.plotting import figure, output_file, show
2
3 p = figure(width=500, height=500)
```

```

4 x = [1, 1, 2, 2]
5 y = [1, 2, 1, 2]
6 p.circle(x, y, radius=.35, alpha=0.5, color='red')
7 output_file('out.html')
8 show(p)

```

Code explanation: This code is very similar to [Listing 14.1](#). The main difference is that in line 6, instead of passing one *X* and one *Y* value, we pass a list of values for each axis. Other changes are straightforward. The result can be seen in [Figure 14.2](#)

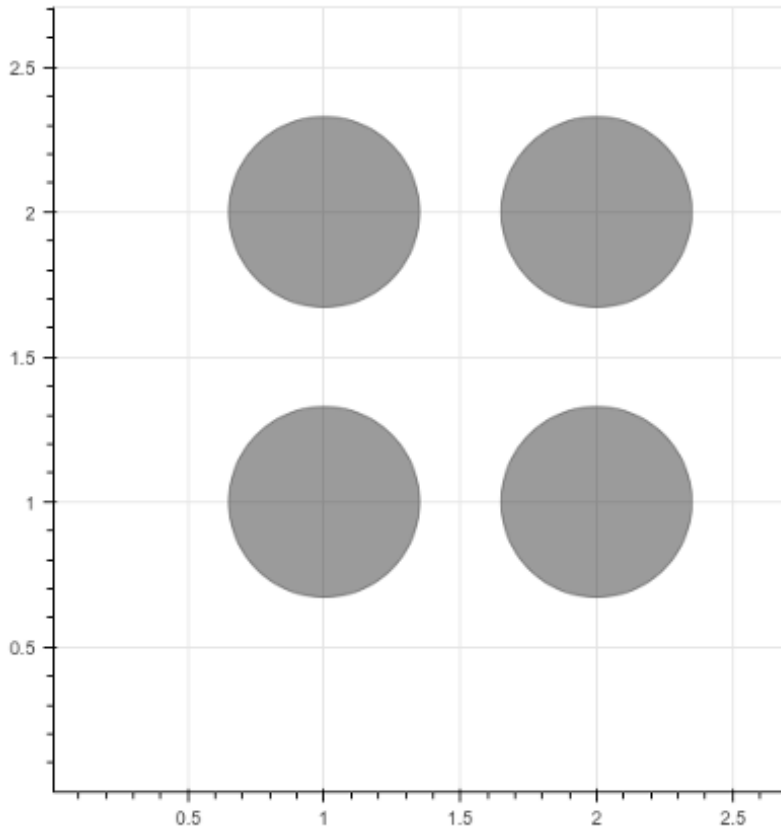


Figure 14.2 Four circles with Bokeh.

14.3.1 A Simple X-Y Plot

With this we are ready to plot two column of data. Even if you are not considering doing this particular type of chart, the elements that appear here are also part of other type of charts, so it is worth following this chapter in the proposed order and not jumping directly into a particular type of graphic. Consider the following table:

```

=====
Time | Mean wt increase
=====
1    | 0.7
2    | 1.4
3    | 2.1
4    | 3
5    | 3.85
6    | 4.55
7    | 5.8
8    | 6.45
=====

```

To make an x-y plot we need the **figure** class. In this case we use new keyword arguments to set a title and a label for each axis. As we did in [Listing 14.2](#) we use **circle** but with a new argument (legend). The **output_file** class is used to set the name of the output HTML file.

Listing 14.3: plot1.py: A minimal plot

```

1 from bokeh.plotting import figure, output_file, show
2
3 x = [1, 2, 3, 4, 5, 6, 7, 8]
4 y = [.7, 1.4, 2.1, 3, 3.85, 4.55, 5.8, 6.45]
5
6 p = figure(title='Mean wt increased vs. time',
7            x_axis_label='Time in days',
8            y_axis_label='% Mean WT increased')
9 p.circle(x, y, legend='Subject 1', size=10)
10 output_file('test.html')
11 show(p)

```

The result of the execution of [Listing 14.3](#) can be seen in [Figure 14.3](#).

14.3.2 Two Data Series Plot

[Listing \(14.4\)](#) adds a new data series and some small customizations. The new data series is a list called **z**. On line 11 there is a new call to the **circle** method, to add the new data series. Note that in this case we add the parameter **line_color** with the value of **'red'** and **fill_color** as **'white'**. Bokeh supports the list of named colors stated at https://www.w3schools.com/colors/colors_names.asp. If you need to include a color that is not in this list, you must enter the RGB code in its place. The last new object is **legend** with its property **location**. This changes the position of the legend box to a place that won't overlap part of the graphic.

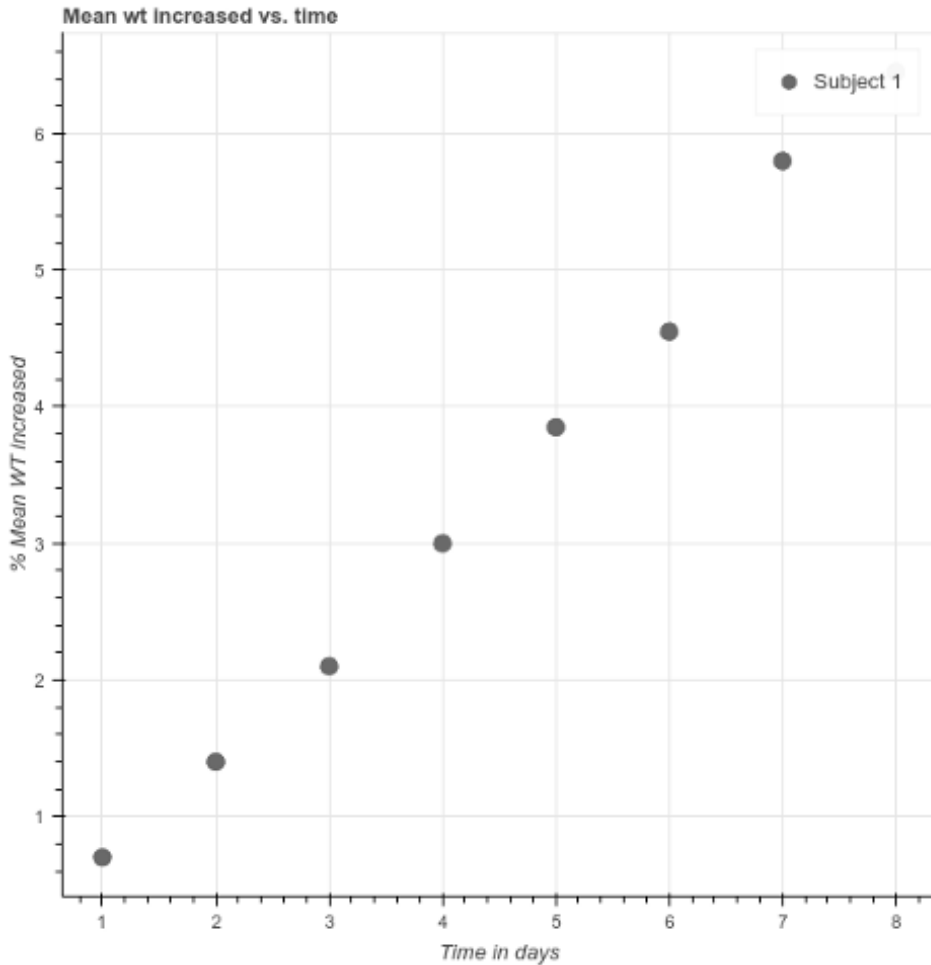


Figure 14.3 A simple plot with Bokeh.

Listing 14.4: plot2.py: Two data series plot

```

1 from bokeh.plotting import figure, output_file, show
2
3 x = [1, 2, 3, 4, 5, 6, 7, 8]
4 y = [.7, 1.4, 2.1, 3, 3.85, 4.55, 5.8, 6.45]
5 z = [.5, 1.1, 1.9, 2.5, 3.1, 3.9, 4.85, 5.2]
6
7 p = figure(title='Mean wt increased vs. time',
8           x_axis_label='Time in days',
9           y_axis_label='% Mean WT increased')
10 p.circle(x, y, legend='Subject 1', size=10)
11 p.circle(x, z, legend='Subject 2', size=10, line_color='red',

```



```
12         fill_color='white')
13 p.legend.location = 'top_left'
14 output_file('test.html')
15 show(p)
```

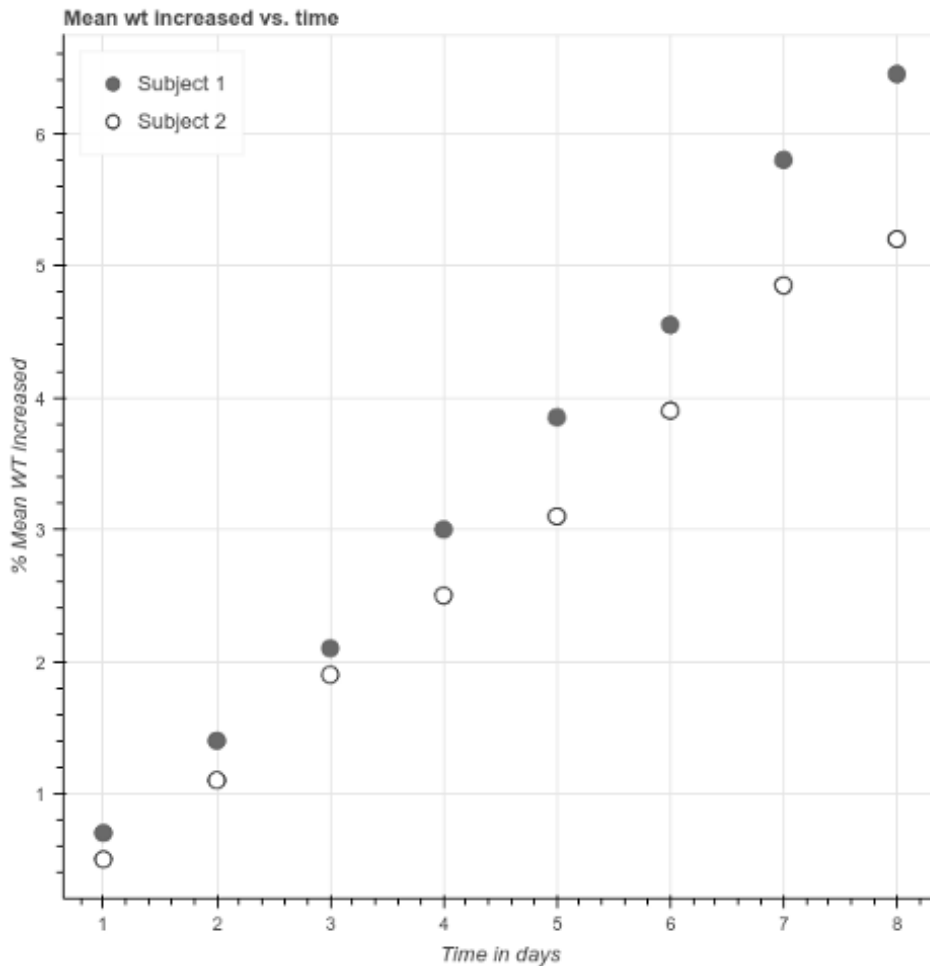


Figure 14.4 A two data series plot with Bokeh.

14.3.3 A Scatter Plot

The scatter plot displays values for two or more variables for a set of data. The dots can be color-coded to display another variable. The following CSV file was made from a work where the intestinal content in various fish species was studied. A principal component analysis (PCA) was made and the data of the first two principals

component, the type of feed and the fish species, are in `fishdata.csv`. For more information on research using this analysis, see the article DOI 10.7717/peerj.550.³

```
index,PC1,PC2,feeds,species
0,0.5,0.8,Crustacea,Epinephelus
1,0.3,0.9,Mosquito larvae,Sebastiscus
2,0.95,0.83,Mosquito larvae,Sebastiscus
3,0.92,1.98,Mosquito larvae,Sebastiscus
4,2.01,1.4,Crustacea,Sebastiscus
5,2.15,1.25,Crustacea, Sebastiscus
6,2.19,1.01,Aquaculture feed,Sebastiscus
7,2.35,0.21,Mosquito larvae,Sebastes
8,2.48,0.87,Crustacea,Sebastes
9,2.53,0.98,Crustacea,Sebastes
10,2.85,1.5,Polychaeta,Sebastes
11,2.93,2.39,Aquaculture feed,Sebastes
12,3.05,3.05,Mosquito larvae,Acanthogobius
13,3.38,3.08,Mosquito larvae,Acanthogobius
14,4.05,2.09,Mosquito larvae,Acanthogobius
15,4.18,2.89,Crustacea,Acanthogobius
16,4.23,1.95,Crustacea,Acanthogobius
17,5.03,1.98,Polychaeta,Acanthogobius
18,5.32,2.05,Aquaculture feed,Acanthogobius
```

[Listing 14.5](#) reads `fishdata.csv` using the **DataFrame** class from the **Pandas** library. The `from_csv` method converts a CSV file into a **DataFrame** object. This object works as a matrix with each column a different data type. In this case we use the **Scatter** class, which is initialized with this **DataFrame** object (`df`). The `x` and `y` parameters are the column names (the column names of the CSV object that was converted into the `df` **DataFrame**). Note that in line 11 we change the `background_fill_alpha` attribute to insert transparency in the legend box.

Listing 14.5: `fishpc.py`: Scatter plot

```
1 from bokeh.charts import Scatter, output_file, show
2 from pandas import DataFrame
3
4 df = DataFrame.from_csv('fishdata.csv')
5
6 scatter = Scatter(df, x='PC1', y='PC2', color='feeds',
7                  marker='species', title=
```

³Asakura T, Sakata K, Yoshida S, Date Y, Kikuchi J. (2014) Noninvasive analysis of metabolic changes following nutrient input into diverse fish species, as investigated by metabolic and microbial profiling approaches. *PeerJ* 2:e550 <https://doi.org/10.7717/peerj.550>.

```

8      'Metabolic variations based on 1H NMR profiling of fishes',
9      xlabel='Principal Component 1: 35.8%',
10     ylabel='Principal Component 2: 15.1%')
11 scatter.legend.background_fill_alpha = 0.3
12 output_file('scatter.html')
13 show(scatter)

```

The resulting graphics can be seen in [Figure 14.5](#).

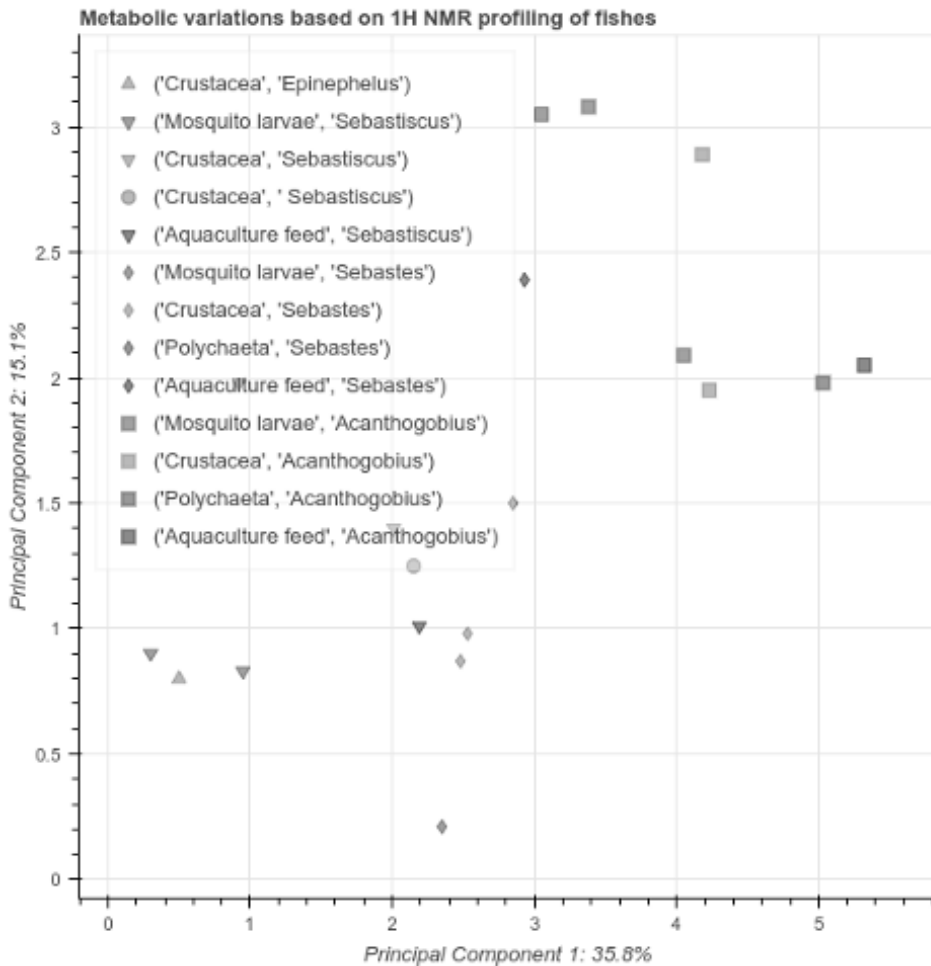


Figure 14.5 Scatter plot graphics.

14.3.4 A Heatmap

A heatmap is a graphic of a matrix where each value is represented as a color, usually the intensity of the color is proportional with the value assigned to its position. There are a lot of uses of heatmaps in biology. In this case we will work

with a DNA microarray experiment. This is used to measure gene expression values. From the following file, we will use the first three values (x, y and lux). The first two are the position in the matrix and the third one is the intensity of the light emitted by the target. The amount of light is relative to the abundance of nucleic acid sequences in the target.

This file is called `GSM188012.CEL`:

x	y	lux	avg	type
0	0	241.3	28.2	16
1	0	10834.8	1384.4	16
2	0	219.0	24.5	16
3	0	11074.5	1287.4	16
(...)				
709	711	416.3	34.1	16
710	711	9177.3	1056.3	16
711	711	437.8	35.4	16

The relevant fields in `GSM188012.CEL` are:

x: Position in the x axis. y: Position in the y axis. lux: Signal intensity.

Now we have all that we need to supply the **HeatMap** class:

Listing 14.6: `heatmap.py`: Plot a gene expression file

```

1 from bokeh.charts import HeatMap, bins, output_file, show
2 import pandas as pd
3
4 DATA_FILE = '../..samples/GSM188012.CEL'
5 dtype = {'x': int, 'y': int, 'lux': float}
6 dataset = pd.read_csv(DATA_FILE, sep='\t', dtype=dtype)
7 hm = HeatMap(dataset, x=bins('x'), y=bins('y'), values='lux',
8               title='Expression', stat='mean')
9 output_file("heatmap7.html", title="heatmap.py example")
10 show(hm)

```

14.3.5 A Chord Diagram

Chord diagrams are used to represent inter-relationships among data in a matrix. The data is arranged radially around a circle with the relationships between the points drawn as arcs connecting the data.

The following table (`fishbacteria.csv`) is based in the work “Fish gut microbiota analysis differentiates physiology and behavior of invasive Asian carp and indigenous American fish” (DOI: 10.1038/ismej.2013.181). They analyzed the gut of different fishes species and measured the number of bacterial species carried. The fields are:

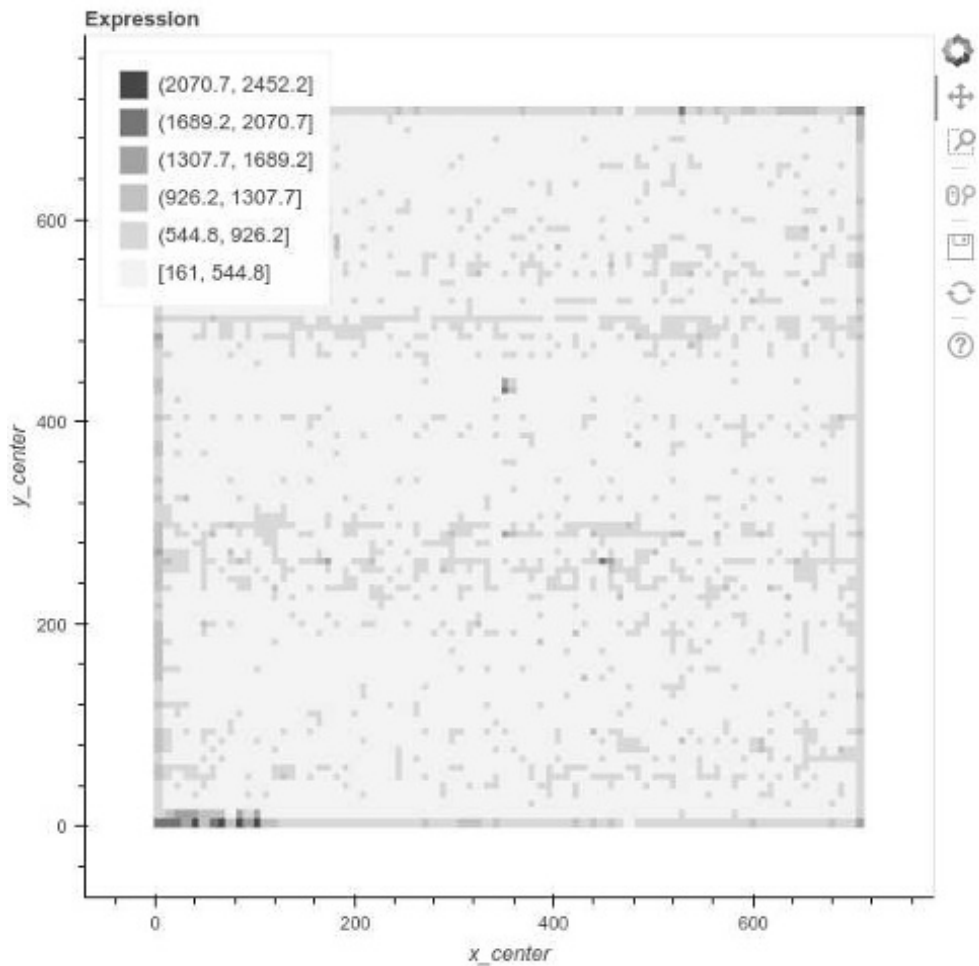


Figure 14.6 A heatmap out of a microarray experiment. This is typically used to study gene expression.

value: This is the number of microbial species found in a given environment, and is measured in operational taxonomic unit (OTUs). **name_x:** This is the name of the phylum (*Acidobacteria* for example). **name_y:** This is the name of an environment, for example *SVCP_H*, which is the hindgut of the Asian silver carp.

```
value,name_x,name_y
5,Acidobacteria,SVCP_H
3,Acidobacteria,SVCP_F
2,Acidobacteria,GZSD_F
2,Acidobacteria,GZSD_H
9,Crenarchaeota,GZSD_F
1,Euryarchaeota,GZSD_H
```

```
1,Euryarchaeota,GZSD_F
(...)
8,Unknown,SVCP_F
(...)
```

To make the Chord diagram, Bokeh provides the **Chord** class. It requires the data, it could be a dictionary or a **DataFrame** object as in this case (converted out of a CSV file). The other needed parameters are **source**, **target** and **value**. Once you pass the name of the variables in the data that corresponds to each parameter, the rest of the [Listing 14.7](#) is identical to previous code:

Listing 14.7: chord.py: A Chord diagram

```
1 from bokeh.charts import output_file, Chord
2 from bokeh.io import show
3 import pandas as pd
4
5 data = pd.read_csv('test3.csv')
6 chord_from_df = Chord(data, source='name_x', target='name_y',
7                       value='value')
8 output_file('chord.html')
9 show(chord_from_df)
```

This code will output the chord diagram that can be seen in [figure 14.7](#).

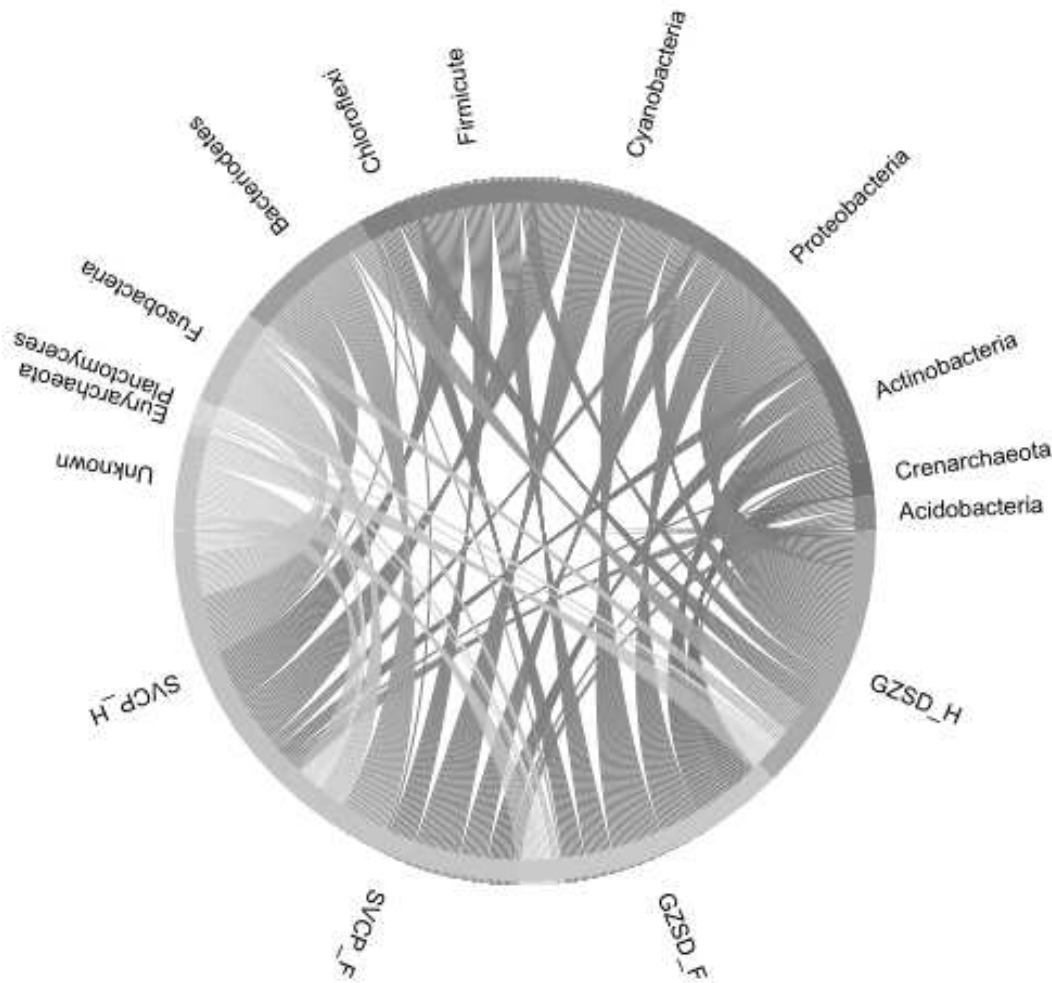


Figure 14.7 A chord diagram.

III

Python Recipes with Commented Source Code



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Sequence Manipulation in Batch

CONTENTS

15.1	Problem Description	315
15.2	Problem One: Create a FASTA File with Random Sequences	315
15.2.1	Commented Source Code	315
15.3	Problem Two: Filter Not Empty Sequences from a FASTA File	316
15.3.1	Commented Source Code	317
15.4	Problem Three: Modify Every Record of a FASTA File	319
15.4.1	Commented Source Code	320

15.1 PROBLEM DESCRIPTION

Sequence manipulation is a pretty common task in most laboratories. Some tasks can be done with the Python Standard Library, but this chapter will show that the Biopython package can help. In this case we will use the SeqRecord and SeqIO modules from the Biopython package.

15.2 PROBLEM ONE: CREATE A FASTA FILE WITH RANDOM SEQUENCES

Random sequences are used as input in some statistical tests. These sequences can also be used to test programs when you don't have real data in the required amount.

In [Listing 15.1](#) we use constants to set the length range and the number of sequences. In this case, the minimum size is 400 nucleotides, the maximum is 1500 nucleotides, and we will generate 500 sequences. This can be modified by changing the constants from line 7 to 9.

15.2.1 Commented Source Code

Listing 15.1: seqiornd.py: Generate random sequences

```
1 import random
2
```

```

3 from Bio.SeqRecord import SeqRecord
4 from Bio.Seq import Seq
5 from Bio import SeqIO
6
7 TOTAL_SEQUENCES = 500
8 MIN_SIZE = 400
9 MAX_SIZE = 1500
10
11 def new_rnd_seq(seq_len):
12     """
13     Generate a random DNA sequence with a sequence length
14     of "sl" (int).
15     return: A string with a DNA sequence.
16     """
17     s = ''
18     while len(s) < seq_len:
19         s += random.choice('ATCG')
20     return s
21
22 with open('randomseqs.txt','w') as new_fh:
23     for i in range(1, TOTAL_SEQUENCES + 1):
24         # Select a random number between MIN_SIZE and MAX_SIZE
25         rsl = random.randint(MIN_SIZE, MAX_SIZE)
26         # Generate the random sequence
27         rawseq = new_rnd_seq(rsl)
28         # Generate a correlative name
29         seqname = 'Sequence_number_{0}'.format(i)
30         rec = SeqRecord(Seq(rawseq), id=seqname, description='')
31         SeqIO.write([rec], new_fh, 'fasta')

```

Code explanation: Generation of the random sequence is done in the *new_rnd_seq* function (from lines 11 to 20). This function is called inside the *for loop* and it is stored as *rawseq*. In line 30 a *SeqRecord* object is created. This object is passed to *SeqIO.write* in line 31.

15.3 PROBLEM TWO: FILTER NOT EMPTY SEQUENCES FROM A FASTA FILE

Sometimes you need to get rid of malformed sequences from a FASTA file. Some programs choke when they receive an empty sequence as the input file. Formatdb, the program used to format BLAST databases, is known behave like this. The code in [Listing 15.2](#) assumes that you have a FASTA file like this:

```
>SSR86 [ssr] : Tomato-EXPEN 2000 map, chr 3
```

```

AGGCCAGCCCCCTTTTCCCTTAAGAACTCTTTGTGAGCTTCCCGCGGTGGCGGCCGCTCTAG
>SSR91 [ssr]
>SSR252 [ssr]
TGGGCAGAGGAGCTCGTANGCATACCGCGAATTGGGTACACTTACCTGGTACCCACCCGGG
TGGAATTCGATGGGCCCCGCGGCCGCTCTAGAAGTACTCTCTCTCT
>SSR257 [ssr]
TGAGAATGAGCACATCGATACGGCAATTGGTACACTTACCTGCGACCCACCCGGGTGGAAA
ATCGATGGGCCCCGCGGCC
>SSR92 [ssr] : Tomato-EXPEN 2000 map, chr 1

```

And it should produce a version of the file without the empty records:

```

>SSR86 [ssr] : Tomato-EXPEN 2000 map, chr 3
AGGCCAGCCCCCTTTTCCCTTAAGAACTCTTTGTGAGCTTCCCGCGGTGGCGGCCGCTCTAG
>SSR252 [ssr]
TGGGCAGAGGAGCTCGTANGCATACCGCGAATTGGGTACACTTACCTGGTACCCACCCGGG
TGGAATTCGATGGGCCCCGCGGCCGCTCTAGAAGTACTCTCTCTCT
>SSR257 [ssr]
TGAGAATGAGCACATCGATACGGCAATTGGTACACTTACCTGCGACCCACCCGGGTGGAAA
ATCGATGGGCCCCGCGGCC

```

15.3.1 Commented Source Code

Listing 15.2: seqio1.py: Filter a FASTA file

```

1 from Bio import SeqIO
2
3 INPUT_FILE = '../samples/fast22.fas'
4 OUTPUT_FILE = 'fast22_out.fas'
5
6 def retseq(seq_fh):
7     """
8     Parse a fasta file and store non empty records
9     into the fullseqs list.
10    :seq_fh: File handle of the input sequence
11    :return: A list with non empty sequences
12    """
13    fullseqs = []
14    for record in SeqIO.parse(seq_fh, 'fasta'):
15        if len(record.seq):
16            fullseqs.append(record)
17    return fullseqs
18

```

```

19 with open(INPUT_FILE) as in_fh:
20     with open(OUTPUT_FILE, 'w') as out_fh:
21         SeqIO.write(retseq(in_fh), out_fh, 'fasta')

```

Although this program does its job, it is not an example of efficient use of computer resources. The list **fullseqs** ends up with the information on every non-empty sequence in the file. For short sequence files this is not noticeable. In a real-world scenario it could bring a server to its knees.

The same program can be adapted for low memory usage. This is accomplished in [Listing 15.3](#) by the use of a *generator*. A *generator* is a special kind of function. Syntactically, a generator and a function are very alike; both have a header with the *def* keyword, a name, and parameters (if any). The most visible difference is that instead of having the word *return* as an exit point, generators have *yield*. The main difference between a function and a generator is that the generator keeps its internal state after being called. The next time the generator is called, it resumes its execution from the point where it was before. This property is used to yield several values, one at a time.

Listing 15.3: seqio2.py: Filter a FASTA file with a generator

```

1 from Bio import SeqIO
2
3 INPUT_FILE = '../..samples/fasta22.fas'
4 OUTPUT_FILE = 'fasta22_out2.fas'
5
6 def retseq(seq_fh):
7     """
8     Parse a fasta file and returns non empty records
9     :seq_fh: File handle of the input sequence
10    :return: Non empty sequences
11    """
12    for record in SeqIO.parse(seq_fh, 'fasta'):
13        if len(record.seq):
14            yield record
15
16 with open(INPUT_FILE) as in_fh:
17     with open(OUTPUT_FILE, 'w') as out_fh:
18         SeqIO.write(retseq(in_fh), out_fh, 'fasta')

```

Code explanation: This code is very similar to [Listing 15.2](#). The first difference that is apparent when they are compared line by line is that in this code there is no empty list to store the sequences (it was called *fullseqs* in [Listing 15.2](#)). Another difference is that in the first code, *retseq* is a function while in the last version it is a generator. Both differences are tightly related: Since generators return elements one

by one, there is no need to use a list. The generator yields one record to *SeqIO.write*, which keeps on calling the generator until it gets a *StopIteration* exception.

There is another way to do the same task without using generators and functions, while still consuming an optimal amount of RAM:

Listing 15.4: `seqio3.py`: Yet another way to filter a FASTA file

```

1 from Bio import SeqIO
2
3 INPUT_FILE = '.././samples/fasta22.fas'
4 OUTPUT_FILE = 'fasta22_out3.fas'
5
6 with open(INPUT_FILE) as in_fh:
7     with open(OUTPUT_FILE, 'w') as out_fh:
8         for record in SeqIO.parse(in_fh, 'fasta'):
9             if len(record.seq):
10                 SeqIO.write([record], out_fh, 'fasta')
```

Note that there is no list creation and there is no *generator* because the *SeqIO.write* function saves the records as soon as it receives them.¹ If [Listing 15.4](#) was shown in the first place, I wouldn't have the chance to show the difference between using a function and a generator.

15.4 PROBLEM THREE: MODIFY EVERY RECORD OF A FASTA FILE

In this problem we have a FASTA file that looks like the one in [Listing 15.5](#):

Listing 15.5: Input file

```

>Protein-X
NYLNNLTVDPDHMKCDNTTGRKGNAPGPCVQRTYVACH
>Protein-Y
MEEPQSDPSVEPPLSQETFSDLWKLLPENNVLSPLPSQAMDDLMLSPDDIEQWFTEDPGPDA
>Protein-Z
MKA AVLAVLVFLTGCAWEFWQQDEPQSQWDRVKDFATVYVDAVKDSGRDYVSQFESST
```

The goal of the exercise is to modify all the sequences by adding the species tag in each sequence name. This kind of file modification may be required for sequence submission for a genetic data bank. A modified FASTA file would look like this:

Listing 15.6: Input file

¹There is some internal small RAM caching but it is not relevant in terms of how this function works.

```
>Protein-X [Rattus norvegicus]
NYLNNLTVDPDHMKCDNTTGRKGNAPGPCVQRTYVACH
>Protein-Y [Rattus norvegicus]
MEEPQSDPSVEPPLSQETFSDLWKLLPENNVLSPLPSQAMDDLMLSPDDIEQWFTEDPGPDA
>Protein-Z [Rattus norvegicus]
MKA AVLAVALVFLTGCQAWFEFWQQDEPQSQWDRVKDFATVYVDAVKDSGRDYVSQFESST
```

Note that in [Listing 15.6](#) there is the tag `[Rattus norvegicus]` in the name of each record. There are several ways to accomplish this task. Here is a version with Biopython `Bio.SeqIO` module ([Listing 15.7](#)) and another that uses just the Standard Python Library ([Listing 15.8](#)).

15.4.1 Commented Source Code

Listing 15.7: `seqiofile3.py`: Add a tag in a FASTA sequence with Biopython

```
1 from Bio import SeqIO
2
3 INPUT_FILE = 'fasta22_out.fas'
4 OUTPUT_FILE = 'fasta33.fas'
5
6 with open(INPUT_FILE) as in_fh:
7     with open(OUTPUT_FILE, 'w') as out_fh:
8         for record in SeqIO.parse(in_fh, 'fasta'):
9             # Modify description
10             record.description += '[Rattus norvegicus]'
11             SeqIO.write([record], out_fh, 'fasta')
```

Even if you can use Biopython to modify a FASTA sequence, in a case like this, it's overkill. The following code shows how to accomplish the same task without Biopython:

Listing 15.8: `seqiofile4.py`: Add a tag in a FASTA sequence

```
1 INPUT_FILE = 'fasta22_out.fas'
2 OUTPUT_FILE = 'fasta33.fas'
3
4 with open(INPUT_FILE) as in_fh:
5     with open(OUTPUT_FILE, 'w') as out_fh:
6         for line in in_fh:
7             if line.startswith('>'):
8                 line = line.replace('\n', '[Rattus norvegicus]\n')
9                 out_fh.write(line)
```

Web Application for Filtering Vector Contamination

CONTENTS

16.1	Problem Description	321
16.1.1	Commented Source Code	322
	HTML Input Form	322
	HTML Output Template	323
	Server Code	323
16.2	Additional Resources	326

16.1 PROBLEM DESCRIPTION

DNA sequences are usually inserted into a cloning vector for manipulation. When sequencing, these constructs frequently produce raw sequences that include segments derived from a vector. If the vector part of the raw sequence is not removed, the finished sequence will be contaminated, spoiling further analysis. There are multiple sources of DNA contamination, like transposons, insertion sequences, organisms infecting our samples, and other organisms used in the same laboratory (e.g., cross contamination from dirty equipment).

Sequence contamination is not a minor issue, since it can lead to several problems:¹

- Time and effort wasted on meaningless analyses
- Misassembly of sequence contigs and false clustering
- Erroneous conclusions drawn about the biological significance of the sequence
- Pollution of public databases
- Delay in the release of the sequence in a public database

¹For information regarding each item, please see NCBI VecScreen program at <http://www.ncbi.nlm.nih.gov/VecScreen/contam.html>.

In order to identify the vector part of a sequence, a BLAST can be done against a vector sequence database (or against any other database that you think your sequence could be contaminated by). To help in removing those sequences, this program takes a sequence or a group of sequences in FASTA format and makes the BLAST against a user-selected database. It identifies the match and the contamination is masked by using the “N” character in the sequence input by the user.

This program works as a web application, so there is an HTML form for the user to enter the data and a Python file to process it.

16.1.1 Commented Source Code

HTML Input Form

Listing 16.1: index.html: Form for vector filter

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head><meta charset="utf-8">
4     <title>Vector Filter</title>
5     <link href="css/bootstrap.min.css" rel="stylesheet">
6   </head>
7   <body style="background-color:#e7f5f5;">
8     <div class="container"><h2>Vector Filter</h2>
9     <form action="/vector_result" method="post">
10       <div class="row">
11         <div class="col-sm-8">
12           <div class="form-group">
13             <label for="seqs">Paste sequence in FASTA format:</label>
14             <p><textarea name="seqs" rows="15" cols="80"></textarea></p>
15           </div>
16         </div>
17       </div>
18       <div class="row">
19         <div class="col-sm-8">
20           <div class="form-group">
21             <label for="prop">Filter by: </label>
22             <div class="radio">
23               <label>
24                 <input type="radio" name="vector" value="customdb">Custom Vectors
25               </label>
26             </div>
27             <div class="radio">
28               <label>
29                 <input type="radio" name="vector" value="ncbivector">NCBI Vector DB

```

```

30     </label>
31 </div>
32 <br>
33 <button type="submit" class="btn btn-primary">Send
34 </button>
35 </div>
36 </div>
37 </div>
38 </form>
39 </div>
40 </body>
41 </html>

```

This code produces, when rendered by a web browser, a page like the one shown in [Figure 16.1](#).

In line 4 there is a call to the Python script that processes the form. This script is shown in [Listing 16.3](#):

HTML Output Template

Listing 16.2: `vector_result.tpl`: Template file

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head><meta charset="utf-8">
4 <title>Vector Filter</title>
5 <link href="css/bootstrap.min.css" rel="stylesheet">
6 </head>
7 <body style="background-color:#e7f5f5;">
8 <div class="container">
9     <h2>Vector Filter Result</h2>
10    <pre>{{finalout}}</pre>
11 </div>
12 </body>
13 </html>

```

Server Code

Listing 16.3: `vector.py`: Web script to filter a DNA sequence

```

1 import os, io
2

```

```

3 from Bio import SeqIO
4 from Bio.SeqRecord import SeqRecord
5 from Bio.Blast import NCBIXML
6 from Bio.Blast.Applications import NcbiblastnCommandline as blastn
7 from bottle import route, post, run, static_file, request, view
8 from tempfile import NamedTemporaryFile
9
10 BLAST_EXE = '/home/sb/opt/ncbi-blast-2.6.0+/bin/blastn'
11 DB_BASE_PATH = '/home/sb/opt/ncbi-blast-2.6.0+/db/'
12 MASK = 'N'
13
14 def create_rel(XMLin):
15     """
16     Create a dictionary that relate the sequence name
17     with the region to mask
18     """
19     bat1 = {}
20     output = io.StringIO()
21     output.write(XMLin)
22     output.seek(0)
23     b_records = NCBIXML.parse(output)
24     for b_record in b_records:
25         for alin in b_record.alignments:
26             for hsp in alin.hsps:
27                 qs = hsp.query_start
28                 qe = hsp.query_end
29                 if qs > qe:
30                     qe, qs = qs, qe
31                 record_id = b_record.query.split(' ')[0]
32                 if record_id not in bat1:
33                     bat1[record_id] = [(qs,qe)]
34                 else:
35                     bat1[record_id].append((qs,qe))
36     return bat1
37
38 def maskseqs(ffh, bat1):
39     """
40     Take a FASTA file and apply the mask using the
41     positions in the dictionary
42     """
43     outseqs = []
44     for record in SeqIO.parse(ffh, 'fasta'):
45         if record.id in bat1:
46             # Generate a mutable sequence object to store

```

```

47         # the sequence with the "mask".
48         mutable_seq = record.seq.tomutable()
49         coords = bat1[record.id]
50         for x in coords:
51             mutable_seq[x[0]:x[1]] = MASK*(x[1]-x[0])
52         seq_rec = SeqRecord(mutable_seq, record.id, '', '')
53         outseqs.append(seq_rec)
54     else:
55         # Leave the sequence as found
56         outseqs.append(record)
57     return outseqs
58
59 @route('/')
60 def index():
61     return static_file('index.html', root='views/')
62
63 @post('/vector_result')
64 @view('vector_result')
65 def result():
66     seqs = request.forms.get('seqs')
67     db = os.path.join(DB_BASE_PATH, 'UniVec_Core')
68     if request.forms.get('vector', 'customdb') == 'customdb':
69         db = os.path.join(DB_BASE_PATH, 'custom')
70     # Create a temporary file
71     with NamedTemporaryFile(mode='w') as fasta_in_fh:
72         # Write the user entered sequence into this temporary file
73         fasta_in_fh.write(seqs)
74         # Flush data to disk without closing and deleting the file,
75         # since that closing a temporary file also deletes it
76         fasta_in_fh.flush()
77         # Get the name of the temporary file
78         file_in = fasta_in_fh.name
79         # Run the BLAST query
80         blastn_cline = blastn(cmd=BLAST_EXE, query=file_in, db=db,
81                               evaluate=.0005, outfmt=5)
82         rh, eh = blastn_cline()
83         # Create contamination position and store it in a dictionary
84         bat1 = create_rel(rh)
85         # Get the sequences masked
86         newseqs = maskseqs(file_in, bat1)
87     with io.StringIO() as fasta_out_fh:
88         SeqIO.write(newseqs, fasta_out_fh, 'fasta')
89         fasta_out_fh.seek(0)
90         finalout = fasta_out_fh.read()

```

```

91     return {'finalout':finalout}
92
93 @route('/css/<filename>')
94 def css_static(filename):
95     return static_file(filename, root='css/')
96
97 run(host='localhost', port=8080)

```

Note that this code assumes that there is a BLAST-formatted database (from line 67 to 69). To create such a base you should run the *makeblastdb* utility that is included in the NCBI BLAST+ package.²

If your sequence file is called *mito.nt*, a *makeblastdb* command may look like this:

```

$ ./makeblastdb -dbtype nucl -in ~/sb/UniVec_Core -title UniVec_Core
Building a new DB, current time: 03/04/2017 19:04:07
New DB name:    /sb/UniVec_Core
New DB title:   UniVec_Core
Sequence type:  Nucleotide
Keep MBits: T
Maximum file size: 1000000000B
Adding sequences from FASTA; added 2822 sequences in 0.1253 seconds.

```

where *-in* means “input file”, *-dbtype nucl* stands for *nucleotide* (*-dbtype prot* is for protein). The *makeblastdb* manual is available from the command line with the option *-h*.

16.2 ADDITIONAL RESOURCES

- K. W. Liao, Y. W. Chang, and S. R. Roffler, Presence of cloning vector sequences in the untranslated region of some genes in GenBank, *J. Biomed. Sci.*, 7(6):529–30, 2000.
- C. Miller, J. Gurd, and A. Brass, A RAPID algorithm for sequence database comparisons: Application to the identification of vector contamination in the EMBL databases, *Bioinformatics*, 15(2):111–21, 1999.
- G. A. Seluja, A. Farmer, M. McLeod, C. Harger, and P. A. Schad, Establishing a method of vector contamination identification in database sequences, *Bioinformatics*, 15(2):106–10, 1999.
- C. Savakis and R. Doelz, Contamination of cDNA sequences in databases, *Science*, 259(5102):1677–8, 1993.

²Look for the appropriate package for your system on this FTP site: <ftp://ftp.ncbi.nih.gov/blast/executables/LATEST>.

Vector Filter

Paste sequence in FASTA format:

```
>KF264557.1 Uncultured bacterium esnapd17 genomic sequence
GTGCCCCGCCGGGTACGGGCACCGTTCCGCGAGCCGGGCAACGACCTGGAAC
GGACCATCGCCGAAGTGTGGTCGGAGGTAAGTGGGCGAGGAGCGCCTCGGCG
CCGACGACAGCTTCTCGACCTCGGCGGCACCTCGTTGCACGCCGGCCGGG
TGCTCAGCCGGCTGCGCGAACGGACCGGCCTGCCGGTGTCTGCTGCAGGACA
TCTTCTTCTTCCCGACCCCGGCCGGCCTGGCCCGGCTGCTCGGCTCCCGCG
AGCCGGTGCAGAGCGGACCGGCCCTGCCGAAGATCGGCAGGAGGAGCCGAT
GAGCGCCGCGTTCCGCCGCCGAGTCACCGTGGATCCGCCGGGGCCGCCCGGC
CCGGCGCCACCATCCGGCTGTTCTGCTTCCCGTACGCGGGGGTCCGGCGCT
CGGCCTTCCGCGACTGGCCCGAGCGGCTGCCCGACACCGTCGAGGTGGTCG
CCGTGCAGCTGCCCGGCCGGGAGGACCGCACGAAGGAACAGCTGCCGGCC
GACGTGCCGACGCTGGCCCGCCGCTGCGCGTTGGCGCTGCTGCCGTACACG
ACGATGCCCTTCGCCCTGTACGGCCACTGCGCCGGCGGGCTGGTCGCCTAC
GAGGTGGCGCAGGAGCTGCGGGCCCGTGGCGCGCGCAGCCCTGAGCTGCT
GCTGGCCGCGGCCCACCCGGCGCCGCACCGGCAGCGCAACCGCGAGCCGG
```

Filter by:

☐ Custom Vectors

☐ NCBI Vector DB

Send

Figure 16.1 HTML form for sequence filtering.

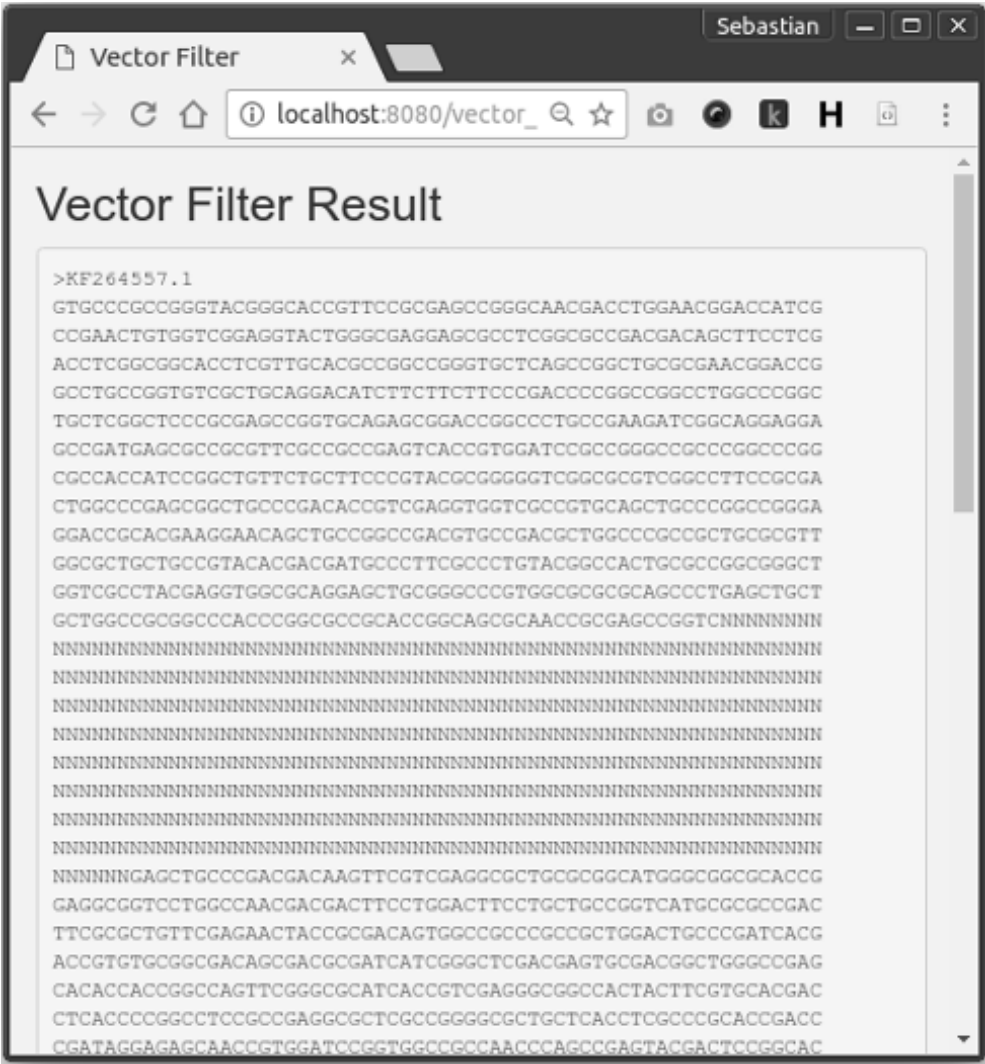


Figure 16.2 HTML form for sequence filtering.

Searching for PCR Primers Using Primer3

CONTENTS

17.1	Problem Description	329
17.2	Primer Design Flanking a Variable Length Region	330
17.2.1	Commented Source Code	331
17.3	Primer Design Flanking a Variable Length Region, with Biopython .	332
17.4	Additional Resources	333

17.1 PROBLEM DESCRIPTION

Primers are small DNA strands (from 15 to 30 base pairs long) that are complementary to a specific spot in a DNA molecule. They are needed for DNA replication to take place. In molecular biology, primers are used for a DNA amplification chain reaction called PCR (**P**olymerase **C**hain **R**eaction). PCR primers have their own characteristics, like specific melting temperature, primer length, need to avoid self-complementarity, and other parameters.¹

PCR primer design is one of the most ubiquitous tasks done in a molecular biology laboratory. There are several programs that help researchers to pick good primers. Some programs are web based, some of them are standalone GUI applications like VectorNTI Suite² and Oligo.net. These programs are suitable for case-by-case study of a few sequences, but they are not the chosen option for automatic batch generation of hundreds of primers, a task that is routinely done in sequencing and fingerprinting projects. One of the most used programs is *primer3*.³ This is due to the high quality of proposed primers and because it can be run in batch and generate multiple primers at once.

Primer3 takes care of primer design; we only need is to prepare the input file for *primer3*. This input file holds the sequence for which the primer should be picked

¹For more information on primer design please see “Additional Resources.”

²VectorNTI web site: <https://www.thermofisher.com/us/en/home/life-science/cloning/vector-nti-software.html>.

³This software is available at <http://sourceforge.net/projects/primer3>; please see the included documentation for how to cite this software if used in a publication.

and other required and optional parameters like desired primer name, primer size, product size, regions to exclude, and others.

A *primer3* input file looks like this:

```
PRIMER_SEQUENCE_ID=<Name>
SEQUENCE=<DNA Sequence in one line>
TARGET=<start>,<length>
PRIMER_OPT_SIZE=<size>
PRIMER_MIN_SIZE=<size>
PRIMER_MAX_SIZE=<size>
PRIMER_NUM_NS_ACCEPTED=<int>
PRIMER_EXPLAIN_FLAG=<int>
PRIMER_PRODUCT_SIZE_RANGE=<start>-<end>
=
```

Each parameter is detailed in the *primer3* README.txt file, although most of them are self-explanatory. The = character is used to terminate the record. Several records can be included in one *primer3* input file.

This recipe chapter is divided in two tasks. The first task will be to generate an input file for *primer3* based in a FASTA file with one sequence inside and one restriction. The second task involves analysis of several sequences for the generation of a multiple sequence primer3 input file.

17.2 PRIMER DESIGN FLANKING A VARIABLE LENGTH REGION

This script should read a FASTA-formatted file with one sequence inside. This sequence has a microsatellite⁴ of variable length where we should avoid doing primer design over it. In fact, we need to assure that the chosen primer flanks this region. We don't know the length or the position of the microsatellite, but we know it is a repeat of the "AAT" sequence and it can be present between 5 to 15 times.

This task could be divided into the following steps:

1. Read the sequence from the FASTA file: Biopython provides the **SeqIO** module that will be used to read the sequence.
2. Detect the region with the microsatellite and store its position: A sliding windows approach will be used. On each possible 45-base-pair-long window⁵ we will count how many times our repeated sequence is present. The chosen window will be the one with the highest number of repetitions inside. We need to store the position of this window.

⁴A **microsatellite** is a region in the chromosome that is characterized by having a small DNA sequence repeated a variable number of times. Since they are inheritable, they can be used to trace relationships between individuals.

⁵This size is estimated by calculating a 3-letter repeat (AAT) and this sequence can be repeated up to 15 times.

3. Generate the primer3 input file: This is trivially accomplished by using the retrieved sequence and the previously stored position as a target.

17.2.1 Commented Source Code

Listing 17.1: primer31.py: Primer design out of one sequence without Biopython

```

1 from Bio import SeqIO
2
3 sfile = open('.././samples/hsc1.fasta')
4 # myseq stores a SeqRecord object generated from the
5 # first record in the fasta file.
6 myseq = SeqIO.read(sfile, "fasta")
7 # title stores the "id" attribute of the SeqRecord object.
8 title = myseq.id
9 seq = str(myseq.seq).upper()
10 win_size = 45
11 i = 0
12 number_l = []
13 # This while is used to walk over the sequence.
14 while i<=(len(seq)-win_size):
15     # Each position of number_l stores the amount of 'AAT'
16     # found on each window.
17     number_l.append(seq[i:i + win_size].count('AAT'))
18     i += 1 # This is the same as i = i+1
19 # pos stores the position of the window with the highest
20 # amount of 'AAT'
21 pos = number_l.index(max(number_l))
22 data = {'title': title, 'seq': seq, 'pos': pos, 'win_size':
23         win_size, 'len_seq': len(seq)}
24 # Saves the data formatted as the input file needed by
25 # primer3.
26 with open('swforprimer3.txt','w') as f_out:
27     with open('template') as tpl:
28         completed = tpl.read().format(**data)
29         f_out.write(completed)

```

This program could process a file like this one:⁶

```

>NC_000001.11:156082546-156140089
GTAGTTTCCCGCCCTTGGGGGCGCGGGGACAAATTCCTTGACCCGAGGAGGATAGGGATGTGGC

```

⁶To change the name of the input file, you have to change line 3.

```
CTTCGGTCTTTCTCGCAGCTCCGGGGCAAGCTAGGAGTGGGATGGAAGTCGAGGTCCCTAATT
TTTTAAGGGGAGGGTGCAGGGGAGAAGGGGTAGTATGCGGAAACAGAGCGGGTATGAAGCTGGCT
AACGCCGCGCGCCCCCTCCAGGACCCGCTCCTGCCCCGCGCCGGTCTCTGGGGGCCCCGCT
TTTTTATGGAATGAGGAGGGGGGCGGGGCGGGGCGGGGAGCCGGGAGCCGGGGGTAGTA
(...)
```

The result of [Listing 17.1](#) produces a file like this one:

```
PRIMER_SEQUENCE_ID=NC_000001.11:156082546-156140089
SEQUENCE=GTAGTTTCCCGCCCTTGGGGGCGCGGGACAAATTCCTTGACCCGAGGAGGATAG<=
GGATGTGGCCTTCGGTCTTTCTCGCAGCTCCGGGGCAAGCTAGGAGTGGGATGGAAGTCGAGG<=
TCCCTAATTTTTTAAGGGGAGGGTGCAGGGGAGAAGGGGTAGTATGCGGAAACAGAGCGGGTATG<=
AAGCTGGCTAACGCCGCGCGCCCCCTCCAGGACCCGCTCCTGCCCCGCGCCGGCCGGTCTCTGG<=
GGGCGCGCTTTTTTATGGAATGAGGAGGGGGGCGGGGCGGGGCGGGGAGCCGGGAGCCG<=
(...)
TARGET=43423,45
PRIMER_OPT_SIZE=18
PRIMER_MIN_SIZE=15
PRIMER_MAX_SIZE=20
PRIMER_NUM_NS_ACCEPTED=0
PRIMER_EXPLAIN_FLAG=1
PRIMER_PRODUCT_SIZE_RANGE=45-57544
```

This file is used as input into **primer3** in this way:

```
$ ./primer3_core < swforprimer3.txt > primer3out.txt
```

17.3 PRIMER DESIGN FLANKING A VARIABLE LENGTH REGION, WITH BIOPYTHON

Biopython can build the input file for primer3 and also the command line.

Listing 17.2: primer32.py: Primer design out of one sequence without Biopython

```
1 from Bio import SeqIO
2 from Bio.Emboss.Applications import Primer3Commandline
3
4 INPUT_SEQUENCE = open('.././samples/hsc1.fasta')
5 OUTPUT_SEQUENCE = 'primer.txt'
6 sfile = open('.././samples/hsc1.fasta')
7 myseq = SeqIO.read(sfile, 'fasta')
8 title = myseq.id
9 seq = str(myseq.seq).upper()
10 win_size = 45
```

```

11 i = 0
12 number_l = []
13 while i <= (len(seq) - win_size):
14     number_l.append(seq[i:i + win_size].count('AAT'))
15     i += 1 # This is the same as i = i+1
16 pos = number_l.index(max(number_l))
17 pr_cl = Primer3Commandline(sequence=INPUT_SEQUENCE, auto=True)
18 pr_cl.outfile = OUTPUT_SEQUENCE
19 pr_cl.oseqsize = 18
20 pr_cl.maxsize = 20
21 pr_cl.minsize = 15
22 pr_cl.explainflag = 1
23 pr_cl.target = (pos, win_size)
24 pr_cl.prangle = (win_size, len(seq))
25 primer_cl()

```

Code explanation: this code uses **Primer3Commandline** (line 17). After you instantiate this class, all parameters are passed as properties (from line 18 to 24). These properties are used to build the input file. The last line runs `eprimer3`, which calls `primer3`.

17.4 ADDITIONAL RESOURCES

- Andreas Untergasser et al. 2007. Primer3Plus, an enhanced web interface to Primer3. (*Nucleic Acids Res.* Source code available at: <http://sourceforge.net/projects/primer3>. The paper above is available at: <https://academic.oup.com/nar/article-lookup/doi/10.1093/nar/gkm306>.
- Integrating PCR theory and bioinformatics into a research-oriented primer design exercise. (*CBE Life Sci Educ.* 2008 Spring;7(1):89–95)
- Enhancements and modifications of primer design program Primer3. (*Bioinformatics.* 2007 May 15;23(10):1289–91. Epub 2007 Mar 22)
- Emboss eprimer3 manual.
<http://emboss.toulouse.inra.fr/cgi-bin/emboss/help/eprimer3>
- Primer3 manual.
http://primer3.sourceforge.net/primer3_manual.htm



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Calculating Melting Temperature from a Set of Primers

CONTENTS

18.1	Problem Description	335
18.1.1	Commented Source Code	336
18.2	Additional Resources	336

18.1 PROBLEM DESCRIPTION

In this case we have a text file full of PCR primers. These primers were obtained from different sources, so their T_m was calculated under different programs and conditions. A researcher may want to make the criteria of T_m of his set of primers uniform.

The first version of the program will output the file formatted as a csv (comma-separated values) file. This kind of file could be opened with a spreadsheet or with a custom-made program. The second version will output the file as an Excel spreadsheet (xls).

Proposed steps to get the melting temperatures of a set of primers:

1. Read the input file line by line.
2. For each line, calculate the melting temperature (T_m) by using the `MeltingTemp` module from Biopython `Bio.SeqUtils`.
3. Print the primer sequence, a comma and, its T_m value.
4. In the Excel case, print the primer sequence in a cell and its T_m value in the next cell in the same row, using `xlwt`.

18.1.1 Commented Source Code

Listing 18.1: fromtxt.py: Primer Tm calculation

```

1 from Bio.SeqUtils import MeltingTemp as MT
2
3 PRIMER_FILE = '../..samples/primers.txt'
4 for line in open(PRIMER_FILE):
5     # prm stores the primer, without 5'- and -3'
6     prm = line[3:len(line)-4].replace(' ','')
7     # .2f is used to print up to decimals.
8     print('{0},{1:.2f}'.format(prm, MT.Tm_staluc(prm)))

```

Version of the same code with Excel output:

Listing 18.2: toexcel.py: Primer Tm calculation, Excel output

```

1 from Bio.SeqUtils import MeltingTemp as MT
2 import xlwt
3
4 PRIMER_FILE = '../..samples/primers.txt'
5 # w is the name of a newly created workbook.
6 w = xlwt.Workbook()
7 # ws is the name of a new sheet in this workbook.
8 ws = w.add_sheet('Result')
9 # These two lines writes the titles of the columns.
10 ws.write(0, 0, 'Primer Sequence')
11 ws.write(0, 1, 'Tm')
12 for index, line in enumerate(open(PRIMER_FILE)):
13     # For each line in the input file, write the primer
14     # sequence and the Tm
15     prm = line[3:len(line)-4].replace(' ','')
16     ws.write(index+1, 0, prm)
17     ws.write(index+1, 1, '{0:.2f}'.format(MT.Tm_staluc(prm)))
18 # Save the spreadsheet into a file.
19 w.save('primerout.xls')

```

18.2 ADDITIONAL RESOURCES

- PCR primer design guidelines.
http://www.premierbiosoft.com/tech_notes/PCR_Primer_Design.html

- *Molecular Biology Techniques Manual*, Vernon E. Coyne, M. Diane James, Sharon J. Reid, and Edward P. Rybicki, eds.
<http://www.mcb.uct.ac.za/pcroptim.htm>
- 10 tips for designing PCR primers that work.
<http://smartnote.miraibio.com/blog/?p=12>
- Nicolas Le Novère. MELTING, computing the melting temperature of nucleic acid duplex. (*Bioinformatics* 2001 17: 1226–1227). doi: 10.1093/bioinformatics/17.12.1226



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Filtering Out Specific Fields from a GenBank File

CONTENTS

19.1	Extracting Selected Protein Sequences	339
19.1.1	Commented Source Code	339
19.2	Extracting the Upstream Region of Selected Proteins	340
19.2.1	Commented Source Code	340
19.3	Additional Resources	341

Genomes for whole organisms are available at Genbank, the most complete genetic sequence database. The National Center for Biotechnology Information (NCBI) at the National Library of Medicine (NLM), National Institutes of Health (NIH), is responsible for producing and distributing the GenBank Sequence Database. Genbank is also the name of the format in which Genbank records are stored (GenBank Flat File Format). Biopython has reading support for this kind of file (with the **Bio.SeqIO** module).

19.1 EXTRACTING SELECTED PROTEIN SEQUENCES

A researcher wants to extract the protein sequences of each NADH found in the *Nicotiana tabacum* mitochondria.

19.1.1 Commented Source Code

Listing 19.1: `genbank1.py`: Extract sequences from a Genbank file

```
1 from Bio import SeqIO, SeqRecord, Seq
2 from Bio.Alphabet import IUPAC
3
4 GB_FILE = '../samples/NC_006581.gb'
5 OUT_FILE = 'nadh.fasta'
6 with open(GB_FILE) as gb_fh:
7     record = SeqIO.read(gb_fh, 'genbank')
```

```

8 seqs_for_fasta = []
9 for feature in record.features:
10     # Each Genbank record may have several features, the program
11     # will walk over all of them.
12     qualifier = feature.qualifiers
13     # Each feature has several parameters
14     # Pick selected parameters.
15     if 'NADH' in qualifier.get('product', [''])[0] and \
16     'product' in qualifier and 'translation' in qualifier:
17         id_ = qualifier['db_xref'][0][3:]
18         desc = qualifier['product'][0]
19         # nadh_sq is a NADH protein sequence
20         nadh_sq = Seq.Seq(qualifier['translation'][0],
21                             IUPAC.protein)
22         # 'srec' is a SeqRecord object from nadh_sq sequence.
23         srec = SeqRecord.SeqRecord(nadh_sq, id=id_,
24                                     description=desc)
25         # Add this SeqRecord object into seqsforfasta list.
26         seqs_for_fasta.append(srec)
27 with open(OUT_FILE, 'w') as outf:
28     # Write all the sequences as a FASTA file.
29     SeqIO.write(seqs_for_fasta, outf, 'fasta')

```

19.2 EXTRACTING THE UPSTREAM REGION OF SELECTED PROTEINS

Regulatory elements are found mostly upstream of the beginning of the genes. They include polyadenylation signals, TATA box, enhancers, and more.

For this program we have a Genbank file and list of genes (cox2, atp6, atp9, cob) whose sequences we want to extract plus the upstream region, up to 1000 base pairs.

19.2.1 Commented Source Code

Listing 19.2: genbank2.py: Extract upstream regions

```

1 from Bio import SeqIO
2 from Bio.SeqRecord import SeqRecord
3
4 GB_FILE = '../samples/NC_006581.gb'
5 OUT_FILE = 'tg.fasta'
6 with open(GB_FILE) as gb_fh:
7     record = SeqIO.read(gb_fh, 'genbank')
8 seqs_for_fasta = []

```

```

9 tg = (['cox2'], ['atp6'], ['atp9'], ['cob'])
10 for feature in record.features:
11     if feature.qualifiers.get('gene') in tg:
12         if feature.type=='gene':
13             # Get the name of the gene
14             genename = feature.qualifiers.get('gene')
15             # Get the start position
16             startpos = feature.location.start.position
17             # Get the required slice
18             newfrag = record.seq[startpos-1000: startpos]
19             # Build a SeqRecord object
20             seq = genename[0] + ' 1000bp upstream'
21             newrec = SeqRecord(newfrag, seq, '', '')
22             seqs_for_fasta.append(newrec)
23 with open(OUT_FILE, 'w') as outf:
24     # Write all the sequences as a FASTA file.
25     SeqIO.write(seqs_for_fasta, outf, 'fasta')

```

19.3 ADDITIONAL RESOURCES

- Benson D.A., Karsch-Mizrachi I., Lipman D.J., Ostell J., Wheeler D.L. GenBank. *Nucleic Acids Res.* 36(Database issue), D25–30 (2008).
<http://www.ncbi.nlm.nih.gov/pubmed/18073190>
- GenBank Flat File Format. Sample record with detailed specifications.
<http://www.ncbi.nlm.nih.gov/Sitemap/samplerecord.html>



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Inferring Splicing Sites

CONTENTS

20.1	Problem Description	343
20.1.1	Infer Splicing Sites with Commented Source Code	345
20.1.2	Sample Run of Estimate Intron Program	347

20.1 PROBLEM DESCRIPTION

An expressed sequence tag or EST is a short sub-sequence of a transcribed spliced nucleotide sequence (either protein-coding or not). They may be used to identify gene transcripts, and are instrumental in gene discovery and gene sequence determination. The identification of ESTs has proceeded rapidly, with approximately 52 million ESTs now available in public databases (e.g., GenBank 5/2008, all species).

An EST is produced by one-shot sequencing of a cloned mRNA (i.e., sequencing several hundred base pairs from an end of a cDNA clone taken from a cDNA library). The resulting sequence is a relatively low-quality fragment whose length is limited by current technology to approximately 500 to 800 nucleotides. Because these clones consist of DNA that is complementary to mRNA, the ESTs represent portions of expressed genes. They may be present in the database as either cDNA/mRNA sequences or as the reverse complement of the mRNA, the template strand.

A way to find splicing sites in a plant sequence is to compare, using BLAST, this sequence with the genome of a known species like *Arabidopsis thaliana* (AT). Then we align our sequence with both the full sequence and the coding DNA sequence (CDS) of the closest match. With this technique we could infer intron, exons, and splicing sites. A script to accomplish this should first do a BLAST search, then use the ID of the result to search the AT sequences in a database. This database, in this case using SQLite, must be made in advance. With all sequences, we use **MultipleAlignCL** from **Biopython** to align them.

The full sequence and CDS can be downloaded from the TAIR website at <https://www.arabidopsis.org/>¹

¹Full sequences are at https://www.arabidopsis.org/download_files/Genes/TAIR10_genome_release/TAIR10_blastsets/TAIR10_seq_20101214_updated. and CDS are at https://www.arabidopsis.org/download_files/Genes/TAIR10_genome_release/TAIR10_blastsets/TAIR10_cds_20101214_updated.

The preparatory program to make de SQLite database with full sequence and CDS ([Listing 20.1](#)).

Listing 20.1: `makedb.py`: Convert data for entering into an SQLite database

```

1 import sqlite3
2 from Bio import SeqIO
3
4 seq_file = open('../samples/TAIR10_seq_20101214_updated')
5 cds_file = open('../samples/TAIR10_cds_20101214_updated')
6 AT_DB_FILE = 'AT.db'
7
8 at_d = {}
9 # Get all sequences from TAIR sequences file.
10 for record in SeqIO.parse(seq_file, 'fasta'):
11     sid = record.id
12     seq = str(record.seq)
13     at_d[sid] = [seq]
14 # Get all sequences from TAIR CDS file.
15 for record in SeqIO.parse(cds_file, 'fasta'):
16     sid = record.id
17     seq = str(record.seq)
18     at_d[sid].append(seq)
19 # Write to a CSV file only the entries of the dictionary that
20 # has data from both sources
21 conn = sqlite3.connect(AT_DB_FILE)
22 c = conn.cursor()
23 c.execute('create table seq(id, cds, full_seq)')
24 for seq_id in at_d:
25     if len(at_d[seq_id])==2:
26         # Write in this order: ID, CDS, FULL_SEQ.
27         c.execute('INSERT INTO seq VALUES (?, ?, ?)',
28                 ((seq_id, at_d[seq_id][1], at_d[seq_id][0])))
29 conn.commit()
30 conn.close()

```

[Listing 20.1](#) populates a new SQLite database called `AT.db`. This database has a table (`seq`) and this table has three fields: `id`, `cds` and `full_seq`.

To format the TAIR cds database to BLAST, run:

```
$ makeblastdb -in TAIR10_cds_20101214_updated -dbtype nucl -out T10
```

20.1.1 Infer Splicing Sites with Commented Source Code

Listing 20.2: `estimateintrons.py`: Estimate introns

```

1  #!/usr/bin/env python
2
3  import argparse
4  import os
5  import sqlite3
6
7  from Bio import SeqIO, SeqRecord, Seq
8  from Bio.Align.Applications import ClustalwCommandline
9  from Bio.Blast import NCBIXML
10 from Bio.Blast.Applications import NcbiblastnCommandline as bn
11 from Bio import AlignIO
12
13 AT_DB_FILE = 'AT.db'
14 BLAST_EXE = '~/opt/ncbi-blast-2.6.0+/bin/blastn'
15 BLAST_DB = '~/opt/ncbi-blast-2.6.0+/db/TAIR10'
16 CLUSTALW_EXE = '../../clustalw2'
17
18 def allgaps(seq):
19     """Return a list with tuples containing all gap positions
20     and length. seq is a string."""
21     gaps = []
22     indash = False
23     for i, c in enumerate(seq):
24         if indash is False and c == '-':
25             c_ini = i
26             indash = True
27             dashn = 0
28         elif indash is True and c == '-':
29             dashn += 1
30         elif indash is True and c != '-':
31             indash = False
32             gaps.append((c_ini, dashn+1))
33     return gaps
34
35 def iss(user_seq):
36     """Infer Splicing Sites from a FASTA file full of EST
37     sequences"""
38
39     with open('forblast','w') as forblastfh:
40         forblastfh.write(str(user_seq.seq))

```



```

41
42     blastn_cline = bn(cmd=BLAST_EXE, query='forblast',
43                       db=BLAST_DB, evaluate='1e-10', outfmt=5,
44                       num_descriptions='1',
45                       num_alignments='1', out='outfile.xml')
46     blastn_cline()
47     b_record = NCBIXML.read(open('outfile.xml'))
48     title = b_record.alignments[0].title
49     sid = title[title.index(' ')+1 : title.index(' |')]
50     # Polarity information of returned sequence.
51     # 1 = normal, -1 = reverse.
52     frame = b_record.alignments[0].hsp[0].frame[1]
53     # Run the SQLite query
54     conn = sqlite3.connect(AT_DB_FILE)
55     c = conn.cursor()
56     res_cur = c.execute('SELECT CDS, FULL_SEQ from seq '
57                        'WHERE ID=?', (sid,))
58     cds, full_seq = res_cur.fetchone()
59     if cds=='':
60         print('There is no matching CDS')
61         exit()
62     # Check sequence polarity.
63     sidcds = '{0}-CDS'.format(sid)
64     sidseq = '{0}-SEQ'.format(sid)
65     if frame==1:
66         seqCDS = SeqRecord.SeqRecord(Seq.Seq(cds),
67                                       id = sidcds,
68                                       name = '',
69                                       description = '')
70         fullseq = SeqRecord.SeqRecord(Seq.Seq(full_seq),
71                                       id = sidseq,
72                                       name='',
73                                       description='')
74     else:
75         seqCDS = SeqRecord.SeqRecord(
76             Seq.Seq(cds).reverse_complement(),
77             id = sidcds, name='', description='')
78         fullseq = SeqRecord.SeqRecord(
79             Seq.Seq(full_seq).reverse_complement(),
80             id = sidseq, name = '', description='')
81     # A tuple with the user sequence and both AT sequences
82     allseqs = (record, seqCDS, fullseq)
83     with open('foralign.txt','w') as trfih:
84         # Write the file with the three sequences

```

```

85     SeqIO.write(allseqs, trifh, 'fasta')
86     # Do the alignment:
87     outfile = '{0}.aln'.format(user_seq.id)
88     cline = ClustalwCommandline(CLUSTALW_EXE,
89                               infile = 'foralign.txt',
90                               outfile = outfile,
91                               )
92     cline()
93     # Walk over all sequences and look for query sequence
94     for seq in AlignIO.read(outfile, 'clustal'):
95         if user_seq.id in seq.id:
96             seqstr = str(seq.seq)
97             gaps = allgaps(seqstr.strip('-'))
98             break
99     print('Original sequence: {0}'.format(user_seq.id))
100    print('\nBest match in AT CDS: {0}'.format(sid))
101    acc = 0
102    for i, gap in enumerate(gaps):
103        print('Putative intron #{0}: Start at position {1}, '
104              'length {2}'.format(i+1, gap[0]-acc, gap[1]))
105        acc += gap[1]
106    print('\n{0}'.format(seqstr.strip('-')))
107    print('\nAlignment file: {0}\n'.format(outfile))
108
109    description = 'Program to infer intron position based on ' \
110                'Arabidopsis Thaliana genome'
111    parser = argparse.ArgumentParser(description=description)
112    ifh = 'Fasta formatted file with sequence to search for introns'
113    parser.add_argument('input_file', help=ifh)
114    args = parser.parse_args()
115    seqhandle = open(args.input_file)
116    records = SeqIO.parse(seqhandle, 'fasta')
117    for record in records:
118        iss(record)

```

20.1.2 Sample Run of Estimate Intron Program

```

$ python estimateintrons.py ../../samples/t3.fasta
Original sequence: secu3

```

```

Best match in AT CDS: AT1G14990.1
Putative intron #1: Start at position 171, length 95
Putative intron #2: Start at position 250, length 153

```

```

CTAGCCACTTCCAACGAGTTGGCCTTGAGATAGAAGGTGAGCCATGTATTGGGAGTGGTAAA<=
CGTATGGAGATTTTCCCTGGCGATCAAAATGCTTAGCCATTATGCAGAATTCAACAGGACCG<=
GAATCTTCAGATTCATAGCCTTTCCCAAGCGCCGCTTTGTACAGCTT-----<=
-----<=
-----AGCTGTGTCGGTCAAAAGTTCGGTGCCAGCAGTCGAAGATGCAT<=
AAAACTGATCTCCCCTGGAATATCCTGCTCTTGTT-----<=
-----<=
-----<=
--GTGTTGTTTGTATAGAAGAATGTGAGGGCAGCAGTGAAGCAGTAGAATCCGGCGTAAGAG<=
ACAGCCCGTCGTAGCTTCTGGATAATTATAACCTCTGAGCGGTCATCCAAGATCATCAT

```

Alignment file: secu3.aln

Web Server for Multiple Alignment

CONTENTS

21.1	Problem Description	349
21.1.1	Web Interface: Front-End. HTML Code	349
21.1.2	Web Interface: Server-Side Script. Commented Source Code ..	351
21.2	Additional Resources	353

21.1 PROBLEM DESCRIPTION

DNA sequences of different organisms are often related. The closer the species, the more similar are their genomes. Some genes are highly conserved while others have extensive arrangement and mutations. Sequence multiple alignment (MSA) helps show the relationship between sequences and infer an evolutionary history.

There are several programs to perform MSA. It is out of the scope of this book to review them, but there are pointers to several papers in “Additional Resources” for those interested in MSA software.

One of these programs is **MUSCLE** (Multiple Sequence alignment by log-expectation), which is characterized by its improved speed and accuracy over currently available programs. Since **MUSCLE** has no graphical user interface (GUI), it is a command-line application, and we will build a GUI using a web server.

The advantage of using a web server it is not only the GUI, but the ability to use it from several computers.

21.1.1 Web Interface: Front-End. HTML Code

The first step is to make the GUI in HTML. Before reinventing the wheel, I searched for a Muscle web server and found one at the EBI website (<http://www.ebi.ac.uk/Tools/muscle>). Inspired by this site, I made the form displayed in Figure 21.1. The HTML code for this form is shown in Listing 21.1.

Listing 21.1: index.html: web interface to Muscle front end

```
1 <!DOCTYPE html><html lang="en">
```

Figure 21.1 Muscle Web interface.

```

2 <head><meta charset="utf-8">
3 <title>Muscle Web Interface</title>
4 <link href="css/bootstrap.min.css" rel="stylesheet">
5 </head>
6 <body style="background-color:#e7f5f5;">
7 <div class="container"><h1>Muscle Web Interface</h1>
8 <form action='muscle_result' method='post'
9   enctype="multipart/form-data">
10 <div class="row"><div class="col-sm-8"><div
11 class="form-group"><label for="iterations">Maximum iterations:
12 </label><select name="iterat" id="iterations">
13 <option value="1" selected="selected">1</option>
14 <option value="4">4</option><option value="8">8</option>
15 <option value="10">10</option><option value="12">12</option>
16 <option value="14">14</option><option value="16">16</option>
17 </select>
18 <label for="format"> Output Format:</label>
19 <select name="output" id="format">
20 <option value="fasta" selected="selected">FASTA</option>
21 <option value="clw">ClustalW2</option>
22 <option value="clwstrict">ClustalW2 (Strict)</option>
23 <option value="html">HTML</option>
24 <option value="msf">MSF</option>
25 </select>
26 <label for="order">Output Order:</label>
27 <select name="outorder" id="order">
28 <option value="group" selected="selected">aligned</option>
29 <option value="stable">input</option>

```

```

30     </select></div></div></div>
31     <div class="row"><div class="col-sm-6">
32         <div class="form-group"><label for="sq">
33             Enter a set of sequences in FASTA format:
34         </label>
35         <textarea name="seq" rows="5" cols="90" id="sq"></textarea>
36     </div></div></div><div class="row"><div class="form-group">
37     <div class="col-sm-3">
38     <label for="upfile">Or upload a file:</label>
39     <input type="file" name="upfile" id="upfile" />
40     </div><div class="col-sm-3">
41     <button type="submit" class="btn btn-primary">
42     Send to Muscle server</button></div></div></div>
43 </form></div></body></html>

```

Template for results page:

Listing 21.2: result.tpl: Template for the result of Muscle server

```

1 <html lang="en">
2   <head>
3     <meta charset="utf-8">
4     <title>Muscle Web Interface</title>
5     <link href="css/bootstrap.min.css" rel="stylesheet">
6   </head>
7   <body>
8     <div class="container">
9       % if bad_option:
10        <h1>Bad Request</h1>
11        Use the options provided in the form. Error in {{bad_opt}}
12      % elif not bad_option:
13        {{!result_output}}
14      % end
15      <p>Go back to the <a href="/">home page</a></p>
16    </div>
17  </body>
18 </html>

```

21.1.2 Web Interface: Server-Side Script. Commented Source Code

Listing 21.3: muscleserver.py: Web interface to Muscle

```

1  #!/usr/bin/env python
2
3  import os
4  import subprocess
5  from tempfile import mkstemp
6
7  from bottle import route, post, run, static_file, request, view
8
9  @route('/')
10 def index():
11     return static_file('index.html', root='views/')
12
13 @post('/muscle_result')
14 @view('result')
15 def result():
16     iterations = request.forms.get('iterat','1')
17     output_type = request.forms.get('output','FASTA')
18     order = request.forms.get('outorder','group')
19     sequence = request.forms.get('seq','')
20     if not sequence:
21         # If the textarea is empty, check the uploaded file
22         sequence = request.files.get('upfile').file.read()
23     badreq = ''
24     # Verify that the user entered valid information.
25     try:
26         int(iterations)
27     except ValueError:
28         badreq = 'iterations'
29     valid_output = ('html', 'fasta', 'msf', 'clw', 'clwstrict')
30     if output_type not in valid_output:
31         badreq = 'output'
32     if order not in ('group', 'stable'):
33         badreq = 'outorder'
34     result_out = ''
35     # Make a random filename for user entered data
36     fi_name = mkstemp('.txt','userdata_')[1]
37     with open(fi_name,'wb') as fi_fh:
38         fi_fh.write(sequence)
39     fo_name = mkstemp('.txt','outfile_')[1]
40     with open('muscle3_error.log','w') as erf:
41         cmd = ['muscle3.8.31_i86linux64', '-in', fi_name,
42               '-out', fo_name, '-quiet', '-maxiters',
43               iterations, '-{}'.format(output_type),
44               '-{}'.format(order)]

```

```

45     p = subprocess.Popen(cmd, stderr=erfh)
46     p.communicate()
47     # Remove the input file
48     os.remove(fi_name)
49     with open(fo_name) as fout_fh:
50         result_out = fout_fh.read()
51     if output_type != 'html':
52         result_out = '<pre>{0}</pre>'.format(result_out)
53     # Remove the output file
54     os.remove(fo_name)
55     return {'bad_opt':badreq, 'result_output':result_out}
56
57 @route('/css/<filename>')
58 def css_static(filename):
59     return static_file(filename, root='css/')
60
61 run(host='localhost', port=8080)

```

Code explanation: Although the code is widely commented, I think it is worth some explanation. There are two main methods: **index()** and **result()**. **index()** is called when the user requests the site and it just displays an HTML form using the `static_file` method. When the user completes the form and presses “Send to Muscle Server.” a POST request is sent to the `/muscle_result` URL. The request is handled by the **result()** method. This method calls the command line that makes the alignment. There is a function to generate files with random names (**mkstemp()**) and it is used instead of having a fixed name for each file. The problem with fixed files is that a web program can be used simultaneously by several users and there is a risk of data override. Temporary files are removed (lines 48 and 54).

21.2 ADDITIONAL RESOURCES

- Edgar, Robert C. (2004), MUSCLE: a multiple sequence alignment method with reduced time and space complexity. *BMC Bioinformatics* 2004, 5:113doi:10.1186/1471-2105-5-113.
- Julie D. Thompson, Benjamin Linard, Odile Lecompte, Olivier Poch. A comprehensive benchmark study of multiple sequence alignment methods: current challenges and future perspectives.
<http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0018093>
- Notredame, C (2007). Recent evolutions of multiple sequence alignment algorithms. *PLOS Computational Biology* 8(3):e123 doi: 10.1371/journal.pcbi.0030123.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Drawing Marker Positions Using Data Stored in a Database

CONTENTS

22.1	Problem Description	355
22.1.1	Preliminary Work on the Data	355
22.1.2	MongoDB Version with Commented Source Code	357

22.1 PROBLEM DESCRIPTION

This program makes a graphical representation of a selected locus in five chromosomes of *Arabidopsis thaliana*.¹ The position data of the locus are stored in a database, so the program has to connect such a database and retrieve the data before plotting it. In this case we use MongoDB as the database back-end. The drawing part is made by using the **BasicChromosome** class from Biopython.

In [Figure 22.1](#) there is an example of what the output looks like.

22.1.1 Preliminary Work on the Data

The raw data used by this program are provided by the Arabidopsis Information Resource² (TAIR). The file is located at their FTP server and can be retrieved with any web browser from ftp://ftp.arabidopsis.org/home/tair/Genes/TAIR7_genome_release/TAIR7_Transcripts_by_map_position.gz. It is a gzipped compressed CSV file with more information than the locus position into the chromosome. From this file we need four fields: `Locus`, `Chromosome`, `Map_start_coordinate` and `Map_end_coordinate`.

Here is a brief sample of TAIR data:

```
Locus Locus_orientation_is_5 Genbank_acc external_id <=
```

¹*Arabidopsis thaliana* (wall cress or mouse-ear cress) is a small flowering plant that is widely used as a model organism in plant biology. It is used in this example because there are completed physical and genetic maps available.

²TAIR website is available at <http://www.arabidopsis.org>.

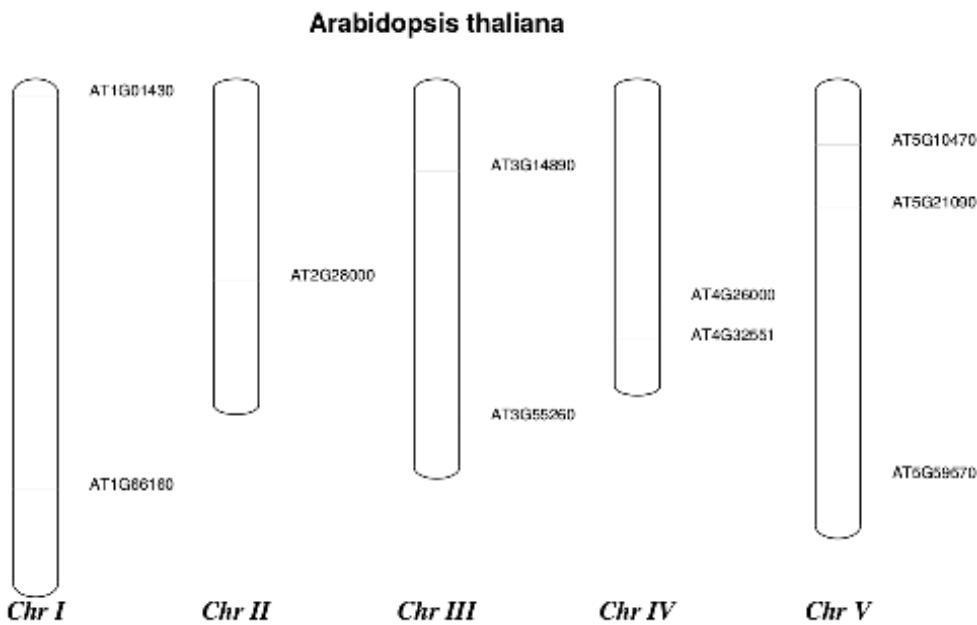


Figure 22.1 Product of [Listing 22.2](#), using the demo dataset (NODBDEMO).

```
Type(1=cDNA 2=EST) Chromosome Transcript_orientation_is_5 <=
Map_start_coordinate Map_end_coordinate
(... cut ...)
AT1G01280 1 BX814827 42472162 1 1 1 112263 113195
AT1G01280 1 BX814827 42472162 1 1 1 113279 113861
AT1G01280 1 AA720028 2733638 2 1 1 112341 112589
AT1G01280 1 AV535036 8695319 2 1 1 112300 112919
AT1G01280 1 AV532990 8693273 2 1 0 113720 113947
AT1G01280 1 BT022023 63003811 1 1 1 112283 113195
AT1G01280 1 BT022023 63003811 1 1 1 113279 113944
(... cut ...)
```

In this dataset, the lowest and highest positions are not properly marked. In the small text snip displayed above, the lower position is 112263 and the highest is 113947, so this text should be translated into the following line:

```
AT1G01280,1,112263,113947
```

Therefore a custom-made script is needed to convert the data for entering into a database.

Listing 22.1: `createdb.py`: Convert the data from a CSV file to insert it into MongoDB

```

1 import csv
2 import gzip
3 import os
4
5 from pymongo import MongoClient, TEXT
6
7 FILE_NAME = '../..samples/TAIR7_Transcripts_by_map_position.gz'
8 CONNECTION_STRING = os.getenv('MONGODB_CS', 'localhost:27017')
9
10 # Get a file handler of an uncompressed file:
11 with gzip.open(FILE_NAME, "rt", newline="") as f_unzip:
12     rows = csv.reader(f_unzip, delimiter='\\t')
13     next(rows) # Skip the header
14     # Dictionary for storing markers and associated information:
15     at_d = {}
16     # Load the dictionary using the data in the file:
17     for row in rows:
18         if row[0] in at_d:
19             chromosome, left_val, right_val = at_d[row[0]]
20             c7 = int(row[7])
21             left = c7 if c7<int(left_val) else left_val
22             c8 = int(row[8])
23             right = c8 if c8>int(right_val) else right_val
24             at_d[row[0]] = (int(chromosome), left, right)
25         else:
26             at_d[row[0]] = (int(row[5]), int(row[7]),
27                             int(row[8]))
28 # Make a list with dictionaries to be stored as documents in
29 # MongoDB
30 markers = []
31 for marker in at_d:
32     markers.append({'marker_id': marker, 'chromosome':
33                     at_d[marker][0], 'start': at_d[marker][1],
34                     'end': at_d[marker][2]})
35 client = MongoClient(CONNECTION_STRING)
36 db = client.TAIRDB
37 collection = db.markers_map
38 collection.insert_many(markers)
39 collection.create_index([('marker_id', TEXT)])

```

22.1.2 MongoDB Version with Commented Source Code

With the database in place, we finally can make a program to retrieve the marker information from the MongoDB database and plot the chromosomes in a PDF document.

The program asks for a list of loci. It checks if each locus conforms to a specific pattern (lines 134 and 167 show how to check a pattern using regex) and then retrieves the data from the database. This program also has two “test modes”: **DB-DEMO** and **NODBDEMO**. These modes are used to test the program without entering all loci by hand. The first mode uses a predefined list of loci (starting at line 148) and then retrieves them from the database. The second mode uses a built-in list of loci with its positions (from line 153) to test the program without a database connection.³ Note that the connection string is taken from an environmental variable. If this variable is not present, it uses the string `'localhost:27017'`. To set up an environmental variable in a Linux/macOS system, use:

```
$ export MONGODB_CS='mongodb://user:pass@dobdomain:port'
```

This program needs **pymongo**, **biopython** and **reportlab** to work:

Listing 22.2: `drawmarker.py`: Draw markers in chromosomes from data extracted from a MongoDB database

```

1 import os
2 import re
3
4 from pymongo import MongoClient
5 from Bio.Graphics import BasicChromosome
6 from reportlab.lib import colors
7
8 CONNECTION_STRING = os.getenv('MONGODB_CS', 'localhost:27017')
9
10 def sortmarkers(crms,end):
11     """ Sort markers into chromosomes """
12     i = 0
13     crms_o = [[] for r in range(len(end))]
14     crms_fo = [[] for r in range(len(end))]
15     for crm in crms:
16         for marker in crm:
17             # add the marker start position at each chromosome.
18             crms_fo[i].append(marker[1])

```

³Having the data inside the program is called *hardcoding*. In most cases it is not recommended since it is a better idea to have the data in an easy-to-change external file. In this case the data is hardcoded since this data is only for debugging purposes.

[illegible]

```

63         chromo[i].append('',None,end[i]-crm_o[0][2]))
64     else:
65         # For chromosomes without markers
66         # Add 3% of each chromosome.
67         chromo[i].append('',None,int(0.03*end[i]))
68         chromo[i].append('',None,end[i]))
69         chromo[i].append('',None,int(0.03*end[i]))
70     i += 1
71     j += 1
72     return chromo
73
74 def addends(chromo):
75     """ Adds a 3% of blank region at both ends for better
76         graphic output.
77     """
78     size = 0
79     for x in chromo:
80         size += x[2]
81     # get 3% of size of each chromosome:
82     endsize = int(float(size)*.03)
83     # add this size to both ends in chromo:
84     chromo.insert(0,('', None, endsize))
85     chromo.append('', None, endsize)
86     return chromo
87
88 def load_chrom(chr_name):
89     """ Generate a chromosome with information
90     """
91     cur_chromosome = BasicChromosome.Chromosome(chr_name[0])
92     chr_segment_info = chr_name[1]
93
94     for seg_info_num in range(len(chr_segment_info)):
95         label, color, scale = chr_segment_info[seg_info_num]
96         # make the top and bottom telomeres
97         if seg_info_num == 0:
98             cur_segment = BasicChromosome.TelomereSegment()
99         elif seg_info_num == len(chr_segment_info) - 1:
100             cur_segment = BasicChromosome.TelomereSegment(1)
101         ## otherwise, they are just regular segments
102         else:
103             cur_segment = BasicChromosome.ChromosomeSegment()
104         if label != "":
105             cur_segment.label = label
106             cur_segment.label_size = 12

```

```

107         if color is not None:
108             cur_segment.fill_color = color
109             cur_segment.scale = scale
110             cur_chromosome.add(cur_segment)
111
112     cur_chromosome.scale_num = max(END) + (max(END)*.04)
113     return cur_chromosome
114
115 def dblookup(atgids):
116     """ Code to retrieve all marker data from name using MongoDB
117     """
118     client = MongoClient(CONNECTION_STRING)
119     db = client.pr4
120     collection = db.markers_map4
121     markers = []
122     for marker in atgids:
123         mrk = collection.find_one({ "marker_id": marker})
124         if mrk:
125             markers.append((marker, (mrk['chromosome'],
126                                     mrk['start'], mrk['end'])))
127         else:
128             print('Marker {0} is not in the DB'.format(marker))
129     return markers
130
131 # Size of each chromosome:
132 END = (30427563, 19696817, 23467989, 18581571, 26986107)
133 gids = []
134 rx_rid = re.compile('^AT[1-5]G\d{5}$')
135 print('Enter AT ID or press 'enter' to stop entering IDs.
136 Valid IDs:
137 AT2G28000
138 AT3G03020
139
140 Also you can enter DBDEMO to use predefined set of markers
141 fetched from a MongoDB database. Enter NODBDEMO to use a
142 predefined set of markers without database access.'')
143 while True:
144     rid = input('Enter Gene ID: ')
145     if not rid:
146         break
147     if rid=='DBDEMO':
148         gids = ['AT3G14890','AT1G66160','AT3G55260','AT5G59570',
149               'AT4G32551','AT1G01430','AT4G26000','AT2G28000',
150               'AT5G21090','AT5G10470']

```



```

151         break
152     elif rid=='NODBDEMO':
153         samplemarkers=[('AT3G14890', (3, 5008749, 5013275)),
154                         ('AT1G66160', (1, 24640827, 24642411)),
155                         ('AT3G55260', (3, 20500225, 20504056)),
156                         ('AT1G10960', (1, 3664385, 3665040)),
157                         ('AT5G23350', (5, 7857646, 7859280)),
158                         ('AT5G15250', (5, 4950414, 4952780)),
159                         ('AT1G55700', (1, 20825263, 20827306)),
160                         ('AT5G21090', (5, 7164583, 7167257)),
161                         ('AT5G10470', (5, 3289228, 3297249)),
162                         ('AT2G28000', (2, 11933524, 11936523)),
163                         ('AT3G03020', (3, 680920, 682009)),
164                         ('AT4G26000', (4, 13197255, 13199845)),
165                         ('AT4G32551', (4, 15707516, 15713587))]
166         break
167     if rx_rid.match(rid):
168         gids.append(rid)
169     else:
170         print("Bad format, please enter it again")
171
172 if rid!='NODBDEMO':
173     samplemarkers = dblookup(gids)
174
175 crms = [[] for r in range(len(END))]
176 for x in samplemarkers:
177     crms[int(x[1][0])-1].append((x[0], x[1][1], x[1][2]))
178
179 crms_o = sortmarkers(crms, END)
180 chromo = getchromo(crms_o, END)
181 all_chr_info = [("I", chromo[0]), ("II", chromo[1]),
182                ("III", chromo[2]), ("IV", chromo[3]),
183                ("V", chromo[4])]
184
185 pdf_organism = BasicChromosome.Organism()
186 for chr_info in all_chr_info:
187     newcrom = (chr_info[0], addends(chr_info[1]))
188     pdf_organism.add(load_chrom(newcrom))
189
190 pdf_organism.draw('at.pdf', 'Arabidopsis thaliana')

```

IV

Appendices



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Collaborative Development: Version Control with GitHub

CONTENTS

A.1	Introduction to version control	366
A.2	Version your code	367
A.3	Share your code	375
A.4	Contribute to other projects	381
A.5	Conclusion	382
A.6	Methods	384
A.7	Additional Resources	384

Introduction to the Appendix

While programs usually start as a single file handled by a single developer, sometimes these grow to include tens or hundreds of files shared by many people working in different places in different time zones. Without some software to handle this workflow, more than one developer might work on the same portion of code or using an outdated version. There is also the case where there is a single programmer who may want to work from different locations (like home and work) and keep track of different versions, without moving files from one location to another. This kind of problem can apply to any text file, not only computer code. So solutions found on this appendix can apply in any document.¹

In the mid-1980s, a professor of Vrije University (Amsterdam), created *Concurrent Versions System (CVS)* while he was working in a C compiler with two students. They faced the type of problem described above, because they were not working together since they all had different schedules. One of the reasons for its popularity, apart from being the first program of its kind, is that it was adopted by high-profile projects such as the development of the Linux kernel. Fast forward to 2017, a new generation of “version control software” as they are generically known

¹There are several services for collaborative editing of text documents like Overleaf (<https://www.overleaf.com>), ShareLatex (<https://www.sharelatex.com/>), Google Docs (<https://docs.google.com/>).

are being used. Today most used program of its kind is *Git*, also created by the author of the Linux kernel, Linus Torvalds.

In this appendix we reprint a paper called “A Quick Introduction to Version Control with Git and GitHub.”

Citation: Blischak JD, Davenport ER, and Wilson G (2016). A Quick Introduction to Version Control with Git and GitHub. *PLoS Comput Biol* 12(1): e1004668. doi:10.1371/journal.pcbi.1004668.

A.1 INTRODUCTION TO VERSION CONTROL

Many scientists write code as part of their research. Just as experiments are logged in laboratory notebooks, it is important to document the code you use for analysis. However, a few key problems can arise when iteratively developing code that make it difficult to document and track which code version was used to create each result. First, you often need to experiment with new ideas, such as adding new features to a script or increasing the speed of a slow step, but you do not want to risk breaking the currently working code. One often utilized solution is to make a copy of the script before making new edits. However, this can quickly become a problem because it clutters your file system with uninformative filenames, e.g. `analysis.sh`, `analysis_02.sh`, `analysis_03.sh`, etc. It is difficult to remember the differences between the versions of the files, and more importantly which version you used to produce specific results, especially if you return to the code months later. Second, you will likely share your code with multiple lab mates or collaborators and they may have suggestions on how to improve it. If you email the code to multiple people, you will have to manually incorporate all the changes each of them sends.

Fortunately, software engineers have already developed software to manage these issues: version control. A version control system (VCS) allows you to track the iterative changes you make to your code. Thus you can experiment with new ideas but always have the option to revert to a specific past version of the code you used to generate particular results. Furthermore, you can record messages as you save each successive version so that you (or anyone else) reviewing the development history of the code is able to understand the rationale for the given edits. Also, it facilitates collaboration. Using a VCS, your collaborators can make and save changes to the code, and you can automatically incorporate these changes to the main code base. The collaborative aspect is enhanced with the emergence of websites that host version controlled code.

In this quick guide, we introduce you to one VCS, Git (<https://git-scm.com/>), and one online hosting site, GitHub (<https://github.com>), both of which are currently popular among scientists and programmers in general. More importantly, we hope to convince you that although mastering a given VCS takes time, you can already achieve great benefits by getting started using a few simple commands. Furthermore, not only does using a VCS solve many common problems when writing code, it can also improve the scientific process. By tracking your code development

Table A.1 Resources

Distributed VCS	Git (https://git-scm.com)
	Mercurial (https://www.mercurial-scm.org/)
	Bazaar (http://bazaar.canonical.com/)
Hosting Site	Github (https://github.com)
	Bitbucket (https://bitbucket.org)
	Gitlab (https://about.gitlab.com/about/)
	Source Forge (https://sourceforge.net/)
Git installation	https://git-scm.com/downloads
Git Tutorials	Software Carpentry (https://swcarpentry.github.io/git-novice/)
	Pro Git (https://git-scm.com/book/)
	A Visual Git Reference (https://marklodato.github.io/visual-git-guide/)
	tryGit (https://try.github.io)
Graphical User Interface for Git	https://git-scm.com/downloads/guis

with a VCS and hosting it online, you are performing science that is more transparent, reproducible, and open to collaboration.²³ There is no reason this framework needs to be limited only to code; a VCS is well-suited for tracking any plain-text files: manuscripts, electronic lab notebooks, protocols, etc.

A.2 VERSION YOUR CODE

The first step is to learn how to version your own code. In this tutorial, we will run Git from the command line of the Unix shell. Thus we expect readers are already comfortable with navigating a file system and running basic commands in such an environment. You can find directions for installing Git for the operating system running on your computer by following one of the links provided in Table A.1. There are many graphical user interfaces (GUIs) available for running Git [Table A.1], which we encourage you to explore, but learning to use Git on the command line is necessary for performing more advanced operations and using Git on a remote machine.

To follow along, first create a folder in your home directory named **thesis**. Next download the three files provided in Supporting Information and place them in the **thesis** directory. Imagine that as part of your thesis you are studying the transcription factor CTCF, and you want to identify high-confidence binding sites in kidney epithelial cells. To do this, you will utilize publicly available ChIP-seq

²³Ram K. Git can facilitate greater reproducibility and increased transparency in science. *Source Code Biol Med.* 2013 Feb;8:7. doi: 10.1186/1751-0473-8-7. pmid:23448176.

³Wilson G, Aruliah D, Brown C, Chue HN, Davis M, Guy R, et al. Best practices for scientific computing. *PLoS Biol.* 2014 Jan;12:e1001745. doi: 10.1371/journal.pbio.1001745. pmid:24415924.

data produced by the ENCODE consortium.⁴ ChIP-seq is a method for finding the sites in the genome where a transcription factor is bound, and these sites are referred to as peaks.⁵ `process.sh` downloads the ENCODE CTCF ChIP-seq data from multiple types of kidney samples and calls peaks (`S1_Data`), `clean.py` filters peaks with a fold change cutoff and merges peaks from the different kidney samples (`S2_Data`), and `analyze.R` creates diagnostic plots on the length of the peaks and their distribution across the genome (`S3_Data`).

If you have just installed Git, the first thing you need to do is provide some information about yourself, since it records who makes each change to the file(s). Set your name and email by running the following lines, but replacing “First Last” and “user@domain” with your full name and email address, respectively.

```
$ git config --global user.name "First Last"
$ git config --global user.email "user@domain"
```

To start versioning your code with Git, navigate to your newly created directory, `~/thesis`. Run the command `git init` to initialize the current folder as a Git repository [Figures A.1, A.2A]. A repository (or repo, for short) refers to the current version of the tracked files as well as all the previously saved versions (Box 1). Only files that are located within this directory (and any subdirectories) have the potential to be version controlled, i.e. Git ignores all files outside of the initialized directory. For this reason, projects under version control tend to be stored within a single directory to correspond with a single Git repository. For strategies on how to best organize your own projects, see Noble, 2009.⁶

```
$ cd ~/thesis
$ ls
analyze.R clean.py process.sh
$ git init
Initialized empty Git repository in ~/thesis/.git/
```

Box 1: Definitions.

- **Version Control System (VCS):** (*noun*) a program that tracks changes to specified files over time and maintains a library of all past versions of those files

⁴ENCODE Project Consortium, Bernstein B, Birney E, Dunham I, Green E, Gunter C, et al. An integrated encyclopedia of DNA elements in the human genome. *Nature*. 2012 Sep;489:57–74. doi: 10.1038/nature11247. pmid:22955616.

⁵Bailey T, Krajewski P, Ladunga I, Lefebvre C, Li Q, Liu T, et al. Practical guidelines for the comprehensive analysis of ChIP-seq data. *PLoS Comput Biol*. 2013 null;9:e1003326. doi: 10.1371/journal.pcbi.1003326. pmid:24244136

⁶Noble W. A quick guide to organizing computational biology projects. *PLoS Comput Biol*. 2009 Jul;5:e1000424. doi: 10.1371/journal.pcbi.1000424. pmid:19649301.

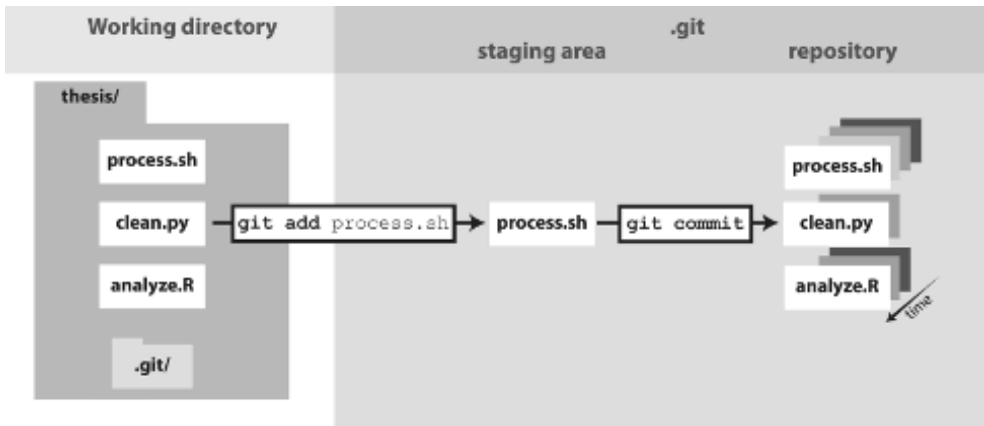


Figure A.1 The git add/commit process: To store a snapshot of changes in your repository, first `git add` any files to the staging area you wish to commit (for example, you've updated the `process.sh` file). Second, type `git commit` with a message. Only files added to the staging area will be committed. All past commits are located in the hidden `.git` directory in your repository.

- **Git:** (*noun*) a version control system
- **repository (repo):** (*noun*) folder containing all tracked files as well as the version control history
- **commit:** (*noun*) a snapshot of changes made to the staged file(s); (*verb*) to save a snapshot of changes made to the staged file(s)
- **stage:** (*noun*) the staging area holds the files to be included in the next commit; (*verb*) to mark a file to be included in the next commit
- **track:** (*noun*) a tracked file is one that is recognized by the Git repository
- **branch:** (*noun*) a parallel version of the files in a repository ([Box 7](#))
- **local:** (*noun*) the version of your repository that is stored on your personal computer
- **remote:** (*noun*) the version of your repository that is stored on a remote server, for instance on GitHub
- **clone:** (*verb*) to create a local copy of a remote repository on your personal computer
- **fork:** (*noun*) a copy of another user's repository on GitHub; (*verb*) to copy a repository, for instance from one user's GitHub account to your own
- **merge:** (*verb*) to update files by incorporating the changes introduced in new commits

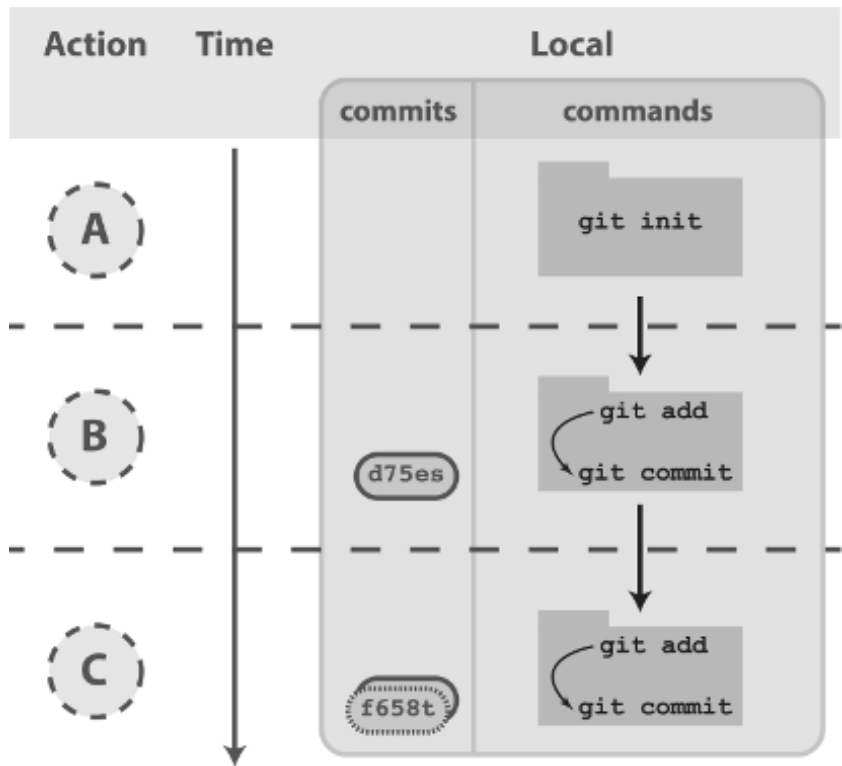


Figure A.2 Working with a local repository: (A) To designate a directory on your computer as a Git repo, type the command `git init`. This initializes the repository and will allow you to track the files located within that directory. (B) Once you have added a file, follow the `git add/commit` cycle to place the new file first into the staging area by typing `git add` to designate it to be committed, and then `git commit` to take the snapshot of that file. The commit is assigned a commit identifier (`d75es`) that can be used in the future to pull up this version or to compare different committed versions of this file. (C) As you continue to add and change files, you should regularly add and commit those changes. Here, an additional commit was done, and the commit log now shows two commit identifiers: `d75es` (from step B) and `f658t` (the new commit). Each commit will generate a unique identifier, which can be examined in reverse chronological order using `git log`.

- **pull:** (*verb*) to retrieve commits from a remote repository and merge them into a local repository
- **push:** (*verb*) to send commits from a local repository to a remote repository
- **pull request:** (*noun*) a message sent by one GitHub user to merge the commits in their remote repository into another user's remote repository

Now you are ready to start versioning your code [Figure A.1]. Conceptually, Git saves snapshots of the changes you make to your files whenever you instruct it to. For instance, after you edit a script in your text editor, you save the updated script to your `thesis` folder. If you tell Git to save a snapshot of the updated document, then you will have a permanent record of the file in that exact version even if you make subsequent edits to the file. In the Git framework, any changes you have made to a script, but have not yet recorded as a snapshot with Git, reside in the working directory only [Figure A.1]. To follow what Git is doing as you record the initial version of your files, use the informative command `git status`.

```
$ git status
On branch master
```

```
Initial commit
```

```
Untracked files:
```

```
(use "git add <file>..." to include in what will be committed)
```

```
analyze.R
clean.py
process.sh
```

```
nothing added to commit but untracked files present (use "git add" <=
to track)
```

There are a few key things to notice from this output. First, the three scripts are recognized as untracked files because you have not told Git to start tracking anything yet. Second, the word “commit” is Git terminology for snapshot. As a noun it means “a version of the code,” e.g. “the figure was generated using the commit from yesterday” (Box 1). This word can also be used as a verb, in which case it means “to save,” e.g. “to commit a change.” Lastly, the output explains how you can track your files using `git add`. Start tracking the file `process.sh`.

```
$ git add process.sh
```

And check its new status.

```
$ git status
On branch master
```

```
Initial commit
```

```
Changes to be committed:
```

```
(use "git rm --cached <file>..." to unstage)
```

```
new file:    process.sh
```

Untracked files:

```
(use "git add <file>..." to include in what will be committed)
```

```
analyze.R
clean.py
```

Since this is the first time that you have told Git about the file `process.sh`, two key things have happened. First, this file is now being tracked, which means Git recognizes it as a file you wish to be version controlled ([Box 1](#)). Second, the changes made to the file (in this case the entire file because it is the first commit) have been added to the staging area [[Figure A.1](#)]. Adding a file to the staging area will result in the changes to that file being included in the next commit, or snapshot of the code ([Box 1](#)). As an analogy, adding files to the staging area is like putting things in a box to mail off, and committing is like putting the box in the mail.

Since this will be the first commit, or first version of the code, use `git add` to begin tracking the other two files and add their changes to the staging area as well. Then create the first commit using the command `git commit`.

```
$ git add clean.py analyze.R
$ git commit -m "Add initial version of thesis code."
[master (root-commit) 660213b] Add initial version of thesis code.
3 files changed, 154 insertions(+)
create mode 100644 analyze.R
create mode 100644 clean.py
create mode 100644 process.sh
```

Notice the flag `-m` was used to pass a message for the commit. This message describes the changes that have been made to the code and is required. If you do not pass a message at the command line, the default text editor for your system will open so you can enter the message. You have just performed the typical development cycle with Git: make some changes, add updated files to the staging area, and commit the changes as a snapshot once you are satisfied with them [[Figure A.2](#)].

Since Git records all of the commits, you can always look through the complete history of a project. To view the record of your commits, use the command `git log`. For each commit, it lists the unique identifier for that revision, author, date, and commit message.

```
$ git log
commit 660213b91af167d992885e45ab19f585f02d4661
Author: First Last <user@domain>
Date:   Fri Aug 21 14:52:05 2015 -0500
```

Add initial version of thesis code.

The commit identifier can be used to compare two different versions of a file, restore a file to a previous version from a past commit, and even retrieve tracked files if you accidentally delete them.

Now you are free to make changes to the files knowing that you can always revert them to the state of this commit by referencing its identifier. As an example, edit `clean.py` so that the fold change cutoff for filtering peaks is more stringent. Here is the current bottom of the file.

```
$ tail clean.py
# Filter based on fold-change over control sample
fc_cutoff = 10
epithelial = epithelial.filter(filter_fold_change,
                                fc = fc_cutoff).saveas()
proximal_tube = proximal_tube.filter(filter_fold_change,
                                       fc = fc_cutoff).saveas()
kidney = kidney.filter(filter_fold_change,
                        fc = fc_cutoff).saveas()
# Identify only those sites that are peaks in all three tissue types
combined = pybedtools.BedTool().multi_intersect(
    i = [epithelial.fn, proximal_tube.fn, kidney.fn])
union = combined.filter(lambda x: int(x[3]) == 3).saveas()
union.cut(range(3)).saveas(data + "/sites-union.bed")
```

Using a text editor, increase the fold change cutoff from 10 to 20.

```
$ tail clean.py
# Filter based on fold-change over control sample
fc_cutoff = 20
epithelial = epithelial.filter(filter_fold_change,
                                fc = fc_cutoff).saveas()
proximal_tube = proximal_tube.filter(filter_fold_change,
                                       fc = fc_cutoff).saveas()
kidney = kidney.filter(filter_fold_change,
                        fc = fc_cutoff).saveas()
# Identify only those sites that are peaks in all three tissue types
combined = pybedtools.BedTool().multi_intersect(
    i = [epithelial.fn, proximal_tube.fn, kidney.fn])
union = combined.filter(lambda x: int(x[3]) == 3).saveas()
union.cut(range(3)).saveas(data + "/sites-union.bed")
```

Because Git is tracking `clean.py`, it recognizes that the file has been changed since the last commit.

```
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working<=
#   directory)
#
#       modified:   clean.py
#
no changes added to commit (use "git add" and/or "git commit -a")
```

The report from `git status` indicates that the changes to `clean.py` are not staged, i.e. they are in the working directory [Figure A.1]. To view the unstaged changes, run the command `git diff`.

```
$ git diff
diff --git a/clean.py b/clean.py
index 7b8c058..76d84ce 100644
--- a/clean.py
+++ b/clean.py
@@ -28,7 +28,7 @@ def filter_fold_change(feature, fc = 1):
     return False

# Filter based on fold-change over control sample
-fc_cutoff = 10
+fc_cutoff = 20
epithelial = epithelial.filter(filter_fold_change,
                                fc = fc_cutoff).saveas()
proximal_tube = proximal_tube.filter(filter_fold_change,
                                       fc = fc_cutoff).saveas()
kidney = kidney.filter(filter_fold_change,
                        fc = fc_cutoff).saveas()
```

Any lines of text that have been added to the script are indicated with a + and any lines that have been removed with a -. Here, we altered the line of code which sets the value of `fc_cutoff`. `git diff` displays this change as the previous line being removed and a new line being added with our update incorporated. You can ignore the first five lines of output because they are directions for other software programs that can merge changes to files. If you wanted to keep this edit, you could add `clean.py` to the staging area using `git add` and then commit the change using `git commit`, as you did above. Instead, this time undo the edit by following the directions from the output of `git status` to “discard changes in the working directory” using the command `git checkout`.

```
$ git checkout -- clean.py
```

```
$ git diff
```

Now `git diff` returns no output because `git checkout` undid the unstaged edit you had made to `clean.py`. And this ability to undo past edits to a file is not limited to unstaged changes in the working directory. If you had committed multiple changes to the file `clean.py` and then decided you wanted the original version from the initial commit, you could replace the argument `--` with the commit identifier of the first commit you made above (your commit identifier will be different; use `git log` to find it). The `--` used above was simply a placeholder for the first argument because by default `git checkout` restores the most recent version of the file from the staging area (if you haven't staged any changes to this file, as is the case here, the version of the file in the staging area is identical to the version in the last commit). Instead of using the entire commit identifier, use only the first seven characters, which is simply a convention since this is usually long enough for it to be unique.

```
$ git checkout 660213b clean.py
```

At this point, you have learned the commands needed to version your code with Git. Thus you already have the benefits of being able to make edits to files without copying them first, to create a record of your changes with accompanying messages, and to revert to previous versions of the files if needed. Now you will always be able to recreate past results that were generated with previous versions of the code (see the command `git tag` for a method to facilitate finding specific past versions) and see the exact changes you have made over the course of a project.

A.3 SHARE YOUR CODE

Once you have your files saved in a Git repository, you can share it with your collaborators and the wider scientific community by putting your code online [Figure A.3]. This also has the added benefit of creating a backup of your scripts and provides a mechanism for transferring your files across multiple computers. Sharing a repository is made easier if you use one of the many online services that host Git repositories [Figure A.1], e.g. GitHub. Note, however, that any files that have not been tracked with at least one commit are not included in the Git repository, even if they are located within the same directory on your local computer (see Box 2 for advice on the types of files that should not be versioned with Git and Box 3 for advice on managing large files).

Box 2: What *not* to version control. You *can* version control any file that you put in a Git repository, whether it is text-based, an image, or giant data files. However, just because you *can* version control something, does not mean you *should*. Git works best for plain text based documents such as your scripts or your manuscript if written in LaTeX or Markdown. This is because for text files, Git saves the entire

file only the first time you commit it and then saves just your changes with each commit. This takes up very little space and Git has the capability to compare between versions (using `git diff`). You can commit a non-text file, but a full copy of the file will be saved in each commit that modifies it. Over time, you may find the size of your repository growing very quickly. A good rule of thumb is to version control anything text based: your scripts or manuscripts if they are written in plain text. Things *not* to version control are large data files that never change, binary files (including Word and Excel documents), and the output of your code.

In addition to the type of file, you need to consider the content of the file. If you plan on sharing your commits publicly using GitHub, ensure you are not committing any files that contain sensitive information, such as human subject data or passwords.

To prevent accidentally committing files you do not wish to track, and to remove them from the output of `git status`, you can create a file called `.gitignore`. In this file, you can list subdirectories and/or file patterns that Git should ignore. For example, if your code produced log files with the file extension `.log`, you could instruct Git to ignore these files by adding `*.log` to `.gitignore`. In order for these settings to be applied to all instances of the repository, e.g. if you clone it onto another computer, you need to add and commit this file.

Box 3: Managing large files Many biological applications require handling large data files. While Git is best-suited for collaboratively writing small text files, nonetheless collaboratively working on projects in the biological sciences necessitates managing this data.

The example analysis pipeline in this tutorial starts by downloading data files in BAM format which contain the alignments of short reads from a ChIP-seq experiment to the human genome. Since these large, binary files are not going to change, there is no reason to version them with Git. Thus hosting them on a remote http (as ENCODE has done in this case) or ftp site allows each collaborator to download it to her machine as needed, e.g., using `wget`, `curl`, or `rsync`. If the data files for your project are smaller, you could also share them via services like Dropbox (www.dropbox.com) or Google Drive (<https://www.google.com/drive/>).

However, some intermediate data files may change over time, and the practical necessity to ensure all collaborators are using the same data set may override the advice to *not* put code output under version control, as described in [Box 2](#). Again returning to the ChIP-seq example, the first step calling the peaks is the most difficult computationally because it requires access to a Unix-like environment and sufficient computational resources. Thus for collaborators that want to experiment with `clean.py` and `analyze.R` without having to run `process.sh`, you could version the data files containing the ChIP-seq peaks (which are in BED format). But since these files are larger than that typically used with Git, you can instead use one

of the solutions for versioning large files within a Git repository without actually saving the file with Git, e.g. git-annex (<https://git-annex.branchable.com/>) or git-fat (<https://github.com/jedbrown/git-fat/>). Recently GitHub has created their own solution for managing large files called Git Large File Storage (LFS) (<https://git-lfs.github.com/>). Instead of committing the entire large file to Git, which quickly becomes unmanageable, it commits a text pointer. This text pointer refers to a specific file saved on a remote GitHub server. Thus when you clone a repository, it only downloads the latest version of the large file. And if you check out an older version of the repository, it automatically downloads the old version of the large file from the remote server. After installing Git LFS, you can manage all the BED files with one command: `git lfs track "*.bed"`. Then you can commit the BED files just like your scripts, and they will automatically be handled with Git LFS. Now if you were to change the parameters of the peak calling algorithm and re-run `process.sh`, you could commit the updated BED files and your collaborators could pull the new versions of the files directly to their local Git repositories.

Below we focus on the technical aspects of sharing your code. However, there are also other issues to consider when deciding if and how you are going to make your code available to others. For quick advice on these subjects, see [Box 4](#) on how to license your code, [Box 5](#) on concerns about being scooped, and [Box 6](#) on the increasing trend of journals to institute sharing policies that require authors to deposit code in a public archive upon publication.

Box 4: Choosing a license Putting software and other material in a public place is not the same as making it publicly usable. In order to do that, the authors must also add a license, since copyright laws in some jurisdictions require people to treat anything that isn't explicitly open as being proprietary.

While dozens of open licenses have been created, the two most widely used are the GNU Public License (GPL) and the MIT/BSD family of licenses. Of these, the MIT/BSD-style licenses put the fewest requirements on re-use, and thereby make it easier for people to integrate *your* software into *their* project.

For an excellent short discussion of these issues, and links to more information, see Jake Vanderplas's blog post from March 2014 at <http://www.astrobetter.com/blog/2014/03/10/the-whys-and-hows-of-licensing-scientific-code/>.

For a more in-depth discussion of the legal implications of different licenses, see Morin et al., 2012.⁷

⁷Morin A, Urban J, Sliz P. A quick guide to software licensing for the scientist-programmer. *PLoS Comput Biol*. 2012 null;8:e1002598. doi: 10.1371/journal.pcbi.1002598. pmid:22844236.

Box 5: Being Scooped One concern scientists frequently have about putting work in progress online is that they will be scooped, e.g., that someone will analyze their data and publish a result that they themselves would have, but hadn't yet. In practice, though, this happens rarely if at all: in fact, the authors are not aware of a single case in which this has actually happened, and would welcome pointers to specific instances. In practice, it seems more likely that making work public early in something like a version control repository, which automatically adds timestamps to content, will help researchers establish their priority.

Box 6: Journal Policies Sharing data, code, and other materials is quickly moving from “desired” to “required.” For example, PLOS’s sharing policy (<http://journals.plos.org/plosone/s/materials-and-software-sharing>) already says, “We expect that all researchers submitting to PLOS will make all relevant materials that may be reasonably requested by others available without restrictions upon publication of the work.” Its policy on software is more specific:

We expect that all researchers submitting to PLOS submissions in which software is the central part of the manuscript will make all relevant software available without restrictions upon publication of the work. Authors must ensure that software remains usable over time regardless of versions or upgrades. . .

It then goes on to specify that software must be based on open source standards, and that it must be put in an archive which is large or long-lived. Granting agencies, philanthropic foundations, and other major sponsors of scientific research are all moving in the same direction, and to our knowledge, none has relaxed or reduced sharing requirements in the last decade.

To begin using GitHub, you will first need to sign up for an account. For the code examples in this tutorial, you will need to replace `username` with the username of your account. Next choose the option to “Create a new repository” (Fig. A.3B, see <https://help.github.com/articles/create-a-repo/>). Call it “thesis” because that is the directory name containing the files on your computer, but note that you can give it a different name on GitHub if you wish. Also, now that the code will be existing in multiple places, you need to learn some more terminology (Box 1). A local repository refers to code that is stored on the machine you are using, e.g. your laptop; whereas, a remote repository refers to the code that is hosted online. Thus, you have just created a remote repository.

Now you need to send the code on your computer to GitHub. The key to this is the URL that GitHub assigns your newly created remote repository. It will have the form <https://github.com/username/thesis.git> (see <https://help.github.com/articles/cloning-a-repository/>). Notice that this URL is using

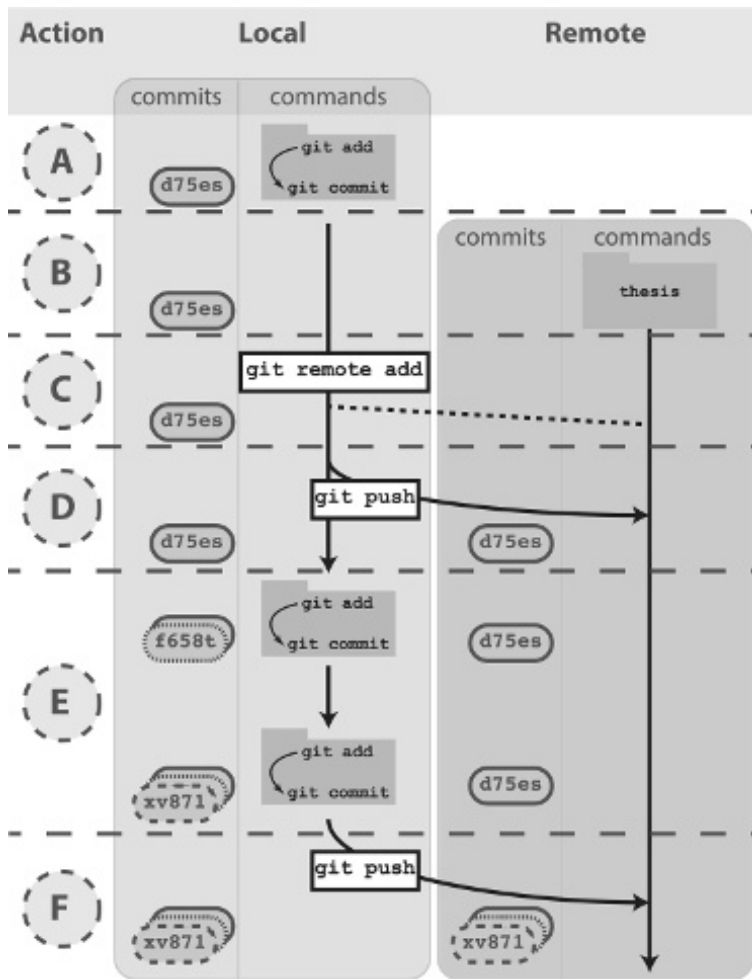


Figure A.3 Working with both a local and remote repository as a single user: (A) On your computer, you commit to a Git repository (commit d75es). (B) On GitHub, you create a new repository called thesis. This repository is currently empty and not linked to the repo on your local machine. (C) The command `git remote add` connects your local repository to your remote repository. The remote repository is still empty, however, because you have not pushed any content to it. (D) You send all the local commits to the remote repository using the command `git push`. Only files that have been committed will appear in the remote repository. (E) You repeat several more rounds of updating scripts and committing on your local computer (commit f658t and then commit xv871). You have not yet pushed these commits to the remote repository, so only the previously pushed commit is in the remote repo (commit d75es). (F) To bring the remote repository up to date with your local repository, you `git push` the two new commits to the remote repository. The local and remote repositories now contain the same files and commit histories.

the HTTPS protocol, which is the quickest to begin using. However it requires you to enter your username and password when communicating with GitHub, so you'll want to consider switching to the SSH protocol once you are regularly using Git and GitHub (see <https://help.github.com/articles/generating-ssh-keys/> for directions). In order to link the local thesis repository on your computer to the remote repository you just created, in your local repository you need to tell Git the URL of the remote repository using the command `git remote add` [Figure A.3C].

```
$ git remote add origin https://github.com/username/thesis.git
```

The name “origin” is a bookmark for the remote repository so that you do not have to type out the full URL every time you transfer your changes (this is the default name for a remote repository, but you could use another name if you like).

Send your code to GitHub using the command `git push` [Figure A.3D].

```
$ git push origin master
```

You first specify the remote repository, “origin.” Second, you tell Git to push to the “master” copy of the repository we will not go into other options in this tutorial, but Box 7 discusses them briefly.

Box 7: Branching Do you ever make changes to your code, but are not sure you will want to keep those changes for your final analysis? Or do you need to implement new features while still providing a stable version of the code for others to use? Using Git, you can maintain parallel versions of your code that you can easily bounce between while you are working on your changes. You can think of it like making a copy of the folder you keep your scripts in, so that you have your original scripts intact but also have the new folder where you make changes. Using Git, this is called branching and it is better than separate folders because 1) it uses a fraction of the space on your computer, 2) keeps a record of when you made the parallel copy (branch) and what you have done on the branch, and 3) there is a way to incorporate those changes back into your main code if you decide to keep your changes (and a way to deal with conflicts). By default, your repository will start with one branch, usually called “master.” To create a new branch in your repository, type `git branch new_branch_name`. You can see what branches a current repository has by typing `git branch`, with the branch you are currently in being marked by a star. To move between branches, type `git checkout branch_to_move_to`. You can edit files and commit them on each branch separately. If you want combine the changes in your new branch with the master branch, you can merge the branches by typing `git merge new_branch_name` while in the master branch.

Pushing to GitHub also has the added benefit of backing up your code in

case anything were to happen to your computer. Also, it can be used to manually transfer your code across multiple machines, similar to a service like Dropbox (www.dropbox.com), but with the added capabilities and control of Git. For example, what if you wanted to work on your code on your computer at home? You can download the Git repository using the command `git clone`.

```
$ git clone https://github.com/username/thesis.git
```

By default, this will download the Git repository into a local directory named “thesis.” Furthermore, the remote “origin” will automatically be added so that you can easily push your changes back to GitHub. You now have copies of your repository on your work computer, your GitHub account online, and your home computer. You can make changes, commit them on your home computer, and send those commits to the remote repository with `git push`, just as you did on your work computer.

Then the next day back at your work computer, you could update the code with the changes you made the previous evening using the command `git pull`.

```
$ git pull origin master
```

This pulls in all the commits that you had previously pushed to the GitHub remote repository from your home computer. In this workflow, you are essentially collaborating with yourself as you work from multiple computers. If you are working on a project with just one or two other collaborators, you could extend this workflow so that they could edit the code in the same way. You can do this by adding them as Collaborators on your repository (Settings -> Collaborators -> Add collaborator, see <https://help.github.com/articles/adding-collaborators-to-a-personal-repository/>). However, with projects with lots of contributors, GitHub provides a workflow for finer-grained control of the code development.

With the addition of a GitHub account and a few commands for sending and receiving code, you can now share your code with others, transfer your code across multiple machines, and set up simple collaborative workflows.

A.4 CONTRIBUTE TO OTHER PROJECTS

Lots of scientific software is hosted online in Git repositories. Now that you know the basics of Git, you can directly contribute to developing the scientific software you use for your research [Figure A.4]. From a small contribution like fixing a typo in the documentation to a larger change such as fixing a bug, it is empowering to be able to improve the software used by you and many other scientists.

When contributing to a larger project with many contributors, you will not be able to push your changes with `git push` directly to the project’s remote repository. Instead you will first need to create your own remote copy of the repository, which on GitHub is called a fork (Box 1). You can fork any repository on GitHub by clicking

the button “Fork” on the top right of the page (see <https://help.github.com/articles/fork-a-repo/>).

Once you have a fork of a project’s repository, you can clone it to your computer and make changes just like a repository you created yourself. As an exercise, you will add a file to the repository that we used to write this paper. First, go to <https://github.com/jdblischak/git-for-science> and choose the “Fork” option to create a git-for-science repository under your GitHub account [Figure A.4B]. In order to make changes, download it to your computer with the command `git clone` from the directory you wish the repo to appear in [Figure A.4C].

```
$ git clone https://github.com/username/git-for-science.git
```

Now that you have a local version, navigate to the subdirectory `readers` and create a text file named as your GitHub username (Fig. A.4D).

```
$ cd git-for-science/readers
$ touch username.txt
```

Add and commit this new file (Fig. A.4D), and then push the changes back to your remote repository on GitHub (Fig. A.4E).

```
$ git add username.txt
$ git commit -m "Add username to directory of readers."
$ git push origin master
```

Currently, the new file you created, `readers/username.txt`, only exists in your fork of `git-for-science`. To merge this file into the main repository, send a pull request using the GitHub interface (Pull request -> New pull request -> Create pull request; Fig. A.4F; see <https://help.github.com/articles/using-pull-requests>). After the pull request is created, we can review your change and then merge it into the main repository. Although this process of forking a project’s repository and issuing a pull request seems like a lot of work to contribute changes, this workflow gives the owner of a project control over what changes get incorporated into the code. You can have others contribute to your projects using the same workflow.

The ability to use Git to contribute changes is very powerful because it allows you to improve the software that is used by many other scientists and also potentially shape the future direction of its development.

A.5 CONCLUSION

Git, albeit complicated at first, is a powerful tool that can improve code development and documentation. Ultimately the complexity of a VCS not only gives users a well-documented “undo” button for their analyses, but it also allows for collaboration and sharing of code on a massive scale. Furthermore, it does not need

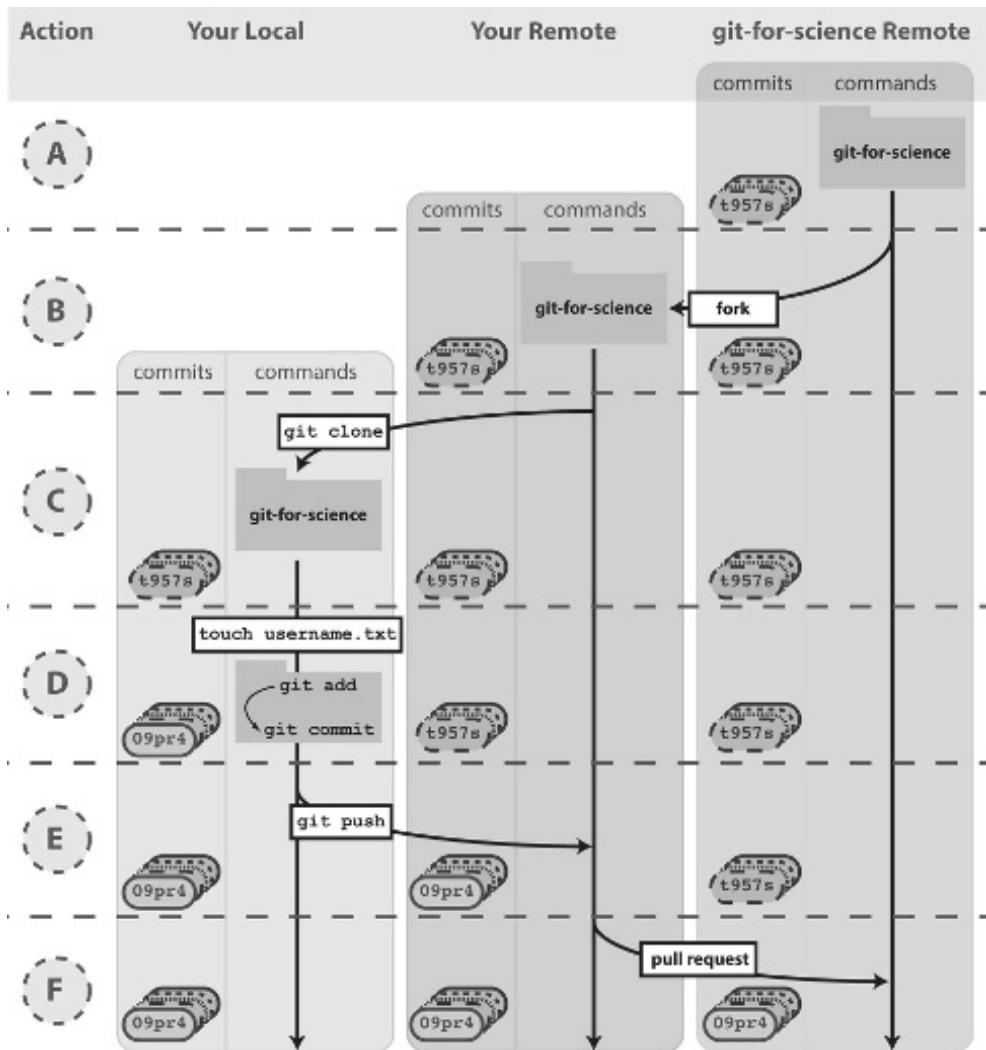


Figure A.4 Contributing to open source projects: (A) Using your internet browser, navigate to <https://github.com/jdblichak/git-for-science>. (B) Click on the *Fork* button to create a copy of this repo on GitHub under your username. (C) On your computer, type `git clone https://github.com/username/git-for-science.git`. (D) Navigate to the readers directory by typing `cd git-for-science/readers/`. Create an empty file that is titled with your GitHub username by typing `touch username.txt`. Commit that new file by adding it to the staging area (`git add username.txt`) and committing with a message (`git commit -m "Add username"`). (E) You have committed your new file locally, and the next step is to push that new commit up to the git-for-science repo under your username on GitHub. To do so, type `git push origin master`. (F) To request to add your commits to the original git-for-science repo, issue a pull request from the git-for-science repo under your username on GitHub. Once your Pull Request is reviewed and accepted, you will be able to see the file you committed in the original git-for-science repository.

to be learned in its entirety to be useful. Instead, you can derive tangible benefits from adopting version control in stages. With a few commands (`git init`, `git add`, `git commit`), you can start tracking your code development and avoid a file system full of copied files [Figure A.2]. Adding a few additional commands (`git push`, `git clone`, `git pull`) and a GitHub account, you can share your code online, transfer your changes across machines, and collaborate in small groups [Figure A.3]. Lastly, by forking public repositories and sending pull requests, you can directly improve scientific software [Figure A.4].

A.6 METHODS

We collaboratively wrote the article in LaTeX (<http://www.latex-project.org/>) using the online authoring platform Authorea (<https://www.authorea.com>). Furthermore, we tracked the development of the document using Git and GitHub. The Git repo is available at <https://github.com/jdblischak/git-for-science>, and the rendered LaTeX article is available at <https://www.authorea.com/users/5990/articles/17489>.

Copyright for this article © 2016 Blischak et al. This is an open access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited

A.7 ADDITIONAL RESOURCES

- Code School GitHub course.
<https://try.github.io/levels/1/challenges/1>
- Hello World, a 10 minutes Git tutorial.
<https://guides.github.com/activities/hello-world/>
- Git cheat sheet.
<http://cheat.errtheblog.com/s/git>
- Improving collaboration with forks.
<https://goo.gl/bUkjQV>
- GVFS: Git Virtual File System.
<https://github.com/Microsoft/GVFS>

Install a Bottle App in PythonAnywhere

CONTENTS

B.1	PythonAnywhere	385
B.1.1	What Is PythonAnywhere	385
B.1.2	Installing a Web App in PythonAnywhere	385

B.1 PYTHONANYWHERE

B.1.1 What Is PythonAnywhere

PythonAnywhere is a web service that allows you to host, run, and code Python in the cloud. It is different from a hosting service because it has some extra services. One service is the “console,” where you can run bash or Python commands. In fact you can choose from Bash, Python, IPython, PyPy, MySQL, and PostgreSQL consoles. You can have a console open in one machine at work, do some work, turn off your machine, go back home and connect to the same console and keep on working from the point you left.

When you enter your account you see a dashboard with the **Consoles** tab already selected (see [Figure B.1](#)).

Another advantage of PythonAnywhere is a “wizard” that guides you to set up a web app. The following section is about how to deploy a Bottle web app using this wizard.

B.1.2 Installing a Web App in PythonAnywhere

To create a web app, go to the “Web” tab and press “Add a new web app” (see [Figure B.2](#)).

This will lead you to the screen seen in [Figure B.3](#) where you can choose between upgrading your account to use a custom domain or keeping the free option with the form `your-account-name.pythonanywhere.com`. Click “Next” if you are OK with the free option.

In [Listing 10.13](#) (page 230) we have a program we could try to install in PythonAnywhere. The program can’t be installed as is; it would required a minor modification.

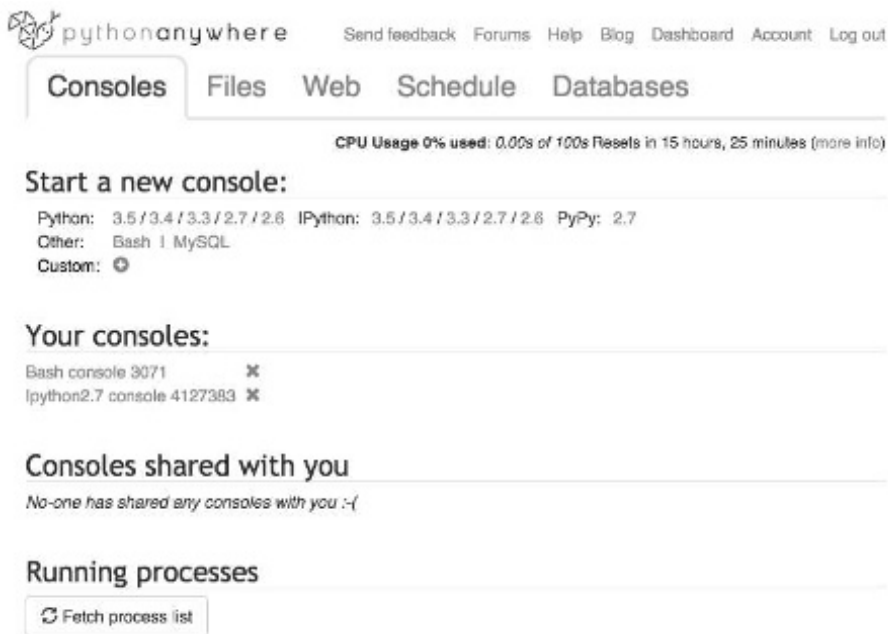


Figure B.1 “Consoles” tab, where you can launch new consoles or resume existing ones.

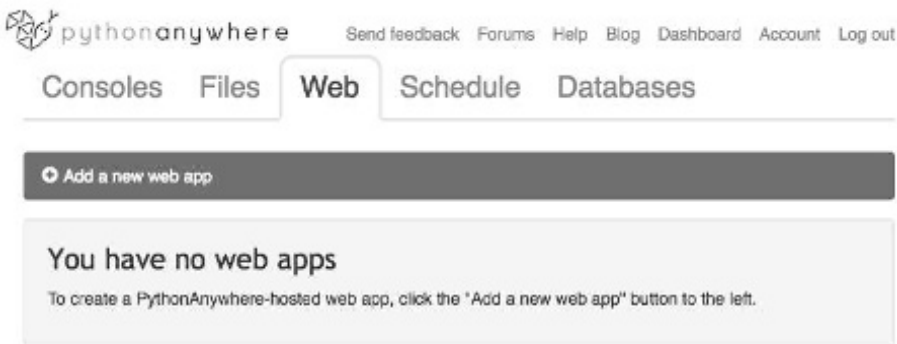


Figure B.2 The “Web” tab in PythonAnywhere, where you can create a web app.

The next step is to select a web framework (Figure B.4). Choose **Bottle**.

Then select a Python version (Figure B.5). Choose 3.5 or better if available.

The next step is to enter the path of the Python file with the app. A sample app will be installed in this file, so enter a non-existent, file name. In this case I entered `/home/sbassi/protcharge/index.py` (Figure B.6). If the directory does not exist, it will be created.

This will lead to a page like the one in Figure B.7. You can try going to your domain using your web browser to see if it is working. The domain is <http://your-account-name.pythonanywhere.com> (in my case is <http://your-account-name.pythonanywhere.com>).

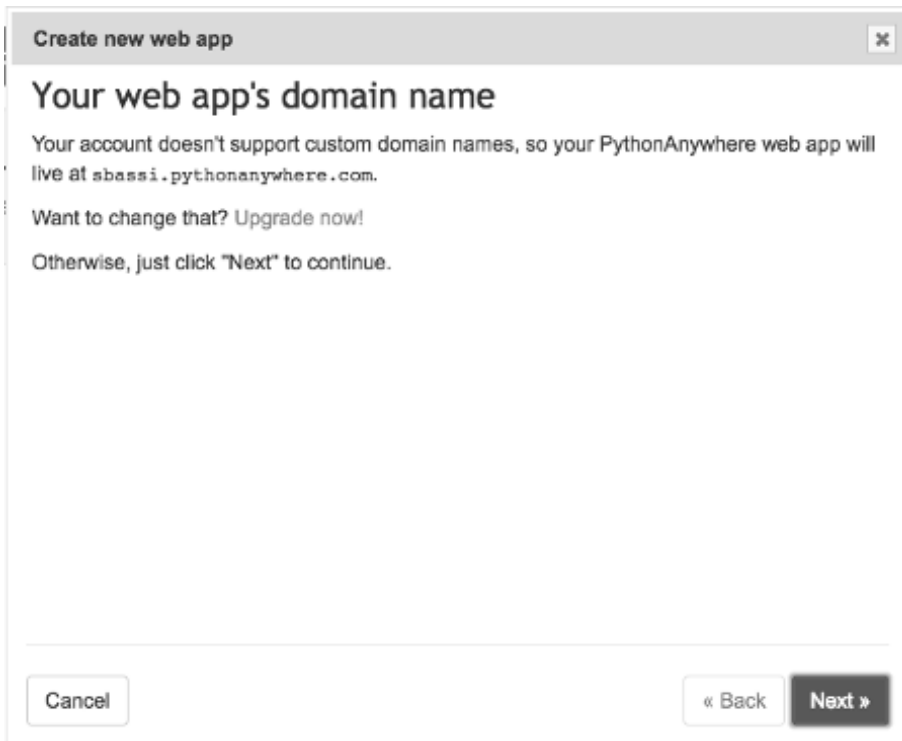


Figure B.3 Screen to opt for upgrading domain type or keeping the free one.

`//sbassi.pythonanywhere.com`). You will see a “Hello from Bottle!” page. The next step is to change the provided sample source file so we can run our web app instead.

In order to change the content of the `index.py` file, go to the “File” tab and click on the file to edit. You should see the contents as in [Figure B.8](#). From that file, you can delete everything except the last line (line 9, `application = default_app()`). In its place, copy the code in [Listing 10.13](#) (from page 230) with two changes: Delete the last line (line 36 in the original, the one that starts with `run`) and add `default_app` in the import line. The resulting `index.py` file can be found in the Github project associated with the book.¹

Once the code is deployed, the templates and auxiliary files must be uploaded. Go to the “Files” tab and create a new directory called `views` (see [Figure B.9](#)).

On this directory, upload the `protchargeformbottle.html` and `result.html` files. This should look like the screenshot in [Figure B.10](#).

Using the same method, create a `css` directory and upload the bootstrap file (`bootstrap.min.css`).

Now the web app is ready to use. Just go to your domain (`http://`

¹The Github project is at <https://github.com/Serulab/Py4Bio> and this file is at <https://github.com/Serulab/Py4Bio/blob/master/code/ch10/index.py>.

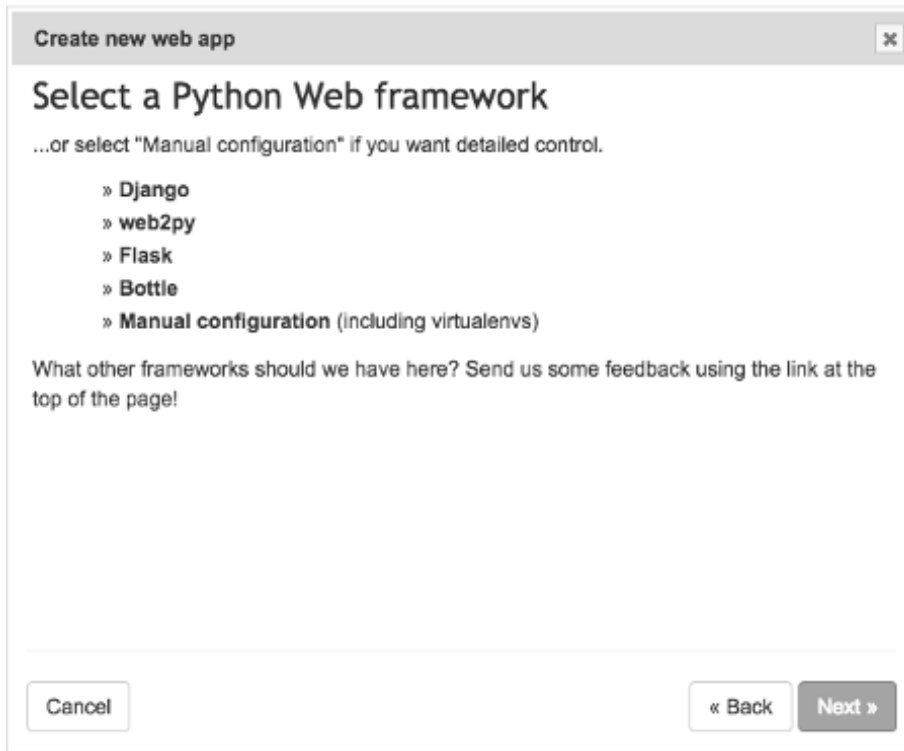


Figure B.4 Select a web framework screen, select Bottle.

`your-account-name.pythonanywhere.com`) and you should see a screen like the one in [Figure B.11](#).

If there is any error, go to the “Web” tab and look for the logs files. They have information that may be useful to debug any issue that may arise.

Create new web app

×

Select a Python version

- » Python 2.7 (Bottle 0.11.6)
- » Python 3.3 (Bottle 0.11.6)
- » Python 3.4 (Bottle 0.12.7)
- » Python 3.5 (Bottle 0.12.9)

Note: If you'd like to use a different version of Bottle to the default version, you can use a virtualenv for your web app. There are [instructions here](#).

Cancel

« Back

Next »

Figure B.5 Select a Python and Bottle version.



Create new web app

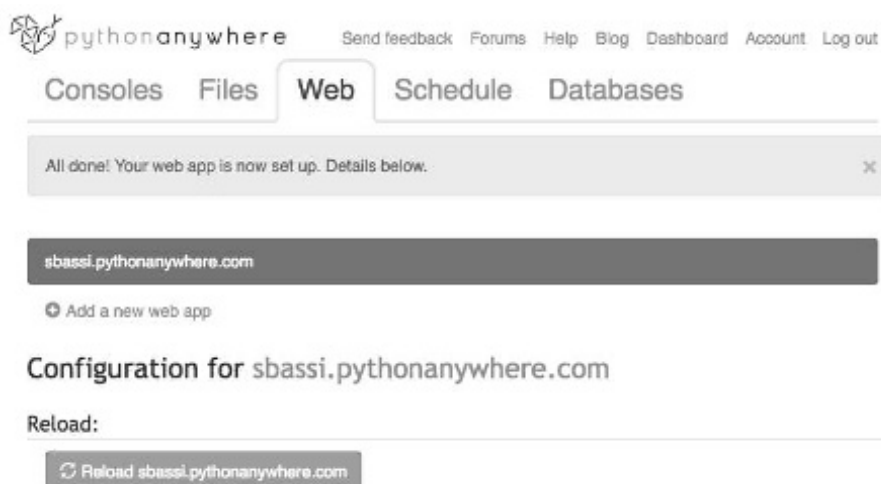
Quickstart new Bottle project

Enter a path for a Python file you wish to use to hold your Bottle app. If this file already exists, its contents will be overwritten with the new app.

Path

Cancel  « Back Next »

Figure B.6 Form to enter the path of the web app.




pythonanywhere Send feedback Forums Help Blog Dashboard Account Log out

Consoles Files **Web** Schedule Databases

All done! Your web app is now set up. Details below.

sbassi.pythonanywhere.com

 Add a new web app

Configuration for sbassi.pythonanywhere.com

Reload:


 Reload sbassi.pythonanywhere.com

Figure B.7 The sample web app is ready to use.



Figure B.8 The “File” tab.

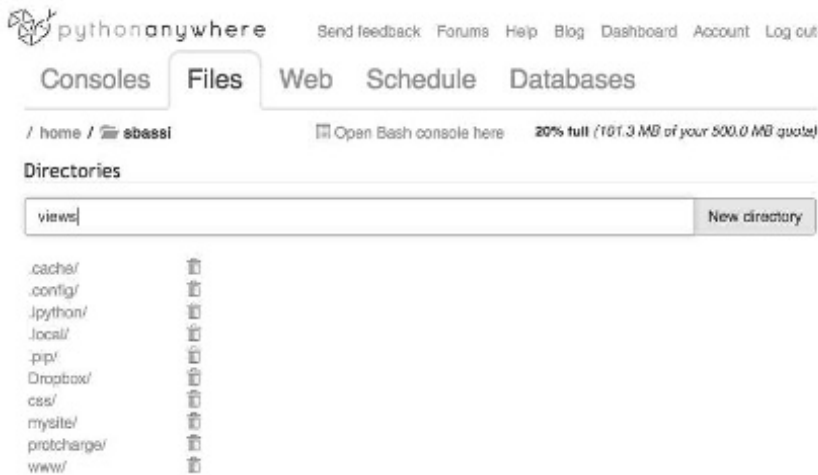


Figure B.9 Form to create a new directory in PythonAnywhere.

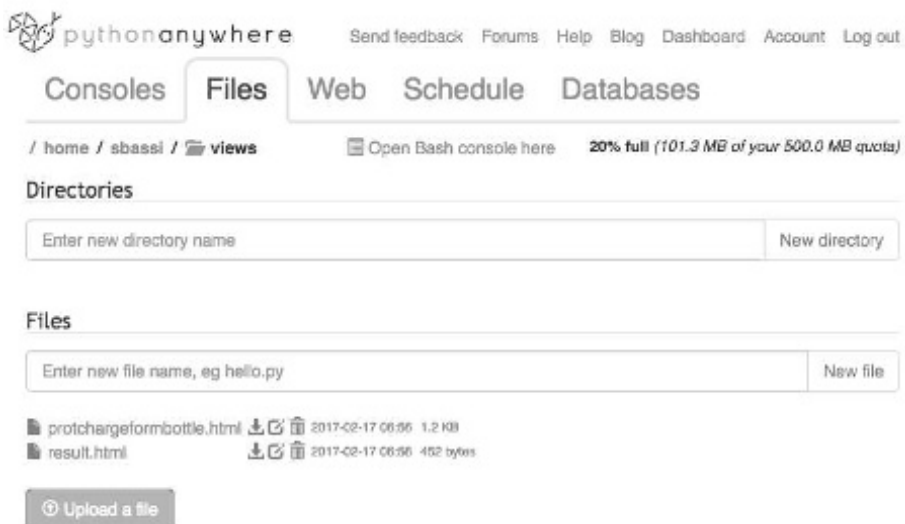
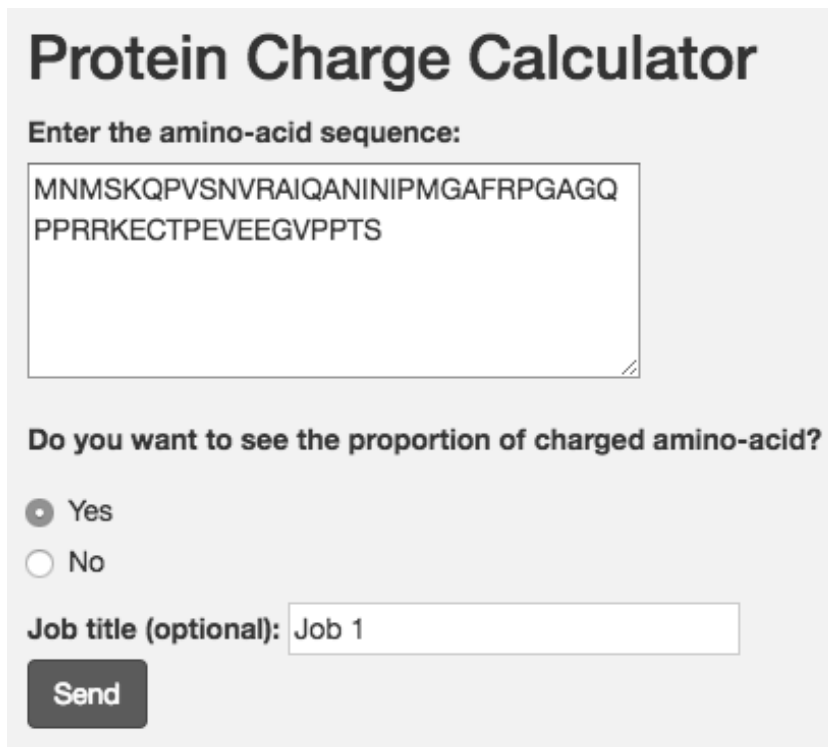


Figure B.10 View and upload files into your account.



Protein Charge Calculator

Enter the amino-acid sequence:

MNMSKQPVSINVRAIQANINIPMGAFRPGAGQ
PPRRKECTPEVEEGVPPTS

Do you want to see the proportion of charged amino-acid?

☒ Yes
☐ No

Job title (optional): Job 1

Send

Figure B.11 Front-end of the program to calculate charge of a protein using Bottle and hosted in PythonAnywhere.

Scientific Python Cheat Sheet

CONTENTS

C.1	Pure Python	394
	Types	394
	Lists	395
	Dictionaries	396
	Sets	396
	Strings	397
	Operators	397
	Control Flow	398
	Functions, Classes, Generators, Decorators	399
C.2	Virtualenv	400
	Create a new virtualenv	400
	Use (activate) a virtualenv	401
	Deactivate a virtualenv	401
	Install a program inside a virtualenv	401
	Get list of programs installed inside a virtualenv	401
	Install a list of programs from a text file	402
C.3	Conda	402
	Create a new Conda environment	402
	Get a list of all installed environments	402
	Use (activate) a Conda environment	402
	Deactivate a Conda environment	403
	Install a program inside a Conda environment	403
	Get list of programs installed inside a Conda environment	403
	Install a list of programs from a text file	403
C.4	IPython	403
	Console	403
	debugger	404
	command line	405
C.5	NumPy	405
	Array initialization	405
	Indexing	406
	Array properties and operations	406
	Boolean arrays	407

	Elementwise operations and math functions	407
	Inner/outer products	408
	Linear algebra/ matrix math	409
	Reading/ writing files	409
	Interpolation, integration, optimization	409
	FFT (Discrete Fourier Transform)	409
	Rounding	410
	Random variables	410
C.6	Matplotlib	410
	Figures and axes	410
	Figures and axes properties	411
	Plotting routines	411
C.7	Scipy	412
	Interpolation	412
	Integration	412
	Linear algebra	413
C.8	Pandas	413
	Data structures	413
	DataFrame	413

This appendix is a reference material. It is based on the *Scientific Python Cheat Sheet*, available at <https://ipgp.github.io/scientific-python-cheat-sheet/>.

Copyright for this article © 2016 Institut de Physique du Globe de Paris. This is an open access article distributed under the terms of the Creative Commons Attribution 4.0 License, Permits almost any use subject to providing credit and license notice.

C.1 PURE PYTHON

This section shows references for basic Python operation. It covers roughly from data typer ([Chapter 2](#)) to flow control ([Chapter 2](#)). There is also a subsection with information on functions and classes, that corresponds with [Chapter 6](#) and [8](#).

Types

```
# integer
a = 2
# float
b = 5.0
# exponential
```

```

c = 8.3e5
# complex
d = 1.5 + 0.5j
# boolean
e = 4 > 5
# string
f = 'word'

```

Lists

```

# manually initialization
a = ['red', 'blue', 'green']
# initialize from iterable
b = list(range(5))
# list comprehension
c = [nu**2 for nu in b]
# conditioned list comprehension
d = [nu**2 for nu in b if nu < 3]
# access element
e = c[0]
# access a slice of the list
f = c[1:2]
# access last element
g = c[-1]
# list concatenation
h = ['re', 'bl'] + ['gr']
# repeat a list
i = ['re'] * 5
# returns index of 're'
['re', 'bl'].index('re')
# add new element to end of list
a.append('yellow')
# add elements from list 'b' to end of list 'a'
a.extend(b)
# insert element in specified position
a.insert(1, 'yellow')
# true if 're' in list
're' in ['re', 'bl']
# true if 'fi' not in list
'fi' not in ['re', 'bl']
# returns sorted list
sorted([3, 2, 1])
# remove and return item at index (default last)
a.pop(2)

```

Dictionaries

```

# dictionary
a = {'red': 'rouge', 'blue': 'bleu'}
# translate item
b = a['red']
# true if dictionary a contains key 'red'
'red' in a
# loop through contents
c = [value for key, value in a.items()]
# return default
d = a.get('yellow', 'no translation found')
# init key with default
a.setdefault('extra', []).append('cyan')
# update dictionary by data from another one
a.update({'green': 'vert', 'brown': 'brun'})
# get list of keys
a.keys()
# get list of values
a.values()
# get list of key-value pairs
a.items()
# delete key and associated with it value
del a['red']
# remove specified key and return the corresponding value
a.pop('blue')

```

Sets

```

# initialize manually
a = {1, 2, 3}
# initialize from iterable
b = set(range(5))
# add new element to set
a.add(13)
# discard element from set
a.discard(13)
# update set with elements from iterable
a.update([21, 22, 23])
# remove and return an arbitrary set element
a.pop()
# true if 2 in set
2 in {1, 2, 3}
# true if 5 not in set

```

```

5 not in {1, 2, 3}
# test whether every element in a is in b
a.issubset(b)
# issubset in operator form
a <= b
# test whether every element in b is in a
a.issuperset(b)
# issuperset in operator form
a >= b
# return the intersection of two sets as a new set
a.intersection(b)
# return the difference of two or more sets as a new set
a.difference(b)
# difference in operator form
a - b
# return the symmetric difference of two sets as a new set
a.symmetric_difference(b)
# return the union of sets as a new set
a.union(b)
# the same as set but immutable
c = frozenset()

```

Strings

```

# assignment
a = 'red'
# access individual characters
char = a[2]
# string concatenation
'red ' + 'blue'
# split string into list
'1, 2, three'.split(',')
# concatenate list into string
','.join(['1', '2', 'three'])

```

Operators

```

# assignment
a = 2
# change and assign
a += 1 (*=, /=)
# addition
3 + 2

```

```

# integer (python2) or float (python3) division
3 / 2
# integer division
3 // 2
# multiplication
3 * 2
# exponent
3 ** 2
# remainder
3 % 2
# absolute value
abs(a)
# equal
1 == 1
# larger
2 > 1
# smaller
2 < 1
# not equal
1 != 2
# logical AND
1 != 2 and 2 < 3
# logical OR
1 != 2 or 2 < 3
# logical NOT
not 1 == 2
# test if a is in b
'a' in b
# test if objects point to the same memory (id)
a is b

```

Control Flow

```

# if/elif/else
a, b = 1, 2
if a + b == 3:
    print('True')
elif a + b == 1:
    print('False')
else:
    print('??')

# for
a = ['red', 'blue', 'green']

```

```

for color in a:
    print(color)

# while
number = 1
while number < 10:
    print(number)
    number += 1

# break
number = 1
while True:
    print(number)
    number += 1
    if number > 10:
        break

# continue
for i in range(20):
    if i % 2 == 0:
        continue
    print(i)

```

Functions, Classes, Generators, Decorators

```

# Function groups code statements and possibly
# returns a derived value
def myfunc(a1, a2):
    return a1 + a2

x = myfunc(a1, a2)

# Class groups attributes (data)
# and associated methods (functions)
class Point(object):
    def __init__(self, x):
        self.x = x
    def __call__(self):
        print(self.x)

x = Point(3)

# Generator iterates without
# creating all values at once

```

```
def firstn(n):
    num = 0
    while num < n:
        yield num
        num += 1

x = [i for i in firstn(10)]

# Decorator can be used to modify
# the behaviour of a function
class myDecorator(object):
    def __init__(self, f):
        self.f = f
    def __call__(self):
        print("call")
        self.f()

@myDecorator
def my_func():
    print('func')

my_func()
```

C.2 VIRTUALENV

Virtualenv allows the user to create isolated Python environments. With these environments the developers can have different projects and each project will not share any library, avoiding conflicts that may arise when one project requires a library that is incompatible with the requirements of another project.

Create a new virtualenv

```
virtualenv venv_name
```

This command creates a directory in relation to where is executed. If you are in `/home/name/`, the resulting directory will be `/home/name/venv_name`.

If you want a virtualenv that runs a specific Python version, do

```
virtualenv --python=/usr/bin/python2.7 venv_name
```

Change `/usr/bin/python2.7` to the location of you desired Python executable. This way you can run a Python version different than the one you have installed as default.

Use (activate) a virtualenv

Once a virtualenv is created, it can be activated in macOS and Linux with:

```
. venv_name/bin/activate
```

In Windows,

```
venv_name\Scripts\activate
```

In any case (Windows, macOS or Linux), this will turn you command line prompt to:

```
(venv_name)
```

The (venv_name) label tells you that the virtualenv is activated. From now on, each Python program you install will be isolated from other installations.

Deactivate a virtualenv

To stop using a virtualenv, and go back to standard environment, use the deactivate command, in macOS or Linux,

```
. venv_name/bin/deactivate
```

In Windows,

```
venv_name\Scripts\deactivate
```

Install a program inside a virtualenv

In macOS and Linux,

```
(venv_name) $ pip install program_name
```

In Windows,

```
(venv_name) $ venv_name\Scripts\pip install program_name
```

Get list of programs installed inside a virtualenv

```
(venv_name) $ pip freeze
```

If you want to export this list, redirect the output to a file,

```
(venv_name) $ pip freeze > filename
```


Install a list of programs from a text file

```
(venv_name) $ pip install -r filename
```

By convention, the file with all the associated libraries is called `requirements.txt`.

C.3 CONDA

Conda is the pip and virtualenv equivalent from the Anaconda Python distribution. If you know how to use pip and virtualenv, you will understand this tool without additional effort.

Create a new Conda environment

Virtual environment in Conda is called “Conda environments”, and are created with,

```
$ conda create -n venv_name
```

You can also create a Conda environment and install a library with one command:

```
$ conda create -n venv_name program_name
```

Get a list of all installed environments

```
$ conda info --envs
```

Use (activate) a Conda environment

In macOS and Linux,

```
$ source activate venv_name
```

In Windows,

```
$ activate venv_name
```

This will turn your command line prompt to:

```
(venv_name)
```

The `(venv_name)` label tells you that the virtualenv is activated. From now on, each Python program you install will be isolated from other installations.

Deactivate a Conda environment

To stop using a Conda environment, and go back to standard environment, type in macOS and Linux:

```
$ source deactivate venv_name
```

In Windows,

```
$ deactivate venv_name
```

Install a program inside a Conda environment

```
(venv_name) $ conda install program_name
```

If the program is not available from conda install, you can use `pip install` inside a Conda environment.

Get list of programs installed inside a Conda environment

```
(venv_name) $ conda list
```

If you want to export this list, there is a specific command for this,

```
(venv_name) $ conda env export > environment.yml
```

Install a list of programs from a text file

```
$ conda env create -f environment.yml
```

C.4 IPYTHON

IPython is an interactive shell. Now is part of the Jupyter Notebook, an open-source web application used to create and share documents that contain live code, equations, visualizations and text. It is mentioned in page 21 because the code of this book is available as a Jupyter Notebook.

Console

```
# Information about the object
```

```
<object>?
```

```
# tab completion
```

```
<object>.<TAB>
```

```
# run scripts / profile / debug
```

```

%run myscript.py

# measure runtime of statement
%timeit range(1000)
# measure script execution time
%run -t myscript.py

# run statement with profiler
%prun <statement>
# sort by key, e.g. "cumulative" or "calls"
%prun -s <key> <statement>
# profile script
%run -p myfile.py

# run script in debug mode
%run -d myscript.py
# jumps to the debugger after an exception
%debug
# run debugger automatically on exception
%pdb

# examine history
%history
# lines 1-5 of last session
%history ~1/1-5

# run shell commands
!make # prefix command with "!"

# clean namespace
%reset

# run code from clipboard
%paste

```

debugger

```

# execute next line
n
# set breakpoint in the main file at line 42
b 42
# set breakpoint in 'myfile.py' at line 42

```

```

b myfile.py:42
# continue execution
c
# show current position in the code
l
# print the 'data' variable
p data
# pretty print the 'data' variable
pp data
# step into subroutine
s
# print arguments that a function received
a
# show all variables in local scope
pp locals()
# show all variables in global scope
pp globals()

```

command line

```

# debug after exception
ipython --pdb -- myscript.py argument1 --option1
# console after finish
ipython -i -- myscript.py argument1 --option1

```

C.5 NUMPY

NumPy is the fundamental package for scientific computing with Python. Provides a N-dimensional array object, tools to integrating with C and Fortran and several mathematical related capabilities.

```
import numpy as np
```

Array initialization

```

# direct initialization
np.array([2, 3, 4])
# single precision array of size 20
np.empty(20, dtype=np.float32)
# initialize 200 zeros
np.zeros(200)
# 3 x 3 integer matrix with ones
np.ones((3,3), dtype=np.int32)

```

```

# ones on the diagonal
np.eye(200)
# array with zeros and the shape of a
np.zeros_like(a)
# 100 points from 0 to 10
np.linspace(0., 10., 100)
# points from 0 to <100 with step 2
np.arange(0, 100, 2)
# 100 log-spaced from 1e-5 -> 1e2
np.logspace(-5, 2, 100)
# copy array to new memory
np.copy(a)

```

Indexing

```

# initialization with 0 - 99
a = np.arange(100)
# set the first three indices to zero
a[:3] = 0
# set indices 2-4 to 1
a[2:5] = 1
# set all but last three elements to 2
a[:-3] = 2
# general form of indexing/slicing
a[start:stop:step]
# transform to column vector
a[None, :]
# return array with values of the indices
a[[1, 1, 3, 8]]
# transform to 10 x 10 matrix
a = a.reshape(10, 10)
# return transposed view
a.T
# transpose array to new axis order
b = np.transpose(a, (1, 0))
# values with elementwise condition
a[a < 2]

```

Array properties and operations

```

# a tuple with the lengths of each axis
a.shape
# length of axis 0

```

```

len(a)
# number of dimensions (axes)
a.ndim
# sort array along axis
a.sort(axis=1)
# collapse array to one dimension
a.flatten()
# return complex conjugate
a.conj()
# cast to integer
a.astype(np.int16)
# convert (possibly multidimensional) array to list
a.tolist()
# return index of maximum along a given axis
np.argmax(a, axis=1)
# return cumulative sum
np.cumsum(a)
# True if any element is True
np.any(a)
# True if all elements are True
np.all(a)
# return sorted index array along axis
np.argsort(a, axis=1)
# return indices where cond is True
np.where(cond)
# return elements from x or y depending on cond
np.where(cond, x, y)

```

Boolean arrays

```

# returns array with boolean values
a < 2
# elementwise logical and
(a < 2) & (b > 10)
# elementwise logical or
(a < 2) | (b > 10)
# invert boolean array
~a

```

Elementwise operations and math functions

```

# multiplication with scalar
a * 5

```

```

# addition with scalar
a + 5
# addition with array b
a + b
# division with b (np.NaN for division by zero)
a / b
# exponential (complex and real)
np.exp(a)
# a to the power b
np.power(a, b)
# sine
np.sin(a)
# cosine
np.cos(a)
# arctan(a/b)
np.arctan2(a, b)
# arcsin
np.arcsin(a)
# degrees to radians
np.radians(a)
# radians to degrees
np.degrees(a)
# variance of array
np.var(a)
# standard deviation
np.std(a, axis=1)

```

Inner/outer products

```

# inner product: a_mi b_in
np.dot(a, b)
# einstein summation convention
np.einsum('ij,kj->ik', a, b)
# sum over axis 1
np.sum(a, axis=1)
# return absolute values
np.abs(a)
# outer sum
a[None, :] + b[:, None]
# outer product
a[None, :] * b[:, None]
# outer product
np.outer(a, b)
# matrix norm

```

```
np.sum(a * a.T)
```

Linear algebra/ matrix math

```
# Find eigenvalues and eigenvectors
evals, evecs = np.linalg.eig(a)
# np.linalg.eig for hermitian matrix
evals, evecs = np.linalg.eigh(a)
```

Reading/ writing files

```
# ascii data from file
np.loadtxt(fname/fobject, skiprows=2, delimiter=',')
# write ascii data
np.savetxt(fname/fobject, array, fmt='%.5f')
# binary data from file
np.fromfile(fname/fobject, dtype=np.float32, count=5)
# write (C) binary data
np.tofile(fname/fobject)
# save as numpy binary (.npy)
np.save(fname/fobject, array)
# load .npy file (memory mapped)
np.load(fname/fobject, mmap_mode='c')
```

Interpolation, integration, optimization

```
# integrate along axis 1
np.trapz(a, x=x, axis=1)
# interpolate function xp, yp at points x
np.interp(x, xp, yp)
# solve a x = b in least square sense
np.linalg.lstsq(a, b)
```

FFT (Discrete Fourier Transform)

```
# complex fourier transform of a
np.fft.fft(a)
# fft frequencies
f = np.fft.fftfreq(len(a))
# shifts zero frequency to the middle
np.fft.fftshift(f)
# real fourier transform of a
```



```
np.fft.rfft(a)
# real fft frequencies
np.fft.rfftfreq(len(a))
```

Rounding

```
# rounds to nearest upper int
np.ceil(a)
# rounds to nearest lower int
np.floor(a)
# rounds to nearest int
np.round(a)
```

Random variables

```
# 100 normal distributed
from np.random import normal, seed, rand, uniform, randint
normal(loc=0, scale=2, size=100)
# resets the seed value
seed(23032)
# 200 random numbers in [0, 1)
rand(200)
# 200 random numbers in [1, 30)
uniform(1, 30, 200)
# 300 random integers in [1, 16)
randint(1, 16, 300)
```

C.6 MATPLOTLIB

A 2D plotting library, somehow similar to Bokeh. Available at <https://matplotlib.org/>.

```
import matplotlib.pyplot as plt
```

Figures and axes

```
# initialize figure
fig = plt.figure(figsize=(5, 2))
# save png image
fig.savefig('out.png')
# fig and 5 x 2 nparray of axes
fig, axes = plt.subplots(5, 2, figsize=(5, 5))
```

```
# add second subplot in a 3 x 2 grid
ax = fig.add_subplot(3, 2, 2)
# multi column/row axis
ax = plt.subplot2grid((2, 2), (0, 0), colspan=2)
# add custom axis
ax = fig.add_axes([left, bottom, width, height])
```

Figures and axes properties

```
# big figure title
fig.suptitle('title')
# adjust subplot positions
fig.subplots_adjust(bottom=0.1, right=0.8, top=0.9, wspace=0.2,
                    hspace=0.5)
# adjust subplots to fit into fig
fig.tight_layout(pad=0.1, h_pad=0.5, w_pad=0.5, rect=None)
# set xlabel
ax.set_xlabel('xbla')
# set ylabel
ax.set_ylabel('ybla')
# sets x limits
ax.set_xlim(1, 2)
# sets y limits
ax.set_ylim(3, 4)
# sets the axis title
ax.set_title('blabla')
# set multiple parameters at once
ax.set(xlabel='bla')
# activate legend
ax.legend(loc='upper center')
# activate grid
ax.grid(True, which='both')
# returns the axes bounding box
bbox = ax.get_position()
# bounding box parameters
bbox.x0 + bbox.width
```

Plotting routines

```
# plots a line
ax.plot(x,y, '-o', c='red', lw=2, label='bla')
# scatter plot
ax.scatter(x,y, s=20, c=color)
```

```

# fast colormesh
ax.pcolormesh(xx, yy, zz, shading='gouraud')
# slower colormesh
ax.colormesh(xx, yy, zz, norm=norm)
# contour lines
ax.contour(xx, yy, zz, cmap='jet')
# filled contours
ax.contourf(xx, yy, zz, vmin=2, vmax=4)
# histogram
n, bins, patch = ax.hist(x, 50)
# show image
ax.imshow(matrix, origin='lower', extent=(x1, x2, y1, y2))
# plot a spectrogram
ax.spectrogram(y, FS=0.1, noverlap=128, scale='linear')
# write text
ax.text(x, y, string, fontsize=12, color='m')

```

C.7 SCIPY

A collection of open source software for scientific computing in Python, it includes most libraries mentioned in this book (Numpy, Matplotlib, IPython, and others).

```
import scipy as sci
```

Interpolation

```

# interpolate data at index positions:
from scipy.ndimage import map_coordinates
pts_new = map_coordinates(data, float_indices, order=3)

# simple 1d interpolator with axis argument:
from scipy.interpolate import interp1d
interpolator = interp1d(x, y, axis=2, fill_value=0.,
    bounds_error=False)
y_new = interpolator(x_new)

```

Integration

```

# definite integral of python
from scipy.integrate import quad
# function/method
value = quad(func, low_lim, up_lim)

```

Linear algebra

```
from scipy import linalg
# Find eigenvalues and eigenvectors
evals, evecs = linalg.eig(a)
# linalg.eig for hermitian matrix
evals, evecs = linalg.eigh(a)
# Matrix exponential
b = linalg.expm(a)
# Matrix logarithm
c = linalg.logm(a)
```

C.8 PANDAS

A library that provides high-performance, easy-to-use data structures and data analysis tools Python.

```
import pandas as pd
```

Data structures

```
# series
s = pd.Series(np.random.rand(1000), index=range(1000))
# time index
index = pd.date_range("13/06/2016", periods=1000)
# DataFrame
df = pd.DataFrame(np.zeros((1000, 3)), index=index,
                  columns=["A", "B", "C"])
```

DataFrame

The *DataFrame* object is used to hold data for further processing. The usual path is to read the data from a CSV file into a *DataFrame* (using a method provided by this object). Once the data is loaded into the *DataFrame*, multiple transformation can be made. In this book this is used in [Chapter 14](#)

```
# read and load CSV file in a DataFrame
df = pd.read_csv("filename.csv")
# get raw data out of DataFrame object
raw = df.values
# get list of columns headers
cols = df.columns
# get data types of all columns
```

```

df.dtypes
# get first 5 rows
df.head(5)
# get basic statistics for all columns
df.describe()
# get index column range
df.index

#column slicing
# (.loc[] and .ix[] are inclusive of the range of values selected)
# select column values as a series by column name (not optimized)
df.col_name
# select column values as a dataframe by column name (not optimized)
df[['col_name']]
# select column values as a series by column name
df.loc[:, 'col_name']
# select column values as a dataframe by column name
df.loc[:, ['col_name']]
# select by column index
df.iloc[:, 0]
# select by column index, but as a dataframe
df.iloc[:, [0]]
# hybrid approach with column name
df.ix[:, 'col_name']
# hybrid approach with column index
df.ix[:, 0]

# row slicin
# print first 2 rows of the dataframe
print(df[:2])
# select first 2 rows of the dataframe
df.iloc[0:2, :]
# select first 3 rows of the dataframe
df.loc[0:2, 'col_name']
# select first 3 rows of the 3 different columns
df.loc[0:2, ['col_name1', 'col_name3', 'col_name6']]
# select first 3 rows and first 3 columns
# Again, .loc[] and .ix[] are inclusive
df.iloc[0:2, 0:2]

# Dicin
# select all rows where col_name < 7

```

```
df[ df.col_name < 7 ]  
# combine multiple boolean indexing conditionals using bit-wise  
# logical operators.  
# Regular Python boolean operators (and, or) cannot be used here.  
# Be sure to encapsulate each conditional in parenthesis to  
# make this work.  
df[ (df.col_name1 < 7) & (df.col_name2 == 0) ]  
# writing to slice  
df[df.recency < 7] = -100
```



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Index

.ace, [204](#)
.phd.1, [203](#)
.py, [30](#)
.pyc, [30](#)
.pyo, [30](#)
.pyw, [30](#)
 __getitem__, [151](#)
 __init__, [142](#)
 __iter__, [151](#)
 __len__, [149](#)
 __repr__, [151](#)
 __setitem__, [152](#)
 __str__, [150](#)

Ace.read, [204](#)
alignment, [180](#)
Anaconda, [35](#), [248](#)
Apache, [215](#)
append(), [47](#)
array, [406](#)
Atom, [33](#)
attributes, [141](#)

BasicChromosome, [355](#)
batch mode, [27](#)
BeautifulSoup, [248](#)
Bio.Align, [167](#)
Bio.AlignInfo, [170](#)
Bio.AlignIO, [169](#), [176](#)
Bio.Alphabet, [163](#)
Bio.Blast.Applications, [178](#)
Bio.Blast.NCBIXML, [180](#), [323](#)
Bio.Clustalw, [172](#)
Bio.Data.CodonTable, [188](#)
Bio.Data.IUPACData, [188](#)
Bio.Graphics, [358](#)
Bio.PDB, [195](#)
Bio.Restriction, [198](#)

Bio.Seq.back_transcribe, [164](#)
Bio.SeqIO, [173](#), [315](#), [317](#), [320](#)
Bio.SeqIO.parse(), [174](#)
Bio.SeqIO.read, [339](#)
Bio.SeqRecord, [315](#)
Bio.Sequencing, [204](#)
Bio.SeqUtils, [335](#)
Bio.SeqUtils.CheckSum, [201](#)
Bio.SeqUtils.MeltingTemp, [201](#)
Bio.SeqUtils.ProtParam, [202](#)
Biopython, [158](#)
BLAST, [43](#), [44](#), [177](#), [326](#)
BLAST record object, [180](#)
block string, [41](#)
Bokeh, [299](#), [300](#)
Bottle, [385](#)
break, [78](#), [79](#)

Canopy, [16](#)
CAP3, [204](#)
capturing groups, [291](#)
cElementTree, [246](#)
CGI, [215](#)
cgi.FieldStorage, [218](#)
checksum, [201](#)
class variables, [141](#)
cloning vector, [321](#)
ClustalW, [171](#)
ClustalX, [172](#)
comment, [29](#)
comments, [29](#)
conda, [402](#)
conditional expressions, [74](#)
Cord Diagram, [309](#)
cPython, [15](#)
crc32, [201](#)
crc64, [201](#)
CSV, [90](#)

- csv, 336
- csv.reader, 92
- csv.Sniffer(), 92
- cursor, 275
- data type, sequence, 40
- data type, unordered, 40
- data types, 40
- database, 256
- database data types, 260
- database, DELETE, 273
- database, UPDATE, 273
- DataFrame, 307, 413
- debugger, 404
- dict, 55
- dictionaries, 396
- dictionary, 54
- dictionary views, 57
- difference, 61
- dir(), 25
- division, 26
- DNA melting temperature, 201
- docstring, 113
- DOM, 246
- EAFP, 131
- editor, 32
- elif, 71
- encapsulation, 142
- encoding comment, 29
- endonuclease, 197
- Entrez, 190
- Entrez.esearch, 193
- Entrez.esummary, 193
- Entrez.read, 193
- EOF, 134
- EOL, 41
- EST, 343
- eUtils, 190
- Excel, 335, 336
- Excel (Read csv), 92
- Excel (Read from an Excel file), 92
- except, 131
- exceptions, 131
- exit(), 27
- extend(), 47
- extensions, 30
- False, 72
- FASTA, 87, 102, 174, 315, 319
- FFT, 409
- file, open, 86, 89
- flow control, 69
- for loop, 75, 79
- frameworks, 232
- frozenset, 63
- functions, 106
- funtion, scope, 108
- GC content, 200
- gcg checksum, 201
- GenBank, 167
- Genbank, 339
- generator, 113, 318
- getcwd(), 98
- Google, 14
- Google App Engine, 235
- gzip, 196
- gzip.GzipFile, 356
- HeatMap, 309
- heatmap, 308
- hello world, 11
- hits, 180
- HSP, 180
- HTML form, 219, 349
- if-else, 69
- import, 115
- indentation, 30
- indexing, 51
- inheritance, 141, 145
- input, 24
- insert(), 47
- Interactive mode, 23
- Intersection, 60
- IPython, 403
- IronPython, 15
- Iterparse, 246

- IUPACAmbiguousDNA, 147
- IUPACUnambiguousDNA, 147
- join(), 44
- Jupyter Notebook, 22
- Jython, 15
- Kodos, 293
- LBYL, 129
- len(), 53
- Linux, 27
- List comprehension, 46
- list(), 45, 54, 62
- Lists, 44
- lists, 395
- lower(), 42, 139
- macOS, 22
- makeblastdb, 326
- mathematical operations, 26
- Matplotlib, 410
- max(), 53
- MeltingTemp, 335
- methods, 141
- min(), 53
- module, testing, 125
- modules, 114
- MongoDB, 278, 355
- multi-paradigm, 10
- MultipleSeqAlignment, 168
- MUSCLE, 349
- Muscle, 351
- MutableSeq, 148, 165
- MySQL, 257, 262, 385
- next(), 187
- None, 72
- objects paradigm, 139
- OOP, 140
- open, 86
 - read(), 86
 - readline(), 86
 - readlines(), 86
- os, 98
 - chdir(), 98
 - getcwd(), 86
 - makedirs(), 99
 - remove(), 98, 351
 - os.rename(), 98
- Pandas, 307
- parse, 85
- pass, 74
- path.exists(), 99
- path.isdir(), 98
- path.isfile(), 98
- path.py, 98
- path.split(), 99
- path.splitext(), 99
- PDB, 194
- pickle, 94
- pip, 117, 118, 401
- polymorphism, 141
- pop(), 47
- PostgreSQL, 385
- pprint, 203
- primary key, 259
- primer, 329
- Primer3, 329
- Primer3Commandline, 332
- print, 23
- procedural, 139
- program, 7
- PROSITE, 196, 295
- PyCharm, 34
- pymongo, 279
- pymysql, 274
- PyMySQL, module, 274
- PyPy, 15, 385
- Python hosting, 234
- Python prompt, 23
- PythonAnywhere, 21, 385
- PYTHONPATH, 119
- quote, double, 41
- quote, single, 41
- quote, triple, 41

- raise, 135
- random.randint, 315
- range(), 76
- RDBMS, 257
- re, 287
 - findall, 288
 - finditer, 289
 - group(), 288
 - match, 289
 - search, 288
 - sub, 294
 - subn, 294
- REGEX, 285, 292
- relational database, 258
- reportlab.lib, 358
- requests, 249
- restriction, 197
- Restriction.RestrictionBatch, 199
- Restriction.Analysis, 199
- SAX, 246
- Scatter Plot, 307
- scatterplot, 306
- Scipy, 412
- ScriptAlias, 215
- seguid, 201
- SELECT, 269
- separator, 24
- Seq, 147, 163
- Seq.tomutable(), 165
- Seq.transcribe, 163
- seq.translate, 163
- SeqRecord, 166
- sequences types, 40
- SeqUtils, 200
- set, 59
- set operations, 60
- sets, 396
- setup.py, 124
- shebang, 27
- shutil, 98, 99
- slicing, 52
- Spyder, 35
- SQLite, 257, 258, 277, 344
- Stackless, 15
- StopIteration, 319
- str(), 25
- string, 40
 - count(), 42
 - find(), 43
 - index(), 43
 - replace(), 42
 - split(), 43
- strings, 397
- Sublime, 32
- subprocess, 351
- SwissProt, 205
- sys
 - argv, 293
 - exc_info(), 134
- TAIR, 343
- tempfile.mkstemp, 351
- translate, 187
- True, 72
- try, 131
- tuple, 49
- Unicode, 41
- union, 60
- VecScreen, 321
- virtualenv, 119, 400
- Visual Studio, 36
- Web framework, 386
- while loop, 77
- Windows, 20, 29, 120
- WinPython, 35
- WSGI, 221
- xlrd, 93
- xlwt, 94, 335, 336
- XML, 179, 237
- yield, 113, 318