

# Bioinformatics with Python

## Cookbook

Second Edition

Learn how to use modern Python bioinformatics libraries and applications to do cutting-edge research in computational biology



Packt

[www.packt.com](http://www.packt.com)

Tiago Antao

# Bioinformatics with Python Cookbook

## Second Edition

Learn how to use modern Python bioinformatics libraries and applications to do cutting-edge research in computational biology

Tiago Antao

**Packt**

BIRMINGHAM - MUMBAI

# Bioinformatics with Python Cookbook

## *Second Edition*

Copyright © 2018 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Commissioning Editor: Sunith Shetty  
Acquisition Editor: Divya Poojari  
Content Development Editor: Ishita Vora  
Technical Editor: Jovita Alva  
Copy Editor: Safis Editing  
Project Coordinator: Namrata Swetta  
Proofreader: Safis Editing  
Indexer: Tejal Daruwale Soni  
Graphics: Jisha Chirayil  
Production Coordinator: Nilesh Mohite

First published: June 2015

Second edition: November 2018

Production reference: 1301118

Published by Packt Publishing Ltd.  
Livery Place  
35 Livery Street  
Birmingham  
B3 2PB, UK.

ISBN 978-1-78934-469-1

[www.packtpub.com](http://www.packtpub.com)



[mapt.io](https://mapt.io)

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

## Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

## Packt.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at [www.packt.com](https://www.packt.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at [customerscare@packtpub.com](mailto:customerscare@packtpub.com) for more details.

At [www.packt.com](https://www.packt.com), you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

## About the author

Tiago Antao is a bioinformatician currently working in the field of genomics. He was originally a computer scientist but he crossed over to computational biology with an MSc in Bioinformatics from the Faculty of Sciences at the University of Porto, Portugal, and a PhD on the spread of drug-resistant malaria from the Liverpool School of Tropical Medicine in the UK. He is one of the co-authors of Biopython, a major bioinformatics package written in Python.

In his post-doctoral career, he has worked with human datasets at the University of Cambridge (UK) and with mosquito whole genome sequence data at the University of Oxford (UK). He is currently working as a research scientist at the University of Montana.

## About the reviewers

Cho-Yi Chen is an Assistant Professor of Biomedical Informatics at the National Yang-Ming University. While growing up, he was an active athlete, with a passion for science and technology, which led him to become an Olympic swimmer and later a scientist in genomics and systems biology. He received his BSc, MSc, and PhD from the National Taiwan University and Academia Sinica, and finished his postdoctoral training at the Dana-Farber Cancer Institute and Harvard University. He is a founding member of Taiwan's Society of Evolution and Computational Biology, and is a recipient of the MOST Young Scholar Fellowship for his dedication to biomedical research. He and his group strive to advance our understanding of cancer and other human diseases.

Pin-Jou Wu is a research assistant in Dr. Cho-Yi Chen's laboratory at the Institute of Biomedical Informatics at National Yang-Ming University, Taiwan. Her research interests lie in circadian rhythm and its relation to human aging and diseases. She enjoys multi-omics data mining by using statistics and machine learning approaches and enjoys discovering a new insight into chronobiology.

She was a research assistant at Tainan District Agriculture Research and Extension Station. She completed her Master's degree in Plant Pathology and Microbiology at National Taiwan University and also completed a full-year exchange program doing a Master's of Science degree at University College Dublin, Ireland.

Yao-Chung Chen is a bioinformatician with a background in molecular biology. After receiving his MS from National Taiwan University, he participated in a transcriptome analysis research project on immune systems in commercial fishes at National Pingtung University of Science and Technology. Previous research experience makes him believe that biologists should know more about programming, especially those who aim to work in bioinformatics.

Aiming to be a pioneer, he wishes to interpret bioinformatics in a more biology-friendly way and encourage more biologists to join this field. Currently, he works as a research assistant and studies chronobiology at the Institute of Biomedical Informatics, National Yang-Ming University.

## Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit [authors.packtpub.com](http://authors.packtpub.com) and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

# Table of Contents

Preface	1
Chapter 1: Python and the Surrounding Software Ecology	8
Introduction	8
Installing the required software with Anaconda	9
Getting ready	9
How to do it...	12
There's more...	13
Installing the required software with Docker	13
Getting ready	13
How to do it...	14
See also	15
Interfacing with R via rpy2	15
Getting ready	15
How to do it...	16
There's more...	22
See also	22
Performing R magic with Jupyter Notebook	23
Getting ready	23
How to do it...	23
There's more...	25
See also	25
Chapter 2: Next-Generation Sequencing	26
Introduction	26
Accessing GenBank and moving around NCBI databases	27
Getting ready	28
How to do it...	28
There's more...	32
See also	33
Performing basic sequence analysis	33
Getting ready	33
How to do it...	34
There's more...	35
See also	36
Working with modern sequence formats	36
Getting ready	36
How to do it...	37
There's more...	43
See also	44

---

## Table of Contents

---

Working with alignment data	45
Getting ready	45
How to do it...	46
There's more...	51
See also	52
Analyzing data in VCF	53
Getting ready	53
How to do it...	54
There's more...	55
See also	56
Studying genome accessibility and filtering SNP data	56
Getting ready	57
How to do it...	58
There's more...	68
See also	69
Processing NGS data with HTSeq	69
Getting ready	70
How to do it...	70
There's more...	73
Chapter 3: Working with Genomes	74
Introduction	74
Working with high-quality reference genomes	75
Getting ready	75
How to do it...	76
There's more...	81
See also	81
Dealing with low-quality genome references	81
Getting ready	82
How to do it...	82
There's more...	86
See also	87
Traversing genome annotations	87
Getting ready	87
How to do it...	88
There's more...	89
See also	90
Extracting genes from a reference using annotations	90
Getting ready	91
How to do it...	91
There's more...	93
See also	94
Finding orthologues with the Ensembl REST API	94
Getting ready	94
How to do it...	95

---

---

## Table of Contents

---

There's more...	98
Retrieving gene ontology information from Ensembl	98
Getting ready	98
How to do it...	99
There's more...	102
See also	103
Chapter 4: Population Genetics	104
Introduction	104
Managing datasets with PLINK	105
Getting ready	106
How to do it...	107
There's more...	111
See also	112
Introducing the Genepop format	112
Getting ready	113
How to do it...	113
See also	117
Exploring a dataset with Bio.PopGen	117
Getting ready	118
How to do it...	118
There's more...	123
See also	123
Computing F-statistics	123
Getting ready	123
How to do it...	124
See also	129
Performing Principal Components Analysis	130
Getting ready	130
How to do it...	130
There's more...	134
See also	135
Investigating population structure with admixture	135
Getting ready	135
How to do it...	136
There's more...	141
Chapter 5: Population Genetics Simulation	142
Introduction	142
Introducing forward-time simulations	143
Getting ready	143
How to do it...	143
There's more...	149
Simulating selection	149
Getting ready	150
How to do it...	150

---

---

## Table of Contents

---

There's more...	156
Simulating population structure using island and stepping-stone models	156
Getting ready	156
How to do it...	157
Modeling complex demographic scenarios	162
Getting ready	162
How to do it...	163
<b>Chapter 6: Phylogenetics</b>	169
Introduction	169
Preparing a dataset for phylogenetic analysis	170
Getting ready	170
How to do it...	170
There's more...	175
See also	176
Aligning genetic and genomic data	176
Getting ready	176
How to do it...	176
Comparing sequences	178
Getting ready	178
How to do it...	178
There's more...	183
Reconstructing phylogenetic trees	183
Getting ready	183
How to do it...	184
There's more...	188
Playing recursively with trees	188
Getting ready	188
How to do it...	189
There's more...	193
Visualizing phylogenetic data	193
Getting ready	194
How to do it...	194
There's more...	200
<b>Chapter 7: Using the Protein Data Bank</b>	201
Introduction	201
Finding a protein in multiple databases	202
Getting ready	202
How to do it...	203
There's more...	206
Introducing Bio.PDB	206
Getting ready	207
How to do it...	207

---

## Table of Contents

---

There's more...	211
Extracting more information from a PDB file	211
Getting ready	211
How to do it...	212
Computing molecular distances on a PDB file	215
Getting ready	216
How to do it...	216
Performing geometric operations	220
Getting ready	220
How to do it...	220
There's more...	223
Animating with PyMOL	223
Getting ready	223
How to do it...	224
There's more...	229
Parsing mmCIF files using Biopython	230
Getting ready	230
How to do it...	230
There's more...	231
Chapter 8: Bioinformatics Pipelines	232
Introduction	232
Introducing Galaxy servers	233
Getting ready	233
How to do it...	234
There's more...	236
Accessing Galaxy using the API	236
Getting ready	236
How to do it...	238
Developing a Galaxy tool	244
Getting ready	244
How to do it...	245
There's more...	247
Using generic pipelines with bioinformatics data	248
Getting ready	248
How to do it...	248
Deploying a variant analysis pipeline with Airflow	250
Getting ready	251
How to do it...	251
There's more...	257
Chapter 9: Python for Big Genomics Datasets	258
Introduction	258
Using high-performance data formats - HDF5	259
Getting ready	259

---

## Table of Contents

---

How to do it...	260
There's more...	264
Doing parallel computing with Dask	264
Getting ready	265
How to do it...	265
There's more...	268
Using high-performance data formats – Parquet	269
Getting ready	269
How to do it...	269
There's more...	270
Computing sequencing statistics using Spark	271
Getting ready	271
How to do it...	272
There's more...	273
Optimizing code with Cython and Numba	274
Getting ready	274
How to do it...	274
There's more...	278
Chapter 10: Other Topics in Bioinformatics	279
Introduction	279
Doing metagenomics with the QIIME 2 Python API	280
Getting ready	280
How to do it...	282
There's more...	285
Inferring shared chromosomal segments with Germline	285
Getting ready	285
How to do it...	287
There's more...	290
Accessing the Global Biodiversity Information Facility via REST	290
How to do it...	291
There's more...	296
Georeferencing GBIF datasets	297
Getting ready	297
How to do it...	297
There's more...	302
Plotting protein interactions with Cytoscape the hard way	303
Getting ready	303
How to do it...	304
There's more...	309
Chapter 11: Advanced NGS Processing	310
Introduction	310
Preparing the dataset for analysis	311
Getting ready	311

---

## Table of Contents

---

How to do it...	312
Using Mendelian error information for quality control	317
How to do it...	317
There's more...	321
Using decision trees to explore the data	321
How to do it...	322
Exploring the data with standard statistics	324
How to do it...	324
There's more...	329
Finding genomic features from sequencing annotations	329
How to do it...	330
There's more...	332
<b>Index</b>	<b>333</b>

# Preface

Whether you are reading this book as a computational biologist or a Python programmer, you will probably relate to the phrase "explosive growth, exciting times." The recent growth in the use of Python is strongly connected with its status as big data's main programming language. The deluge of data in biology, mostly from genomics and proteomics, makes bioinformatics one of the forefront applications of data science. There is a massive need for bioinformaticians to analyze all this data; of course, one of the main tools is Python. We will not only talk about the programming language but also the whole community and software ecology behind it.

When you choose Python to analyze your data, you expect to get an extensive set of libraries, ranging from statistical analysis to plotting, parallel programming, machine learning, and bioinformatics. However, you actually get even more than this; the community has a tradition of providing good documentation, reliable libraries, and frameworks. It is also friendly and supportive of all its participants.

In this book, we will present practical solutions to modern bioinformatics problems using Python. Our approach will be hands-on; we will address important topics, such as next-generation sequencing, genomics, population genetics, phylogenetics, and proteomics.

At this stage, you probably know the language reasonably well and are aware of the basic analysis methods in your field of research. You will dive directly into relevant complex computational biology problems and learn how to tackle them with Python. This is not your first Python book or your first biology lesson; this is where you will find reliable and pragmatic solutions to realistic and complex problems.

The first edition of this book took several high-risk decisions a few years ago, considering Docker, Jupyter Notebook, and even Python 3 were not obvious choices. These choices worked perfectly well. The second edition once again uses these technologies, which are now standard in the field. Probably due to bioinformatics being a more mature field, there are no high-risk options now. There is new content on pipelines, parallel processing systems, and file formats, but none of them are unsafe bets.

## Who this book is for

This book is for data scientists, bioinformatics analysts, researchers, and Python developers who want to address intermediate-to-advanced biological and bioinformatics problems using a recipe-based approach. Working knowledge of Python programming language is expected.

## What this book covers

Chapter 1, Python and the Surrounding Software Ecology, tells you how to set up a modern bioinformatics environment with Python. This chapter discusses how to deploy software using Docker, interface with R, and interact with the IPython Notebook.

Chapter 2, Next-Generation Sequencing, provides concrete solutions to deal with next-generation sequencing data. This chapter teaches you how to deal with large FASTQ, BAM, and VCF files. It also discusses data filtering.

Chapter 3, Working with Genomes, not only deals with high-quality references—such as the human genome—but also discusses how to analyze other low-quality references typical in nonmodel species. It introduces GFF processing, teaches you to analyze genomic feature information, and discusses how to use gene ontologies.

Chapter 4, Population Genetics, describes how to perform population genetics analysis of empirical datasets. For example, on Python, we could perform Principal Components Analysis, computer<sup>s</sup>For structure/admixture plots.

Chapter 5, Population Genetics Simulation, covers simuPOP, an extremely powerful Python-based forward-time population genetics simulator. This chapter shows you how to simulate different selection and demographic regimes. It also briefly discusses coalescent simulation.

Chapter 6, Phylogenetics, uses complete sequences of recently sequenced Ebola viruses to perform real phylogenetic analysis, which includes tree reconstruction and sequence comparisons. This chapter discusses recursive algorithms to process tree-like structures.

Chapter 7, Using the Protein Data Bank, focuses on processing PDB files, for example, performing the geometric analysis of proteins. This chapter takes a look at protein visualization.

Chapter 8, Bioinformatics Pipelines, introduces two types of pipelines. The first type of pipeline is Python-based Galaxy, a widely used system with a web-interface targeting mostly non-programming users, although bioinformaticians might still have to interact with it programmatically. The second type is Airflow, a type of pipeline that targets programmers.

Chapter 9, Python for Big Genomics Datasets, discusses high-performance programming techniques necessary to handle big datasets. It briefly discusses parallel processing with Dask and Spark. Code optimization frameworks (such as Numba or Cython) are introduced. Finally, efficient file formats such as HDF5 or Parquet are presented.

Chapter 10, Other Topics in Bioinformatics, talks about how to analyze data made available by the Global Biodiversity Information Facility (GBIF) and how to use Cytoscape, a powerful platform to visualize complex networks. This chapter also looks at how to work with geo-referenced data and map-based services.

Chapter 11, Advanced NGS Processing, covers advanced programming techniques to filter NGS data. These include the use of Mendelian datasets that are then analyzed by standard statistics and machine learning techniques.

## To get the most out of this book

Modern bioinformatics analysis is normally performed on a Linux server. Most of our recipes will also work on macOS. It will also work on Windows in theory, but this is not recommended. If you do not have a Linux server, you can use a free virtual machine emulator, such as VirtualBox, to run it on a Windows/macOS computer. An alternative that we explore in the book is to use Docker as a container, which can be used on Windows and macOS.

As modern bioinformatics is a big data discipline, you will need a reasonable amount of memory; at least 8 GB on a native Linux machine, probably 16 GB on a macOS/Windows system, but more would be better. A broadband internet connection will also be necessary to download the real and hands-on datasets used in the book.

Python is a requirement. With few exceptions, the code will need Python 3. Many free Python libraries will also be required and these will be presented in the book. Biopython, NumPy, SciPy, and Matplotlib are used in almost all chapters. Although Jupyter Notebook is not strictly required, it's highly encouraged. Different chapters will also require various bioinformatics tools. All the tools used in the book are freely available and thorough instructions are provided in the relevant chapters of this book.

## Download the example code files

You can download the example code files for this book from your account at [www.packt.com](http://www.packt.com). If you purchased this book elsewhere, you can visit [www.packt.com/support](http://www.packt.com/support) and register to have the files emailed directly to you.

You can download the code files by following these steps:

- Log in or register at [www.packt.com](http://www.packt.com).
- Select the SUPPORT tab.
- Click on Code Downloads & Errata.
- Enter the name of the book in the Search box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- ZipEg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Bioinformatics-with-Python-Cookbook-Second-Edition>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

## Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: [https://www.packtpub.com/sites/default/files/downloads/9781789344691\\_ColorImages.pdf](https://www.packtpub.com/sites/default/files/downloads/9781789344691_ColorImages.pdf).

## Conventions used

There are a number of text conventions used throughout this book.

**CodeInText:** Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "We will read the data from our file using R's `read.delim` function."

A block of code is set as follows:

```
import os
from IPython.display import Image
import rpy2.robj as robjects
import pandas as pd
from rpy2.robj import pandas2ri
```

Any command-line input or output is written as follows:

```
conda install r-essentials r-gridextra
```

**Bold:** Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "On the top menu choose User, inside choose Preferences."



Warnings or important notes appear like this.



Tips and tricks appear like this.

## Sections

In this book, you will find several headings that appear frequently (Getting ready, How to do it..., How it works..., There's more..., and See also).

To give clear instructions on how to complete a recipe, use these sections as follows:

### Getting ready

This section tells you what to expect in the recipe and describes how to set up any software or any preliminary settings required for the recipe.

### How to do it...

This section contains the steps required to follow the recipe.

## How it works...

This section usually consists of a detailed explanation of what happened in the previous section.

## There's more...

This section consists of additional information about the recipe in order to make you more knowledgeable about the recipe.

## See also

This section provides helpful links to other useful information for the recipe.

## Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at [customercare@packtpub.com](mailto:customercare@packtpub.com).

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit [www.packt.com/submit-errata](http://www.packt.com/submit-errata), selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at [copyright@packt.com](mailto:copyright@packt.com) with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit [authors.packtpub.com](http://authors.packtpub.com).

## Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit [packt.com](http://packt.com).

# 1

# Python and the Surrounding Software Ecology

In this chapter, we will cover the following recipes:

- Installing the required software with Anaconda
- Installing the required software with Docker
- Interfacing with R via rpy2
- Performing R magic with Jupyter Notebook

## Introduction

We will start by installing the required software. This will include the Python distribution, some fundamental Python libraries, and external bioinformatics software. Here, we will also be concerned with the world outside Python. In bioinformatics and big data, R is also a major player; therefore, you will learn how to interact with it via rpy2, which is a Python/R bridge. We will also explore the advantages that the IPython framework (via Jupyter Notebook) can give us in order to efficiently interface with R. This chapter will set the stage for all of the computational biology that we will perform in the rest of this book.

As different users have different requirements, we will cover two different approaches for installing the software. One approach is using the Anaconda Python (<http://docs.anaconda.io/>) distribution, and another approach to install the software is via Docker (a server virtualization method based on containers sharing the same operating system kernel—<https://www.docker.com/>). If you are using a Windows-based operating system, you are strongly encouraged to consider changing your operating system or use Docker via some of the existing options on Windows. On macOS, you might be able to install most of the software natively, though Docker is also available.

# Installing the required software with Anaconda

Before we get started, we need to install some prerequisite software. The following sections will take you through the software and the steps needed to install them. An alternative way to start is to use the Docker recipe, after which everything will be taken care for you via a Docker container.

If you are already using a different Python version, you are strongly encouraged to consider Anaconda, as it has become the de facto standard for data science. Also, it is the distribution that will allow you to install software from Bioconda (<https://bioconda.github.io/>).

## Getting ready

Python can be run on top of different environments. For instance, you can use Python inside the Java Virtual Machine (JVM) (via Jython) or with .NET (with IronPython). However, here, we are concerned not only with Python, but also with the complete software ecology around it; therefore, we will use the standard (CPython) implementation, since the JVM and .NET versions exist mostly to interact with the native libraries of these platforms. A potentially viable alternative would be to use the PyPy implementation of Python (not to be confused with Python Package Index (PyPI)).

Save for noted exceptions, we will be using Python 3 only. If you were starting with Python and bioinformatics, any operating system will work, but here, we are mostly concerned with intermediate to advanced usage. So, while you can probably use Windows and macOS, most heavy-duty analysis will be done on Linux (probably on a Linux cluster). Next-generation sequencing (NGS) data analysis and complex machine learning is mostly performed on Linux clusters.

If you are on Windows, you should consider upgrading to Linux for your bioinformatics work because most modern bioinformatics software will not run on Windows. macOS will be fine for almost all analyses, unless you plan to use a computer cluster, which will probably be Linux-based.

If you are on Windows or macOS and do not have easy access to Linux, don't worry. Modern virtualization software (such as VirtualBox and Docker) will come to your rescue, which will allow you to install a virtual Linux on your operating system. If you are working with Windows and decide that you want to go native and not use Anaconda, be careful with your choice of libraries; you are probably safer if you install the 32-bit version for everything (including Python itself).



If you are on Windows, many tools will be unavailable to you.



Bioinformatics and data science are moving at breakneck speed; this is not just hype, it's a reality. When installing software libraries, choosing a version might be tricky. Depending on the code that you have, it might not work with some old versions, or maybe not even work with a newer version. Hopefully, any code that you use will indicate the correct dependencies—though this is not guaranteed.

The software developed for this book is available at <https://github.com/PacktPublishing/Bioinformatics-with-Python-Cookbook-Second-Edition>. To access it, you will need to install Git. Alternatively, you can download the ZIP file that GitHub makes available (indeed, getting used to Git may be a good idea because lots of scientific computing software is being developed with it).

Before you install the Python stack properly, you will need to install all the external non-Python software that you will be interoperating with. The list will vary from chapter to chapter, and all chapter-specific packages will be explained in their respective chapters. Some less common Python libraries may also be referred to in their specific chapters. Fortunately, since the first edition of this book, most bioinformatics software can be easily installed with conda using the Bioconda project.

If you are not interested in a specific chapter, you can skip the related packages and libraries. Of course, you will probably have many other bioinformatics applications around—such as Burrows-Wheeler Aligner (bwa) or Genome Analysis Toolkit (GATK) for NGS—but we will not discuss these because we do not interact with them directly (although we might interact with their outputs).

You will need to install some development compilers and libraries, all of which are free. On Ubuntu, consider installing the build-essential package (apt-get it), and on macOS, consider Xcode (<https://developer.apple.com/xcode/>).

In the following table, you will find a list of the most important Python software:

Name	Application	URL	Purpose
Project Jupyter	All chapters	<a href="https://jupyter.org/">https://jupyter.org/</a>	Interactive computing
pandas	All chapters	<a href="https://pandas.pydata.org/">https://pandas.pydata.org/</a>	Data processing
NumPy	All chapters	<a href="http://www.numpy.org/">http://www.numpy.org/</a>	Array/matrix processing
SciPy	All chapters	<a href="https://www.scipy.org/">https://www.scipy.org/</a>	Scientific computing
Biopython	All chapters	<a href="https://biopython.org/">https://biopython.org/</a>	Bioinformatics library
PyVCF	NGS	<a href="https://pyvcf.readthedocs.io">https://pyvcf.readthedocs.io</a>	VCF processing
Pysam	NGS	<a href="https://github.com/pysam-developers/pysam">https://github.com/pysam-developers/pysam</a>	SAM/BAM processing
HTSeq	NGS/Genomes	<a href="https://htseq.readthedocs.io">https://htseq.readthedocs.io</a>	NGS processing
simuPOP	Population genetics	<a href="http://simupop.sourceforge.net/">http://simupop.sourceforge.net/</a>	Population genetics simulation
DendroPY	Phylogenetics	<a href="https://dendropy.org/">https://dendropy.org/</a>	Phylogenetics
scikit-learn	Machine learning/population genetics	<a href="http://scikit-learn.org">http://scikit-learn.org</a>	Machine learning library
PyMol	Proteomics	<a href="https://pymol.org">https://pymol.org</a>	Molecular visualization
rpy2	Introduction	<a href="https://rpy2.readthedocs.io">https://rpy2.readthedocs.io</a>	R interface
seaborn	All chapters	<a href="http://seaborn.pydata.org/">http://seaborn.pydata.org/</a>	Statistical chart library
Cython	Big data	<a href="http://cython.org/">http://cython.org/</a>	High performance
Numba	Big data	<a href="https://numba.pydata.org/">https://numba.pydata.org/</a>	High performance
Dask	Big data	<a href="http://dask.pydata.org">http://dask.pydata.org</a>	Parallel processing

We have taken a somewhat conservative approach in most of the recipes with regard to the processing of tabular data. While we use pandas every now and then, most of the time, we use standard Python. As time advances and pandas becomes more pervasive, it will probably make sense to just process all tabular data with it (if it fits in-memory).

## How to do it...

Take a look at the following steps to get started:

Start by downloading the Anaconda distribution from <https://www.anaconda.com/download>. Choose Python version 3. In any case, this is not fundamental, because Anaconda will let you use Python 2 if you need it. You can accept all the installation defaults, but you may want to make sure that the conda binaries are in your path (do not forget to open a new window so that the path is updated). If you have another Python distribution, be careful with your PYTHONPATH and existing Python libraries. It's probably better to unset your PYTHONPATH. As much as possible, uninstall all other Python versions and installed Python libraries.

Let's go ahead with the libraries. We will now create a new conda environment called bioinformatics with biopython=1.70, as shown in the following command:

```
conda create -n bioinformatics biopython biopython=1.70
```

Let's activate the environment, as follows:

```
source activate bioinformatics
```

Let's add the bioconda and conda-forge channel to our source list:

```
conda config --add channels bioconda  
conda config --add channels conda-forge
```

Also, install the core packages:

```
conda install scipy matplotlib jupyter-notebook pip pandas cython  
numba scikit-learn seaborn pysam pyvcf simuPOP dendropy rpy2
```

Some of them will probably be installed with the core distribution anyway.

We can even install R from conda:

```
conda install r-essentials r-gridextra
```

r-essentials installs a lot of R packages, including ggplot2, which we will use later. We also install r-gridextra, since we will be using it in the Notebook.

## There's more...

Compared to the first edition of this book, this recipe is now highly simplified. There are two main reasons for this: the bioconda package, and the fact that we only need to support Anaconda as it has become a standard. If you feel strongly against using Anaconda, you will be able to install many of the Python libraries via pip. You will probably need quite a few compilers and build tools—not only C compilers, but also C++ and Fortran.

## Installing the required software with Docker

Docker is the most widely-used framework for implementing operating system-level virtualization. This technology allows you to have an independent container: a layer that is lighter than a virtual machine, but still allows you to compartmentalize software. This mostly isolates all processes, making it feel like each container is a virtual machine. Docker works quite well at both extremes of the development spectrum: it's an expedient way to set up the content of this book for learning purposes, and may become your platform of choice for deploying your applications in complex environments. This recipe is an alternative to the previous recipe.

However, for long-term development environments, something along the lines of the previous recipe is probably your best route, although it can entail a more laborious initial setup.

## Getting ready

If you are on Linux, the first thing you have to do is install Docker. The safest solution is to get the latest version from <https://www.docker.com/>. While your Linux distribution may have a Docker package, it may be too old and buggy (remember the "advancing at breakneck speed" thing we mentioned?).

If you are on Windows or macOS, do not despair; take a look at the Docker site. Various options are available there to save you, but there is no clear-cut formula, as Docker advances quite quickly on those platforms. A fairly recent computer is necessary to run our 64-bit virtual machine. If you have any problems, reboot your machine and make sure that on the BIOS, VT-X or AMD-V is enabled. At the very least, you will need 6 GB of memory, preferably more.



This will require a very large download from the internet, so be sure that you have plenty of bandwidth. Also, be ready to wait for a long time.

## How to do it...

Follow these steps to get started:

Use the following command on your Docker shell:

```
docker build -t bio  
https://raw.githubusercontent.com/PacktPublishing/Bioinformatics-with-Python-Cookbook-Second-Edition/master/docker/Dockerfile
```

On Linux, you will either need to have root privileges or be added to the Docker Unix group.

Now you are ready to run the container, as follows:

```
docker run -ti -p 9875:9875 -v YOUR_DIRECTORY:/data bio
```

Replace YOUR\_DIRECTORY with a directory on your operating system. This will be shared between your host operating system and the Docker container.

YOUR\_DIRECTORY will be seen in the container on /data and vice versa.

-p 9875:9875 will expose the container TCP port 9875 on the host computer port 9875.

Especially on Windows (and maybe on macOS), make sure that your directory is actually visible inside the Docker shell environment. If not, check the Docker documentation on how to expose directories.

You are now ready to use the system. Point your browser to <http://localhost:9875> and you should get the Jupyter environment.

If this does not work on Windows, check the Docker documentation (<https://docs.docker.com/>) on how to expose ports.

## See also

- Docker is the most widely used containerization software and has seen enormous growth in usage in recent times. You can read more about it at <https://www.docker.com/>.
- A security-minded alternative to Docker is rkt, which can be found at <https://coreos.com/rkt/>.
- If you are not able to use Docker; for example, if you do not have permissions, as will be the case on most computer clusters, then take a look at Singularity at <https://www.sylabs.io/singularity/>.

## Interfacing with R via rpy2

If there is some functionality that you need and you cannot find it in a Python library, your first port of call is to check whether it's implemented in R. For statistical methods, R is still the most complete framework; moreover, some bioinformatics functionalities are also only available in R, most probably offered as a package belonging to the Bioconductor project.

rpy2 provides a declarative interface from Python to R. As you will see, you will be able to write very elegant Python code to perform the interfacing process. To show the interface (and try out one of the most common R data structures, the DataFrame, and one of the most popular R libraries, ggplot2), we will download its metadata from the Human 1,000 Genomes Project (<http://www.1000genomes.org/>). This is not a book on R, but we want to provide interesting and functional examples.

## Getting ready

You will need to get the metadata file from the 1,000 Genomes sequence index. Please check <https://github.com/PacktPublishing/Bioinformatics-with-Python-Cookbook-Second-Edition/blob/master/Datasets.ipynb> and download the sequence.index file. If you are using Jupyter Notebook, open the Chapter01/Interfacing\_R.ipynb file and just execute the wget command on top.

This file has information about all of the FASTQ files in the project (we will use data from the Human 1,000 Genomes Project in the chapters to come). This includes the FASTQ file, the sample ID, and the population of origin, and important statistical information per lane, such as the number of reads and number of DNA bases read.

## How to do it...

Follow these steps to get started:

Let's start by doing some imports:

```
import os
from IPython.display import Image
import rpy2.robj as robjects
import pandas as pd
from rpy2.robj import pandas2ri
from rpy2.robj import default_converter
from rpy2.robj.conversion import localconverter
```

We will be using pandas on the Python side. R DataFrames map very well to pandas.

We will read the data from our file using R's `read.delim` function:

```
read_delim = robjects.r('read.delim')
seq_data = read_delim('sequence.index', header=True,
stringsAsFactors=False)
#In R:
# seq.data <- read.delim('sequence.index', header=TRUE,
stringsAsFactors=FALSE)
```

The first thing that we do after importing is access the `read.delim` R function, which allows you to read files. The R language specification allows you to put dots in the names of objects. Therefore, we have to convert a function name to `read_delim`. Then, we call the function name proper; note the following highly declarative features. Firstly, most atomic objects, such as strings, can be passed without conversion. Secondly, argument names are converted seamlessly (barring the dot issue). Finally, objects are available in the Python namespace (but objects are actually not available in the R namespace; more about this later).

For reference, I have included the corresponding R code. I hope it's clear that it's an easy conversion. The `seq_data` object is a DataFrame. If you know basic R or pandas, you are probably aware of this type of data structure; if not, then this is essentially a table: a sequence of rows where each column has the same type.

Let's perform a basic inspection of this DataFrame, as follows:

```
print('This dataframe has %d columns and %d rows' %
(seq_data.ncol, seq_data.nrow))
print(seq_data.colnames)
#In R:
```

---

```
# print(colnames(seq.data))
# print(nrow(seq.data))
# print(ncol(seq.data))
```

Again, note the code similarity.

You can even mix styles using the following code:

```
my_cols = robjects.r.ncol(seq_data)
print(my_cols)
```

You can call R functions directly; in this case, we will call ncol if they do not have dots in their name; however, be careful. This will display an output, not 26 (the number of columns), but [26], which is a vector that's composed of the element 26. This is because, by default, most operations in R return vectors. If you want the number of columns, you have to perform my\_cols[0]. Also, talking about pitfalls, note that R array indexing starts with 1, whereas Python starts with 0.

Now, we need to perform some data cleanup. For example, some columns should be interpreted as numbers, but they are read as strings:

```
as_integer = robjects.r('as.integer')
match = robjects.r.match

my_col = match('READ_COUNT', seq_data.colnames)[0] # vector
returned
print('Type of read count before as.integer: %s' % seq_data[my_col
- 1].rclass[0])
seq_data[my_col - 1] = as_integer(seq_data[my_col - 1])
print('Type of read count after as.integer: %s' % seq_data[my_col -
1].rclass[0])
```

The match function is somewhat similar to the index method in Python lists. As expected, it returns a vector so that we can extend it. It's also 1-indexed, so we subtract 1 when working on Python. The as\_integer function will convert a column into integers. The first print will show strings (values surrounded by " ), whereas the second print will show numbers.

We will need to massage this table a bit more; details on this can be found on the Notebook, but here, we will finalize getting the DataFrame to R (remember that while it's an R object, it's actually visible on the Python namespace):

```
import rpy2.robjects.lib.ggplot2 as ggplot2
```

This will create a variable in the R namespace called seq.data, with the content of the DataFrame from the Python namespace. Note that after this operation, both objects will be independent (if you change one, it will not be reflected on the other).



While you can perform plotting on Python, R has default built-in plotting functionalities (which we will ignore here). It also has a library called ggplot2 that implements the Grammar of Graphics (a declarative language to specify statistical charts).

With regard to our concrete example based on the Human 1,000 Genomes Project, we will first plot a histogram with the distribution of center names, where all sequencing lanes were generated. We will use ggplot for this:

```
from rpy2.robjects.functions import SignatureTranslatedFunction

ggplot2.theme = SignatureTranslatedFunction(ggplot2.theme,
init_prm_translate = {'axis_text_x': 'axis.text.x'})

bar = ggplot2.ggplot(seq_data) + ggplot2.geom_bar() +
ggplot2.aes_string(x='CENTER_NAME') +
ggplot2.theme(axis_text_x=ggplot2.element_text(angle=90, hjust=1))
robjects.r.png('out.png', type='cairo-png')
bar.plot()
dev_off = robjects.r('dev.off')
dev_off()
```

The second line is a bit uninteresting, but is an important piece of boilerplate code. One of the R functions that we will call has a parameter with a dot in its name. As Python function calls cannot have this, we must map the axis.text.x R parameter name to the axis\_text\_r Python name in the function theme. We monkey patch it (that is, we replace ggplot2.theme with a patched version of itself).

We then draw the chart itself. Note the declarative nature of ggplot2 as we add features to the chart. First, we specify the seq\_data DataFrame, then we use a histogram bar plot called geom\_bar, followed by annotating the x variable (CENTER\_NAME). Finally, we rotate the text of the x axis by changing the theme. We finalize this by closing the R printing device.

We can now print the image on the Jupyter Notebook:

```
Image(filename='out.png')
```

The following chart is produced:

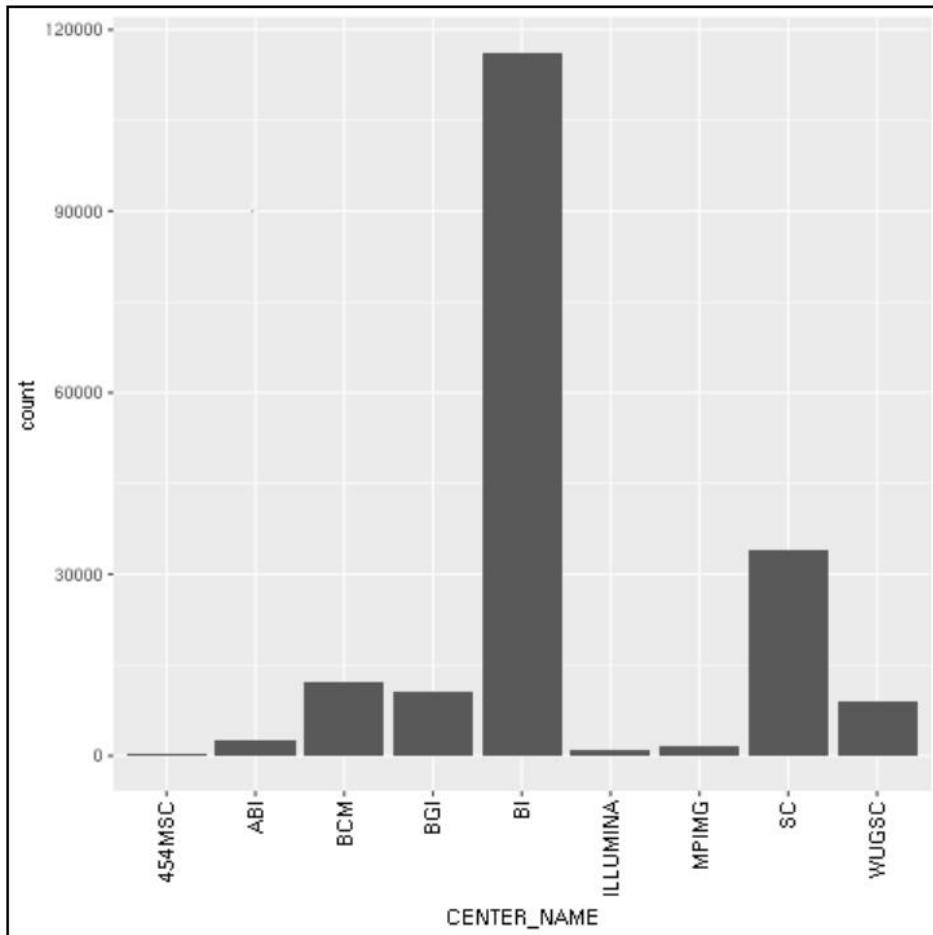


Figure 1: The ggplot2-generated histogram of center names, which is responsible for sequencing the lanes of the human genomic data from the 1,000 Genomes Project

As a final example, we will now do a scatter plot of read and base counts for all the sequenced lanes for Yoruban (YRI) and Utah residents with ancestry from Northern and Western Europe (CEU), using the Human 1,000 Genomes Project (the summary of the data of this project, which we will use thoroughly, can be seen in the Working with modern sequence formats recipe in Chapter 2, Next-Generation Sequencing). We are also interested in the differences between the different types of sequencing (exome, high, and low coverage). First, we generate a DataFrame only just YRI and CEU lanes, and limit the maximum base and read counts:

```
robjects.r('yri_ceu <- seq.data[seq.data$POPULATION %in% c("YRI",  
"CEU") & seq.data$BASE_COUNT < 2E9 & seq.data$READ_COUNT < 3E7, ]')  
yri_ceu = robjects.r('yri_ceu')
```

We are now ready to plot:

```
scatter = ggplot2.ggplot(yri_ceu) +  
ggplot2.aes_string(x='BASE_COUNT', y='READ_COUNT',  
shape='factor(POPULATION)', col='factor(ANALYSIS_GROUP)') +  
ggplot2.geom_point()  
robjects.r.png('out.png')  
scatter.plot()
```

Hopefully, this example (refer to the following screenshot) makes the power of the Grammar of Graphics approach clear. We will start by declaring the DataFrame and the type of chart in use (the scatter plot implemented by `geom_point`).

Note how easy it is to express that the shape of each point depends on the POPULATION variable and the color on the ANALYSIS\_GROUP:

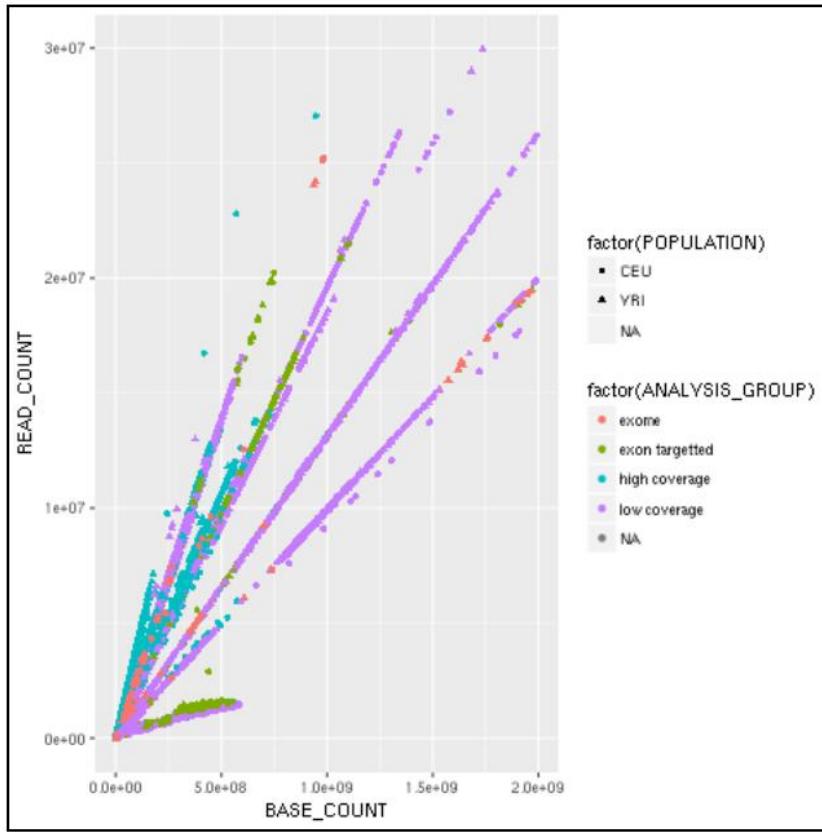


Figure 2: The ggplot2-generated scatter plot with base and read counts for all sequencing lanes read; the color and shape of each dot reflects categorical data (population and the type of data sequenced)

Because the R DataFrame is so close to pandas, it makes sense to convert between the two, as that is supported by rpy2:

```
pd_yri_ceu = pandas2ri.ri2py(yri_ceu)
del pd_yri_ceu['PAIRED_FASTQ']
no_paired = pandas2ri.py2ri(pd_yri_ceu)
robjects.r.assign('no.paired', no_paired)
robjects.r("print(colnames(no.paired))")
```

We start by importing the necessary conversion module. We then convert the R DataFrame (note that we are converting `yri_ceu` in the R namespace, not the one on the Python namespace). We delete the column that indicates the name of the paired FASTQ file on the pandas DataFrame and copy it back to the R namespace. If you print the column names of the new R DataFrame, you will see that `PAIRED_FASTQ` is missing.

## There's more...

It's worth repeating that the advances in the Python software ecology are occurring at a breakneck pace. This means that if a certain functionality is not available today, it might be released sometime in the near future. So, if you are developing a new project, be sure to check for the very latest developments on the Python front before using functionality from an R package.

There are plenty of R packages for Bioinformatics in the Bioconductor project (<http://www.bioconductor.org/>). This should probably be your first port of call in the R world for bioinformatics functionalities. However, note that there are many R Bioinformatics packages that are not on Bioconductor, so be sure to search the wider R packages on Comprehensive R Archive Network (CRAN) (refer to CRAN at <http://cran.r-project.org/>).

There are plenty of plotting libraries for Python. Matplotlib is the most common library, but you also have a plethora of other choices. In the context of R, it's worth noting that there is a `ggplot2`-like implementation for Python based on the Grammar of Graphics description language for charts, and this is called—surprise, surprise—`ggplot!` (<http://yhat.github.io/ggpy/>).

## See also

- There are plenty of tutorials and books on R; check the R web page (<http://www.r-project.org/>) for documentation.
- For Bioconductor, check the documentation at [http://manuals.bioinformatics.ucr.edu/home/R\\_BioCondManual](http://manuals.bioinformatics.ucr.edu/home/R_BioCondManual)
- If you work with NGS, you might also want to check high throughput sequence analysis with Bioconductor at <http://manuals.bioinformatics.ucr.edu/home/ht-seq>.

- The rpy library documentation is your Python gateway to R, and can be found at <https://rpy2.bitbucket.io/>.
- The Grammar of Graphics is described in a book aptly named The Grammar of Graphics, Leland Wilkinson, Springer.
- In terms of data structures, similar functionality to R can be found in the pandas library. You can find some tutorials at <http://pandas.pydata.org/pandas-docs/dev/tutorials.html>. The book, Python for Data Analysis, Wes McKinney, O'Reilly Media, is also an alternative to consider.

## Performing R magic with Jupyter Notebook

You have probably heard of, and maybe used, the Jupyter Notebook. Among many other features, Jupyter provides a framework of extensible commands called magics (actually, this only works with the IPython kernel of Jupyter, but that is the one we are concerned with), which allow you to extend the language in many useful ways. There are magic functions to deal with R. As you will see in our example, it makes R interfacing much more declarative and easy. This recipe will not introduce any new R functionalities, but hopefully, it will make it clear how IPython can be an important productivity boost for scientific computing in this regard.

### Getting ready

You will need to follow the previous Getting ready steps of the Interfacing with R via rpy2 recipe. The Notebook is Chapter01/R\_magic.ipynb. The Notebook is more complete than the recipe presented here, and includes more chart examples. For brevity here, we will only concentrate on the fundamental constructs to interact with R using magics.

### How to do it...

This recipe is an aggressive simplification of the previous one because it illustrates the conciseness and elegance of R magics:

The first thing you need to do is load R magics and ggplot2:

```
import rpy2.robjects as robjects
import rpy2.robjects.lib.ggplot2 as ggplot2%load_ext rpy2.ipython
```

Note that the % starts an IPython-specific directive. Just as a simple example, you can write %R print(c(1, 2)) on a Jupyter cell.

Check out how easy it is to execute the R code without using the robjects package. Actually, rpy2 is being used to look under the hood.

Let's ~~read~~ the sequence.index file that was downloaded in the previous recipe:

```
%%R
seq.data <- read.delim('sequence.index', header=TRUE,
stringsAsFactors=FALSE)
seq.data$READ_COUNT <- as.integer(seq.data$READ_COUNT)
seq.data$BASE_COUNT <- as.integer(seq.data$BASE_COUNT)
```

You can then specify that the whole cell should be interpreted as an R code by using %%R (note the double %%).

We ~~can~~ now transfer the variable to the Python namespace:

```
seq_data = %R seq.data
print(type(seq_data)) # pandas dataframe!
```

The type of the DataFrame is not a standard Python object, but a pandas DataFrame. This is a departure from previous versions of the R magic interface.

As we ~~have~~ a pandas DataFrame, we can operate on it quite easily using pandas' interface:

```
my_col = list(seq_data.columns).index("CENTER_NAME")
seq_data['CENTER_NAME'] = seq_data['CENTER_NAME'].apply(lambda x:
x.upper())
```

Let's ~~put~~ this DataFrame back in the R namespace, as follows:

```
%R -i seq_data
%R print(colnames(seq_data))
```

The -i argument informs the magic system that the variable that follows on the Python space is to be copied in the R namespace. The second line just shows that the DataFrame is indeed available in R. The name that we are using is different from the original—it's seq\_data instead of seq.data.

Let's do some final cleanup (for details, see the previous recipe) and print the same bar chart as before:

```
%%R
bar <- ggplot(seq_data) + aes(factor(CENTER_NAME)) + geom_bar() +
theme(axis.text.x = element_text(angle = 90, hjust = 1))
print(bar)
```

The R magic system also allows you to reduce code, as it changes the behavior of the interaction of R with IPython. For example, in the ggplot2 code of the previous recipe, you do not need to use the .png and dev.off R functions, as the magic system will take care of this for you. When you tell R to print a chart, it will magically appear in your Notebook or graphical console.

## There's more...

The R magics have seemed to have changed quite a lot over time in terms of interface. For example, I updated the R code for the first edition of this book a few times. The current version of DataFrame assignment returns pandas objects, which is a major change. Be careful with the version of Jupyter that you use as the %R code can be quite different. If this code does not work and you are using an older version, consult the Notebooks of the first edition of this book, as they might help.

## See also

- For basic instructions on IPython magics, see <https://github.com/ipython/ipython/blob/master/examples/IPython%20Kernel/Script%20Magics.ipyn and https://github.com/ipython/ipython/blob/master/examples/IPython%20Kernel/Cell%20Magics.ipynb>
- A list of default extensions is available at <https://ipython.readthedocs.io/en/stable/config/extensions/index.html>
- A list of third-party magic extensions can be found at <https://github.com/ipython/ipython/wiki/Extensions-Index>

# 2

# Next-Generation Sequencing

In this chapter, we will cover the following recipes:

- Accessing GenBank and moving around NCBI databases
- Performing basic sequence analysis
- Working with modern sequence formats
- Working with alignment data
- Analyzing data in VCF
- Studying genome accessibility and filtering SNP data
- Processing NGS data with HTSeq

## Introduction

Next-generation sequencing (NGS) is one of the fundamental technological developments of the decade in life sciences. Whole genome sequencing (WGS), RAD-Seq, RNA-Seq, Chip-Seq, and several other technologies are routinely used to investigate important biological problems. These are also called high-throughput sequencing technologies, and with good reason: they generate vast amounts of data that needs to be processed. NGS is the main reason that computational biology has become a big-data discipline. More than anything else, this is a field that requires strong bioinformatics techniques.

Here, we will not discuss each individual NGS technique per se (this would require a whole book on its own). We will use an existing WGS dataset and the 1,000 Genomes Project to illustrate the most common steps necessary to analyze genomic data. The recipes presented here will be easily applicable to other genomic sequencing approaches. Some of them can also be used for transcriptomic analysis (for example, RNA-Seq). The recipes are also species-independent, so you will be able to apply them to any other species for which you have sequenced data. The biggest difference in processing data from different species is related to genome size, diversity, and the quality of the assembled genome (if it exists for your species). These will not affect the automated Python part of NGS processing much. In any case, we will discuss different genomes in the next chapter, Chapter 3, Working with Genomes.

As this is not an introductory book, you are expected to know at least what FASTA, FASTQ, Binary Alignment Map (BAM), and Variant Call Format (VCF) files are. I will also make use of the basic genomic terminology without introducing it (such as exomes, nonsynonymous mutations, and so on). You are required to be familiar with basic Python. We will leverage this knowledge to introduce the fundamental libraries in Python to perform the NGS analysis. Here, we will follow the flow of a standard bioinformatics pipeline.

However, before we delve into real data from a real project, let's get comfortable with accessing existing genomic databases and basic sequence processing—a simple start before the storm.

## Accessing GenBank and moving around NCBI databases

Although you may have your own data to analyze, you will probably need existing genomic datasets. Here, we will look at how to access such databases at the National Center for Biotechnology Information (NCBI). We will not only discuss GenBank, but also other databases at NCBI. Many people refer (wrongly) to the whole set of NCBI databases as GenBank, but NCBI includes the nucleotide database and many others, for example, PubMed.

As sequencing analysis is a long subject, and this book targets intermediate to advanced users, we will not be very exhaustive with a topic that is, at its core, not very complicated. Nonetheless, it's a good warm-up for the more complex recipes that we will see at the end of this chapter.

## Getting ready

We will use Biopython, which you installed in Chapter 1, Python and the Surrounding Software Ecology. Biopython provides an interface to Entrez, the data retrieval system made available by NCBI. This recipe is made available in the Chapter02/Accessing\_Databases.ipynb Notebook.

You will be accessing a live API from NCBI. Note that the performance of the system may vary during the day. Furthermore, you are expected to be a "good citizen" while using it. You will find some recommendations at [http://www.ncbi.nlm.nih.gov/books/NBK25497/#chapter2.Usage\\_Guidelines\\_and\\_Requirements](http://www.ncbi.nlm.nih.gov/books/NBK25497/#chapter2.Usage_Guidelines_and_Requirements). Notably, you are required to specify an email address with your query. You should try to avoid large number of requests (100 or more) during peak times (between 9.00 a.m. and 5.00 p.m. American Eastern Time on weekdays), and do not post more than three queries per second (Biopython will take care of this for you). It's not only good citizenship, but you risk getting blocked if you over use NCBI's servers (a good reason to give a real email address, because NCBI may try to contact you).



## How to do it...

Now, let's look at how we can search and fetch data from NCBI databases:

We will start by importing the relevant module and configuring the email address:

```
from Bio import Entrez, SeqIO  
Entrez.email = 'put@your.email.here'
```

We will also import the module to process sequences. Do not forget to put in the correct email address.

We will now try to find the chloroquine resistance transporter (CRT) gene in *Plasmodium falciparum* (the parasite that causes the deadliest form of malaria) on the nucleotide database:

```
handle = Entrez.esearch(db='nucleotide', term='CRT[Gene Name] AND  
"Plasmodium falciparum"[Organism]')  
rec_list = Entrez.read(handle)  
if rec_list['RetMax'] < rec_list['Count']:  
    handle = Entrez.esearch(db='nucleotide', term='CRT[Gene Name]  
AND "Plasmodium falciparum"[Organism], retmax=rec_list['Count'])  
    rec_list = Entrez.read(handle)
```

We will search the nucleotide database for our gene and organism (for the syntax of the search string, check the NCBI website). Then, we will read the result that is returned. Note that the standard search will limit the number of record references to 20, so if you have more, you may want to repeat the query with an increased maximum limit. In our case, we will actually override the default limit with retmax. The Entrez system provides quite a few sophisticated ways to retrieve large number of results (for more information, check the Biopython or NCBI Entrez documentation). Although you now have the IDs of all of the records, you still need to retrieve the records properly.

Now, let's try to retrieve all of these records. The following query will download all matching nucleotide sequences from GenBank, which is 481, at the time of writing this book. You probably won't want to do this all the time:

```
id_list = rec_list['IdList']  
hdl = Entrez.efetch(db='nucleotide', id=id_list, rettype='gb')
```

Well, in this case, go ahead and do it. However, be careful with this technique, because you will retrieve a large amount of complete records, and some of them will have fairly large sequences inside. You risk downloading a lot of data (which would be a strain both on your side and on NCBI servers).

There are several ways around this. One way is to make a more restrictive query and/or download just a few at a time and stop when you have found the one that you need. The precise strategy will depend on what you are trying to achieve. In any case, we will retrieve a list of records in the GenBank format (which includes sequences, plus a lot of interesting metadata).

Let's read and parse the result:

```
recs = list(SeqIO.parse(hdl, 'gb'))
```

Note that we have converted an iterator (the result of SeqIO.parse) to a list. The advantage of doing this is that we can use the result as many times as we want (for example, iterate many times over), without repeating the query on the server. This saves time, bandwidth, and server usage if you plan to iterate many times over. The disadvantage is that it will allocate memory for all records. This will not work for very large datasets; you might not want to do this conversion genome-wide as in the next chapter, Chapter 3, Working with Genomes. We will return to this topic in the last part of this book. If you are doing interactive computing, you will probably prefer to have a list (so that you can analyze and experiment with it multiple times), but if you are developing a library, an iterator will probably be the best approach.

We will now just concentrate on a single record. This will only work if you used the exact same preceding query:

```
for rec in recs:  
    if rec.name == 'KM288867':  
        break  
    print(rec.name)  
    print(rec.description)
```

The `rec` variable now has our record of interest. The `rec.description` file will contain its human-readable description.

Now, let's extract some sequence features, which contain information such as gene products and exon positions on the sequence:

```
for feature in rec.features:  
    if feature.type == 'gene':  
        print(feature.qualifiers['gene'])  
    elif feature.type == 'exon':  
        loc = feature.location  
        print(loc.start, loc.end, loc.strand)  
    else:  
        print('not processed:\n%s' % feature)
```

If the feature.type is gene, we will print its name, which will be in the qualifiers dictionary. We will also print all the locations of exons. Exons, as with all features, have locations in this sequence: a start, an end, and the strand from where they are read. While all the start and end positions for our exons are ExactPosition, note that Biopython supports many other types of positions. One type of position is BeforePosition, which specifies that a location point is before a certain sequence position. Another type of position is BetweenPosition, which gives the interval for a certain location start/end. There are quite a few more position types; these are just some examples.

Coordinates will be specified in such a way that you will be able to retrieve the sequence from a Python array with ranges easily, so generally, the start will be one before the value on the record, and the end will be equal. The issue of coordinate systems will be revisited in future recipes.

For other feature types, we simply print them. Note that Biopython will provide a human-readable version of the feature when you print it.

We will now look at the annotations on the record, which are mostly metadata that is not related to the sequence position:

```
for name, value in rec.annotations.items():
    print('%s=%s' % (name, value))
```

Note that some values are not strings; they can be numbers or even lists (for example, the taxonomy annotation is a list).

Last but not least, you can access the fundamental piece of information, the sequence:

```
print(len(rec.seq))
```

The sequence object will be the main topic of our next recipe.

## There's more...

There are many more databases at NCBI. You will probably want to check the Sequence Read Archive (SRA) database (previously known as Short Read Archive) if you are working with NGS data. The SNP database contains information on single-nucleotide polymorphisms (SNPs), whereas the protein database has protein sequences, and so on. A full list of databases in Entrez is linked in the See also section of this recipe.

Another database that you probably already know about with regard to NCBI is PubMed, which includes a list of scientific and medical citations, abstracts, and even full texts. You can also access it via Biopython. Furthermore, GenBank records often contain links to PubMed. For example, we can perform this on our previous record, as shown here:

```
from Bio import Medline
refs = rec.annotations['references']
for ref in refs:
    if ref.pubmed_id != "":
        print(ref.pubmed_id)
        handle = Entrez.efetch(db='pubmed', id=[ref.pubmed_id],
                               rettype='medline', retmode='text')
        records = Medline.parse(handle)
        for med_rec in records:
            for k, v in med_rec.items():
                print('%s: %s' % (k, v))
```

This will take all reference annotations, check whether they have a PubMed identifier, and then access the PubMed database to retrieve the records, parse them, and then print them.

The output per record is a Python dictionary. Note that there are many references to external databases on a typical GenBank record.

Of course, there are many other biological databases outside NCBI, such as Ensembl (<http://www.ensembl.org>) and UCSC Genome Bioinformatics (<http://genome.ucsc.edu/>). The support for many of these databases in Python will vary a lot.

An introductory recipe on biological databases would not be complete without at least a passing reference to BLAST. Basic local alignment search tool (BLAST) is an algorithm that assesses the similarity of sequences. NCBI provides a service that allows you to compare your sequence of interest against its own database. Of course, you can use have your local BLAST database instead of using NCBI's service. Biopython provides extensive support for this, but as this is too introductory, I will just refer you to the Biopython tutorial.

## See also

- You can find more examples on the Biopython tutorial at <http://biopython.org/DIST/docs/tutorial/Tutorial.html>
- A list of accessible NCBI databases can be found at <http://www.ncbi.nlm.nih.gov/gquery/>
- A great Q&A site where you can find help for your problems with databases and sequence analysis is Biostars (<http://www.biostars.org>); you can use it for all of the content in this book, not just for this recipe

## Performing basic sequence analysis

We will now do some basic analysis on DNA sequences. We will work with FASTA files and do some manipulation, such as reverse complementing or transcription. As with the previous recipe, we will use Biopython, which you installed in Chapter 1, Python and the Surrounding Software Ecology. These two recipes provide you with the necessary introductory building blocks with which we will perform all the modern NGS analysis and then genome processing in this and the next chapter, Chapter 3, Working with Genomes.

## Getting ready

If you are using Jupyter Notebook, then open Chapter02/Basic\_Sequence\_Processing.ipynb. If not, you will need to download a FASTA sequence. We will use the human Lactase (LCT) gene as an example; you can get this using your knowledge from the previous recipe, by using the Entrez research interface:

```
from Bio import Entrez, SeqIO
Entrez.email = "your@email.here"
hdl = Entrez.efetch(db='nucleotide', id=['NM_002299'], rettype='fasta') #
Lactase gene
seq = SeqIO.read(hdl, 'fasta')
```

Note that our example sequence is available on the Biopython sequence record.

## How to do it...

Let's take a look at the following steps:

As our sequence of interest is available in a Biopython sequence object, let's start by saving it to a FASTA file on our local disk:

```
from Bio import SeqIO
w_hdl = open('example.fasta', 'w')
w_seq = seq[11:5795]
SeqIO.write([w_seq], w_hdl, 'fasta')
w_hdl.close()
```

The SeqIO.write function takes a list of sequences to write (not just a single one). Be careful with this idiom. If you want to write many sequences (and you could easily write millions with NGS), do not use a list (as shown in the preceding code), because this will allocate massive amounts of memory. Either use an iterator, or use the SeqIO.write function several times with a subset of the sequence on each write.

In most situations, you will actually have the sequence on the disk, so you will be interested in reading it:

```
recs = SeqIO.parse('example.fasta', 'fasta')
for rec in recs:
    seq = rec.seq
    print(rec.description)
    print(seq[:10])
    print(seq.alphabet)
```

Here, we are concerned with processing a single sequence, but FASTA files can contain multiple records. The Python idiom to perform this is quite easy. To read a FASTA file, you just use standard iteration techniques, as shown in the following code. For our example, the preceding code will print the following:

```
NM_002299.3 Homo sapiens lactase (LCT), mRNA
ATGGAGCTGT
SingleLetterAlphabet()
```

Note that we printed seq[:10]. The sequence object can use typical array slices to get part of a sequence.

We will now change the alphabet of our sequence:

```
from Bio import Seq
from Bio.Alphabet import IUPAC
seq = Seq.Seq(str(seq), IUPAC.unambiguous_dna)
```

Probably the biggest value of the sequence object (compared to a simple string) comes from the alphabet information. The sequence object will be able to impose useful constraints and operations on the underlying string, based on the expected alphabet. The original alphabet in the FASTA file is not very informative, but in this case, we know that we have a DNA alphabet. Therefore, we will create a new sequence with a more informative alphabet.

As we now have an unambiguous DNA, we can transcribe it as follows:

```
rna = seq.transcribe()
print(rna)
```

Note that the seq constructor takes a string, not a sequence. You will see that the alphabet of the rna variable is now IUPACUnambiguousRNA.

Finally, we can translate our gene into a protein:

```
prot = seq.translate()
print(prot)
```

Now, we have a protein alphabet with the annotation that there is a stop codon (this means that our protein is complete).

## There's more...

Much more can be said about the management of sequences in Biopython, but this is mostly introductory material that you can find in the Biopython tutorial. I think it's important to give you a taste of sequence management, mostly for completion purposes. To support readers who might have some experience in other fields of bioinformatics, but are just starting with sequence analysis, there are, nonetheless, a few points that you should be aware of:

- When you perform a RNA translation to get your protein, be sure to use the correct genetic code. Even if you are working with "common" organisms (such as humans), remember that the mitochondrial genetic code is different.

- Biopython's Seq object is much more flexible than what's shown here. For some good examples, refer to the Biopython tutorial. However, this recipe will be enough for the work we need to do with FASTQ files (see the next recipe).
- To deal with strand-related issues, there are, as expected, sequence functions like reverse\_complement.

## See also

- Genetic codes known to Biopython are the ones specified by NCBI, available at <http://www.ncbi.nlm.nih.gov/Taxonomy/Utils/wprintgc.cgi>
- As in the previous recipe, the Biopython tutorial is your main port of call and is available at <http://biopython.org/DIST/docs/tutorial/Tutorial.html>
- Be sure to also check the Biopython SeqIO page at <http://biopython.org/wiki/SeqIO>

# Working with modern sequence formats

Here, we will work with FASTQ files, the standard format output used by modern sequencers. You will learn how to work with quality scores per base and also consider the variations in output coming from different sequencing machines and databases. This is the first recipe that will use real data (big data) from the Human 1,000 Genomes Project. We will start with a brief description of the project.

## Getting ready

The Human 1,000 Genomes Project aims to catalog worldwide human genetic variation and takes advantage of modern sequencing technology to do WGS. This project makes all data publicly available, which includes output from sequencers, sequence alignments, and SNP calls, among many other artifacts. The name "1,000 Genomes" is actually a misnomer, because it currently includes more than 2,500 samples. These samples are divided into 26 populations, spanning the whole planet. We will mostly use data from four populations: African Yorubans (YRI), Utah Residents with Northern and Western European Ancestry (CEU), Japanese in Tokyo (JPT), and Han Chinese in Beijing (CHB). The reason we chose these specific populations is because they were the first ones that came from HapMap, an old project with similar goals. They used genotyping arrays to find out more about the quality of this subset. We will revisit the 1,000 Genomes, and HapMap projects in Chapter 4, Population Genetics.

Next-generation datasets are generally very large. As we will be using real data, some of the files that you will download will be big. While I have tried to choose the smallest real examples possible, you will still need a good network connection and a considerably large amount of disk space. Waiting for the download will probably be your biggest hurdle in this recipe, but data management is a serious problem with NGS. In real life, you will need to budget time for data transfer, allocate disk space (which can be financially costly), and consider backup policies. The most common initial mistake with NGS is to think that these problems are trivial, but they are not. An operation such as copying a set of BAM files to a network, or even to your computer, will become a headache. Be prepared. After downloading large files, at the very least, you should check that the size is correct. Some databases offer MD5 checksums. You can compare these checksums with the ones on the files you downloaded by using tools like md5sum.



If you use Jupyter Notebook, do not forget to download the data, as specified on the first cell of Chapter02/Working\_with\_FASTQ.ipynb. If not, download the SRR003265.filt.fastq.gz file, which is linked in <https://github.com/PacktPublishing/Bioinformatics-with-Python-Cookbook-Second-Edition/blob/master/Datasets.ipynb>. This is a fairly small file (27 MB) and represents part of the sequenced data of a Yoruban female (NA18489). If you refer to the 1,000 Genomes Project, you will see that the vast majority of FASTQ files are much bigger (up to two orders of magnitude bigger).

The processing of FASTQ sequence files will mostly be performed using Biopython.

## How to do it...

Before we start coding, let's take a look at the FASTQ file, in which you will have many records, as shown in the following code:

```
@SRR003258.1 30443AAXX:1:1:1053:1999 length=51
ACCCCCCCCCACCCCCCCCCCCCCCCCCCCCCACACACACCAACAC
+
=IIIIIIII5IIIIII>III+GIIIIIIIIII(IIII01&III
```

Line 1 starts with @, followed by a sequence identifier and a description string. The description string will vary from a sequencer or a database source, but will normally be amenable to automated parsing.

The second line has the sequence DNA, which is just like a FASTA file. The third line is a +, sometimes followed by the description line on the first line.

The fourth line contains quality values for each base that's read on line two. Each letter encodes a Phred quality score ([http://en.wikipedia.org/wiki/Phred\\_quality\\_score](http://en.wikipedia.org/wiki/Phred_quality_score)), which assigns a probability of error to each read. This encoding can vary a bit among platforms. Be sure to check for this on your specific platform.

Let's take a look at the following steps:

Let's open the file:

```
import gzip
from Bio import SeqIO
recs = SeqIO.parse(gzip.open('SRR003265.filt.fastq.gz'),'rt',
encoding='utf-8'), 'fastq')
rec = next(recs)
print(rec.id, rec.description, rec.seq)
print(rec.letter_annotations)
```

We will open a GZIP file so that we can use the Python gzip module. We will also specify the fastq format. Note that some variations in this format will impact the interpretation of the Phred quality scores. You may want to specify a slightly different format. Refer to <http://biopython.org/wiki/SeqIO> for all formats.



You should usually store your FASTQ files in a compressed format. Not only do you gain a lot of disk space, as these are text files, but you probably also gain some processing time. Although decompressing is a slow process, it can still be faster than reading a much bigger (uncompressed) file from a disk.

We print the standard fields and quality scores from the previous recipe into rec.letter\_annotations. As long as we choose the correct parser, Biopython will convert all the Phred encoding letters to logarithmic scores, which we will use soon.

For now, don't do this:

```
recs = list(recs) # do not do it!
```

Although this might work with some FASTA files (and with this very small FASTQ file), if you do something like this, you will allocate memory so that you can load the complete file in memory. With an average FASTQ file, this is the best way to crash your computer. As a rule, always iterate over your file. If you have to perform several operations over it, you have two main options. The first option is perform a single iteration or all operations at once. The second option is open a file several times and repeat the iteration.

Now, let's take a look at the distribution of nucleotide reads:

```
from collections import defaultdict
recs = SeqIO.parse(gzip.open('SRR003265.filt.fastq.gz', 'rt',
encoding='utf-8'), 'fastq')
cnt = defaultdict(int)
for rec in recs:
    for letter in rec.seq:
        cnt[letter] += 1
tot = sum(cnt.values())
for letter, cnt in cnt.items():
    print('%s: %.2f %d' % (letter, 100. * cnt / tot, cnt))
```

We will reopen the file again and use defaultdict to maintain a count of nucleotide references in the FASTQ file. If you have never used this Python standard dictionary type, you may want to consider it because it removes the need to initialize dictionary entries, assuming default values for each type.



Note that there is a residual number for N calls. These are calls in which a sequencer reports an unknown base. In our FASTQ file example, we have cheated a bit because we used a filtered file (the fraction of N calls will be quite low). Expect a much bigger number of N calls in a file that came out of the sequencer unfiltered. In fact, you can even expect something more with regards to the spatial distribution of N calls.

Let's plot the distribution of Ns according to its read position:

```
%matplotlib inline
import seaborn as sns
import matplotlib.pyplot as plt
recs = SeqIO.parse(gzip.open('SRR003265.filt.fastq.gz', 'rt',
encoding='utf-8'), 'fastq')
n_cnt = defaultdict(int)
for rec in recs:
    for i, letter in enumerate(rec.seq):
        pos = i + 1
        if letter == 'N':
```

```
n_cnt[pos] += 1
seq_len = max(n_cnt.keys())
positions = range(1, seq_len + 1)
fig, ax = plt.subplots(figsize=(16,9))
ax.plot(positions, [n_cnt[x] for x in positions])
ax.set_xlim(1, seq_len)
```

The first line only works on IPython and Jupyter Notebook (you should remove it on a standard Python implementation) and it will inline any plots. We then import the seaborn library. Although we do not use it explicitly at this point, this library has the advantage of making matplotlib plots look better, because it tweaks the default matplotlib style.

We then open the file to parse again (remember that you do not use a list, but iterate again). We iterate through the file and get the position of any references to N. Then, we plot the distribution of Ns as a function of the distance from the start of the sequence:

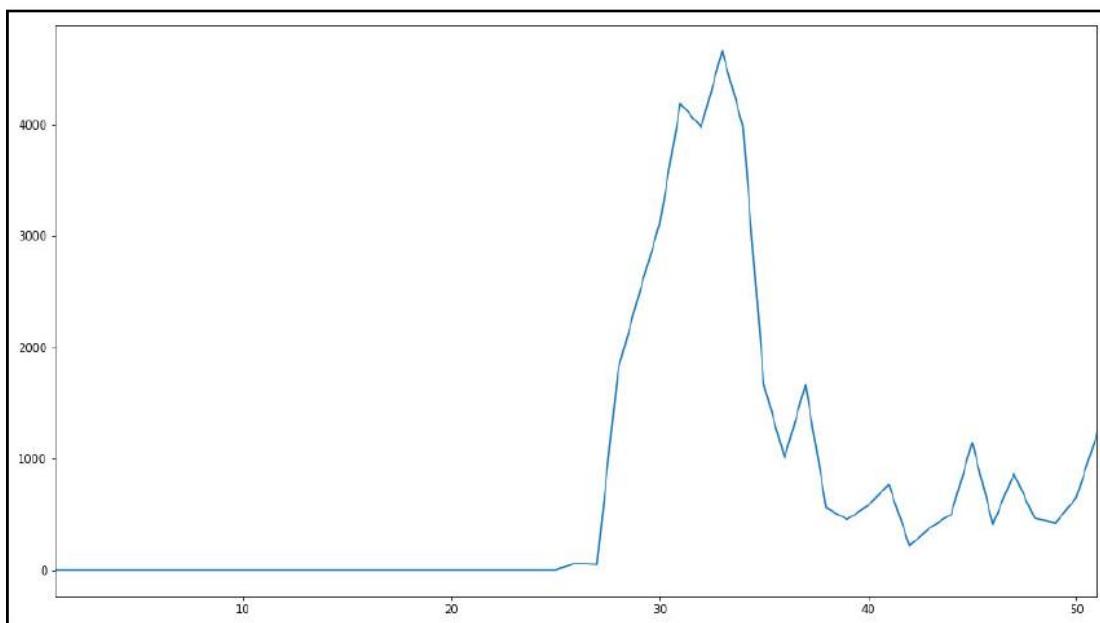


Figure 1: The number of N calls as a function of the distance from the start of the sequencer read

You will see that until position 25, there are no errors. This is not what you will get from a typical sequencer output. Our example file is already filtered, and the 1,000 genomes filtering rules enforce that no N calls can occur before position 25.

While we cannot study the behavior of Ns in this dataset before position 25 (feel free to use one of your own unfiltered FASTQ files with this code in order to see how Ns distribute across the read position), we can see that after position 25, the distribution is far from uniform. There is an important lesson here, which is that the quantity of uncalled bases is position-dependent. So, what about the quality of reads?

Let's study the distribution of Phred scores (that is, the quality of our reads):

```
recs = SeqIO.parse(gzip.open('SRR003265.filt.fastq.gz', 'rt',
    encoding='utf-8'), 'fastq')
cnt_qual = defaultdict(int)
for rec in recs:
    for i, qual in
        enumerate(rec.letter_annotations['phred_quality']):
        if i < 25:
            continue
        cnt_qual[qual] += 1
tot = sum(cnt_qual.values())
for qual, cnt in cnt_qual.items():
    print('%d: %.2f %d' % (qual, 100. * cnt / tot, cnt))
```

We will start by reopening the file (again) and initializing a default dictionary. We then get the phred\_quality letter annotation, but we ignore sequencing positions that are up to 24 base pairs from the start (because of the filtering of our FASTQ file, if you have an unfiltered file, you may want to drop this rule). We add the quality score to our default dictionary, and finally print it.

As a short reminder, the Phred quality score is a logarithmic representation of the probability of an accurate call. This probability is given as  $10^{-\frac{Q}{10}}$ . So, a Q of 10 represents a 90 percent call accuracy, 20 represents 99 percent call accuracy, and 30 will be 99.9 percent. For our file, the maximum accuracy will be 99.99 percent (40). In some cases, values of 60 are possible (99.9999 percent accuracy).



More interestingly, we can plot the distribution of qualities according to their read position:

```
recs = SeqIO.parse(gzip.open('SRR003265.filt.fastq.gz', 'rt',
    encoding='utf-8'), 'fastq')
qual_pos = defaultdict(list)
for rec in recs:
    for i, qual in
        enumerate(rec.letter_annotations['phred_quality']):
        if i < 25 or qual == 40:
            continue
        pos = i + 1
        qual_pos[pos].append(qual)
vps = []
poses = list(qual_pos.keys())
poses.sort()
for pos in poses:
    vps.append(qual_pos[pos])
fig, ax = plt.subplots(figsize=(16,9))
sns.boxplot(data=vps, ax=ax)
ax.set_xticklabels([str(x) for x in range(26, max(qual_pos.keys()) + 1)])
```

In this case, we will ignore both positions sequenced as 25 base pairs from the start (again, remove this rule if you have unfiltered sequencer data) and the maximum quality score for this file (40). However, in your case, you can consider starting your plotting analysis with the maximum. You may want to check the maximum possible value for your sequencer hardware. Generally, as most calls can be performed with maximum quality, you may want to remove them if you are trying to understand where quality problems lie.

Note that we are using seaborn's boxplot function; we are only using this because the output looks slightly better than the standard Matplotlib boxplot. If you prefer not to depend on seaborn, just use the stock matplotlib function. In this case, you will call `ax.boxplot(vps)` instead of `sns.boxplot(data=vps, ax=ax)`.

As expected, the distribution is not uniform, as shown in the following screenshot:

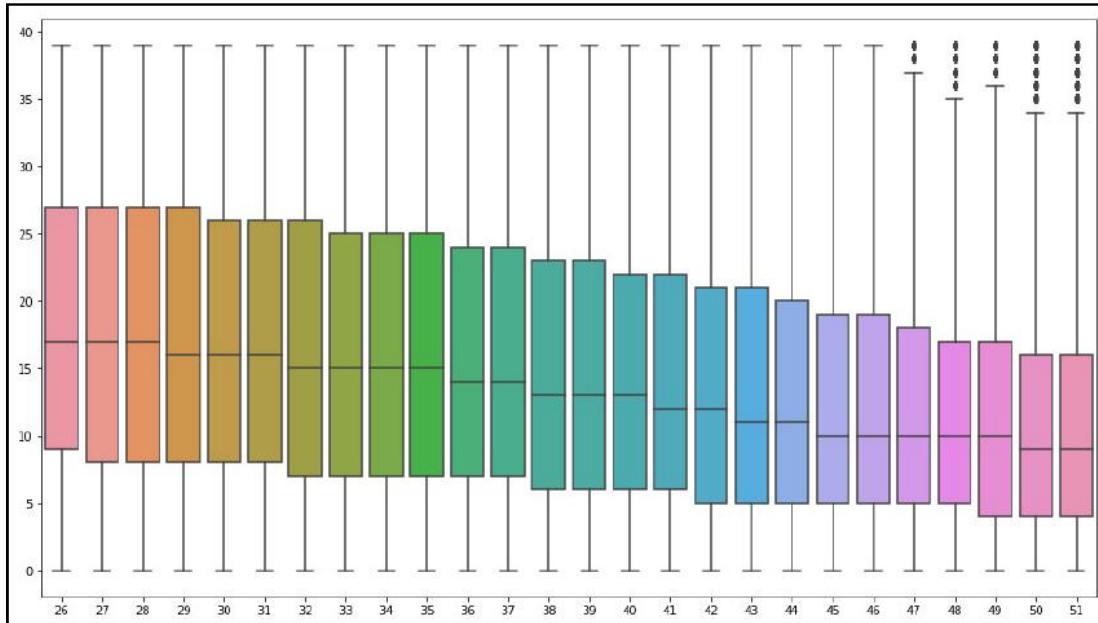


Figure 2: The distribution of Phred scores as a function of the distance from the start of the sequencer read

## There's more...

Although it's impossible to discuss all the variations of output coming from sequencer files, paired-end reads are worth mentioning because they are common and require a different processing approach. With paired-end sequencing, both ends of a DNA fragment are sequenced with a gap in the middle (called the insert). In this case, two files will be produced: X\_1.FASTQ and X\_2.FASTQ. Both files will have the same order and exact same number of sequences. The first sequence will be in X\_1 pairs with the first sequence of X\_2, and so on. With regards to the programming technique, if you want to keep the pairing information, you might perform something like this:

```
f1 = gzip.open('X_1.filt.fastq.gz', 'rt', encoding='utf-8')
f2 = gzip.open('X_2.filt.fastq.gz', 'rt', encoding='utf-8')
recs1 = SeqIO.parse(f1, 'fastq')
recs2 = SeqIO.parse(f2, 'fastq')
cnt = 0
```

```
for rec1, rec2 in zip(recs1, recs2):
    cnt +=1
print('Number of pairs: %d' % cnt)
```

The preceding code reads all pairs in order and just counts the number of pairs. You will probably want to do something more, but this exposes a dialect that is based on the Python zip function that allows you to iterate through both files simultaneously. Remember to replace X for your FASTQ prefix.



Note that the preceding code will most probably crash Python 2 as the zip function is eager in Python 2, (that is, it will read all records before needing them). Indeed, the lazy behavior of iterators in Python 3 is one of the many features that makes it more suitable for big data analysis. If you really need to use Python 2, then consider the itertools module, which provides lazy implementations of common iterators.

Finally, if you are sequencing human genomes, you may want to use sequencing data from Complete Genomics. In this case, read the There's more section in the next recipe, where we briefly discuss Complete Genomics data.

## See also

- The Wikipedia page ([http://en.wikipedia.org/wiki/FASTQ\\_format](http://en.wikipedia.org/wiki/FASTQ_format)) on the FASTQ format is quite informative
- You can find more information on the 1,000 Genomes Project at <http://www.1000genomes.org/>
- Information about the Phred quality score can be found at [http://en.wikipedia.org/wiki/Phred\\_quality\\_score](http://en.wikipedia.org/wiki/Phred_quality_score)
- Illumina provides a good introduction page to paired-end reads at <https://www.illumina.com/science/technology/next-generation-sequencing/paired-end-vs-single-read-sequencing.html>
- The paper Computational methods for discovering structural variation with next-generation sequencing from Medvedev et al on nature methods (<http://www.nature.com/nmeth/journal/v6/n11s/abs/nmeth.1374.html>); note that this is not open access

## Working with alignment data

After you receive your data from the sequencer, you will normally use a tool such as Burrows-Wheeler Aligner (bwa) to align your sequences to a reference genome. Most users will have a reference genome for their species. You can read more on reference genomes in the next chapter, Chapter 3, Working with Genomes.

The most common representation for aligned data is the sequence alignment map (SAM) format. Due to the massive size of most of these files, you will probably work with its compressed version (BAM). The compressed format is indexable for extremely fast random access (for example, to speedily find alignments to a certain part of a chromosome). Note that you will need to have an index for your BAM file, which is normally created by the tabix utility of SAMtools. SAMtools is probably the most widely-used tool for manipulating SAM/BAM files.

## Getting ready

As discussed in the previous recipe, we will use data from the 1,000 Genomes Project. We will use the exome alignment for chromosome 20 of female NA18489. This is just 312 MB. The whole exome alignment for this individual is 14.2 GB, and the whole genome alignment (at a low coverage of 4x) is 40.1 GB. This data is a paired-end with reads of 76 bp. This is common nowadays, but slightly more complex to process. We will take this into account. If your data is not paired, just simplify the following recipe appropriately.

As usual, if you use Notebook, the cell at the top of Chapter02/Working\_with\_BAM.ipynb will download the data for you. If you don't use Notebooks, get the data from our dataset list at <https://github.com/PacktPublishing/Bioinformatics-with-Python-Cookbook-Second-Edition/blob/master/Datasets.ipynb>. The files you will want are NA18490\_20\_exome.bam and NA18490\_20\_exome.bam.bai.

We will use pysam, a Python wrapper to the SAMtools C API. This was installed in Chapter 1, Python and the Surrounding Software Ecology.

## How to do it...

Before you start coding, note that you can inspect the BAM file using samtools view -h (this is if you have SAMtools installed, which we recommend, even if you use the Genome Analysis Toolkit (GATK) or something else for variant calling). We suggest that you take a look at the header file and the first few records. The SAM format is too complex to be described here. There is plenty of information on the internet about it; nonetheless, sometimes, there's some really interesting information buried in these header files.



One of the most complex operations in NGS is to generate good alignment files from raw sequence data. It not only calls the aligner, but also cleans up data. Now, in the @PG headers of high quality BAM files, you will find the actual command lines used for most, if not all, of the procedures used to generate this BAM file. In our example BAM file, you will find all the information needed to run bwa, SAMtools, the GATK IndelRealigner, and the Picard application suite to clean up data. Remember that while you can generate BAM files easily, the programs after it will be quite picky in terms of correctness of the BAM input. For instance, if you use GATK's variant caller to generate genotype calls, the files will have to be extensively cleaned. The header of other BAM files can thus provide you with the best way to generate yours. A final recommendation is that if you do not work with human data, try to find good BAMs for your species, because the parameters of a given program may be slightly different. Also, if you use something other than the WGS data, check for similar types of sequencing data.

Let's take a look at the following steps:

Let's inspect the header files:

```
import pysam
bam =
pysam.AlignmentFile('NA18489.chrom20.ILLUMINA.bwa.YRI.exome.2012121
1.bam', 'rb')
headers = bam.header
for record_type, records in headers.items():
    print(record_type)
    for i, record in enumerate(records):
        if type(record) == dict:
            print('\t%d' % (i + 1))
            for field, value in record.items():
                print('\t\t%s\t%s' % (field, value))
        else:
            print('\t\t%s' % record)
```

The header is represented as a dictionary (where the key is the record\_type). As there can be several instances of the same record\_type, the value of the dictionary is a list (where each element is, again, a dictionary, or sometimes a string containing tag/value pairs).

We will now inspect a single record. The amount of data per record is quite complex. Here, we will focus on some of the fundamental fields for paired-end reads. Check the SAM file specification and the pysam API documentation for more details:

```
for rec in bam:  
    if rec.cigarstring.find('M') > -1 and rec.cigarstring.find('S')  
        > -1 and not rec.is_unmapped and not rec.mate_is_unmapped:  
        break  
    print(rec.query_name, rec.reference_id,  
          bam.getrname(rec.reference_id), rec.reference_start,  
          rec.reference_end)  
    print(rec.cigarstring)  
    print(rec.query_alignment_start, rec.query_alignment_end,  
          rec.query_alignment_length)  
    print(rec.next_reference_id,  
          rec.next_reference_start, rec.template_length)  
    print(rec.is_paired, rec.is_proper_pair, rec.is_unmapped,  
          rec.mapping_quality)  
    print(rec.query_qualities)  
    print(rec.query_alignment_qualities)  
    print(rec.query_sequence)
```

Note that the BAM file object is iterable over its records. We will transverse it until we find a record whose CIGAR string contains a match and a soft clip.

The CIGAR string gives an indication of the alignment of individual bases. The clipped part of the sequence is the part that the aligner failed to align (but is not removed from the sequence). We will also want the read, its mate ID, and position (of the pair, as we have paired-end reads) that was mapped to the reference genome.

First, we print the query template name, followed by the reference ID. The reference ID is a pointer to the name of the sequence on the given references on the lookup table of references. An example will make this clear. For all records on this BAM file, the reference ID is 19 (a non-informative number), but if you apply `pysam.getname(19)`, you will get 20, which is the name of the chromosome. So, do not confuse the reference ID (in this case, 19) with the name of the chromosome (20). This is then followed by the reference start and reference end. `pysam` is 0-based, not 1-based. So, be careful when you convert coordinates to other libraries. You will notice that the start and end for this case is 59,996 and 60,048, which means an alignment of 52 bases. Why is there only 52 bases when the read size is 76 (remember, the read size used in this BAM file)? The answer can be found on the CIGAR string, which in our case will be 52M24S, which is a 52 bases match, followed by 24 bases that were soft-clipped.

Then, we print where the alignment starts and ends and calculate its length. By the way, you can compute this by looking at the CIGAR string. It starts at 0 (as the first part of the read is mapped) and ends at 52. The length is 76 again.

Now, we query the mate (something that you will only do if you have paired-end reads). We get its reference ID (as shown in the previous code), its start position, and a measure of the distance between both pairs. This measure of distance only makes sense if both mates are mapped to the same chromosome.

We then plot the Phred score (refer to the previous recipe on Phred scores) for the sequence, and then only for the aligned part. Finally, we print the sequence (don't forget to do this!). This is the complete sequence, not the clipped one (of course, you can use the preceding coordinates to clip).

Now, let's plot the distribution of the successfully mapped positions in a subset of sequences in the BAM file:

```
%matplotlib inline
import seaborn as sns
import matplotlib.pyplot as plt
counts = [0] * 76
for n, rec in enumerate(bam.fetch('20', 0, 10000000)):
    for i in range(rec.query_alignment_start,
                    rec.query_alignment_end):
        counts[i] += 1
freqs = [x / (n + 1.) for x in counts]
fig, ax = plt.subplots(figsize=(16,9))
ax.plot(range(1, 77), freqs)
```

We will start by initializing an array to keep the count for the entire 76 positions. Note that we then fetch only the records for chromosome 20 between positions 0 and 10 Mbp. We will just use a small part of the chromosome here. It's fundamental to have an index (generated by tabix) for these kinds of fetch operations; the speed of execution will be completely different.

We traverse all records in the 10 Mbp boundary. For each boundary, we get the alignment start and end, and increase the counter of mappability among the positions that were aligned. Finally, we convert this into frequencies, and then plot it, as shown in the following graph:

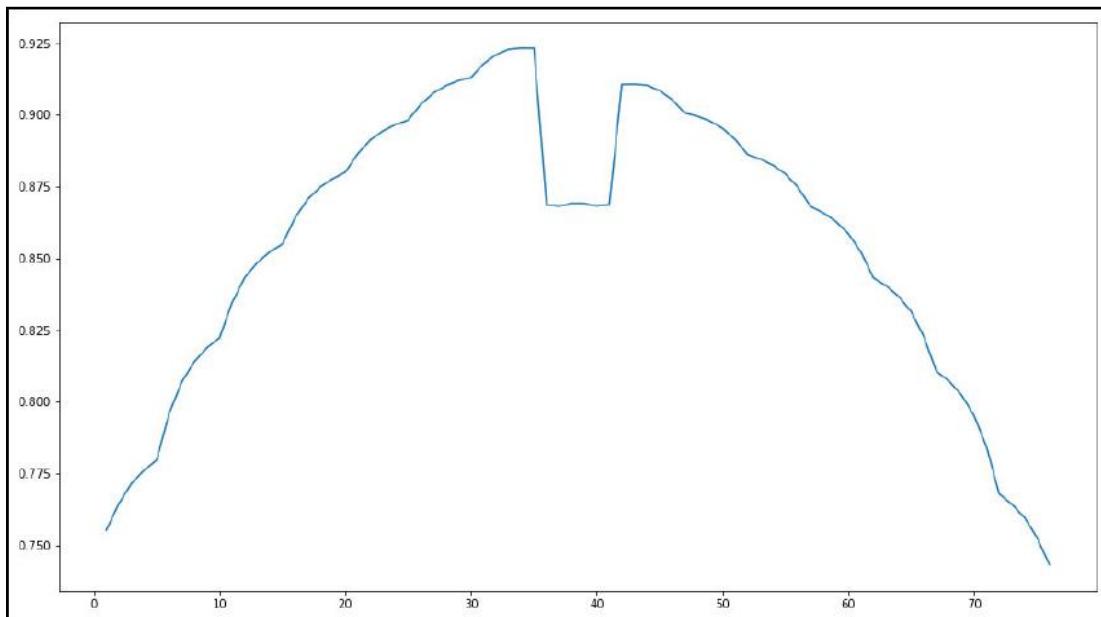


Figure 3: The percentage of mapped calls as a function of the position from the start of the sequencer read

It's quite clear that the distribution of mappability is far from being uniform; it's worse at the extremes, with a drop in the middle.

Finally, let's get the distribution of Phred scores across the mapped part of the reads. As you may suspect, this is probably not going to be uniform:

```
from collections import defaultdict
import numpy as np
phreds = defaultdict(list)
for rec in bam.fetch('20', 0, None):
    for i in range(rec.query_alignment_start,
    rec.query_alignment_end):
        phreds[i].append(rec.query_qualities[i])
maxs = [max(phreds[i]) for i in range(76)]
tops = [np.percentile(phreds[i], 95) for i in range(76)]
medians = [np.percentile(phreds[i], 50) for i in range(76)]
bottoms = [np.percentile(phreds[i], 5) for i in range(76)]
medians_fig = [x - y for x, y in zip(medians, bottoms)]
tops_fig = [x - y for x, y in zip(tops, medians)]
maxs_fig = [x - y for x, y in zip(maxs, tops)]
fig, ax = plt.subplots(figsize=(16,9))
ax.stackplot(range(1, 77), (bottoms, medians_fig,tops_fig))
ax.plot(range(1, 77), maxs, 'k-')
```

Here, we again use default dictionaries that allow you to use a bit of initialization code. We now fetch from start to end and create a list of Phred scores in a dictionary whose index is the relative position in the sequence read.

We then use NumPy to calculate the 95th, 50th (median), and 5th percentiles, along with the maximum of quality scores per position. For most computational biology analysis, having a statistical summarized view of the data is quite common. So, you're probably already familiar with not only percentile calculations, but also with other Pythonic ways to calculate means, standard deviations, maximums, and minimums.

Finally, we will perform a stacked plot of the distribution of Phred scores per position. Due to the way matplotlib expects stacks, we have to subtract the value of the lower percentile from the one before with the stackplot call. We can use the list for the bottom percentiles, but we have to correct the median and the top as follows:

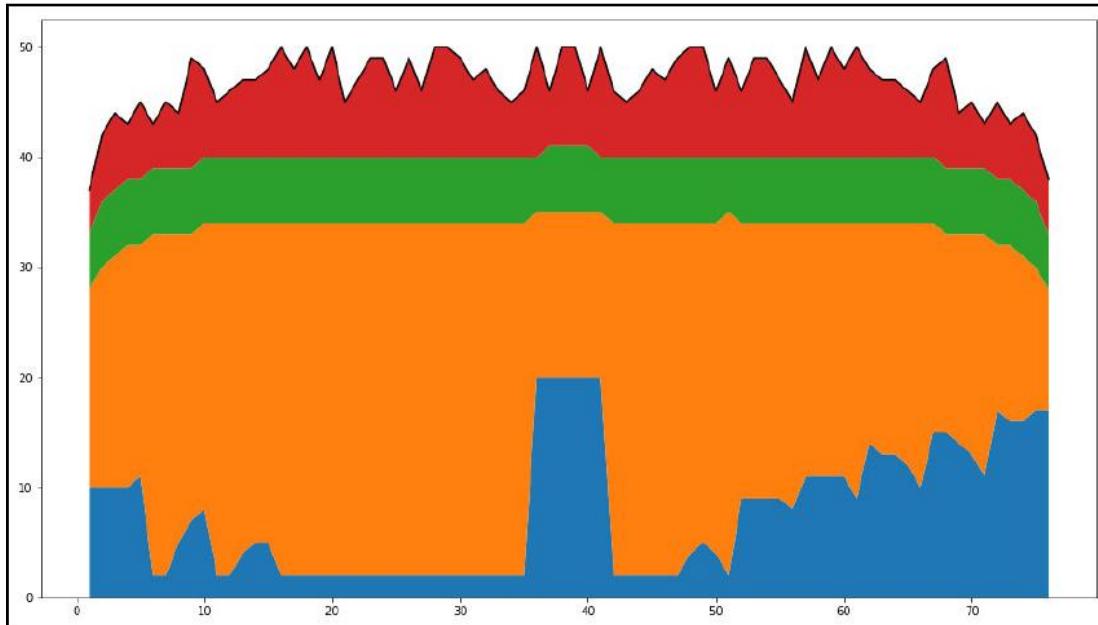


Figure 4: The distribution of Phred scores as a function of the position in the read; the bottom blue color spans from 0 to the 5th percentile; the green color up to the median, red to the 95th percentile, and purple to the maximum

## There's more...

Although we will discuss data filtering in the Studying genome accessibility and filtering SNP data recipe in this chapter, it's not our objective to explain the SAM format in detail or give a detailed course in data filtering. This task will require a book of its own, but with the basics of pysam, you can navigate through SAM/BAM files. However, in the last recipe of this chapter, we will take a look at extracting genome-wide metrics from BAM files (via annotations on VCF files that represent metrics of BAM files) for the purpose of understanding the overall quality of our dataset.

You will probably have very large data files to work with. It's possible that some BAM processing will take too much time. One of the first approaches to reduce the computation time is subsampling. For example, if you subsample at 10 percent, you ignore 9 records out of 10. For many tasks, such as some of the analysis done for the quality assessment of BAM files, subsampling at 10 percent (or even 1 percent) will be enough to get the gist of the quality of the file.

If you use human data, you may have your data sequenced at Complete Genomics. In this case, the alignment files will be different. Although Complete Genomics provides tools to convert to standard formats, you might be served better if you use their own data.

## See also

- The SAM/BAM format is described at <http://samtools.github.io/hts-specs/SAMv1.pdf>
- You can find an introductory explanation to the SAM format on the Abecassis group wiki page at <http://genome.sph.umich.edu/wiki/SAM>
- If you really need to get complex statistics from BAM files, Alistair Miles' pysamstats library is your port of call, at <https://github.com/alimanfoo/pysamstats>
- To convert your raw sequence data to alignment data, you will need an aligner; the most widely used is the bwa (<http://bio-bwa.sourceforge.net/>)
- Picard (surely a reference to Star Trek: The Next Generation) is the most commonly used tool to clean up BAM files; refer to <http://broadinstitute.github.io/picard/>
- The technical forum for sequence analysis is SEQanswers (<http://seqanswers.com/>)
- I would like to repeat the recommendation on Biostars here (which is referred to in the previous recipe); it's a treasure trove of information and has a very friendly community, at <http://www.biostars.org/>
- If you have the Complete Genomics data, take a look at their FAQ at <http://www.completegenomics.com/customer-support/faqs/>

## Analyzing data in VCF

After running a genotype caller (for example, GATK or SAMtools), you will have a VCF file reporting on genomic variations, such as SNPs, insertions/deletions (INDELs), copy number variations (CNVs), and so on. In this recipe, we will discuss VCF processing with the PyVCF module.

### Getting ready

While NGS is all about big data, there is a limit to how much I can ask you to download as a dataset for this book. I believe that 2 to 20 GB of data for a tutorial is asking too much. While the 1,000 Genomes' VCF files with realistic annotations are in this order of magnitude, we will want to work with much less data here. Fortunately, the Bioinformatics community has developed tools to allow for the partial download of data. As part of the SAMtools/htslib package (<http://www.htslib.org/>), you can download tabix and bgzip, which will take care of data management. On the command line, perform the following:

```
tabix -fh ftp://ftp-trace.ncbi.nih.gov/1000genomes/ftp/release/20130502/supporting/vcf_with_sample_level_annotation/ALL.chr22.phase3_shapeit2_mvncall_integrated_v5_extra_anno.20130502.genotypes.vcf.gz 22:1-17000000 | bgzip -c > genotypes.vcf.gz

tabix -p vcf genotypes.vcf.gz
```

If the preceding link does not work, be sure to check the dataset page at <https://github.com/PacktPublishing/Bioinformatics-with-Python-Cookbook-Second-Edition/blob/master/Datasets.ipynb> for an update.

The first line will partially download the VCF file for chromosome 22 (up to 17 Mbp) of the 1,000 Genomes Project. Then, bgzip will compress it.

The second line will create an index, which we will need for direct access to a section of the genome. As usual, you have the code to do this in a Notebook (Chapter02/Working\_with\_VCF.ipynb file).

## How to do it...

Take a look at the following steps:

Let's start by inspecting the information that we can get per record:

```
import vcf
v = vcf.Reader(filename='genotypes.vcf.gz')
print('Variant Level information')
infos = v.infos
for info in infos:
    print(info)
print('Sample Level information')
fmcts = v.formats
for fmt in fmcts:
    print(fmt)
```

We start by inspecting the annotations that are available for each record (remember that each record encodes a variant, such as SNP, CNV, INDELS, and so on, and the state of that variant per sample). At the variant (record) level, we find AC: the total number of ALT alleles in called genotypes, AF: the estimated allele frequency, NS: the number of samples with data, AN: the total number of alleles in called genotypes, and DP: the total read depth, among others. There are others, but they are mostly specific to the 1,000 Genomes Project (here, we will try to be as general as possible). Your own dataset may have more annotations (or none of these).

At the sample level, there are only two annotations in this file: GT—genotype, and DP—the per sample read depth. You have the per variant (total) read depth and the per sample read depth; be sure not to confuse both.

Now that we know which information is available, let's inspect a single VCF record:

```
v = vcf.Reader(filename='genotypes.vcf.gz')
rec = next(v)
print(rec.CHROM, rec.POS, rec.ID, rec.REF, rec.ALT, rec.QUAL,
rec.FILTER)
print(rec.INFO)
print(rec FORMAT)
samples = rec.samples
print(len(samples))
sample = samples[0]
print(sample.called, sample.gt_alleles, sample.is_het,
sample.phased)
print(int(sample['DP']))
```

We will start by retrieving the standard information: the chromosome, position, identifier, reference base, (typically just one), alternative bases (you can have more than one, but it's not uncommon as a first filtering approach to only accept a single ALT, for example, only accept biallelic SNPs), quality (as you might expect, Phred-scaled), and filter status. Regarding the filter status, remember that whatever the VCF file says, you may still want to apply extra filters (as in the next recipe).

We then print the additional variant-level information (AC, AS, AF, AN, DP, and so on), followed by the sample format (in this case, DP and GT). Finally, we count the number of samples and inspect a single sample to check whether it was called for this variant. Also, the reported alleles, heterozygosity, and phasing status (this dataset happens to be phased, which is not that common) are included.

Let's check the type of variant and the number of nonbiallelic SNPs in a single pass:

```
from collections import defaultdict
f = vcf.Reader(filename='genotypes.vcf.gz')
my_type = defaultdict(int)
num_alts = defaultdict(int)
for rec in f:
    my_type[rec.var_type, rec.var_subtype] += 1
    if rec.is_snp:
        num_alts[len(rec.QUAL)] += 1
print(num_alts)
print(my_type)
```

We will now use the now common Python default dictionary. We find that this dataset has INDELS (both insertions and deletions), CNVs, and, of course, SNPs (roughly two-thirds being transitions with one-third transversions). There is a residual number (79) of tri-allelic SNPs.

## There's more...

The purpose of this recipe is to get you up to speed with the PyVCF module. At this stage, you should be comfortable with the API. We will not spend too much time on usage details because this will be the main purpose of the next recipe: using the VCF module to study the quality of your variant calls.

It will probably not be a shocking revelation that PyVCF is not the fastest module on earth. The file format (highly text-based) makes processing a time-consuming task. There are two main strategies for dealing with this problem. One strategy is parallel processing, which we will discuss in the last chapter, Chapter 9, Python for Big Genomics Datasets. The second strategy is to convert to a more efficient format; we will provide an example of this in Chapter 4, Population Genetics. Note that VCF developers are working on a binary (BCF) version to deal with parts of these problems (<http://www.1000genomes.org/wiki/analysis/variant-call-format/bcf-binary-vcf-version-2>).

## See also

- The specification for VCF is available at <http://samtools.github.io/hts-specs/VCFv4.2.pdf>
- GATK is one of the most widely used variant callers; check out <https://www.broadinstitute.org/gatk/> for details
- SAMtools and htllib are used for variant calling and SAM/BAM management; check out <http://httplib.org> for details

## Studying genome accessibility and filtering SNP data

While the previous recipes were focused on giving an overview of Python libraries to deal with alignment and variant call data, in this recipe, we will concentrate on actually using them with a clear purpose in mind.

If you are using NGS data, chances are that your most important file to analyze is a VCF file, which is produced by a genotype caller such as SAMtools, mpileup, or GATK. The quality of your VCF calls may need to be assessed and filtered. Here, we will put in place a framework to filter SNP data. Rather than giving you filtering rules (an impossible task to be performed in a general way), we will give you procedures to assess the quality of your data. With this, you can devise your own filters. Be sure to check Chapter 11, Advanced NGS Processing for more tips on filtering.

## Getting ready

In the best-case scenario, you have a VCF file with proper filters applied. If this is the case, you can just go ahead and use your file. Note that all VCF files will have a FILTER column, but this might not mean that all of the proper filters were applied. You have to be sure that your data is properly filtered.

In the second case, which is one of the most common, your file will have unfiltered data, but you'll have enough annotations and can apply hard filters (there is no need for programmatic filtering). If you have a GATK annotated file, refer to <http://gatkforums.broadinstitute.org/discussion/2806/howto-apply-hard-filters-to-a-call-set>.

In the third case, you have a VCF file that has all the annotations that you need, but you may want to apply more flexible filters (for example, "if read depth > 20, accept if mapping quality > 30; otherwise, accept if mapping quality > 40").

In the fourth case, your VCF file does not have all the necessary annotations and you have to revisit your BAM files (or even other sources of information). In this case, the best solution is to find whatever extra information you can find and create a new VCF file with the required annotations. Some genotype callers (such as GATK) allow you to specify which annotations you want; you may also want to use extra programs to provide more annotations. For example, SnpEff (<http://snpeff.sourceforge.net/>) will annotate your SNPs with predictions of their effect (for example, if they are in exons, are they coding or non-coding?).

It's impossible to provide a clear-cut recipe. It will vary with your type of sequencing data, your species of study, and your tolerance to errors, among other variables. What we can do is provide a set of typical analysis that is done for high-quality filtering.

In this recipe, we will not use data from the Human 1,000 Genomes Project. We want dirty unfiltered data that has a lot of common annotations that can be used to filter it. We will use data from the Anopheles gambiae 1,000 Genomes Project (Anopheles is the mosquito vector involved in the transmission of the parasite that causes malaria), which makes filtered and unfiltered data available. You can find more information on this project at <http://www.malariagen.net/projects/vector/ag1000g>.

We will get a part of the centromere of chromosome 3L for around 100 mosquitoes, which is followed by a part somewhere in the middle of this chromosome (and index both):

```
tabix -fh  
ftp://ngs.sanger.ac.uk/production/ag1000g/phase1/preview/ag1000g.AC.phase1.  
AR1.vcf.gz 3L:1-200000 |bgzip -c > centro.vcf.gz  
tabix -fh  
ftp://ngs.sanger.ac.uk/production/ag1000g/phase1/preview/ag1000g.AC.phase1.
```

```
AR1.vcf.gz 3L:21000001-21200000 |bgzip -c > standard.vcf.gz  
tabix -p vcf centro.vcf.gz  
tabix -p vcf standard.vcf.gz
```

If the links do not work, be sure to check <https://github.com/PacktPublishing/Bioinformatics-with-Python-Cookbook-Second-Edition/blob/master/Datasets>.

ipynb for updates. As usual, the code for downloading this data is available on the Chapter02/Filtering\_SNPs.ipynb Notebook.

Finally, a word of warning on this recipe: the level of Python here will be slightly more complicated than usual. The more general code we will write will be easier to reuse for your specific case. We will use functional programming techniques (lambda functions) and the partial function application extensively.

## How to do it...

Take a look at the following steps:

Let's start by plotting the distribution of variants across the genome in both files:

```
%matplotlib inline  
from collections import defaultdict  
import functools  
import numpy as np  
import seaborn as sns  
import matplotlib.pyplot as plt  
import vcf  
  
def do_window(recs, size, fun):  
    start = None  
    win_res = []  
    for rec in recs:  
        if not rec.is_snp or len(rec.ALT) > 1:  
            continue  
        if start is None:  
            start = rec.POS  
        my_win = 1 + (rec.POS - start) // size  
        while len(win_res) < my_win:  
            win_res.append([])  
            win_res[my_win - 1].extend(fun(rec))  
    return win_res  
  
wins = {}
```

```

size = 2000
names = ['centro.vcf.gz', 'standard.vcf.gz']
for name in names:
    recs = vcf.Reader(filename=name)
    wins[name] = do_window(recs, size, lambda x: [1])

```

We will start by performing the required imports (as usual, remember to remove the first line if you are not using the IPython Notebook). Before I explain the function, note what we are doing.

For both files, we will compute windowed statistics. We will divide our file, which includes 200,000 bp of data, in windows of size 2,000 (100 windows). Every time we find a biallelic SNP, we will add a one to the list related to this window in the window function.

The window function will take a VCF record (a SNP, `rec.is_snp`, that is not biallelic `len(rec.ALT) == 1`), determine the window where that record belongs (by performing an integer division of `rec.POS` by `size`), and extend the list of results of that window by the function passed to it as the `fun` parameter (which, in our case, is just a one).

So, now we have a list of 100 elements (each representing 2,000 base pairs). Each element will be another list that will have a one for each biallelic SNP found.

So, if you have 200 SNPs in the first 2,000 base pairs, the first element of the list will have 200 ones.

Let's continue as follows:

```

def apply_win_funcs(wins, funs):
    fun_results = []
    for win in wins:
        my_funs = {}
        for name, fun in funs.items():
            try:
                my_funs[name] = fun(win)
            except:
                my_funs[name] = None
        fun_results.append(my_funs)
    return fun_results

stats = {}
fig, ax = plt.subplots(figsize=(16, 9))
for name, nwins in wins.items():
    stats[name] = apply_win_funcs(nwins, {'sum': sum})
    x_lim = [i * size for i in range(len(stats[name]))]

```

```

    ax.plot(x_lim, [x['sum'] for x in stats[name]], label=name)
    ax.legend()
    ax.set_xlabel('Genomic location in the downloaded segment')
    ax.set_ylabel('Number of variant sites (bi-allelic SNPs)')
    fig.suptitle('Number of bi-allelic SNPs along the genome',
    fontsize='xx-large')

```

Here, we perform a plot that contains statistical information for each of our 100 windows. `apply_win_funs` will calculate a set of statistics for every window. In this case, it will sum all the numbers in the window. Remember that every time we find an SNP, we add 1 to the window list. This means that if we have 200 SNPs, we will have 200 1s; hence, summing them will return 200.

So, we are able to compute the number of SNPs per window in an apparently convoluted way. Why we perform things with this strategy will become apparent soon. However, for now, let's check the result of this computation for both files, as shown in the following graph:

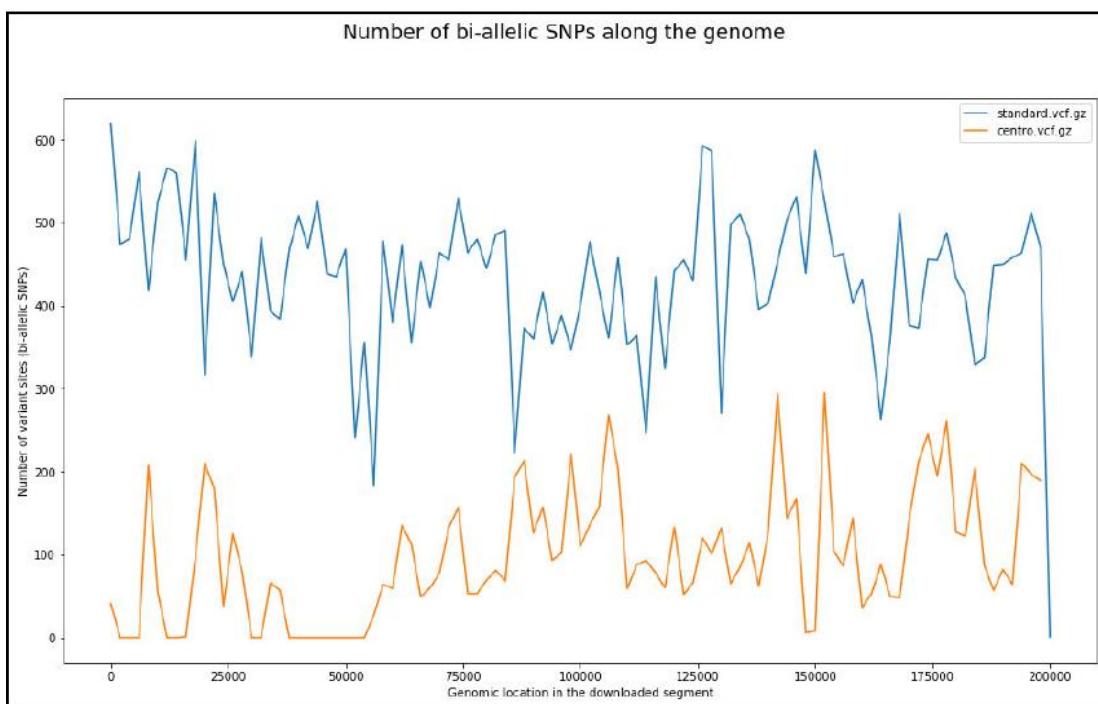


Figure 5: The number of biallelic SNPs distributed of windows of 2,000 bp of size for an area of 200 kbp near the centromere (orange), and in the middle of chromosome (blue); both areas come from chromosome 3L for circa 100 Ugandan mosquitoes from the Anopheles 1,000 Genomes Project



Note that the amount of SNPs in the centromere is smaller than in the middle of the chromosome. This is expected because both calling variants in chromosomes are more difficult than calling in the middle. Also, there is probably less genomic diversity in centromeres. If you are used to humans or other mammals, you will find the density of variants obnoxiously high—that's mosquitoes for you!

Let's take a look at the sample-level annotation. We will inspect mapping quality zero (refer to [https://www.broadinstitute.org/gatk/guide/tooldocs/org\\_broadinstitute\\_gatk\\_tools\\_walkers\\_annotator\\_MappingQualityZeroBySample.php](https://www.broadinstitute.org/gatk/guide/tooldocs/org_broadinstitute_gatk_tools_walkers_annotator_MappingQualityZeroBySample.php) for details), which is a measure of how well sequences involved in calling this variant map clearly to this position. Note that there is also a MQ0 annotation at the variant level:

```
mq0_wins = {}
size = 5000

def get_sample(rec, annot, my_type):
    res = []
    samples = rec.samples
    for sample in samples:
        if sample[annot] is None: # ignoring nones
            continue
        res.append(my_type(sample[annot]))
    return res

for vcf_name in vcf_names:
    recs = vcf.Reader(filename=vcf_name)
    mq0_wins[vcf_name] = do_window(recs, size,
        functools.partial(get_sample, annot='MQ0', my_type=int))
```

Start inspecting this by looking at the last for; we will perform a windowed analysis by reading the MQ0 annotation from each record. We perform this by calling the get\_sample function, which will return our preferred annotation (in this case, MQ0), which has been cast with a certain type (my\_type=int). We use the partial application function here. Python allows you to specify some parameters of a function and wait for other parameters to be specified later. Note that the most complicated thing here is the functional programming style. Also, note that it makes it very easy to compute other sample-level annotations. Just replace MQ0 with AB, AD, GQ, and so on. You immediately have a computation for that annotation. If the annotation is not of the integer type, no problem; just adapt my\_type. It's a difficult programming style if you are not used to it, but you will reap its benefits very soon.

Now, let's print the median and the top 75 percent percentile for each window (in this case, with a size of 5,000):

```
stats = {}
colors = ['b', 'g']
i = 0
fig, ax = plt.subplots(figsize=(16, 9))
for name, nwins in mq0_wins.items():
    stats[name] = apply_win_funs(nwins, {'median':np.median, '75':functools.partial(np.percentile, q=75)})
    x_lim = [j * size for j in range(len(stats[name]))]
    ax.plot(x_lim, [x['median'] for x in stats[name]], label=name,
            color=colors[i])
    ax.plot(x_lim, [x['75'] for x in stats[name]], '--',
            color=colors[i])
    i += 1
ax.legend()
ax.set_xlabel('Genomic location in the downloaded segment')
ax.set_ylabel('MQ0')
fig.suptitle('Distribution of MQ0 along the genome', fontsize='xx-large')
```

Note that we now have two different statistics on `apply_win_funs` (percentile and median). Again, we pass functions as parameters (`np.median` and `np.percentile`), with partial function application done on `np.percentile`. The result is as follows:

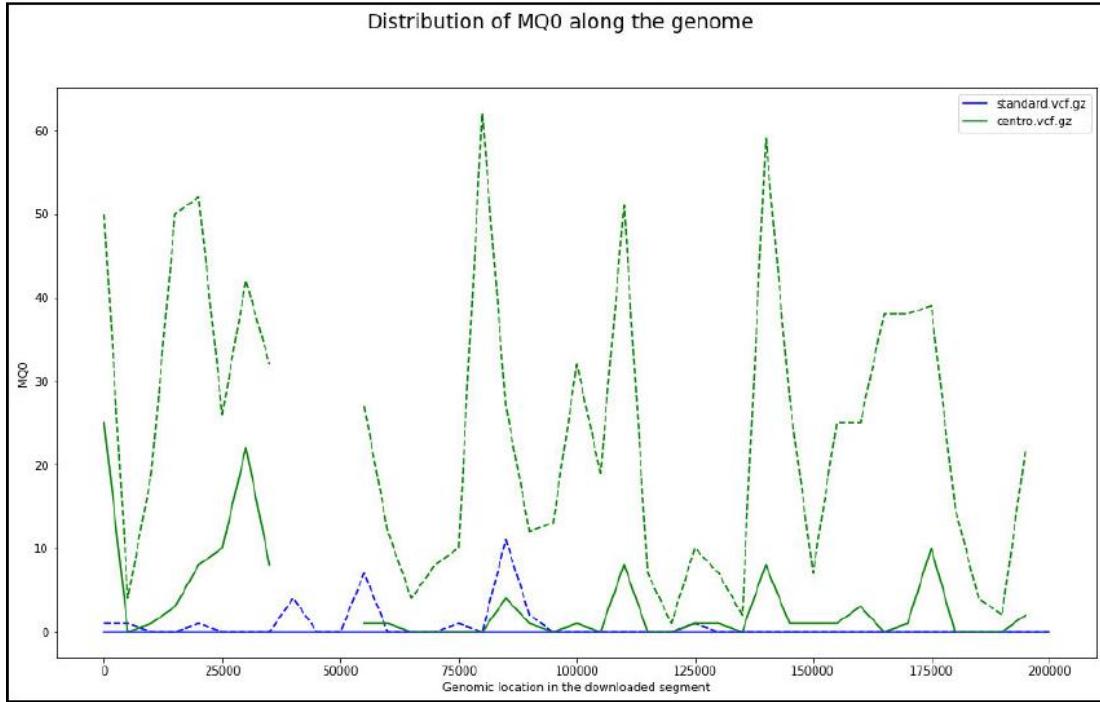


Figure 6: Median (continuous line) and 75th percentile (dashed) of MQ0 of sample SNPs distributed on windows of 5,000 bp of size for an area of 200 kbp near the centromere (blue) and in the middle of chromosome (green); both areas come from chromosome 3L for circa 100 Ugandan mosquitoes from the Anopheles 1,000 Genomes Project

For the standard.vcf.gz file, the median MQ0 is plotted at the very bottom and is almost unseen. This is good as it suggests that most sequences involved in the calling of variants map clearly to this area of the genome. For the centro.vcf.gz file, MQ0 is of poor quality. Furthermore, there are areas where the genotype caller cannot find any variants at all (hence the incomplete chart).

Let's compare heterozygosity with (DP), the sample-level annotation. Here, we will plot the fraction of heterozygosity calls as a function of the Sample Read Depth (DP) for every SNP. First, we will explain the result and then the code that generates it.

The following graph shows the fraction of calls that are heterozygous at a certain depth:

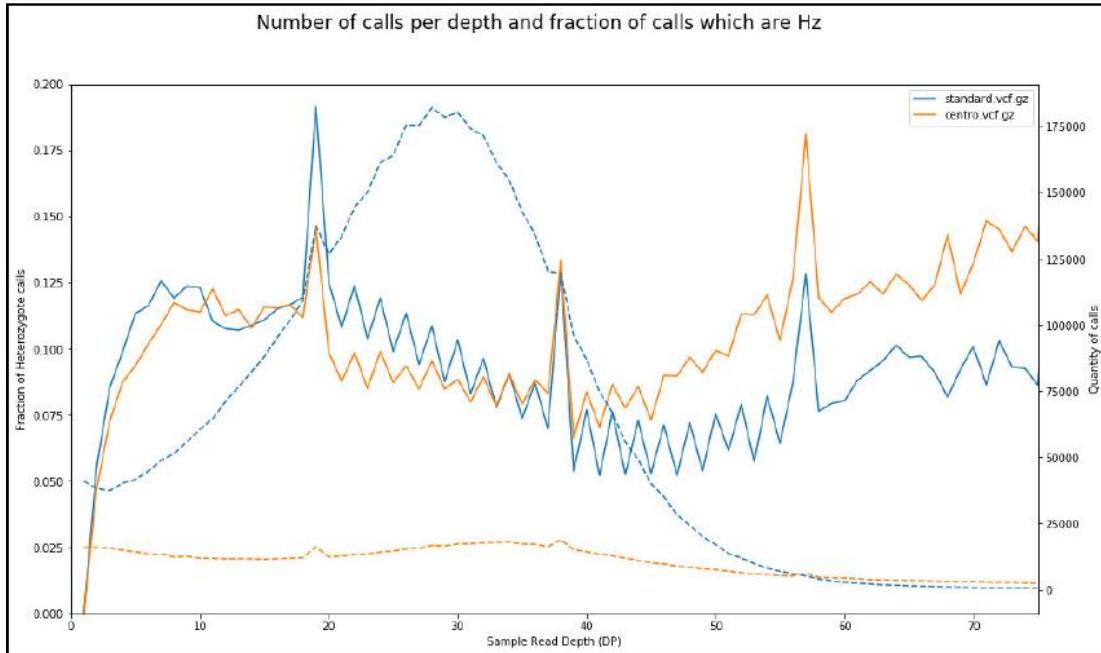


Figure 7: The continuous line represents the fraction of heterozygosity calls computed at a certain depth; in orange is the centromeric area; in blue is the "standard" area; the dashed lines represent the number of sample calls per depth; both areas come from chromosome 3L for circa 100 Ugandan mosquitoes from the Anopheles 1,000 Genomes Project

In the preceding graph, there are two considerations to take into account. At a very low depth, the fraction of heterozygote calls is biased, that is, low. This makes sense, as the number of reads per position does not allow you to make a correct estimate of the presence of both alleles in a sample. Therefore, you should not trust calls at very low depth.

As expected, the number of calls in the centromere is way lower than outside it. The distribution of SNPs outside the centromere follows a common pattern that you can expect in many datasets.

The code for this is as follows:

```
def get_sample_relation(recs, f1, f2):
    rel = defaultdict(int)
    for rec in recs:
        if not rec.is_snp:
```

```

        continue
for sample in rec.samples:
    try:
        v1 = f1(sample)
        v2 = f2(sample)
        if v1 is None or v2 is None:
            continue # We ignore Nones
        rel[(v1, v2)] += 1
    except:
        pass # This is outside the domain (typically None)
return rel
rels = {}
for vcf_name in vcf_names:
    recs = vcf.Reader(filename=vcf_name)
    rels[vcf_name] = get_sample_relation(recs, lambda s: 1 if
s.is_het else 0, lambda s: int(s['DP']))

```

Start by looking for the for loop. Again, we use functional programming; the `get_sample_relation` function will traverse all SNP records and apply two functional parameters. The first parameter determines heterozygosity, whereas the second parameter acquires the sample DP (remember that there is also a variant of DP).

Now, since the code is as complex as it is, I opted for a naive data structure to be returned by `get_sample_relation`: a dictionary where the key is a pair of results (in this case, heterozygosity and DP) and the sum of SNPs that share both values. There are more elegant data structures with different trade-offs. For this, there's SciPy sparse matrices, pandas' DataFrames, or you may want to consider PyTables. The fundamental point here is to have a framework that is general enough to compute relationships between a couple of sample annotations.

Also, be careful with the dimension space of several annotations. For example, if your annotation is of the float type, you might have to round it (if not, the size of your data structure might become too big).

Now, let's take a look at the plotting codes. Let's perform this in two parts. Here is part one:

```

def plot_hz_rel(dps, ax, ax2, name, rel):
    frac_hz = []
    cnt_dp = []
    for dp in dps:
        hz = 0.0
        cnt = 0
        for khz, kdp in rel.keys():
            if kdp != dp:

```

```

        continue
        cnt += rel[khz, dp]
        if khz == 1:
            hz += rel[(khz, dp)]
        frac_hz.append(hz / cnt)
        cnt_dp.append(cnt)
    ax.plot(dps, frac_hz, label=name)
    ax2.plot(dps, cnt_dp, '--', label=name)

```

This function will take a data structure, as generated by `get_sample_relation`, expecting that the first parameter of the key tuple is the heterozygosity state (0=homozygote, 1=heterozygote) and the second parameter is DP. With this, it will generate two lines: one with the fraction of samples (which are heterozygotes at a certain depth) and the other with the SNP count.

Now, let's call this function:

```

fig, ax = plt.subplots(figsize=(16, 9))
ax2 = ax.twinx()
for name, rel in rels.items():
    dps = list(set([x[1] for x in rel.keys()]))
    dps.sort()
    plot_hz_rel(dps, ax, ax2, name, rel)
    ax.set_xlim(0, 75)
    ax.set_ylim(0, 0.2)
    ax2.set_ylabel('Quantity of calls')
    ax.set_ylabel('Fraction of Heterozygote calls')
    ax.set_xlabel('Sample Read Depth (DP)')
    ax.legend()
fig.suptitle('Number of calls per depth and fraction of calls which are Hz', fontsize='xx-large')

```

Here, we will use two axes. On the left-hand side, we will have the fraction of heterozygous SNPs. On the right-hand side, we will have the number of SNPs. We then call `plot_hz_rel` for both data files. The rest is standard Matplotlib code.

Finally, let's compare the DP variant with a categorical variant level annotation (EFF). EFF is provided by SnpEff and tells us (among many other things) the type of SNP (for example, intergenic, intronic, coding synonymous, and coding nonsynonymous). The Anopheles dataset provides this useful annotation. Let's start by extracting variant-level annotations and the functional programming style:

```

def get_variant_relation(recs, f1, f2):
    rel = defaultdict(int)

```

```

for rec in recs:
    if not rec.is_snp:
        continue
    try:
        v1 = f1(rec)
        v2 = f2(rec)
        if v1 is None or v2 is None:
            continue # We ignore Nones
        rel[(v1, v2)] += 1
    except:
        pass
return rel

```

The programming style here is similar to `get_sample_relation`, but we will not delve into any samples. Now, we define the type of effects that we'll work with and convert its effect into an integer (as it will allow us to use it as an index, for example, matrices). Now, think about coding a categorical variable:

```

accepted_eff = ['INTERGENIC', 'INTRON', 'NON_SYNONYMOUS_CODING',
'SYNONYMOUS_CODING']

def eff_to_int(rec):
    try:
        for annot in rec.INFO['EFF']:
            #We use the first annotation
            master_type = annot.split('(')[0]
            return accepted_eff.index(master_type)
    except ValueError:
        return len(accepted_eff)

```

We will now traverse the file; the style should be clear to you now:

```

eff_mq0s = {}
for vcf_name in vcf_names:
    recs = vcf.Reader(filename=vcf_name)
    eff_mq0s[vcf_name] = get_variant_relation(recs, lambda r:
eff_to_int(r), lambda r: int(r.INFO['DP']))

```

Finally, we plot the distribution of DP using the SNP effect:

```

fig, ax = plt.subplots(figsize=(16,9))
vcf_name = 'standard.vcf.gz'
bp_vals = [[] for x in range(len(accepted_eff) + 1)]
for k, cnt in eff_mq0s[vcf_name].items():
    my_eff, mq0 = k
    bp_vals[my_eff].extend([mq0] * cnt)
sns.boxplot(data=bp_vals, sym='', ax=ax)
ax.set_xticklabels(accepted_eff + ['OTHER'])

```

```
ax.set_ylabel('DP (variant)')
fig.suptitle('Distribution of variant DP per SNP type',
            fontsize='xx-large')
```

Here, we just print a boxplot for the noncentromeric file, as shown in the following diagram. The results are as expected: SNPs in coding areas will probably have more depth, because they are in more complex regions that are easier to call than intergenic SNPs:

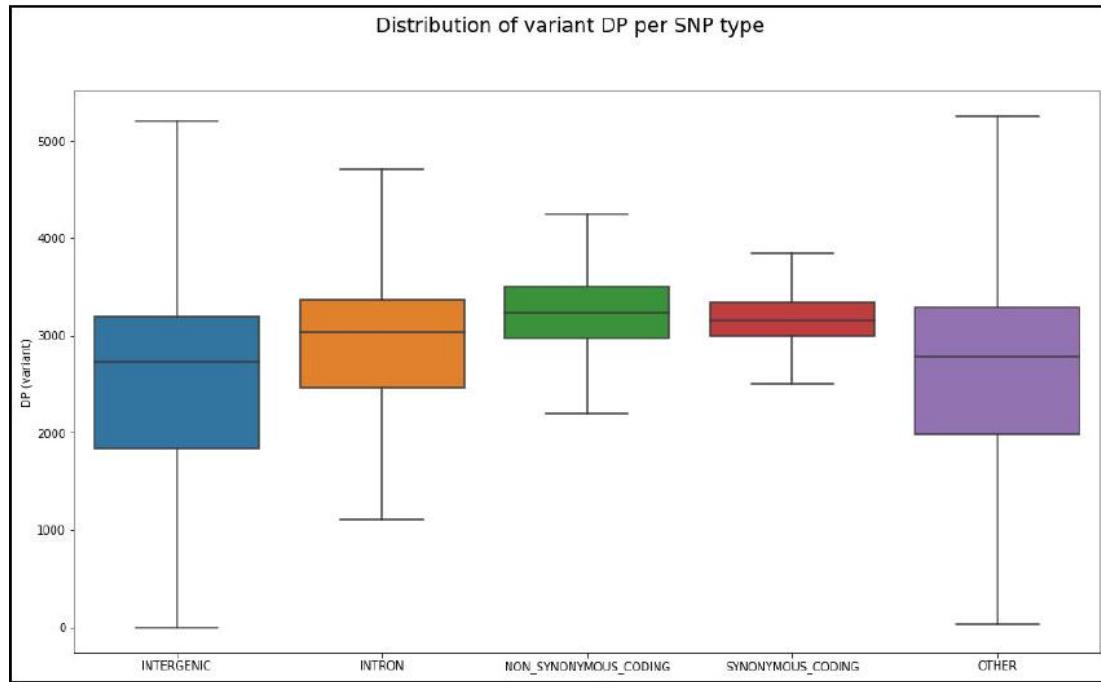


Figure 8: Boxplot for the distribution of variant read depth across different SNP effects

## There's more...

The whole issue of filtering SNPs and other genome features will need a book on its own. This approach will depend on the type of sequencing data that you have, the number of samples, and potential extra information (for example, a pedigree among samples).

This recipe is very complex as it is, but parts of it are profoundly naive (there is a limit regarding the complexity that I can force on you in a simple recipe). For example, the window code does not support overlapping windows. Also, data structures are simplistic. However, I hope that they give you an idea of the general strategy to process genomic, high-throughput sequencing data. You can read more in Chapter 11, Advanced NGS Processing.

## See also

- There are many filtering rules, but I would like to draw your attention to the need for a reasonably good coverage (clearly above 10x). Refer to Meynet et al., Variant detection sensitivity and biases in whole genome and exome sequencing, at <http://www.biomedcentral.com/1471-2105/15/247/>.
- bcbio-nextgen is a Python-based pipeline for high-throughput sequencing analysis, which is worth checking out (<https://bcbio-nextgen.readthedocs.org>).

## Processing NGS data with HTSeq

HTSeq (<https://htseq.readthedocs.io>) is an alternative library that's used for processing NGS data. Most of the functionality made available by HTSeq is actually available in other libraries covered in this book, but you should be aware of it as an alternative way of processing NGS data. HTSeq supports, among others, FASTA, FASTQ, SAM (via pysam), VCF, GFF, and Browser Extensible Data (BED) file formats. It also includes a set of abstractions for processing (mapped) genomic data, encompassing concepts like genomic positions and intervals or alignments. A complete examination of the features of this library is beyond our scope, so we will concentrate on a small subset of features. We will take this opportunity to also introduce the BED file format.

The BED format allows for the specification of features for annotations tracks. It has many uses, but it's common to load BED files into genome browsers to visualize features. Each line includes information about at least the position (chromosome, start and end) and also optional fields such as name or strand. Full details about the format can be found at <https://genome.ucsc.edu/FAQ/FAQformat.html#format1>.

## Getting ready

Our simple example will use data from the region where the LCT gene is located in the human genome. The LCT gene codifies lactase, an enzyme involved in the digestion of lactose.

We will take this information from Ensembl. Go to [http://uswest.ensembl.org/Homo\\_sapiens/Gene/Summary?db=core;g=ENSG00000115850](http://uswest.ensembl.org/Homo_sapiens/Gene/Summary?db=core;g=ENSG00000115850) and choose Export data. The Output format should be BED Format. Gene information should be selected (you can choose more if you want). For convenience, a downloaded file is available in the Chapter02 directory, called LCT.bed.

The Notebook for this code is called Chapter02/Processing\_BED\_with\_HTSseq.ipynb.

Take a look at the file before we start. An example of a few lines of this file is as follows:

```
track name=gene description="Gene information"
2    135836529    135837180    ENSE00002202258 0      -
2    135833110    135833190    ENSE00001660765 0      -
2    135789570    135789798    NM_002299.2.16 0      -
2    135787844    135788544    NM_002299.2.17 0      -
2    135836529    135837169    CCDS2178.117 0      -
2    135833110    135833190    CCDS2178.116 0      -
```

The fourth column is the feature name. This will vary widely from file to file, and you will have to check it each and every time. However, in our case, it seems apparent that we have Ensembl Exons (ENSE...), Genbank records (NM\_...), and coding region information (CCDS) from the CCDS database (<https://www.ncbi.nlm.nih.gov/CCDS/CcdsBrowse.cgi>).

## How to do it...

Take a look at the following steps:

We will start by setting up a reader for our file. Remember that this file has already been supplied to you, and should be in your current work directory:

```
from collections import defaultdict
import re
import HTSeq

lct_bed = HTSeq.BED_Reader('LCT.bed')
```

We are now going to extract all the types of features via its name:

```
feature_types = defaultdict(int)
for rec in lct_bed:
    last_rec = rec
    feature_types[re.search('([A-Z]+)', rec.name).group(0)] += 1
print(feature_types)
```

Remember that this code is specific to our example. You will have to adapt it to your case.



You will find that the preceding code uses a regular expression. Be careful with regular expressions, as they tend to generate read-only code that is difficult to maintain. You might have better alternatives. In any case, regular expressions exist and you will find them from time to time.

The output for our case is as follows:

```
defaultdict(<class 'int'>, {'ENSE': 27, 'NM': 17, 'CCDS': 17})
```

We stored the last record so that we can inspect it:

```
print(last_rec)
print(last_rec.name)
print(type(last_rec))
interval = last_rec.iv
print(interval)
print(type(interval))
```

There are many fields available, most notably name and interval. For the preceding code, the output is as follows:

```
<GenomicFeature: BED line 'CCDS2178.11' at 2: 135788543 ->
135788322 (strand '-')
CCDS2178.11
<class 'HTSeq.GenomicFeature'>
2:[135788323,135788544)-
<class 'HTSeq._HTSeq.GenomicInterval'>
```

Let's dig deeper into the interval:

```
print(interval.chrom, interval.start, interval.end)
print(interval.strand)
print(interval.length)
print(interval.start_d)
print(interval.start_as_pos)
print(type(interval.start_as_pos))
```

The output is as follows:

```
2 135788323 135788544
-
221
135788543
2:135788323/
<class 'HTSeq._HTSeq.GenomicPosition'>
```

Note the genomic position (chromosome, start and end). The most complex issue is how to deal with the strand. If the feature is coded in the negative strand, you have to be careful with processing. HTSeq offers the start\_d and end\_d fields to help you with this (that is, they will be reversed with regards to the start and end if the strand is negative).

Finally, let's extract some statistics from our coding regions (CCDS records). We will use CCDS, since it's probably than the better curated database here:

```
exon_start = None
exon_end = None
sizes = []
for rec in lct_bed:
    if not rec.name.startswith('CCDS'):
        continue
    interval = rec.iv
    exon_start = min(interval.start, exon_start or interval.start)
    exon_end = max(interval.length, exon_end or interval.end)
    sizes.append(interval.length)
sizes.sort()
print("Num exons: %d / Begin: %d / End %d" % (len(sizes),
exon_start, exon_end))
print("Smaller exon: %d / Larger exon: %d / Mean size: %.1f" %
(sizes[0], sizes[-1], sum(sizes)/len(sizes)))
```

The output should be self-explanatory:

```
Num exons: 17 / Begin: 135788323 / End 135837169
Smaller exon: 79 / Larger exon: 1551 / Mean size: 340.2
```

## There's more...

The BED format can be a bit more complex than this. Furthermore, the preceding code is based on quite specific premises with regard to the contents of our file. However, this example should be enough to get you started. At its worst, the BED format is really not very complicated.

HTSeq has much more functionality. This recipe is mostly provided as a starting point for the whole package. HTSeq has functionality that can be used as an alternative to most of the recipes that we've covered thus far.

# 3

# Working with Genomes

In this chapter, we will cover the following recipes:

- Working with high-quality reference genomes
- Dealing with low-quality reference genomes
- Traversing genome annotations
- Extracting genes from a reference using annotations
- Finding orthologues using the Ensembl REST API
- Retrieving gene ontology information from Ensembl

## Introduction

Many tasks in computational biology are dependent on the existence of reference genomes. If you are performing sequence alignment, finding genes, or studying the genetics of populations, you will be directly or indirectly using a reference genome. In this chapter, we will develop some recipes for working with reference genomes and dealing with references of a varying quality—which can vary from high quality, as with the human genome, to problematic with non-model species. We will also look at how to deal with genome annotations (working with text databases that will point us to interesting features in the genome) and extract sequence data using the annotation information. We will also try to find some gene orthologues across species. Finally, we will access a Gene Ontology (GO) database.

# Working with high-quality reference genomes

In this recipe, you will learn about a few general techniques to manipulate reference genomes. As an illustrative example, we will study the GC content—the fraction of the genome that is based on guanine-cytosine in *Plasmodium falciparum*, the most important parasite species that causes malaria. Reference genomes are normally made available as FASTA files.

## Getting ready

Organism genomes come in widely different sizes, ranging from viruses such as HIV, which is 9.7 kbp, to bacteria such as *E. coli*, to protozoans like *Plasmodium falciparum*, with a 22 Mbp spread across 14 chromosomes, mitochondrion, and apicoplast, to the fruit fly with three autosomes, a mitochondrion, and X/Y sex chromosomes, to humans with their three Gbp pairs spread across 22 autosomes, X/Y chromosomes, and mitochondria, all the way up to *Paris japonica*, a plant with 150 Gbp of genome. Along the way, you have different ploidy and different sex chromosome organizations.



As you can see, different organisms have very different genome sizes. This difference can be of several orders of magnitude. This can have significant implications for your programming style. Working with a large genome will require you to be more conservative with the usage of memory. Unfortunately, larger genomes would benefit from more speed-efficient programming techniques (as you have much more data to analyze); these are conflicting requirements. The general rule is that you have to be much more careful with efficiency (both speed and memory) with larger genomes.

To make this recipe less of a burden, we will use a small eukaryotic genome from *P. falciparum*. This genome still has many typical features of larger genomes (for example, multiple chromosomes). Therefore, it's a good compromise between complexity and size. Note that with a genome of the size of *P. falciparum*, it will be possible to perform many operations by loading the whole genome in-memory. However, we opted for a programming style that can be used with bigger genomes (for example, mammals) so that you can use this recipe in a more general way, but feel free to use more memory-intensive approaches with small genomes like this.

We will use Biopython, which you installed in Chapter 1, Python and the Surrounding Software Ecology. As usual, this recipe is available for the Jupyter Notebook at Chapter03/Reference\_Genome.ipynb in the code bundle of this book.

## How to do it...

Let's take a look at the following steps:

We will start by inspecting the description of all of the the sequences on the reference genome FASTA file:

```
from Bio import SeqIO
genome_name = 'PlasmoDB-9.3_Pfalciparum3D7_Genome.fasta'
recs = SeqIO.parse(genome_name, 'fasta')
for rec in recs:
    print(rec.description)
```

This code should look familiar from the previous chapter, Chapter 2, Next-Generation Sequencing. Let's take a look at part of the output:

Pf3D7_05_v3	organism=Plasmodium_falciparum_3D7	version=2012-02-01	length=1343557	S0=chromosome
Pf3D7_10_v3	organism=Plasmodium_falciparum_3D7	version=2012-02-01	length=1687056	S0=chromosome
Pf3D7_07_v3	organism=Plasmodium_falciparum_3D7	version=2012-02-01	length=1445207	S0=chromosome
Pf3D7_03_v3	organism=Plasmodium_falciparum_3D7	version=2012-02-01	length=1067971	S0=chromosome
Pf3D7_13_v3	organism=Plasmodium_falciparum_3D7	version=2012-02-01	length=2925236	S0=chromosome
Pf3D7_11_v3	organism=Plasmodium_falciparum_3D7	version=2012-02-01	length=2038340	S0=chromosome
Pf3D7_14_v3	organism=Plasmodium_falciparum_3D7	version=2012-02-01	length=3291936	S0=chromosome
Pf3D7_09_v3	organism=Plasmodium_falciparum_3D7	version=2012-02-01	length=1541735	S0=chromosome
Pf3D7_01_v3	organism=Plasmodium_falciparum_3D7	version=2012-02-01	length=649851	S0=chromosome
Pf3D7_12_v3	organism=Plasmodium_falciparum_3D7	version=2012-02-01	length=2271494	S0=chromosome
Pf3D7_08_v3	organism=Plasmodium_falciparum_3D7	version=2012-02-01	length=1472805	S0=chromosome
Pf3D7_06_v3	organism=Plasmodium_falciparum_3D7	version=2012-02-01	length=1418242	S0=chromosome
Pf3D7_04_v3	organism=Plasmodium_falciparum_3D7	version=2012-02-01	length=1200490	S0=chromosome
Pf3D7_02_v3	organism=Plasmodium_falciparum_3D7	version=2012-02-01	length=947182	S0=chromosome
M76611	organism=Plasmodium_falciparum_3D7	version=2012-02-01	length=5967	S0=mitochondrial_chromosome
PFC10_API_IRAB	organism=Plasmodium_falciparum_3D7	version=2012-02-01	length=34242	S0=apicoplast_chromosome

Different genome references will have different description lines, but they will generally have important information. In this example, you can see that we have chromosomes, mitochondria, and apicoplast. We can also view chromosome sizes, but we will take the value from the sequence length instead.

Let's parse the description line to extract the chromosome number. We will retrieve the chromosome size from the sequence, and compute the GC content across chromosomes on a window basis:

```
from Bio import SeqUtils
recs = SeqIO.parse(genome_name, 'fasta')
```

```
chrom_sizes = {}
chrom_GC = {}
block_size = 50000
min_GC = 100.0
max_GC = 0.0
for rec in recs:
    if rec.description.find('SO=chromosome') == -1:
        continue
    chrom = int(rec.description.split('_')[1])
    chrom_GC[chrom] = []
    size = len(rec.seq)
    chrom_sizes[chrom] = size
    num_blocks = size // block_size + 1
    for block in range(num_blocks):
        start = block_size * block
        if block == num_blocks - 1:
            end = size
        else:
            end = block_size + start + 1
        block_seq = rec.seq[start:end]
        block_GC = SeqUtils.GC(block_seq)
        if block_GC < min_GC:
            min_GC = block_GC
        if block_GC > max_GC:
            max_GC = block_GC
    chrom_GC[chrom].append(block_GC)
print(min_GC, max_GC)
```

We have performed a windowed analysis of all chromosomes, similar to what you have seen in the previous chapter, Chapter 2, Next-Generation Sequencing. We started by defining a window size of 50 kbp. This is appropriate for *P. falciparum* (feel free to vary its size), but you will want to consider other values for genomes with chromosomes that are orders of magnitude different from this.

Note that we are re-reading the file. With such a small genome, it would have been feasible (in step one) to do an in-memory load of the whole genome. By all means, feel free to try this programming style for small genomes – it's faster! However, this code is more generalized for larger genomes.

Note that in the for loop, we ignore the mitochondrion and apicoplast by parsing the SO entry to the description. The chrom\_sizes dictionary will maintain the size of chromosomes.

The chrom\_GC dictionary is our most interesting data structure and will have a list of a fraction of the GC content for each 50 kbp window. So, for chromosome 1, which has a size of 640,851 bp, there will be 14 entries because this chromosome size has 14 blocks of 50 kbp.



Be aware of two unusual features of the *P. falciparum* genome: the genome is very AT-rich, that is, GC-poor. Therefore, the numbers that you will get will be very low. Also, chromosomes are ordered based on size (as is common), but starting with the smallest size. The usual convention is to start with the largest size (such as with genomes in humans).

Now, let's perform a genome plot of the GC distribution. We will use shades of blue for the GC content. However, for high outliers, we will use shades of red. For low outliers, we will use shades of yellow:

```
from reportlab.lib import colors
from reportlab.lib.units import cm
from Bio.Graphics import BasicChromosome

chroms = list(chrom_sizes.keys())
chroms.sort()
biggest_chrom = max(chrom_sizes.values())
my_genome = BasicChromosome.Organism(output_format="png")
my_genome.page_size = (29.7*cm, 21*cm)
telomere_length = 10
bottom_GC = 17.5
top_GC = 22.0
for chrom in chroms:
    chrom_size = chrom_sizes[chrom]
    chrom_representation = BasicChromosome.Chromosome('Cr %d' % chrom)
    chrom_representation.scale_num = biggest_chrom
    tel = BasicChromosome.TelomereSegment()
    tel.scale = telomere_length
    chrom_representation.add(tel)
    num_blocks = len(chrom_GC[chrom])
    for block, gc in enumerate(chrom_GC[chrom]):
        my_GC = chrom_GC[chrom][block]
        body = BasicChromosome.ChromosomeSegment()
        if my_GC > top_GC:
            body.fill_color = colors.Color(1, 0, 0)
        elif my_GC < bottom_GC:
            body.fill_color = colors.Color(1, 1, 0)
        else:
            my_color = (my_GC - bottom_GC) / (top_GC - bottom_GC)
            body.fill_color = colors.Color(my_color, my_color, 1)
```

```
if block < num_blocks - 1:  
    body.scale = block_size  
else:  
    body.scale = chrom_size % block_size  
chrom_representation.add(body)  
tel = BasicChromosome.TelomereSegment(inverted=True)  
tel.scale = telomere_length  
chrom_representation.add(tel)  
my_genome.add(chrom_representation)  
my_genome.draw('falciparum.png', 'Plasmodium falciparum')
```

The first line converts the return of the keys method to a list. This was redundant in Python 2, but not in Python 3, where the keys method has a specific dict\_keys return type.

We will draw the chromosomes in order (hence the sort). We will need the size of the biggest chromosome (14, in P. falciparum) to make sure that the size of chromosomes is printed with the correct scale (the biggest\_chrom variable).

We then create an A4-sized representation of an organism with a PNG output. Note that we will draw very small telomeres of 10 bp. This will produce a rectangular-like chromosome. You can make the telomeres bigger, giving it a roundish representation, or you may have the arguably better idea of using the correct telomere size for your species.

We declare that anything with a GC content below 17.5 percent or above 22.0 percent will be considered an outlier. Remember that for most other species, this will be much higher.

We then print these chromosomes: they are bounded by telomeres and composed of 50 kbp chromosome segments (the last segment is sized with the remainder). Each segment will be colored in blue, with a red-green component based on the linear normalization between two outlier values. Each chromosome segment will either be 50 kbp, or potentially smaller if it's the last one of the chromosome. The output is shown in the following diagram:

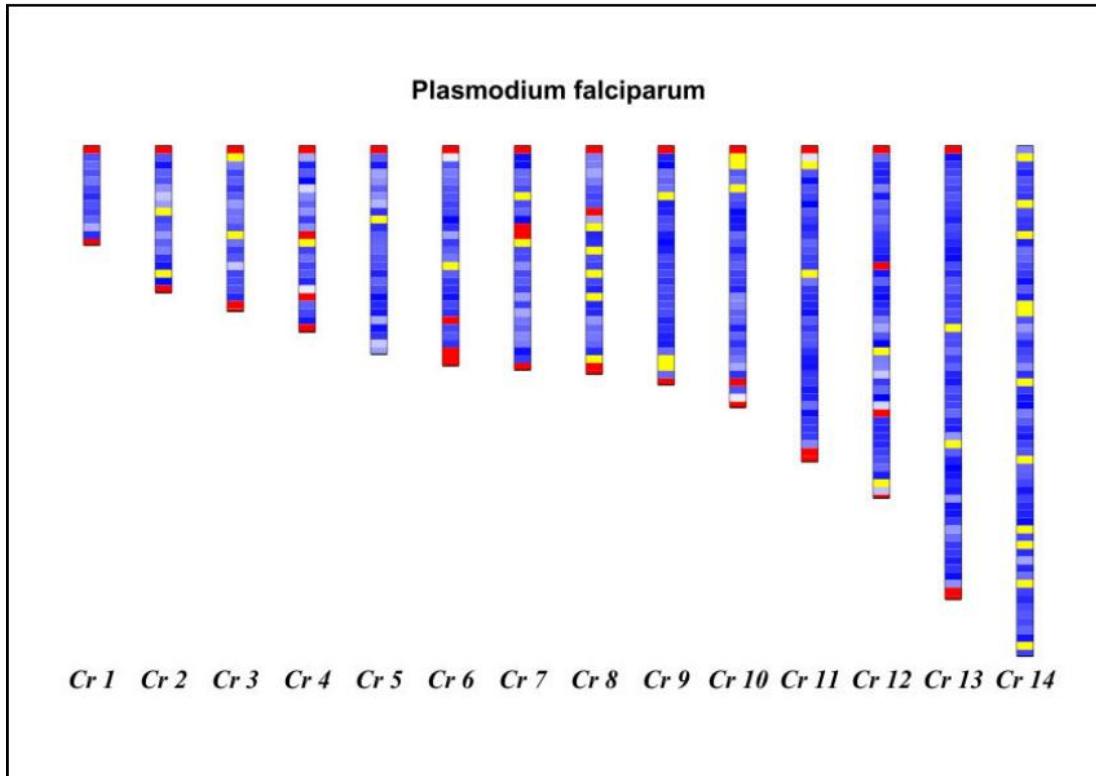


Figure 1: The 14 chromosomes of *Plasmodium falciparum*, color-coded with the GC content (red is more than 22 percent, yellow less than 17 percent, and the blue shades represent a linear gradient between both numbers)



Biopython code has evolved over time, from before Python was such a fashionable language. In the past, the availability of libraries was quite limited. The usage of reportlab can be seen mostly as a legacy issue. I suggest that you learn just enough from it to use it with Biopython. If you are planning on learning a modern plotting library in Python, you will probably want to consider matplotlib, Bokeh, HoloViews, or Python's version of ggplot (or other visualization alternatives, such as Mayavi, Visualization Toolkit (VTK), or even Blender API).

Finally, you can print the image inline in the Notebook:

```
from IPython.core.display import Image  
Image("falciparum.png")
```

## There's more...

*P. falciparum* is a reasonable example of a eukaryote with a small genome that allows you to perform a small data exercise with enough features, while still being useful for most eukaryotes. Of course, there are no sex chromosomes (such as X/Y in humans), but these should be easy to process because reference genomes do not deal with ploidy issues.

*P. falciparum* does have a mitochondrion, but we will not deal with it here due to space constraints. Biopython does have the functionality to print circular genomes, which you can also use with bacteria. With regards to bacteria and viruses, these genomes are much easier to process because their size is very small.

## See also

- You can find many reference genomes of model organisms in Ensembl at <http://www.ensembl.org/info/data/ftp/index.html>.
- As usual, National Center for Biotechnology Information (NCBI) also provides a large list of genomes at <http://www.ncbi.nlm.nih.gov/genome/browse/>.
- There are plenty of websites dedicated to a single organism (or a set of related organisms). Apart from PlasmoDB (<http://plasmadb.org/plasmo/>), from which you downloaded the *P. falciparum* genome, you will find VectorBase (<https://www.vectorbase.org/>) in the next recipe for disease vectors. FlyBase (<http://flybase.org/>) for *Drosophila* is also worth mentioning, but do not forget to search for your organism of interest.

## Dealing with low-quality genome references

Unfortunately, not all reference genomes will have the quality of *P. falciparum*. Apart from some model species (for example, humans, or the common fruit fly *Drosophila melanogaster*) and a few others, most reference genomes could use some improvement. In this recipe, we will look at how to deal with reference genomes of lower quality.

## Getting ready

In keeping with the malaria theme, we will use the reference genomes of two mosquitoes that are vectors of malaria: *Anopheles gambiae* (which is the most important vector of malaria and can be found in sub-Saharan Africa) and *Anopheles atroparvus*, a malaria vector in Europe (while the disease has been eradicated in Europe, this vector is still around). The *A. gambiae* genome is of reasonable quality. Most chromosomes have been mapped, although the Y chromosome still needs some work. There is a fairly large unknown chromosome, probably composed of bits of X and Y chromosomes, as well as midgut microbiota. This genome has a reasonable amount of positions that are not called (that is, you will find Ns instead of ACTGs). The *A. atroparvus* genome is still in the scaffold format. Unfortunately, this is what you will find for many non-model species.

Note that we will up the ante a bit. The *Anopheles* genome is one order of magnitude bigger than the *P. falciparum* genome (but one order of magnitude smaller than most mammals).

We will use Biopython, which you installed in Chapter 1, Python and the Surrounding Software Ecology. As usual, this recipe is available from the Jupyter Notebook at Chapter03/Low\_Quality.ipynb, in the code bundle of this book.

## How to do it...

Let's take a look at the following steps:

Let's start by listing the chromosomes of the *A. gambiae* genome:

```
import gzip
from Bio import SeqIO
gambiae_name = 'gambiae.fa.gz'
atroparvus_name = 'atroparvus.fa.gz'
recs = SeqIO.parse(gzip.open(gambiae_name, 'rt', encoding='utf-8'),
'fasta')
for rec in recs:
    print(rec.description)
```

This will produce the following output:

```
chromosome:AgamP3:2L:1:49364325:1 chromosome 2L
chromosome:AgamP3:2R:1:61545105:1 chromosome 2R
chromosome:AgamP3:3L:1:41963435:1 chromosome 3L
chromosome:AgamP3:3R:1:53200684:1 chromosome 3R
chromosome:AgamP3:UNKN:1:42389979:1 chromosome UNKN
chromosome:AgamP3:X:1:24393108:1 chromosome X
chromosome:AgamP3:Y_unplaced:1:237045:1 chromosome Y_unplaced
```

The code is quite straightforward. We use the gzip module because the files of larger genomes are normally compressed. We can see four chromosome arms (2L, 2R, 3L, and 3R), the X chromosome, and the Y chromosome, which is quite small and has a name that all but indicates that it might not be in the best state. Also, the unknown (UNKN) chromosome is actually a quite large proportion of the reference genome, to the tune of a chromosome arm.

Do not perform this with *An. atroparvus* or you will get more than a thousand entries, courtesy of the scaffold status.

We will now check uncalled positions (Ns) and their distribution for the *A. gambiae* genome:

```
recs = SeqIO.parse(gzip.open(gambiae_name, 'rt', encoding='utf-8'),
    'fasta')
chrom_Ns = {}
chrom_sizes = {}
for rec in recs:
    chrom = rec.description.split(':')[2]
    if chrom in ['UNKN', 'Y_unplaced']:
        continue
    chrom_Ns[chrom] = []
    on_N = False
    curr_size = 0
    for pos, nuc in enumerate(rec.seq):
        if nuc in ['N', 'n']:
            curr_size += 1
            on_N = True
        else:
            if on_N:
                chrom_Ns[chrom].append(curr_size)
                curr_size = 0
            on_N = False
    if on_N:
        chrom_Ns[chrom].append(curr_size)
    chrom_sizes[chrom] = len(rec.seq)
for chrom, Ns in chrom_Ns.items():
    size = chrom_sizes[chrom]
    print('%s (%s): %%Ns (%.1f), num Ns: %d, max N: %d' %
          (chrom, size, 100 * sum(Ns) / size, len(Ns), max(Ns)))
```

This code will take some time to run, so please be patient, because we will inspect each and every base pair of autosomes. As usual, we will reopen and re-read the file to save memory.

We have two dictionaries: one dictionary with chromosome sizes and another with the distribution of the sizes of runs of Ns. To calculate the runs of Ns, we traverse all autosomes (noting when an N position starts and ends). We then print the basic statistics of the distribution of Ns:

```
2L (49364325): %Ns (1.7), num Ns: 957, max N: 28884
2R (61545105): %Ns (2.3), num Ns: 1658, max N: 36427
3L (41963435): %Ns (2.9), num Ns: 1272, max N: 31063
3R (53200684): %Ns (1.8), num Ns: 1128, max N: 24292
X (24393108): %Ns (4.1), num Ns: 1287, max N: 21132
```

So, for the 2L chromosome arm (with a size of 49 Mbp), 1.7 percent are N calls divided by 957 runs. The biggest run is 28884 bps. Note that the X chromosome has the highest fraction of positions with Ns.

We will now turn our attention to the *A. Atroparvus* genome. Let's count the number of scaffolds, along with the distribution of scaffold sizes:

```
import numpy as np
recs = SeqIO.parse(gzip.open(atroparvus_name, 'rt',
encoding='utf-8'), 'fasta')
sizes = []
size_N = []
for rec in recs:
    size = len(rec.seq)
    sizes.append(size)
    count_N = 0
    for nuc in rec.seq:
        if nuc in ['n', 'N']:
            count_N += 1
    size_N.append((size, count_N / size))
print(len(sizes), np.median(sizes), np.mean(sizes),
max(sizes), min(sizes),
np.percentile(sizes, 10), np.percentile(sizes, 90))
```

This code is similar to the previous point, but we do print slightly more detailed statistics using NumPy, so we get the following:

```
(1371, 8304.0, 163596.0, 20238125, 1004, 1563.0, 56612.0)
```

Thus, we have 1371 scaffolds (against seven entries on the *A. gambiae* genome) with a median size of 8304.0 (mean of 163536.0). The biggest scaffold has 2 Mbp and the smallest scaffold has 1,004 bp. The tenth percentile for size is 1563.0 and the ninetieth is 56612.0.

Finally, let's plot the fraction of the scaffold, that is, N, as a function of its size:

```
%matplotlib inline
import matplotlib.pyplot as plt
small_split = 4800
large_split = 540000
fig, axs = plt.subplots(1, 3, figsize=(16, 9), squeeze=False,
sharey=True)
xs, ys = zip(*[(x, 100 * y) for x, y in size_N if x <=
small_split])
axs[0, 0].plot(xs, ys, '.')
xs, ys = zip(*[(x, 100 * y) for x, y in size_N if x > small_split
and x <= large_split])
axs[0, 1].plot(xs, ys, '.')
axs[0, 1].set_xlim(small_split, large_split)
xs, ys = zip(*[(x, 100 * y) for x, y in size_N if x > large_split])
axs[0, 2].plot(xs, ys, '.')
axs[0, 0].set_ylabel('Fraction of Ns', fontsize=12)
axs[0, 1].set_xlabel('Contig size', fontsize=12)
fig.suptitle('Fraction of Ns per contig size', fontsize=26)
```

The preceding code will generate the output shown in the following diagram, in which we split the chart into three parts based on the scaffold size: one for scaffolds with less than 4,800 bp, one for scaffolds between 4,800 and 540,000 bp, and one for larger ones. The fraction of Ns is very low for small scaffolds (always below 3.5%); for medium scaffolds it has large variance (sizes between 0 and above 90 percent), and tighter variance (between 0 and 25 percent) for the largest scaffolds:

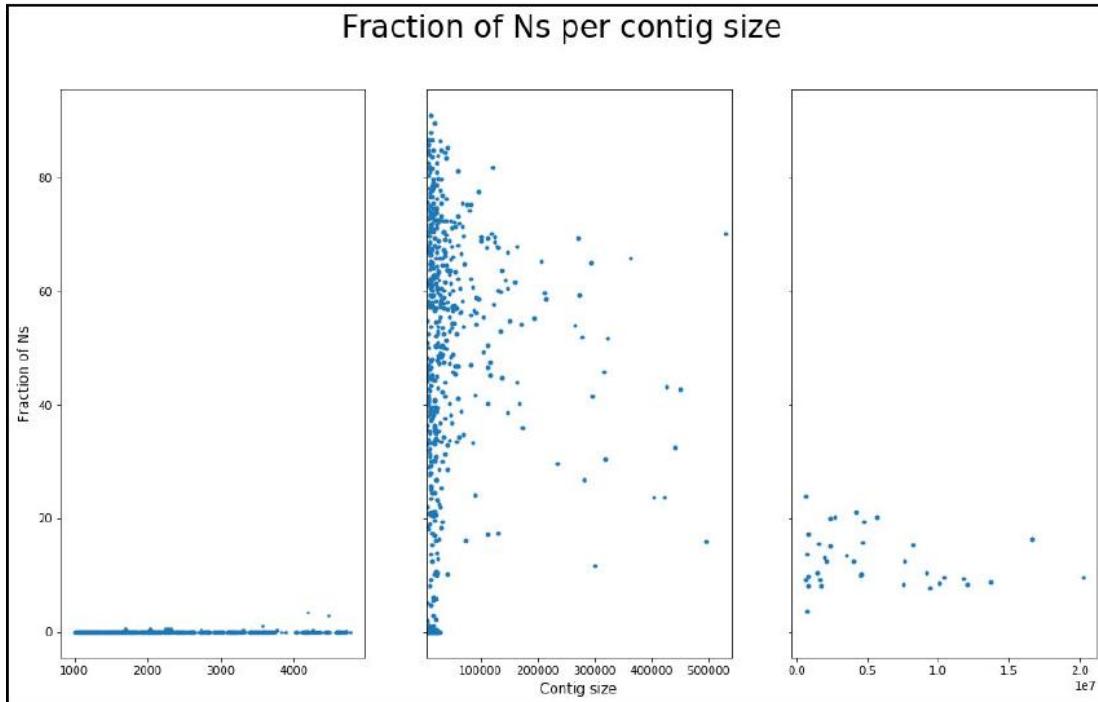


Figure 2: Fraction of scaffolds that are N as a function of their size

## There's more...

Sometimes, reference genomes carry extra information, for example, the *Anopheles gambiae* genome is soft masked. This means that some procedures were run on the genome to identify areas of low complexity (which are normally more problematic to analyze). This will be annotated by capitalization: ACTG will be high complexity, whereas actg will be low.

Reference genomes with lots of scaffolds are more than an inconvenient hassle. For example, very small scaffolds (say, below 2,000 bp) may have mapping problems when using an aligner (such as Burrows-Wheeler Aligner (bwa)), especially at the extremes (most scaffolds will have mapping problems at their extremes, but these will be of a much larger proportion of the scaffold if it's small). If you are using a reference genome like this to align, you will want to consider ignoring the pair information (assuming that you have paired-end reads) when mapping to small scaffolds, or at least measure the impact of the scaffold size in the performance of your aligner. In any case, the general comment is that you should be careful, because the scaffold size and number will rear its ugly head from time to time.

With these genomes, only complete ambiguity (N) was identified. Note that other genome assemblies will give you an intermediate code between the total ambiguity and certainty (ACTG).

## See also

- Tools like RepeatMasker are used to find areas of the genome with low complexity; check out <http://www.repeatmasker.org/> for more information
- IUPAC ambiguity codes may be useful to have in hand when processing other genomes; check out <http://www.bioinformatics.org/sms/iupac.html> for more information

## Traversing genome annotations

Having a genome sequence is interesting, but we will want to extract features from it, such as genes, exons, and coding sequences. This type of annotation information is made available in Generic Feature Format (GFF) and General Transfer Format (GTF) files. In this recipe, we will look at how to parse and analyze GFF files, using the annotation of the *Anopheles gambiae* genome as an example.

## Getting ready

Use the Chapter03/Annotations.ipynb Notebook file, which is provided in the code bundle of this book.

## How to do it...

Let's take a look at the following steps:

Let's start by creating an annotation database with gffutils, based on our GFF file:

```
import gffutils
import sqlite3
try:
    db = gffutils.create_db('gambiae.gff.gz', 'ag.db')
except sqlite3.OperationalError:
    db = gffutils.FeatureDB('ag.db')
```

The gffutils library creates a SQLite database to store annotations efficiently. Here, we will try to create the database, but if it already exists, we will use the existing one. This step can be time-consuming.

Now, we will list all the available feature types and count them:

```
print(list(db.featuretypes()))
for feat_type in db.featuretypes():
    print(feat_type, db.count_features_of_type(feat_type))
```

These features will include contigs, genes, exons, transcripts, and so on. Note that we will use the gffutils package's featuretypes function. It will return a generator, but we will convert it to a list (it's safe to do so here).

Let's list all of the contigs:

```
for contig in db.features_of_type('contig'):
    print(contig)
```

This will show us that there is annotation information for all chromosome arms and sex chromosomes, mitochondrion, and the unknown chromosome.

Now, let's extract a lot of useful information per chromosome, such as the number of genes, number of transcripts per gene, number of exons, and so on:

```
from collections import defaultdict
num_mRNAs = defaultdict(int)
num_exons = defaultdict(int)
max_exons = 0
max_span = 0
for contig in db.features_of_type('contig'):
    cnt = 0
    for gene in db.region((contig.seqid, contig.start, contig.end),
```

```

featuretype='gene'):
    cnt += 1
    span = abs(gene.start - gene.end) # strand
    if span > max_span:
        max_span = span
        max_span_gene = gene
    my_mRNAs = list(db.children(gene, featuretype='mRNA'))
    num_mRNAs[len(my_mRNAs)] += 1
    if len(my_mRNAs) == 0:
        exon_check = [gene]
    else:
        exon_check = my_mRNAs
    for check in exon_check:
        my_exons = list(db.children(check, featuretype='exon'))
        num_exons[len(my_exons)] += 1
        if len(my_exons) > max_exons:
            max_exons = len(my_exons)
            max_exons_gene = gene
        print('contig %s, number of genes %d' % (contig.seqid, cnt))
        print('Max number of exons: %s (%d)' % (max_exons_gene.id,
max_exons))
        print('Max span: %s (%d)' % (max_span_gene.id, max_span))
        print(num_mRNAs)
        print(num_exons)

```

We will traverse all contigs (features\_of\_type), extracting all genes (region). In each gene, we count the number of alternative transcripts. If there are none (note that this is probably an annotation issue and not a biological one), we count the exons (children). If there are several transcripts, we count the exons per transcript. We also account for the span size to check for the gene spanning the largest region. We follow a similar procedure to find the gene and the largest number of exons. Finally, we print a dictionary with the distribution of the number of alternative transcripts per gene (num\_mRNAs) and the distribution of number of exons per transcript (num\_exons).

## There's more...

There are many variations of the GFF/GTF format. There are different GFF versions and many unofficial variations. If possible, choose the GFF Version 3. However, the ugly truth is that you will find it very difficult to process files. The gffutils library tries as best as it can to accommodate this. Indeed, much of the documentation of this library is concerned with helping you process all kinds of awkward variations (refer to <https://github.com/brentp/gffutils/blob/master/examples.html>).

There is an alternative to using gffutils (either because your GFF file is strange or because you do not like the library interface, or its dependency on a SQL backend). Parse the file yourself manually. If you look at the format, you will notice that it's not very complex. If you are only performing a one-off operation, then maybe manual parsing is good enough. Of course, one-off operations tend to not be that good in the long run.

Also, note that the quality of annotations tends to vary a lot. As the quality increases, so does the complexity. Just check the human annotation for an example of this. You can expect that, over time, as our knowledge of organisms evolves, the quality and complexity of annotations will increase.

## See also

- The GFF spec can be found at <https://www.sanger.ac.uk/resources/software/gff/spec.html>.
- Probably the best explanation on the GFF format, along with the most common versions and GTF, can be found at <http://gmod.org/wiki/GFF3>.
- Somewhat related is the Browser Extensible Data (BED) format to specify annotations; this is mostly used interactively to specify tracks for genome viewers. You can find it at <http://genome.ucsc.edu/FAQ/FAQformat.html#format1>, although processing is quite trivial.

## Extracting genes from a reference using annotations

In this recipe, we will look at how to extract a gene sequence with the help of an annotation file to get its coordinates against a reference FASTA. We will use the *Anopheles gambiae* genome, along with its annotation file (as per the previous two recipes). We will first extract the voltage-gated sodium channel (VGSC) gene, which is involved in resistance to insecticides.

## Getting ready

If you have followed the previous two recipes, you will be ready. If not, download the *Anopheles gambiae* FASTA file, along with the GTF file. You also need to prepare the gffutils database:

```
import gffutils
import sqlite3
try:
    db = gffutils.create_db('gambiae.gff.gz', 'ag.db')
except sqlite3.OperationalError:
    db = gffutils.FeatureDB('ag.db')
```

As usual, you will find all of this in the Chapter03/Getting\_Gene.ipynb Notebook file.

## How to do it...

Let's take a look at the following steps:

Let's start by retrieving the annotation information for our gene:

```
import gzip
from Bio import Alphabet, Seq, SeqIO
gene_id = 'AGAP004707'
gene = db[gene_id]
print(gene)
print(gene.seqid, gene.strand)
```

The gene\_id was retrieved from VectorBase, an online database of the genomics of disease vectors. For other specific cases, you will need to know the ID of your gene (which will be dependent on species and database). The output will be as follows:

```
2L VectorBase gene 2358158 2431617 . + .
ID=AGAP004707;biotype=protein_coding
2L +
```

Note that the gene is on the 2L chromosome arm and coded in the positive direction (+ strand).

Let's hold the sequence for the 2L chromosome arm in memory (it's just a single chromosome, so we will indulge):

```
recs = SeqIO.parse(gzip.open('gambiae.fa.gz', 'rt',
encoding='utf-8'), 'fasta')
for rec in recs:
    print(rec.description)
    if rec.description.split(':')[2] == gene.seqid:
        my_seq = rec.seq
        break
print(my_seq.alphabet)
```

The output will be as follows:

```
chromosome:AgamP3:2L:1:49364325:1 chromosome 2L
SingleLetterAlphabet()
```

Note the alphabet of the sequence.

Let's create a function to construct a gene sequence for a list of CDSs:

```
def get_sequence(chrom_seq, CDSs, strand):
    seq = Seq.Seq("", alphabet=Alphabet.IUPAC.unambiguous_dna)
    for CDS in CDSs:
        my_cds = Seq.Seq(str(my_seq[CDS.start - 1:CDS.end]),
alphabet=Alphabet.IUPAC.unambiguous_dna)
        seq += my_cds
    return seq if strand == '+' else seq.reverse_complement()
```

This function will receive a chromosome sequence (in our case, the 2L arm), a list of coding sequences (retrieved from the annotation file), and the strand.

There are several important issues to note here. We will construct a sequence of the unambiguous DNA type (we will need this for translation). The SingleLetterAlphabet of the previous step will have to be converted. We also have to be very careful with the start and end of the sequence (note that the GFF file is 1-based, whereas the Python array is 0-based). Finally, we return the reverse complement if the strand is negative.

Although we have the gene\_id at hand, we actually want just one of the transcripts of the three available for this gene, so we need to choose one:

```
mRNAs = db.children(gene, featuretype='mRNA')
for mRNA in mRNAs:
    print(mRNA.id)
    if mRNA.id.endswith('RA'):
        break
```

We will now get the coding sequence for our transcript, then get the gene sequence, and translate it:

```
CDSs = db.children(mRNA, featuretype='CDS', order_by='start')
gene_seq = get_sequence(my_seq, CDSs, gene.strand)
print(len(gene_seq), gene_seq)
prot = gene_seq.translate()
print(len(prot), prot)
```

Let's get the gene that is coded in the negative strand direction. We will just take the gene next to VGSC (which happens to be the negative strand):

```
reverse_transcript_id = 'AGAP004708-RA'
reverse_CDSs = db.children(reverse_transcript_id,
featuretype='CDS', order_by='start')
reverse_seq = get_sequence(my_seq, reverse_CDSs, '-')
print(len(reverse_seq), reverse_seq)
reverse_prot = reverse_seq.translate()
print(len(reverse_prot), reverse_prot)
```

Here, I evaded getting all of the information about the gene, and just hardcoded the transcript ID. The point is that you should make sure your code works, irrespective of the strand.

## There's more...

This is a simple recipe that exercises several concepts that have been presented in this and the previous chapter, Chapter 2, Next Generation Sequencing. While it's conceptually trivial, it's unfortunately full of booby traps.



When using different databases, be sure that the genome assembly versions are synchronized. It would be a serious and potentially silent bug to use different versions. Remember that different versions (at least on the major version number) have different coordinates. For example, position 1,234 on chromosome 3 on build 36 of the human genome will probably refer to a different SNP than 1,234 on build 38. With human data, you will probably find a lot of chips on build 36, and plenty of whole genome sequences on build 37, whereas the most recent human assembly is build 38. With our Anopheles example, you will have versions 3 and 4 around. This will happen with most species. So, be aware!

There is also the issue of 0-indexed arrays in Python versus 1-indexed genomic databases. Nonetheless, be aware that some genomic databases may also be 0-indexed.

There are also two sources of confusion: the transcript versus the gene choice, as in more rich annotation databases. Here, you will have several alternative transcripts (if you want to look at a rich-to-the-point-of-confusing database, refer to the human annotation database). Also, fields tagged with exon will have more information compared to the coding sequence. For this purpose, you will want the CDS field.

Finally, there is the strand issue, where you will want to translate based on the reverse complement.

## See also

- You can download MySQL tables for Ensembl at <http://www.ensembl.org/info/data/mysql.html>.
- The UCSC genome browser can be found at <http://genome.ucsc.edu/>. Be sure to check the download area at <http://hgdownload.soe.ucsc.edu/downloads.html>.
- With a reference to genomes, you can find GTFs of model organisms in Ensembl at <http://www.ensembl.org/info/data/ftp/index.html>.
- A simple explanation between CDSs and exons can be found at <https://www.biostars.org/p/65162/>.

## Finding orthologues with the Ensembl REST API

Here, we will examine how to look for orthologues for a certain gene. This simple recipe will not only introduce orthology retrieval, but also how to use REST APIs on the web to access biological data. Last, but surely not least, it will serve as an introduction on how to access the Ensembl database using the programmatic API.

In our example, we will try to find any orthologue for the human lactase (LCT) gene on the horse genome.

## Getting ready

This recipe will not require any pre-downloaded data, but as we are using the web APIs, internet access will be needed. The amount of data being transferred will be limited.

We will also make use of the requests library to access Ensembl. The request API is an easy-to-use wrapper for web requests. Of course, you can use the standard Python libraries, but these are much more cumbersome.

As usual, you can find this content in the Chapter03/Orthology.ipynb Notebook file.

## How to do it...

Let's take a look at the following steps:

We will start by creating a support function to perform a web request:

```
import requests

ensembl_server = 'http://rest.ensembl.org'

def do_request(server, service, *args, **kwargs):
    url_params = ""
    for a in args:
        if a is not None:
            url_params += '/' + a
    req = requests.get('%s/%s%s' % (server, service, url_params),
                       params=kwargs, headers={'Content-Type': 'application/json'})
    if not req.ok:
        req.raise_for_status()
    return req.json()
```

We start by importing the requests library and specifying the root URL. Then, we create a simple function that will take the functionality to be called (see the following examples) and generate a complete URL. It will also add optional parameters and specify the payload to be of the JSON type (just to get a default JSON answer). It will return the response in JSON format. This is typically a nested Python data structure of lists and dictionaries.

Then, we check all the available species on the server, which is around 110 at the time of writing this book:

```
answer = do_request(ensembl_server, 'info/species')
for i, sp in enumerate(answer['species']):
    print(i, sp['name'])
```

Note that this will construct the URL starting with the prefix `http://rest.ensembl.org/info/species` for the REST request. The link above will not work on your browser, by the way; it is to be used only via a REST API.

We will now try to find any HGNC databases on the server related to human data:

```
ext_dbs = do_request(ensembl_server, 'info/external_dbs',
    'homo_sapiens', filter='HGNC%')
print(ext_dbs)
```

We restrict the search to human-related databases (`homo_sapiens`). We also filter databases starting with HGNC (this filtering uses the SQL notation). HGNC is the HUGO database. We want to make sure that it's available, because the HUGO database is responsible for curating human gene names and maintains our LCT identifier.

Now that we know that the LCT identifier is probably available, we want to retrieve the Ensembl ID for the gene, as shown in the following code:

```
answer = do_request(ensembl_server, 'lookup/symbol',
    'homo_sapiens', 'LCT')
print(answer)
lct_id = answer['id']
```



Different databases, as you probably know by now, will have different IDs for the same object. We will need to resolve our LCT identifier to the Ensembl ID. When you deal with external databases that relate to the same objects, ID translation between databases will probably be your first task.

Just for your information, we can now get the sequence of the area containing the gene. Note that this is probably the whole interval, and if you want to recover the gene, you will have to use a procedure similar to what we used in the previous recipe:

```
lct_seq = do_request(ensembl_server, 'sequence/id', lct_id)
print(lct_seq)
```

We can also inspect other databases known to Ensembl; refer to the following gene:

```
lct_xrefs = do_request(ensembl_server, 'xrefs/id', lct_id)
for xref in lct_xrefs:
    print(xref['db_display_name'])
    print(xref)
```

You will find different kinds of databases, such as the Vertebrate Genome Annotation (Vega) project, UniProt (see Chapter 7, Using the Protein Data Bank), and WikiGene.

Let's get the orthologues for this gene on the horse genome:

```
hom_response = do_request(ensembl_server, 'homology/id', lct_id,
                           type='orthologues', sequence='none')
homologies = hom_response['data'][0]['homologies']
for homology in homologies:
    print(homology['target']['species'])
    if homology['target']['species'] != 'equus_caballus':
        continue
    print(homology)
    print(homology['taxonomy_level'])
    horse_id = homology['target']['id']
```

We could have actually acquired the orthologues directly for the horse by specifying a target\_species parameter on do\_request. However, this code allows you to inspect all available orthologues.

You will get quite a lot of information about an orthologue, such as the taxonomic level of orthology (Boreoeutheria—placental mammals is the closest phylogenetic level between humans and horses), the Ensembl ID of the orthologue, the dN/dS ratio (non-synonymous to synonymous mutations), and the CIGAR string (refer to the previous chapter, Chapter 2, Next-Generation Sequencing) of differences among sequences. By default, you will also get the alignment of the orthologous sequence, but I have removed it to unclog the output.

Finally, let's look for the horse\_id Ensembl record:

```
horse_req = do_request(ensembl_server, 'lookup/id', horse_id)
print(horse_req)
```

From this point onward, you can use the previous recipe methods to explore the LCT horse orthologue.

## There's more...

You can find a detailed explanation of all the functionalities available at <http://rest.ensembl.org/>. This includes all the interfaces and Python code snippets, among other languages.

If you are interested in paralogues, this information can be retrieved quite trivially from the preceding recipe. On the call to homology/id, just replace the type with paralogues.

If you have heard of Ensembl, you have probably heard of an alternative service from UCSC: the Genome Browser (<http://genome.ucsc.edu/>). From the perspective of the user interface, they are on the same level. From a programmatic perspective, Ensembl is probably more mature. Access to NCBI Entrez databases was covered in the previous chapter, Chapter 2, Next Generation Sequencing.

Another completely different strategy to interface programmatically with Ensembl will be to download raw tables and inject them into a local MySQL database. Be aware that this will be quite an undertaking in itself (you will probably just want to load a very small subset of tables). However, if you intend to be very intensive in terms of usage, you may have to consider creating a local version of part of the database. If this is the case, you may want to reconsider the UCSC alternative, as it's as good as Ensembl from the local database perspective.

## Retrieving gene ontology information from Ensembl

In this recipe, we will introduce the usage of gene ontology information again by querying the Ensembl REST API. Gene ontologies are controlled vocabularies to annotate genes and gene products. These are made available as trees of concepts (with more general concepts near the top of the hierarchy). There are three domains for gene ontologies: a cellular component, the molecular function, and the biological process.

## Getting ready

As with the previous recipe, we do not require any pre-downloaded data, but as we are using web APIs, internet access will be needed. The amount of data transferred will be limited.

As usual, you can find this content in the Chapter03/Gene\_Ontology.ipynb Notebook file. We will make use of the do\_request function, which is defined in the first step of the previous recipe (Finding orthologues with the Ensembl REST API). To draw GO trees, we will use pygraphviz, a graph-drawing library.

## How to do it...

Let's take a look at the following steps:

Let's start by retrieving all GO terms associated with the LCT gene (you can find out how to retrieve the Ensembl ID in the previous recipe). Remember that you will need the do\_request function from the previous recipe:

```
lct_id = 'ENSG00000115850'  
refs = do_request(ensembl_server, 'xrefs/id',  
lct_id, external_db='GO', all_levels='1')  
print(len(refs))  
print(refs[0].keys())  
for ref in refs:  
    go_id = ref['primary_id']  
    details = do_request(ensembl_server, 'ontology/id', go_id)  
    print('%s %s %s' % (go_id, details['namespace'],  
    ref['description']))  
    print('%s\n' % details['definition'])
```

Note the free-form definition and the varying namespace for each term. The first two of the eleven reported items in the loop are as follows (this may change when you run it, because the database may have been updated):

```
GO:0000016 molecular_function lactase activity  
"Catalysis of the reaction: lactose + H2O = D-glucose + D-  
galactose." [EC:3.2.1.108]
```

```
GO:0004553 molecular_function hydrolase activity, hydrolyzing O-  
glycosyl compounds  
"Catalysis of the hydrolysis of any O-glycosyl bond." [GOC:mah]
```

Let's concentrate on the lactase activity molecular function and retrieve more detailed information about it (the following go\_id comes from the previous step):

```
go_id = 'GO:0000016'  
my_data = do_request(ensembl_server, 'ontology/id', go_id)  
for k, v in my_data.items():  
    if k == 'parents':
```

```

for parent in v:
    print(parent)
    parent_id = parent['accession']
else:
    print("%s: %s" % (k, str(v)))
parent_data = do_request(ensembl_server, 'ontology/id', parent_id)
print(parent_id, len(parent_data['children']))

```

We print the lactase activity record (which is currently a node of the GO tree molecular function) and retrieve a list of potential parents. There is a single parent for this record. We retrieve it and print the number of children.

Let's retrieve all the general terms for the lactase activity molecular function (again, the parent and all other ancestors):

```

refs = do_request(ensembl_server, 'ontology/ancestors/chart',
go_id)
for go, entry in refs.items():
    print(go)
    term = entry['term']
    print('%s %s' % (term['name'], term['definition']))
    is_a = entry.get('is_a', [])
    print('\tis a: %s\n' % ', '.join([x['accession'] for x in
is_a]))

```

We retrieve the ancestor list by following the is\_a relationship (refer to the GO sites in the See also section for more details on the types of possible relationships).

Let's define a function to create a dictionary with the ancestor relationship for a term, along with some summary information for each term returned in a pair:

```

def get_upper(go_id):
    parents = {}
    node_data = {}
    refs = do_request(ensembl_server, 'ontology/ancestors/chart',
go_id)
    for ref, entry in refs.items():
        my_data = do_request(ensembl_server, 'ontology/id', ref)
        node_data[ref] = {'name': entry['term']['name'],
'children': my_data['children']}
        try:
            parents[ref] = [x['accession'] for x in entry['is_a']]
        except KeyError:
            pass # Top of hierarchy
    return parents, node_data

```

Finally, we will print a tree of relationships for the lactase activity term. For this, we will use the pygraphviz library:

```
parents, node_data = get_upper(go_id)
import pygraphviz as pgv
g = pgv.AGraph(directed=True)
for ofs, ofs_parents in parents.items():
    ofs_text = '%s\n%s' % (node_data[ofs]['name'].replace(',', ',\n'), ofs)
    for parent in ofs_parents:
        parent_text = '%s\n%s' %
(node_data[parent]['name'].replace(',', ',\n'), parent)
        children = node_data[parent]['children']
        if len(children) < 3:
            for child in children:
                if child['accession'] in node_data:
                    continue
                g.add_edge(parent_text, child['accession'])
        else:
            g.add_edge(parent_text, '...%d...' % (len(children) -
1))
    g.add_edge(parent_text, ofs_text)
print(g)
g.graph_attr['label']='Ontology tree for Lactase activity'
g.node_attr['shape']='rectangle'
g.layout(prog='dot')
g.draw('graph.png')
```

The following output shows the ontology tree for the lactase activity term:

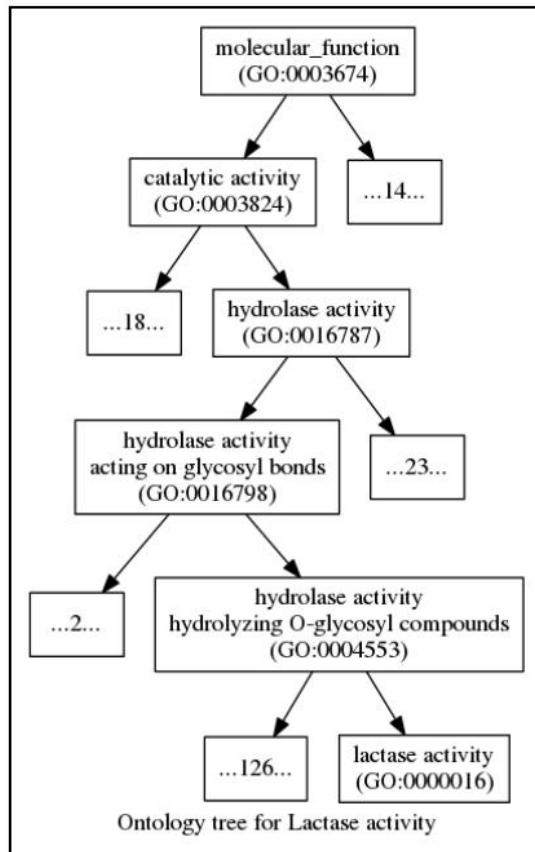


Figure 3: An ontology tree for the term "lactase activity" (the terms at the top are more general); the top of the tree is molecular\_function; for all ancestral nodes, the number of extra offspring is also noted (or enumerated, if less than three)

## There's more...

If you are interested in gene ontologies, your main port of call will be <http://www.geneontology.org>, where you will find much more information on this topic. Apart from molecular\_function, gene ontology also has a biological process and a cellular component. In our recipes, we have followed the hierarchical relationship is\_a, but others do exist partially. For example, "mitochondrial ribosome" (GO:0005761) is a cellular component, and is part of "mitochondrial matrix" (refer to <http://www.amigo.geneontology.org/amigo/term/GO:0005761#display-lineage-tab> and click on the Graph Views).

As with the previous recipe, you can download the MySQL dump of a gene ontology database (you may prefer to interact with the data in that way). For this, see <http://geneontology.org/page/download-go-annotations>. Again, expect to allocate some time to understanding the relational database schema. Also, note that there are many alternatives to Graphviz for plotting trees and graphs. We will return to this topic in the course of this book.

## See also

- As stated before, more so than Ensembl, the main resource for gene ontologies is <http://geneontology.org/>.
- For visualization, we are using the pygraphviz library, which is a wrapper on top of Graphviz (<http://www.graphviz.org/>).
- There are very good user interfaces for GO data, for example, AmiGO (<http://amigo.geneontology.org>) and QuickGO (<http://www.ebi.ac.uk/QuickGO/>).
- One of the most common analyses performed with GO is the gene enrichment analysis to check whether some GO terms are overexpressed or underexpressed in a certain gene set. The geneontology.org server uses Panther (<http://go.pantherdb.org/>), but other alternatives are available (such as DAVID, at <http://david.abcc.ncifcrf.gov/>).

# 4

# Population Genetics

In this chapter, we will cover the following recipes:

- Managing datasets with PLINK
- Introducing the Genepop format
- Exploring a dataset with Bio.PopGen
- Computing F-statistics
- Performing Principal Components Analysis
- Investigating population structure with admixture

## Introduction

Population genetics is the study of the changes of frequency of alleles in a population on the basis of selection, drift, mutation, and migration. The previous chapters focused mainly on data processing and cleanup; this is the first chapter in which we will actually infer interesting biological results.

There is a lot of interesting population genetics analysis based on sequence data, but as we already have quite a few recipes for dealing with sequence data, we will divert our attention somewhere else. Also, we will not cover genomic structural variation such as copy number variations (CNVs) or inversions here. We will concentrate on analyzing SNP data, which is one of the most common data types. We will perform many standard population genetic analyses with Python, such as fixation index, computing F-statistics, Principal Components Analysis (PCA), and study population structure.

We will use Python mostly as a scripting language that glues together applications that perform necessary computations, which is the old-fashioned way of doing things. Having said that, as the Python software ecology is still evolving, you can at least perform the PCA in Python using scikit-learn.

There is no such thing as a default file format for population genetics data. The bleak reality of this field is that there are plenty of formats, most of them developed with a specific application in mind; therefore, it's not generically applicable. Some of the efforts to create a more general format (or even just a file converter to support many formats) met with limited success. Furthermore, as our knowledge of genomics increases, we will require new formats anyway (for example, to support some kind of previously unknown genomic structural variation). Here, we will work with the formats of two widely used applications. One is PLINK (<https://www.cog-genomics.org/plink2>), which was originally developed to perform genome-wide association studies (GWAS) with human data, but has many more applications. The second format is Genepop (<http://kimura.univ-montp2.fr/~roussel/Genepop.htm>), which is widely used in the conservation genetics community. If you have next-generation sequencing (NGS) sequencing data, you may question, why not use variant call format (VCF)? Well, a VCF file is normally annotated to help with sequencing analysis, which you do not need at this stage (you should now have a filtered dataset). If you convert your Single-Nucleotide Polymorphism (SNP) calls from VCF to PLINK, you will get roughly a 95 percent reduction in terms of size (this is in comparison to a compressed VCF). More importantly, the computational cost of processing a VCF file is much bigger (think of processing all this highly-structured text) than the cost of the other two formats.

This chapter is closely tied to the next one. Here, we will work with real empirical data, whereas in the next chapter, Chapter 5, Population Genetics Simulation, we will simulate data. However, if you are interested in analyzing population genetics data, be sure to read the next chapter, where we will cover some ground analysis.

First, let's start with a discussion on file format issues and then continue to discuss interesting data analysis.

## Managing datasets with PLINK

Here, we will manage our dataset using PLINK. We will create subsets of our main dataset (from the HapMap project) that are suitable for analysis in the following recipes.



Note that neither PLINK nor any similar programs were developed for their file formats. There was probably no objective to become a default file standard for population genetics data. In this field, you will need to be ready to convert from format to format (for this, Python is quite appropriate) because every application that you will use will probably have its own quirky requirements. The most important point to learn from this recipe is that it's not formats that are being used, although these are relevant, but the "file conversion mentality". Apart from this, some of the steps in this recipe also convey genuine analytical techniques that you may want to consider using, for example, subsampling or linkage disequilibrium (LD) pruning.

## Getting ready

Throughout this chapter, we will use data from the International HapMap Project. You may recall that we used data from the 1,000 Genomes Project in Chapter 2, Next-Generation Sequencing, and that the HapMap project is in many ways the precursor of the 1,000 Genomes Project; instead of whole genome sequencing, genotyping was used. Most of the samples of the HapMap project were used in the 1,000 Genomes Project, so if you have read the recipes in Chapter 2, Next-Generation Sequencing, you will already have an idea of the dataset (including the available population). I will not introduce the dataset much more, but you can refer to Chapter 2, Next-Generation Sequencing, and, of course, the HapMap site (<http://www.hapmap.org>) for more information. Remember that we have genotyping data for many individuals split across populations around the globe. We will refer to these populations by their acronyms. Here is the list taken from <http://www.sanger.ac.uk/resources/downloads/human/hapmap3.html>:

Acronym	Population
ASW	African ancestry in Southwest USA
CEU	Utah residents with Northern and Western European ancestry from the CEPH collection
CHB	Han Chinese in Beijing, China
CHD	Chinese in Metropolitan Denver, Colorado
GIH	Gujarati Indians in Houston, Texas
JPT	Japanese in Tokyo, Japan
LWK	Luhya in Webuye, Kenya
MXL	Mexican ancestry in Los Angeles, California
MKK	Maasai in Kinyawa, Kenya
TSI	Toscani in Italy
YRI	Yoruba in Ibadan, Nigeria



We will be using data from the HapMap project which has, in practice, been replaced by the 1,000 Genomes Project. For the purpose of teaching population genetics programming techniques in Python, the HapMap Project dataset is more manageable than the 1,000 Genomes Project, as the data is considerably smaller. The HapMap samples are a subset of the 1,000 Genomes' samples. If you do research in human population genetics, you are strongly advised to use the 1,000 Genomes Project as a base dataset.

This will require a fairly big download (approximately 1 GB), which will have to be uncompressed. For this, you will need bzip2, which is available at <http://www.bzip.org/>. Make sure that you have approximately 20 GB of disk space for this chapter.

Decompress the PLINK file using the following commands:

```
bunzip2 hapmap3_r2_b36_fwd.consensus.qc.poly.map.bz2  
bunzip2 hapmap3_r2_b36_fwd.consensus.qc.poly.ped.bz2
```

Now, we have PLINK files; the MAP file has information on the marker position across the genome, whereas the PED file has actual markers for each individual, along with some pedigree information. We also downloaded a metadata file that contains information about each individual. Take a look at all these files and familiarize yourself with them. As usual, this is also available in the Chapter04/Data\_Formats.ipynb Notebook file, where everything has been taken care of.

Finally, most of this recipe will make heavy usage of PLINK (you should have installed at least version 1.9 from <https://www.cog-genomics.org/plink/2.0/>, not version 1.0x). Python will mostly be used as the glue language to call PLINK.

## How to do it...

Take a look at the following steps:

Let's get the metadata for our samples. We will load the population of each sample and note all individuals that are offspring of others in the dataset:

```
from collections import defaultdict  
f = open('relationships_w_pops_121708.txt')  
pop_ind = defaultdict(list)  
f.readline() # header  
offspring = []  
for l in f:  
    toks = l.rstrip().split('\t')
```

```
fam_id = toks[0]
ind_id = toks[1]
mom = toks[2]
dad = toks[3]
if mom != '0' or dad != '0':
    offspring.append((fam_id, ind_id))
pop = toks[-1]
pop_ind[pop].append((fam_id, ind_id))
f.close()
```

This will load a dictionary where population is the key (CEU, YRI, and so on) and its value is the list of individuals in that population. This dictionary will also store information on whether the individual is the offspring of another. Each individual is identified by the family and individual ID (which are found on the PLINK file). The file provided by the HapMap project is a simple tab-delimited file, which is not difficult to process. While we are reading the files using standard Python text processing, this is a typical example where pandas would help.

There is an important point to make here: the reason this information is provided on a separate, ad hoc file is because the PLINK format makes no provision for the population structure (this format makes provision only for the case/control information for which PLINK was designed). This is not a flaw of the format, as it was never designed to support standard population genetic studies (it's a GWAS tool). However, this is a general feature of population genetics' data formats: whichever you end up working with, there will be something important missing.

We will use this metadata in other recipes in this chapter. We will also perform some consistency analysis between the metadata and the PLINK file, but we will defer this to the next recipe.

Now, let's subsample the dataset at 10 percent and 1 percent of the number of markers, as follows:

```
import os
os.system('plink --recode --file
hapmap3_r2_b36_fwd.consensus.qc.poly --noweb --out hapmap10 --thin
0.1 --geno 0.1')
os.system('plink --recode --file
hapmap3_r2_b36_fwd.consensus.qc.poly --noweb --out hapmap1 --thin
0.01 --geno 0.1')
```

Actually, with Jupyter Notebook, you can just do this instead:

```
!plink --recode --file hapmap3_r2_b36_fwd.consensus.qc.poly --noweb  
--out hapmap10 --thin 0.1 --geno 0.1  
!plink --recode --file hapmap3_r2_b36_fwd.consensus.qc.poly --noweb  
--out hapmap1 --thin 0.01 --geno 0.1
```

Note the subtlety that you will not really get 1 or 10 percent of the data; each marker will have a 1 or 10 percent chance of being selected, so you will get approximately 1 or 10 percent of the markers.

Obviously, as the process is random, different runs will produce different marker subsets. This will have important implications further down the road. If you want to replicate the exact same result, you can nonetheless use the --seed option.

We will also remove all SNPs that have a genotyping rate of lower than 90 percent (with the --geno 0.1 parameter).

There is nothing special about Python in this code, but there are two reasons you may want to subsample your data. First, if you are performing exploratory analysis of your own dataset, you may want to start with a smaller version because it will be easy to process. Also, you will have a broader view of your data. Second, some analytical methods may not require all your data (indeed, some methods might not be even able to use all of your data). Be very careful with the last point though; that is, for every method that you use to analyze your data, be sure that you understand the data requirements for the scientific questions you want to answer. Feeding too much data may be okay normally (even if you pay a time and memory penalty), but feeding too little will lead to unreliable results.



Now, let's generate subsets with just the autosomes (that is, let's remove the sex chromosomes and mitochondria), as follows:

```
def get_non_auto_SNPs(map_file, exclude_file):  
    f = open(map_file)  
    w = open(exclude_file, 'w')  
    for l in f:  
        toks = l.rstrip().split('\t')  
        chrom = int(toks[0])  
        rs = toks[1]  
        if chrom > 22:  
            w.write('%s\n' % rs)  
    w.close()  
get_non_auto_SNPs('hapmap1.map', 'exclude1.txt')
```

```
get_non_auto_SNPs('hapmap10.map', 'exclude10.txt')
os.system('plink --recode --file hapmap1 --noweb --out hapmap1_auto
--exclude exclude1.txt')
os.system('plink --recode --file hapmap10 --noweb --out
hapmap10_auto --exclude exclude10.txt')
```

Let's create a function that generates a list with all SNPs not belonging to autosomes. In PLINK, this means a chromosome number above 22 (for 22 human autosomes). If you use another species, be careful with your chromosome coding because PLINK is geared toward human data. If your species are diploid and have less than 23 autosomes and a sex determination system, that is, X/Y, this will be straightforward; if not, refer to [https://www.cog-genomics.org/plink2/input#allow\\_extra\\_chr](https://www.cog-genomics.org/plink2/input#allow_extra_chr) for some alternatives (similar to the --allow-extra-chr flag).

We then create autosome-only PLINK files for subsample datasets of 10 and 1 percent (prefixed as hapmap10\_auto and hapmap1\_auto).

Let's create some datasets without offspring. These will be needed for most population genetic analysis, which requires unrelated individuals to a certain degree:

```
os.system('plink --file hapmap10_auto --filter-founders --recode --
out hapmap10_auto_noofs')
```

This step is representative of the fact that most population genetic analysis require samples to be unrelated to a certain degree. Obviously, as we know that some offspring are in HapMap, we remove them. However, note that with your dataset, you are expected to be much more refined than this. For instance, run plink --genome or use another program to detect related individuals. The fundamental point here is that you have to dedicate some effort to detect related individuals in your samples; this is not a trivial task.

We will also generate an LD-pruned dataset, as required by many PCA and admixture algorithms, as follows:

```
os.system('plink --file hapmap10_auto_noofs --indep-pairwise 50 10
0.1 --out keep')
os.system('plink --file hapmap10_auto_noofs --extract keep.prune.in
--recode --out hapmap10_auto_noofs_ld')
```



The first step generates a list of markers to be kept if the dataset is LD-pruned. This uses a sliding window of 50 SNPs, advancing by 10 SNPs at a time with a cut value of 0.1. The second step extracts SNPs from the list that was generated earlier.

Let's `recode` a couple of cases in different formats:

```
os.system('plink --file hapmap10_auto_noofs_ld --recode12 tab --out  
hapmap10_auto_noofs_ld_12')  
os.system('plink --make-bed --file hapmap10_auto_noofs_ld --out  
hapmap10_auto_noofs_ld')
```

The first operation will convert a PLINK format that uses nucleotide letters from the ACTG to another, which recodes alleles with 1 and 2. We will use this in the Performing Principal Components Analysis recipe later.

The second operation recodes a file in a binary format. If you work inside PLINK (using the many useful operations that PLINK has), the binary format is probably the most appropriate format (offering, for example, smaller file size). We will use this in the admixture recipe.

We will also extract a single chromosome (2) for analysis. We will start with the autosome dataset, which has been subsampled at 10 percent:

```
os.system('plink --recode --file hapmap10_auto_noofs --chr 2 --out  
hapmap10_auto_noofs_2')
```

## There's more...

There are many reasons why you might want to create different datasets for analysis. You may want to perform some fast initial exploration of data; for example, if the analysis algorithm that you plan to use has some data format requirements or a constraint on the input, such as the number of markers or relationships among individuals. Chances are that you will have lots of subsets to analyze (unless your dataset is very small to start with, for instance, a microsatellite dataset).

This may seem like a minor point, but it's not: be very careful with file naming (note that I have followed some simple conventions while generating filenames). Make sure that the name of the file gives some information about subset options. When you perform the downstream analysis, you will want to be sure that you choose the correct dataset; you will want your dataset management to be agile and reliable, above all. The worst thing that can happen is that you create an analysis with an erroneous dataset that does not obey constraints required by software.

At the time of writing this book, there are two PLINK versions: 1.x, and the fast approaching version 2. I strongly suggest that you use version 2 with beta as well, because the speed and memory improvements in version 2 are impressive.

The LD-pruning that we used is somewhat standard for human analysis, but be sure to check the parameters, especially if you are using non-human data.

The HapMap file that we downloaded is based on an old version of the reference genome (build 36). As stated in the previous chapter, Chapter 3, Working with Genomes, be sure to use annotations from build 36 if you plan to use this file for more analysis of your own.

This recipe will set the stage for the following recipes, and its results will be used extensively.

## See also

- The Wikipedia page [http://en.wikipedia.org/wiki/Linkage\\_disequilibrium](http://en.wikipedia.org/wiki/Linkage_disequilibrium) on LD is a good place to start
- The website of PLINK <https://www.cog-genomics.org/plink/2.0/> is very well documented, something that many genetics software lacks

## Introducing the Genepop format

The Genepop format is used in many conservation genetics studies. It's the format of the Genepop application and is the de facto format for much population genetics analysis. If you come from other fields (for example, those that have a lot of sequencing experience), you may not have heard of it, but this format is widely used (as its citation record proves) and is worth a look. Here, we will convert some datasets from previous recipes to this format and introduce the Genepop parser in Biopython.

## Getting ready

You will need to run the previous recipe because its output is required for this one. I have a small library to help with basic data conversion and charting. You can find this code at <https://github.com/tiagoantao/pygenomics>, and you can install it with pip:

```
pip install pygenomics
```

Note that at this stage, we will not use the Genepop application (this will change in the next recipe), so no need to install it for now.

As usual, this is available in the Chapter04/Genepop\_Format.ipynb Notebook file, but it will still require you to run the previous Notebook file in order to generate the required files.

## How to do it...

Take a look at the following steps:

Let's load the metadata (we will use a simplified version from the previous recipe) as follows:

```
from collections import defaultdict
f = open('relationships_w_pops_121708.txt')
pop_ind = defaultdict(list)
f.readline() # header
for line in f:
    toks = line.rstrip().split('\t')
    fam_id = toks[0]
    ind_id = toks[1]
    pop = toks[-1]
    pop_ind[pop].append((fam_id, ind_id))
f.close()
```

Let's check for consistency between the PLINK data file and the metadata, as we will need to clean up population mappings to generate a Genepop file, as shown in the following code:

```
all_inds = []
for inds in pop_ind.values():
    all_inds.extend(inds)
for line in open('hapmap1.ped'):
    toks = line.rstrip().replace(' ', '\t').split('\t')
    fam = toks[0]
    ind = toks[1]
```

```
if (fam, ind) not in all_inds:  
    print('Problems with %s/%s' % (fam, ind))
```

The preceding code generates a list with all of the individuals coming from the metadata file. Then, it will open hapmap1.ped, which has the pedigree information at 1 percent sampling (I have chosen 1 percent because 1 will be much faster to process than 10 or 100 percent samples; we only need pedigree, not genetic information) and compare the information on both. It will report all individuals that are on the PED file, but not on the metadata file.

We perform a replacement procedure on each PED line because you can find some PLINK files that are space-separated, whereas others are tab-separated.

In a perfect world, this will output nothing, but there is one incorrect entry. This entry (which has a family ID of 2469 and an individual ID of NA20281) is not consistent with the family ID reported on the metadata.

For your own dataset, always be sure to thoroughly compare your data with your metadata and check for consistency problems.

With all your sources of data (if you have more than one), make sure that they are consistent among themselves. If not, at least annotate all problematic cases. Better yet, take action to understand and correct any underlying problems. The default assumption should be that there are problems (not that everything is sound). Although you have produced the data yourself, check it. Bugs and typos are assured to happen. Making errors is normal, and checking for them is fundamental. Being overconfident is a sign of inexperience.



Let's convert some datasets from PLINK to the Genepop format:

```
from genomics.popgen.plink.convert import to_genepop  
to_genepop('hapmap1_auto', 'hapmap1_auto', pop_ind)  
to_genepop('hapmap10', 'hapmap10', pop_ind)  
to_genepop('hapmap10_auto', 'hapmap10_auto', pop_ind)  
to_genepop('hapmap10_auto_noofs_ld', 'hapmap10_auto_noofs_ld',  
pop_ind)  
to_genepop('hapmap10_auto_noofs_2', 'hapmap10_auto_noofs_2',  
pop_ind)
```

We will maintain the prefix of all files, hence the equal string on the first and second parameters. This will be prepended with .ped and .map to find input files. The second parameter will be prepended with .gp to generate the Genepop file, whereas .pops will contain the order of populations on the Genepop file. Take a look at both generated files so that you're familiar with the content, although we will dissect the result a bit more in the next recipe.

As PLINK has no population structure information, we need to pass the pop\_ind dictionary. This dictionary will be used to create a Genepop file that's structured by population.

This uses a function provided by my package to convert PLINK to Genepop data. This will take some time to run. Note that we are just converting subsampled data as this is done to make things computationally more efficient in downstream analysis, but be aware that in many of your own analyses, you may need the complete dataset. The function will ignore individuals without population, which means that it will exclude the individual with the wrong family ID detected in the consistency step. A will be converted to 1, C to 2, T to 3, and G to 4. Although a .pops file will be produced with the order of populations on the output file, this will always be lexicographically ordered.

If you are curious about how this function works, feel free to take a look at <https://gitlab.com/tiagoantao/pygenomics/blob/master/genomics/popgen/plink/convert.py>. Be forewarned that it contains text processing.

Biopython<sup>41</sup> provides an in-memory parser for Genepop files; let's get a small taste of it by opening the autosome file sampled at 1 percent:

```
from Bio.PopGen.GenePop import read
rec = read(open('hapmap1_auto.gp'))
print('Number of loci %d' % len(rec.loci_list))
print('Number of populations %d' % len(rec.pop_list))
print('Population names: %s' % ', '.join(rec.pop_list))
print('Individuals per population %s' % ', '.join([str(len(inds))
    for inds in rec.populations]))
ind = rec.populations[1][0]
print('Individual %s, SNP %s, alleles: %d %d' % (ind[0],
    rec.loci_list[0], ind[1][0][0], ind[1][0][1]))
del rec
```

The output is as follows:

```
Header: 8
Number of loci 13937
Number of populations 11
```

```
Population names: 2436/NA19983, 1459/NA12865, NA18594/NA18594,  
NA18140/NA18140, NA20881/NA20881, NA19007/NA19007, NA19372/NA19372,  
M005/NA19652, 2581/NA21371, NA20757/NA20757, Y105/NA19099  
Individuals per population 82, 165, 84, 85, 88, 86, 90, 77, 171,  
88, 167  
Individual 1328/NA06989, SNP 1/rs12032637/1455245, alleles: 1 1
```

The default assumption about population names on Genepop is that somehow the last individual is used to identify a population. As this is slightly ad hoc, we will also generate a .pop file (as in the previous recipe) with the names of populations. As the marker sampling process in the previous recipe is stochastic, you will probably see a slightly different number of loci.

As the whole dataset is in memory, we can directly access any individual of any population. This is what we perform to print the last line; that is, we access the first individual of the second population and print its name, along with the alleles of the first SNP (which are 3 and 2, thus coding T and C). The first SNP is called 1/rs12032637/1455245. Here, 1 represents the chromosome number, the middle ID is the SNP rs ID (the identifier on NCBI's dbSNP database), and the last number is the chromosome position against the human reference genome build 36.

At the end, we delete the record because it takes up a lot of memory.



Note that some of these outputs depend on how the Genepop was coded (in the to\_genepop function) and are based on that. For example, the coding of ACTG to 1234 is arbitrary (just a convenience) or the fact that populations are lexicographically ordered, or that loci names include the rs ID and position chromosomes. If you receive your files from another source, you will have to check whatever conventions they have used (which may or may not be convenient to you). If you generate your own files, be sure to use conventions that will be useful downstream. Of course, this argument is generalizeable: you can apply it to other file formats as long as they have any form of built-in flexibility.

More realistically, we will use the large file parser for most modern datasets because it won't load the whole in-memory file, but provide an iterator instead, as follows:

```
from Bio.PopGen.GenePop.LargeFileParser import read as read_large

def count_individuals(fname):
    rec = read_large(open(fname))
    pop_sizes = []
```

```
for line in rec.data_generator():
    if line == ():
        pop_sizes.append(0)
    else:
        pop_sizes[-1] += 1
return pop_sizes
print('Individuals per population %s' % ',join([str(len(inds))
    for inds in count_individuals('hapmap1_auto.gp')]))
print(len(read_large(open('hapmap10.gp')).loci_list))
print(len(read_large(open('hapmap10_auto.gp')).loci_list))
print(len(read_large(open('hapmap10_auto_noofs_ld.gp')).loci_list))
```

The `count_individuals` function shows you how you can traverse a Genepop file using the large file parser; when you iterate over it, if you find an empty tuple, it's a marker of a new population. Anything else is an individual, composed of tuples (pair) with an individual name and a list of loci (which we will not read here). Individuals per population will return the exact same values as in the previous step.

We then print the number of loci on three different files: 10 percent sampling, 10 percent sampling with only autosomes, and 10 percent sampling of autosomes with LD-pruning. The output, which will vary due to stochasticity generating the files, reflects that the first file has more markers than the second file (as the second file is a subset of the first file, removing sex chromosomes and mitochondria) and the last file will have far fewer markers because it's a LD-pruned subset of the second file.

## See also

- There is actually a Genepop interface on the web at <http://genepop.curtin.edu.au> that you can use for manual examples (especially with small files)

## Exploring a dataset with Bio.PopGen

In this recipe, we will perform an initial exploratory analysis of one of our generated datasets.

We will analyze the 10 percent sampling of chromosome 2 without the offspring. We will look for monomorphic loci (in this case, SNPs) across populations, along with how to research minimum allele frequencies and expected heterozygosites.

## Getting ready

You will need to have run the previous two recipes and should have the `hapmap10_auto_noofs_2.gp` and `hapmap10_auto_noofs_2.pops` files downloaded. We will also use the metadata file that we downloaded in the first recipe. For this code to work, you will need to install Genepop from either <http://kimura.univ-montp2.fr/~rousset/Genepop.htm> or, if you're using Anaconda Python, by using `conda install -c bioconda genepop`. We will use the interface provided by Biopython to execute Genepop and parse its output files.

There is a Notebook file with this recipe, called `Chapter04/Exploratory_Analysis.ipynb`, but it will still require running the previous two Notebooks in this chapter in order for the required files to be generated.

## How to do it...

Take a look at the following steps:

Let's load population names and execute Genepop externally to compute genotypic frequencies, as follows:

```
from Bio.PopGen.GenePop import Controller as gpc
ctrl = gpc.GenePopController()
my_pops = [l.rstrip() for l in open('hapmap10_auto_noofs_2.pops')]
num_pops = len(my_pops)
pop_iter, loci_iter =
ctrl.calc_allele_genotype_freqs('hapmap10_auto_noofs_2.gp')
```

First, we create a controller (an object that allows you to interact with the Genepop application). Then, we load population names. Finally, we compute the genotypic information, which may take some time. Our controller will return two iterators, exposing results per population and per loci.

We will use a relatively small dataset, which makes running Genepop in a single go feasible. If you have a larger dataset, be it in the number of individuals or in the number of loci, you may need to split the file into smaller chunks and run several Genepop instances in parallel, each working with part of the data. We will discuss this in Chapter 8, Python for Big Genomics Datasets.



Let's go through all loci statistics and retrieve information for each population of fixed alleles, minimum allele frequencies, and the number of reads:

```
from collections import defaultdict
fix_pops = [0 for i in range(num_pops)]
num_reads = [defaultdict(int) for i in range(num_pops)]
num_buckets = 20
MAFs = []
for i in range(num_pops):
    MAFs.append([0] * num_buckets)
for locus_data in loci_iter:
    locus_name = locus_data[0]
    allele_list = locus_data[1]
    pop_of_loci = locus_data[2]
    for i in range(num_pops):
        locus_num_reads = pop_of_loci[i][2]
        num_reads[i][locus_num_reads] += 1
        maf = min(pop_of_loci[i][1])
        if maf == 0:
            fix_pops[i] += 1
        else:
            bucket = min([num_buckets - 1, int(maf * 2 *
                num_buckets)])
            MAFs[i][bucket] += 1
```

We initialize three data structures: one to count the number of monomorphic loci per population, another to count the number of reads per loci and per population, and finally, one to hold the minimum allele frequency per population.

The minimum allele frequency will be held in bins (values between 0 and 0.025 will go in the first bin, values between 0.025 and 0.05 will go in the second bin, and so on, until 0.475 and 0.5).

We then go through the loci iterator provided in the previous entry. We extract the locus name, list of alleles for the loci, and per population information for the locus. We extract the number of alleles read (twice the number of samples as a rule) and allele frequencies. We use the minimum to calculate the MAF and infer whether the locus is monomorphic (MAF of 0).

Let's plot the results, as follows:

```
import numpy as np
import matplotlib.pyplot as plt
fig, axs = plt.subplots(3, figsize=(16, 9), squeeze=False)
axs[0, 0].bar(range(num_pops), fix_pops)
axs[0, 0].set_xlim(0, 11)
axs[0, 0].set_xticks(0.5 + np.arange(num_pops))
axs[0, 0].set_xticklabels(my_pops)
axs[0, 0].set_title('Monomorphic positions')
axs[1, 0].bar(range(num_pops), [np.max(list(vals.keys())) for
vals in num_reads])
axs[1, 0].set_xlim(0, 11)
axs[1, 0].set_xticks(0.5 + np.arange(num_pops))
axs[1, 0].set_xticklabels(my_pops)
axs[1, 0].set_title('Maximum number of allele reads per loci')
for pop in [0, 7, 8]:
    axs[2, 0].plot(MAFs[pop], label=my_pops[pop])
axs[2, 0].legend()
axs[2, 0].set_xticks(range(num_buckets + 1))
axs[2, 0].set_xticklabels(['%.3f' % (x / (num_buckets * 2)) for x
in range(num_buckets + 1)])
axs[2, 0].set_title('MAF bundled in bins of 0.025')
```

The output is seen in the following graph and includes three subplots: one with the number of fixed (monomorphic) alleles per population, another with the maximum number of allele reads per loci (mostly, a proxy of a number of individuals processed per population), and finally, the distribution of MAF for three populations. This is the population chosen with the least number of samples (ASW and MEX) and one with the greatest number of samples (MKK).

This was done to illustrate sampling effects; ASW and MEX are bumpy. This is most probably due to there being less of the sample size to influence the distribution of MAFs (fewer values become possible), whereas MKK is smoother:

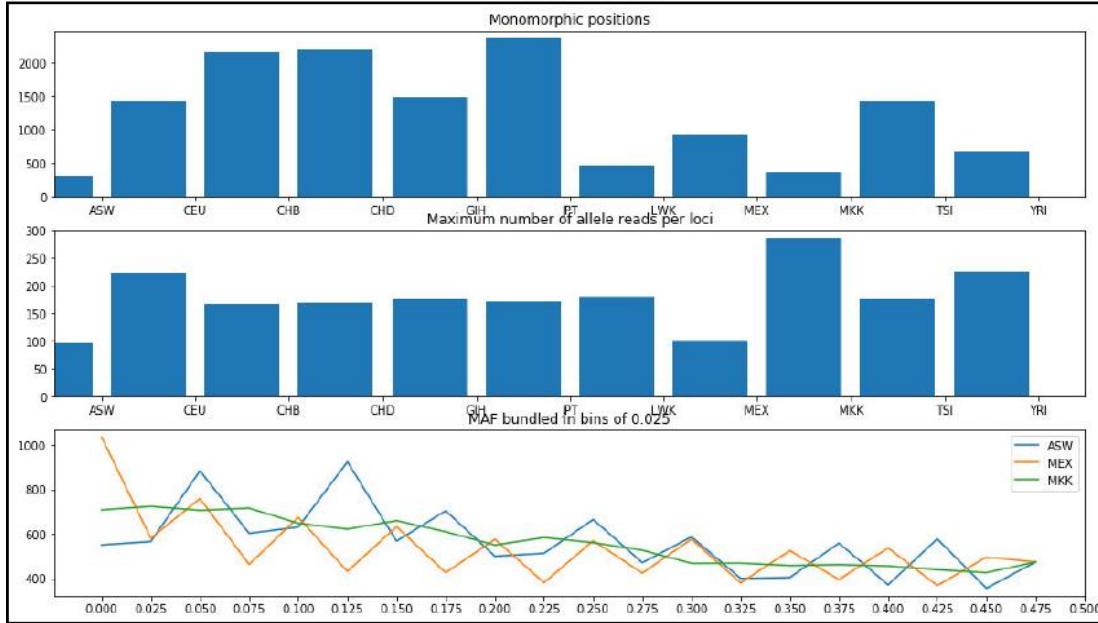


Figure 1: Three subplots: the top one includes a count of fixed SNPs per population, the second one includes the maximum allele reads per population, and the bottom one includes the distribution of MAF for three of the eleven populations

Now, let's traverse the same result, but population-wise, to compute the expected heterozygosites per population and per loci:

```
exp_hes = []
for pop_data in pop_iter:
    pop_name, allele = pop_data
    print(pop_name)
    exp_vals = []
    for locus_name, vals in allele.items():
        geno_list, heterozygosity, allele_cnts, summary = vals
        cexp_ho, cobs_ho, cexp_he, cobs_he = heterozygosity
        exp_vals.append(cexp_he / (cexp_he + cexp_ho))
    exp_hes.append(exp_vals)
```

Now, we traverse the iterator with the population information. We extract the population name (remember from the previous recipe that this is not very informative) and then go through each locus and extract the expected number of homozygotes and convert those to the expected heterozygosity.

Note that this iterator is still somewhat memory intensive; it will load all loci for a single population in memory, which is not very scalable if you have millions of SNPs.

Let's plot the distribution of expected heterozygosites per population, as follows:

```
fig = plt.figure(figsize=(16, 9))
ax = fig.add_subplot(111)
ax.boxplot(exp_hes, ax=ax)
ax.set_title('Distribution of expected Heterozygosity')
ax.set_xticks(1 + np.arange(num_pops))
ax.set_xticklabels(my_pops)
```

The output can be seen in the following screenshot:

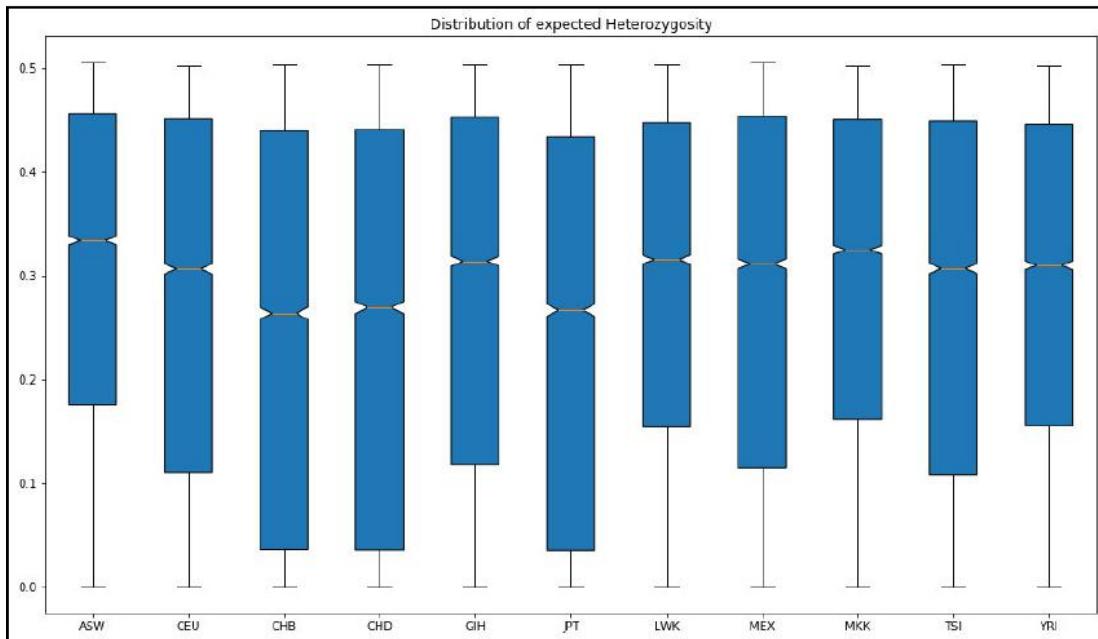


Figure 2: The distribution of expected heterozygosity across eleven populations of the HapMap project for chromosome 2, subsampled at 10 percent

## There's more...

The truth is that for population genetic analysis, nothing beats R; you are definitely encouraged to take a look at the existing R libraries for population genetics. Do not forget that there is a Python-R bridge, which was discussed in Chapter 1, Python and the Surrounding Software Ecology.

Most of the analysis presented here will be computationally costly if done on bigger datasets (remember that we are only using chromosome 2, subsampled at 10 percent). Chapter 9, Python for Big Genomics Datasets, will discuss ways to address this.

## See also

- A list of R packages for statistical genetics is available at <http://www.cran.r-project.org/web/views/Genetics.html>
- If you need to know more about population genetics, I recommend the book Principles of Population Genetics, by Daniel L. Hartl and Andrew G. Clark, Sinauer Associates

## Computing F-statistics

Nearly 100 years ago, Sewall Wright developed F-statistics to quantify inbreeding effects at a certain level of population subdivision. The most widely used of these statistics and is mostly interpreted as the genetic variation caused by the population structure.

## Getting ready

You will need to have run the first two recipes and should have the hapmap10\_auto\_noofs\_2.gp and hapmap10\_auto\_noofs\_2.pops files downloaded. We will also use the metadata file that we downloaded in the first recipe. For the type of comparison that we will perform here, it's important to assure that there is little relatedness among sampled individuals, so we want to remove the offspring at the very least. For efficiency, we will use only chromosome 2, subsampled at 10 percent.

For this code to work, you will need to install Genepop from <http://kimura.univ-montp2.fr/~rousset/Genepop.htm> or, with Anaconda Python, use conda install -c bioconda genepop. We will use the interface provided by Biopython to execute Genepop and parse its output files. These requirements are the same as for the previous recipe.

There is a Notebook file with the 03\_PopGen/F-stats.ipynb recipe in it, but it will still require you to run the first two Notebook files in this chapter in order to generate the files that are required.

## How to do it...

Take a look at the following steps:

First, let's compute F-statistics ( $F_{ST}$  and  $F_{IS}$ ) for our dataset with all 11 populations, as follows:

```
from Bio.PopGen.GenePop import Controller as gpc
my_pops = [l.rstrip() for l in open('hapmap10_auto_noofs_2.pops')]
num_pops = len(my_pops)
ctrl = gpc.GenePopController()
(multi_fis, multi fst, multi_fit), f_iter =
ctrl.calc_fst_all('hapmap10_auto_noofs_2.gp')
print(multi_fis, multi fst, multi_fit)
```

As with the previous recipe, we will first load population names and initialize a controller to interact with the Genepop application. Then, we will calculate various F-statistics.

The function will return a loci iterator that returns several F-statistics per loci as the last parameter. You may be tempted to compute the average F-statistic by looping through the iterator; while this is interesting, the multilocus F-statistics are not trivial to compute. For example, you will want to give more weightage to a loci with a larger MAF. Genepop provides multilocus  $F_{ST}$  and  $F_{IS}$  as its first three parameters before the iterator.

Let's traverse the loci results and put them on arrays, as follows:

```
fst_vals = []
fis_vals = []
fit_vals = []
for f_case in f_iter:
    name, fis, fst, fit, qinter, qintra = f_case
    fst_vals.append(fst)
    fis_vals.append(fis)
    fit_vals.append(fit)
```

This code assumes that you can fit all the values in memory. If your dataset is large, you may want to consider using a slightly more sophisticated approach. Refer to Chapter 8, Python for Big Genomics Datasets, for ideas.

Let's plot the summary distributions:

```
sns.set_style("whitegrid")
fig = plt.figure(figsize=(16, 9))
ax = fig.add_subplot(1, 1, 1)
ax.hist(fst_vals, 50, color='r')
ax.set_title('FST, FIS and FIT distributions')
ax.set_xlabel('FST')
ax = fig.add_subplot(2, 3, 2)
sns.violinplot([fis_vals], ax=ax, vert=False)
ax.set_yticklabels(['FIS'])
ax.set_xlim(-.15, 0.4)
ax = fig.add_subplot(2, 3, 3)
sns.violinplot([fit_vals], ax=ax, vert=False)
ax.set_yticklabels(['FIT'])
ax.set_xlim(-.15, 0.4)
```

The results can be seen in the following diagram. These results depict the distributions of  $F_{ST}$ ,  $F_{IS}$  and  $F_{IT}$  across chromosome 2:

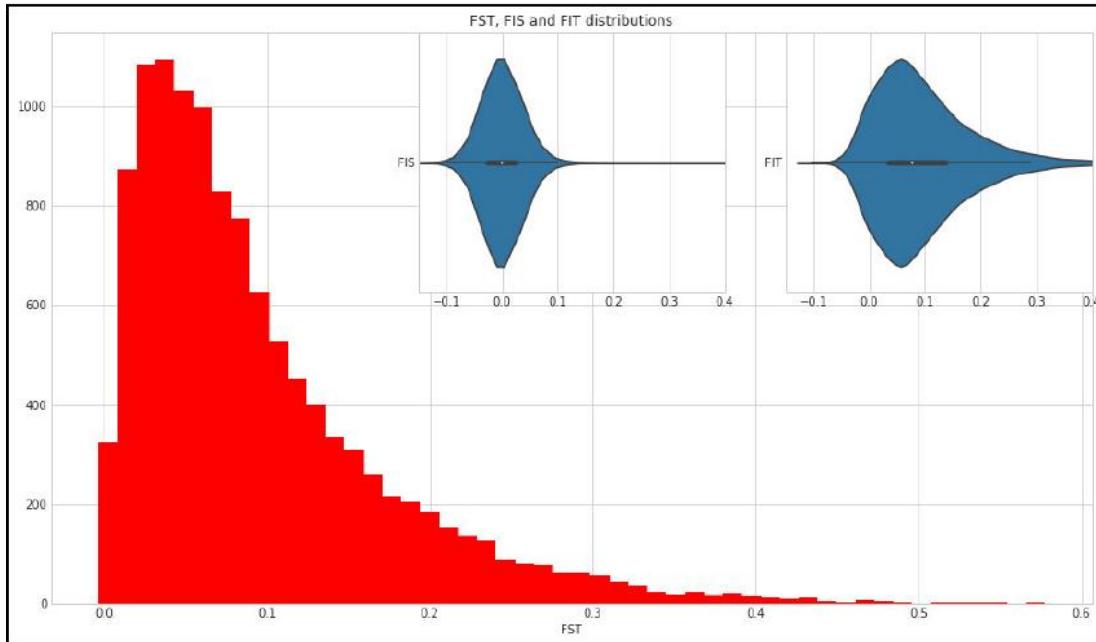


Figure 3: In the large chart, we can see a histogram. These small charts have violin plots.

Let's compute the average pair-wise  $F_{ST}$  among all populations, as shown in the following code:

```
fpair_iter, avg = ctrl.calc_fst_pair('hapmap10_auto_noofs_2.gp')
```

Remember that now, we will be comparing all pairs of populations, so we will have a number of SNPs times 55 values (the number of combinations possible with our 11 HapMap populations). You can access all of these values by using fpair\_iter. However, for now, we will concentrate on avg, which reports the multilocus pair-wise  $F_{ST}$  for all 55 combinations.

Let's plot the distance matrix across populations based on the multilocus pair-wise F<sub>ST</sub> as follows:

```

min_pair = min(avg.values())
max_pair = max(avg.values())
arr = np.ones((num_pops - 1, num_pops - 1, 3), dtype=float)
sns.set_style("white")
fig = plt.figure(figsize=(16, 9))
ax = fig.add_subplot(111)
for row in range(num_pops - 1):
    for col in range(row + 1, num_pops):
        val = avg[(col, row)]
        norm_val = (val - min_pair) / (max_pair - min_pair)
        ax.text(col - 1, row, '%.3f' % val, ha='center')
        if norm_val == 0.0:
            arr[row, col - 1, 0] = 1
            arr[row, col - 1, 1] = 1
            arr[row, col - 1, 2] = 0
        elif norm_val == 1.0:
            arr[row, col - 1, 0] = 1
            arr[row, col - 1, 1] = 0
            arr[row, col - 1, 2] = 1
        else:
            arr[row, col - 1, 0] = 1 - norm_val
            arr[row, col - 1, 1] = 1
            arr[row, col - 1, 2] = 1
ax.imshow(arr, interpolation='none')
ax.set_xticks(range(num_pops - 1))
ax.set_xticklabels(my_pops[1:])
ax.set_yticks(range(num_pops - 1))
ax.set_yticklabels(my_pops[:-1])

```

In the following diagram, we will draw an upper triangular matrix, where the background color of a cell represents the measure of differentiation; white means less different (lower F<sub>ST</sub>) and blue means more different (higher F<sub>ST</sub>). The lowest value between CHB and CHD is represented in yellow, and the biggest value between JPT and YRI is represented in magenta. The value on each cell is the average pair-wise F<sub>ST</sub> between these two populations:

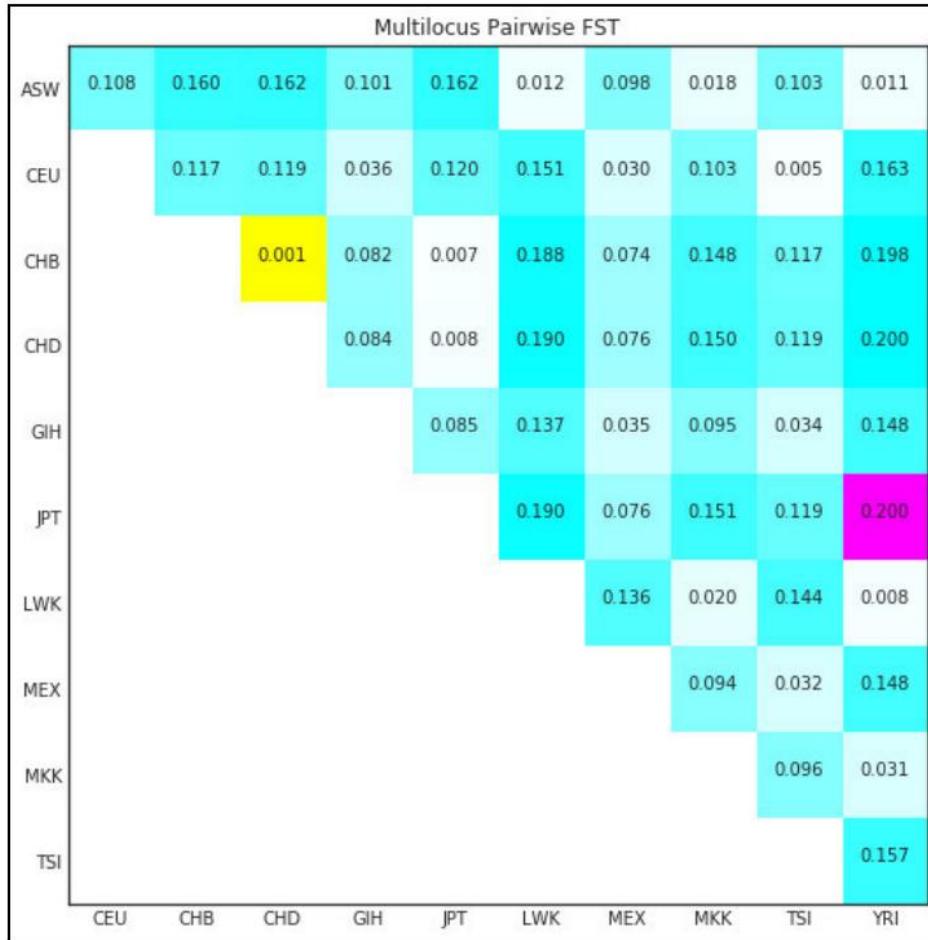


Figure 4: The average pairwise FSTs 11 populations of the HapMap project for chromosome 2

Finally, let's check the values of pair-wise comparisons between Yorubans (YRI) and Utah residents with Northwest European ancestry (CEU) around the LCT gene that resides on chromosome 2:

```
pop_ceu = my_pops.index('CEU')
pop_yri = my_pops.index('YRI')
start_pos = 136261886 # b36end_pos = 136350481
all_fsts = []
inside_fsts = []
for locus_pfst in fpair_iter:
    name = locus_pfst[0]
```

```

pfst = locus_pfst[1]
pos = int(name.split('/')[-1]) # dependent
my_fst = pfst[(pop_yri, pop_ceu)]
if my_fst == '-': # Can be this
    continue
all_fsts.append(my_fst)
if pos >= start_pos and pos <= end_pos:
    inside_fsts.append(my_fst)
print(inside_fsts)
print('%.2f/%.2f/%.2f' % (np.median(all_fsts), np.mean(all_fsts),
                           np.percentile(all_fsts, 90)))

```

The output can be seen here:

```
[0.1106, 0.2485]
0.08/0.13/0.33
```

This is done by iterating over the whole result. On one side, you get the values for the whole chromosome, whereas on the other, you get the values for the region around LCT (and MCM6, which is another gene in the neighborhood). We then print pairwise F<sub>ST</sub> region and some statistical information about the whole chromosome.

Note that you will definitely have different results in the region; your random subsampled file will surely have other markers. If you are unlucky enough, you may even not get any markers (although this is unlikely).

Note how the reported values in the LCT area are generally much higher than the median and, even in some cases, the ninetieth percentile. This is because LCT is known to be under the selection of the CEU population (giving that population the ability to digest milk into adult age). F<sub>ST</sub> is a statistic that can help you perform selection scans (to find genes that may be under selection), and genes that are under directional selection will probably have SNPs with high F<sub>ST</sub>.

## See also

- F-statistics is an immensely complex topic and I will direct you firstly to the Wikipedia page at [http://en.wikipedia.org/wiki/F\\_-statistics](http://en.wikipedia.org/wiki/F_-statistics)
- A very good explanation can be found on Holsinger, Weir paper, and Nature Reviews Genetics (Genetics in geographically structured populations: defining, estimating, and interpreting  $F_{ST}$ ) <http://www.nature.com/nrg/journal/v10/n9/abs/nrg2611.html>

# Performing Principal Components Analysis

PCA is a statistical procedure that's used to perform a reduction of the dimension of a number of variables to a smaller subset that is linearly uncorrelated. Its practical application in population genetics is assisting with the visualization of the relationships between the individuals that are being studied.

While most of the recipes in this chapter make use of Python as a glue language (Python calls external applications that actually do most of the work), with PCA, we have an option: we can either use an external application (for example, EIGENSOFT SmartPCA), or use scikit-learn and perform everything on Python. We will perform both.

## Getting ready

You will need to run the first recipe in order to make use of the `hapmap10_auto_noofs_ld_12` PLINK file (with alleles recoded as 1 and 2). PCA requires LD-pruned markers; we will not risk using the offspring here because it will probably bias the result. We will use the recoded PLINK file with alleles as 1 and 2 because this makes processing easier with SmartPCA and scikit-learn.

As with the second recipe, if you are not using Docker, you will also be using some of the code that I have produced. You can find this code at <https://github.com/tiagoantao/pygenomics>. You can install it with the following command:

```
pip install pygenomics
```

For this recipe, you will need to download EIGENSOFT (<http://www.hsph.harvard.edu/alkes-price/software/>), which includes the SmartPCA application that we will use.

There is a Notebook file in the Chapter04/PCA.ipynb recipe, but you will still need to run the first recipe.

## How to do it...

Take a look at the following steps:

Let's load the metadata, as follows:

```
f = open('relationships_w_pops_121708.txt')
ind_pop = {}
f.readline() # header
for l in f:
```

```

toks = l.rstrip().split('\t')
fam_id = toks[0]
ind_id = toks[1]
pop = toks[-1]
ind_pop['/'.join([fam_id, ind_id])] = pop
f.close()
ind_pop['2469/NA20281'] = ind_pop['2805/NA20281']

```

In this case, we will add an entry that is consistent with what is available in the PLINK file.

Let's convert the PLINK file into the EIGENSOFT format:

```

from genomics.popgen.plink.convert import to_eigen
to_eigen('hapmap10_auto_noofs_ld_12', 'hapmap10_auto_noofs_ld_12')

```

This uses a function that I have written to convert from PLINK to the EIGENSOFT format. This is mostly text manipulation—not exactly the most exciting code.

Now we will run SmartPCA and parse its results, as follows:

```

from genomics.popgen.pca import smart
ctrl = smart.SmartPCAController('hapmap10_auto_noofs_ld_12')
ctrl.run()
wei, wei_perc, ind_comp =
smart.parse_evec('hapmap10_auto_noofs_ld_12.evec',
'hapmap10_auto_noofs_ld_12.eval')

```

Again, this will use a couple of functions from pygenomics to control SmartPCA and then parse the output. The code is typical for this kind of operation, and while you are invited to inspect it, it's quite straightforward.

The parse function will return PCA weights (which we will not use, but you should inspect), normalized weights, and then principal components (usually up to PC 10) per individual.

Then we plot PC1 and PC2, as shown in the following code:

```

from genomics.popgen.pca import plot
plot.render_pca(ind_comp, 1, 2, cluster=ind_pop)

```

This will produce the following diagram. We will supply the plotting function and the population information retrieved from the metadata, which allows you to plot each population with a different color. The results are very similar to published results; we will find four groups. Most Asian populations are located on top, the African populations are located on the right-hand side, and the European populations are located at the bottom. Two more admixed populations (GIH and MEX) are located in the middle:

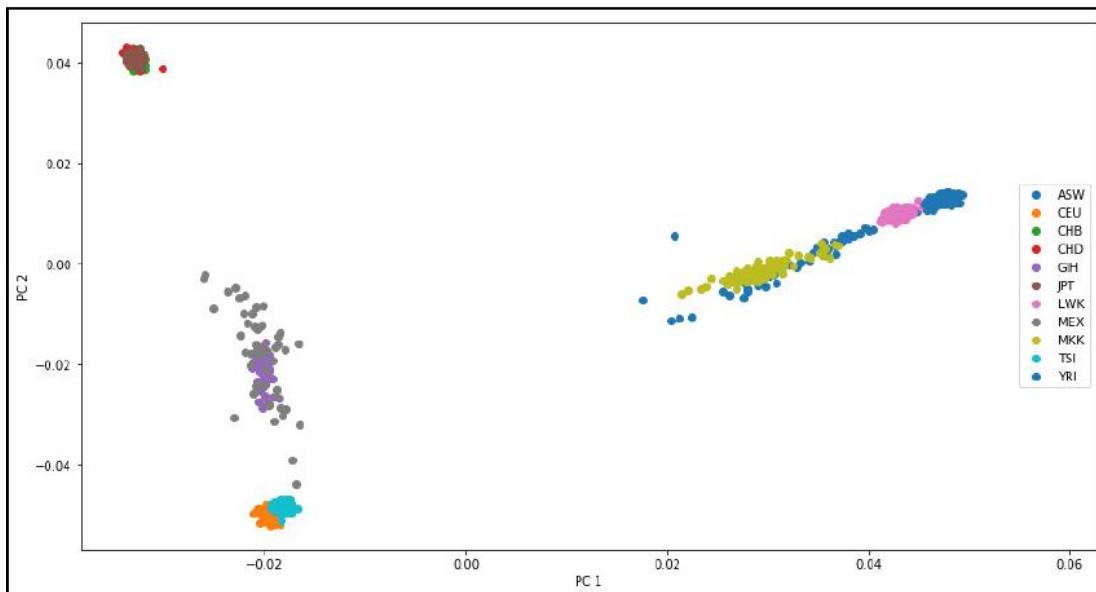


Figure 5: PC1 and PC2 of the HapMap data, as produced by SmartPCA



Note that PCA plots can be symmetrical in any axis across runs, as signal does not matter. What matters is that the clusters should be the same and that the distances between individuals (and these clusters) should be similar.

Now, let's turn to a PCA plot that's produced by Python libraries only. To be able to run scikit-learn PCA on our data, let's get the individual order of the PED file and the number of SNPs first, as follows:

```
f = open('hapmap10_auto_noofs_ld_12.ped')
ninds = 0
ind_order = []
for l in f:
    ninds += 1
```

```

toks = l[:100].replace(' ', '\t').split('\t') # for speed
fam_id = toks[0]
ind_id = toks[1]
ind_order.append('%s/%s' % (fam_id, ind_id))
nsnps = (len(line.replace(' ', '\t').split('\t')) - 6) // 2
f.close()

```

Then, we will create an array that's required for the PCA function to read in the PED file:

```

import numpy as np
pca_array = np.empty((ninds, nsnps), dtype=int)
f = open('hapmap10_auto_noofs_ld_12.ped')
for ind, l in enumerate(f):
    snps = l.replace(' ', '\t').split('\t')[6:]
    for pos in range(len(snps) // 2):
        a1 = int(snps[2 * pos])
        a2 = int(snps[2 * pos])
        my_code = a1 + a2 - 2
        pca_array[ind, pos] = my_code
f.close()

```

This code will be slow to execute.

The most important part of the code is the coding of alleles. The ability for PCA to produce meaningful results relies on good coding here. Remember that we will use a PLINK file that has a 1 and 2 allele coding. We will use the following strategy, that is, 11s are converted into 0s, 12s (and 21) are converted into 1s, and 22s are converted into 2s.

We can now call the scikit-learn PCA function, which requests 8 components:

```

from sklearn.decomposition import PCA
my_pca = PCA(n_components=8)
my_pca.fit(pca_array)
trans = my_pca.transform(pca_array)

```

Finally, let's print eight PCs, as follows:

```

sc_ind_comp = {}
for i, ind_pca in enumerate(trans):
    sc_ind_comp[ind_order[i]] = ind_pca
plot.render_pca_eight(sc_ind_comp, cluster=ind_pop)

```

We will use a different function to perform the plotting here; you will be able to see up to component 8.

The result is qualitatively similar to the SmartPCA version (it would be a worrying situation if it had been otherwise). Note that this is a vertical mirror image from the previous diagram; as discussed previously, swapping signals on PCA is not a major issue at all:

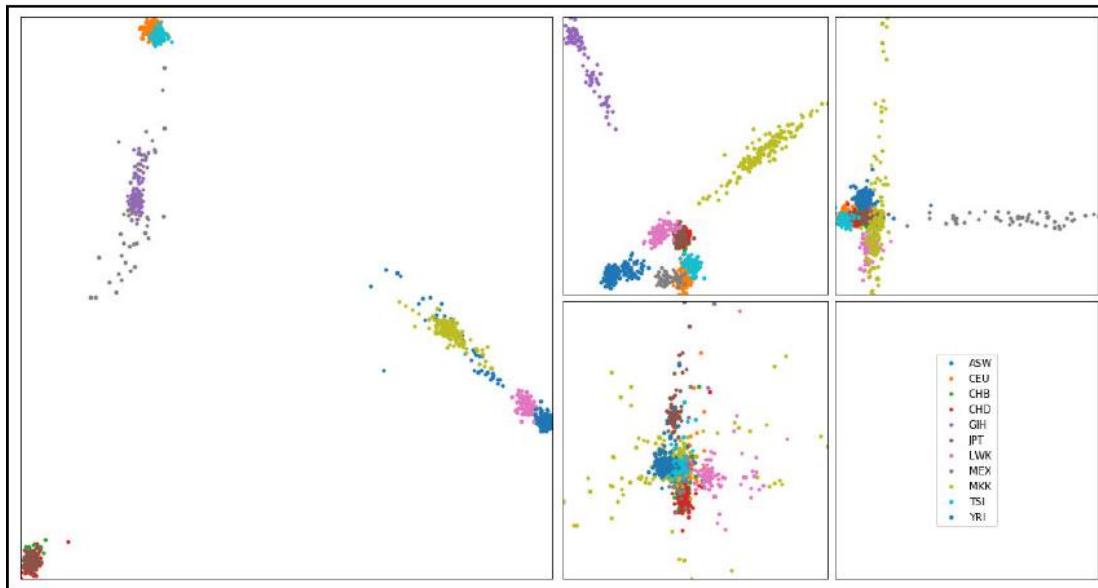


Figure 6: PC1 to PC8 of the HapMap data, as produced by scikit-learn

## There's more...

An interesting question here is, which method should you use? SmartPCA or scikit-learn? The results are similar, so if you are performing your own analysis, you are free to choose. However, if you publish your results in a scientific journal, SmartPCA is probably a safer choice because it's based on the published piece of software in the field of genetics; reviewers will probably prefer this.

## See also

- The paper that probably popularized the use of PCA in genetics was Novembre et al., Genes mirror geography within Europe on nature, where a PCA of Europeans maps almost perfectly to the map of Europe. This can be found at <http://www.nature.com/nature/journal/v456/n7218/abs/nature07331.html>. Note that there is nothing in PCA that assures it will map to geographical features (just check our PCA earlier).
- The SmartPCA is described in Patterson et al., Population Structure and Eigenanalysis, PLoS Genetics, at <http://journals.plos.org/plosgenetics/article?id=10.1371/journal.pgen.0020190>.
- A discussion of the meaning of PCA can be found in McVean's paper on A Genealogical Interpretation of Principal Components Analysis, PLoS Genetics, at <http://journals.plos.org/plosgenetics/article?id=10.1371/journal.pgen.1000686>.

## Investigating population structure with admixture

A typical analysis in population genetics was the one popularized by the program structure (<https://web.stanford.edu/group/pritchardlab/structure.html>), which is used to study population structure. This type of software is used to infer how many populations exist (or how many ancestral populations generated the current population) and to identify potential migrants and admixed individuals. Structure was developed quite some time ago, when far fewer markers were genotyped (at that time, this was mostly a handful of microsatellites) and faster versions were developed, including one from the same laboratory called fastStructure (<http://rajanil.github.io/fastStructure/>). Here, we will use Python to interface with a program of the same type that was developed at UCLA, called admixture (<http://software.genetics.ucla.edu/admixture/>).

## Getting ready

You will need to run the first recipe in order to use the hapmap10\_auto\_noofs\_ld binary PLINK file. Again, we will use a 10 percent subsampling of autosomes that have been LD-pruned with no offspring.

As in the second recipe, if you are not using Docker, you might not be able to use the code that I have produced; you can find these code files at <https://github.com/tiagoantao/pygenomics>. You can install it with the following command:

```
pip install pygenomics
```

In theory, for this recipe, you will need to download admixture (<https://www.genetics.ucla.edu/software/admixture/>). However, in this case, I will provide the outputs of running admixture on the HapMap data that we will use because running admixture takes a lot of time. You can either use the results available or run admixture yourself. There is a Notebook file for this in the Chapter04/Admixture.ipynb recipe, but you will still need to run the recipe first.

## How to do it...

Take a look at the following steps:

First, let's define our k (a number of ancestral populations) range of interest, as follows:

```
k_range = range(2, 10) # 2..9
```

Let's run admixture for all our ks (alternatively, you can skip this step and use the example data provided):

```
for k in k_range:  
    os.system('admixture --cv=10 hapmap10_auto_noofs_ld.bed %d >  
    admix.%d' % (k, k))
```

This is the worst possible way of running admixture and will probably take more than 3 hours if you do it like this. This is because it will run all ks from two to nine in a sequence. There are two things that you can do to speed this up: use the multithreaded option (-j), which admixture provides, or run several applications in parallel. Here, I have to assume a worst-case scenario where you only have a single core and thread available, but you should be able to run this more efficiently by parallelizing. We will discuss this issue at length in the last chapter.



TIP

We will need the order of individuals in the PLINK file, as admixture outputs individual results in this order:

```
f = open('hapmap10_auto_noofs_ld.fam')
ind_order = []
for l in f:
    toks = l.rstrip().replace(' ', '\t').split('\t')
    fam_id = toks[0]
    ind_id = toks[1]
    ind_order.append((fam_id, ind_id))
f.close()
```

The cross-validation error gives a measure of the "best" k, as follows:

```
import matplotlib.pyplot as plt
CVs = []
for k in k_range:
    f = open('admix.%d' % k)
    for l in f:
        if l.find('CV error') > -1:
            CVs.append(float(l.rstrip().split(' ')[-1]))
            break
    f.close()
fig = plt.figure(figsize=(16, 9))
ax = fig.add_subplot(111)
ax.set_title('Cross-Validation error')
ax.set_xlabel('K')
ax.plot(k_range, CVs)
```

The following graph plots the CV between a K of 2 and 9; lower is better. It should be clear from this graph that we should run maybe some more Ks (indeed, we have 11 populations; if not more, we should at least run up to 11), but due to computation costs, we stopped at 9.

It will be a very technical debate on whether there is such a thing as the "best" K; modern scientific literature suggests that there may not be a "best" K; these results are worthy of some interpretation. I think it's important that you are aware of this before you go ahead and interpret K results:

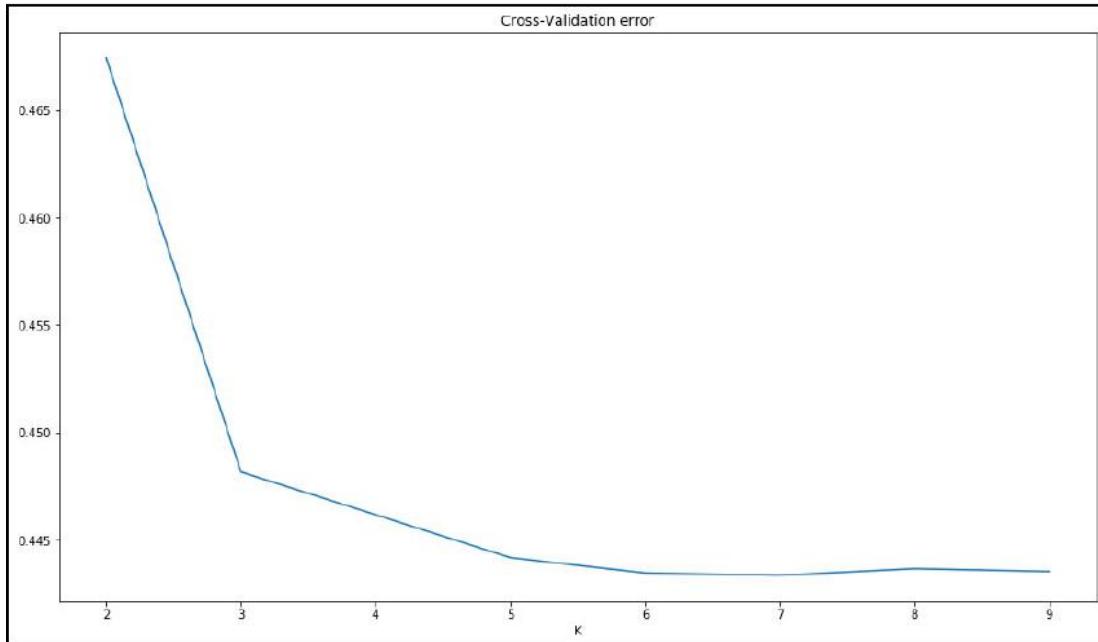


Figure 7: The error per K

We will need the metadata for the population information:

```
f = open('relationships_w_pops_121708.txt')
pop_ind = defaultdict(list)
f.readline() # header
for l in f:
    toks = l.rstrip().split('\t')
    fam_id = toks[0]
    ind_id = toks[1]
    if (fam_id, ind_id) not in ind_order:
        continue
    mom = toks[2]
    dad = toks[3]
    if mom != '0' or dad != '0':
        continue
    pop = toks[-1]
    pop_ind[pop].append((fam_id, ind_id))
f.close()
```

We will ignore individuals that are not in the PLINK file.

Let's load the individual component, as follows:

```
def load_Q(fname, ind_order):
    ind_comps = {}
    f = open(fname)
    for i, l in enumerate(f):
        comps = [float(x) for x in l.rstrip().split(' ')]
        ind_comps[ind_order[i]] = comps
    f.close()
    return ind_comps
comps = {}
for k in k_range:
    comps[k] = load_Q('hapmap10_auto_noofs_ld.%d.Q' % k, ind_order)
```

Admixture produces a file with the ancestral component per individual (for an example, look at any of the generated Q files); there will be as many components as the number of Ks that you decided to study. Here, we will load the Q file for all Ks that we studied and store them in a dictionary where the individual ID is the key.

Then, we cluster individuals, as follows:

```
from genomics.popgen.admix import cluster
ordering = {}
for k in k_range:
    ordering[k] = cluster(comps[k], pop_ind)
```

Remember that individuals were given components of ancestral populations by admixture; we would like to order them as per their similarity in terms of ancestral components (not by their order in the PLINK file). This is not a trivial exercise and requires a clustering algorithm.

Furthermore, we do not want to order all of them; we want to order them in each population and then order each population accordingly.

For this purpose, I have some clustering code available at <https://gitlab.com/tiagoantao/pygenomics/blob/master/genomics/popgen/admix/kinit.py>. This is far from perfect, but allows you to perform some plotting that still looks reasonable. My code makes use of the SciPy clustering code. I suggest you to take a look (by the way, it's not very difficult to improve on it).

With a sensible individual order, we can now plot the admixture:

```
from genomics.popgen.admix import plot
plot.single(comps[4], ordering[4])
fig = plt.figure(figsize=(16, 9))
plot.stacked(comps, ordering[7], fig)
```

This will produce two charts; the second chart is shown in the following diagram (the first chart is actually a variation of the third admixture plot from the top).

The first figure of K=4 requires components per individual and its order. It will plot all individuals, ordered and split by population.

The second chart will perform a set of stacked plots of admixture from K=2 to 9. It requires a figure object (as the dimension of this figure can vary widely with the number of stacked admixtures that you require). The individual order will typically follow one of the Ks (we have chosen a K of 7 here).

Note that all Ks are worthy of some interpretation (for example, K=2 separates the African population from others and K=3 separates the European population and shows the admixture of GIH and MEX):

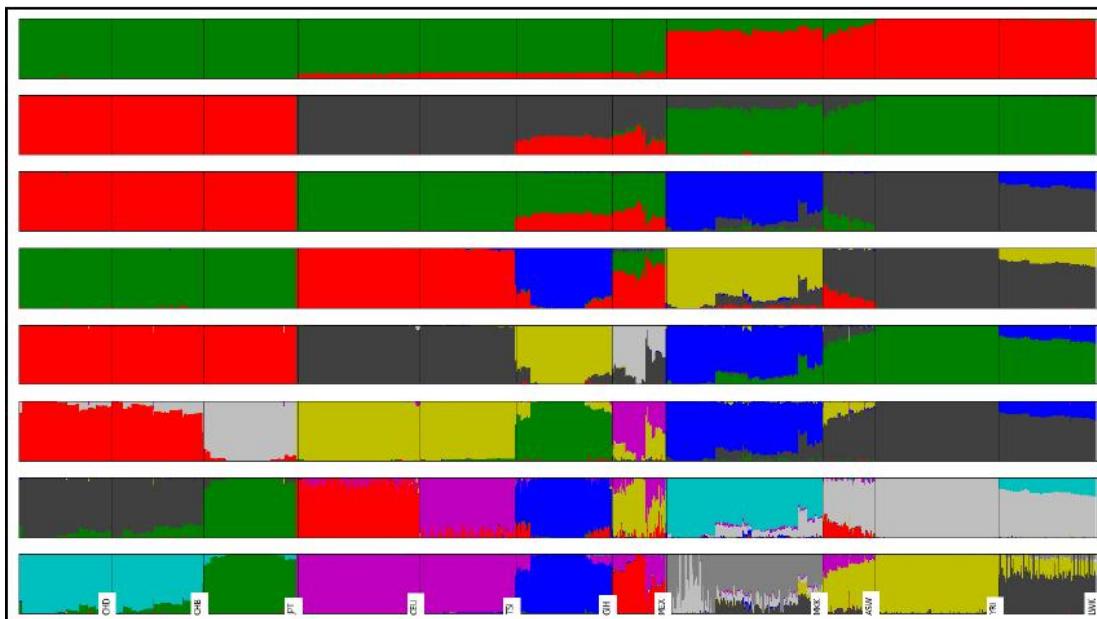


Figure 8: Stacked admixture plot (between K of 2 and 9) for the HapMap example

## There's more...

Unfortunately, you cannot run a single instance of admixture to get a result. The best practice is to actually run 100 instances and get the one with the best log likelihood (which is reported in the admixture output). Obviously, I cannot ask you to run 100 instances for each of the 7 different Ks for this recipe (we are talking about two weeks of computation), but you will probably have to perform this if you want to have publishable results. A cluster (or at least a very good machine) is required to run this. You can use Python to go through outputs and select the best log likelihood. After selecting the result with the best log likelihood for each K, you can easily apply this recipe to plot the output.

# 5

# Population Genetics Simulation

In this chapter, we will cover the following recipes:

- Introducing forward-time simulations
- Simulating selection
- Simulating population structure using island and stepping-stone models
- Modeling complex demographic scenarios

## Introduction

In Chapter 4, Population Genetics, we used Python to analyze population genetics datasets based on real data. In this chapter, we will look at how we can use Python to simulate population genetics data. From teaching, to developing new statistical methods, to analyzing the performance of existing methods, simulated datasets have plenty of applications.

There are two kinds of simulation: one is the coalescent, going backward in time, while the second is forward-time simulation. Coalescent simulation is computationally less expensive because only the most recent generation of individuals needs to be completely rendered; previous generations only need parents of the next generation to be maintained. On the other hand, this severely limits what can be simulated because we need to complete populations to make decisions on, for example, which individuals mate. Forward-time simulations are computationally more demanding and normally more complex to code, but they allow you to have much more flexibility.

In this chapter, we will use the Python-based forward-time simulator called simuPOP to model very complex scenarios and look at how we can analyze its results. Be aware that you will need to know about basic population genetics to understand this chapter.

# Introducing forward-time simulations

We will start with a simple recipe to code the bare minimum with simuPOP. simuPOP is probably the most flexible and powerful forward-time simulator available, and it's Python-based. You will be able to simulate almost anything in terms of demography and genomics, save for complex genome structural variation (for example, inversions or translocations).

## Getting ready

simuPOP programming may appear difficult, but it will make sense if you understand its event-oriented model. As you might expect, there is a meta-population composed of individuals with a predefined genomic structure. Starting with an initial population that you prepare, a set of initial operators is applied. Then, every time a generation ticks, a set of pre-operators are applied, followed by a mating step that generates the new population for the next cycle. This is followed by a final set of post-operators that are applied again. This cycle (pre-operations, mating, and post-operations) repeats for as many generations as you desire.

The most important part of getting ready is preparing yourself for this model; we will go through a simple example now. As usual, you can find this in the Chapter05/Basic\_SimuPOP.ipynb Notebook file.

## How to do it...

Take a look at the following steps:

Let's start by initializing the variables and basic data structures, as follows:

```
from collections import OrderedDict
num_loci = 10
pop_size = 100
num_gens = 10
init_ops = OrderedDict()
pre_ops = OrderedDict()
post_ops = OrderedDict()
```

Here, we specify that we want to simulate 10 loci, a population size of 100, and just 10 generations. We then prepare three ordered dictionaries. These will maintain our operators.

Note that we use `OrderedDict()`. Ordered dictionaries will return keys in the order that they were inserted. This is important because the order of operators is relevant, for example, if we print the result of a statistic—this will be dependent on it having been computed before.

We will now create a population object and some basic operators, as follows:

```
import simuPOP as sp

pops = sp.Population(pop_size, loci=[1] * num_loci)
init_ops['Sex'] = sp.InitSex()
init_ops['Freq'] = sp.InitGenotype(freq=[0.5, 0.5])
post_ops['Stat-freq'] = sp.Stat(alleleFreq=sp.ALL_AVAIL)
post_ops['Stat-freq-eval'] = sp.PyEval(r"""\%d %.2f\n' % (gen,
alleleFreq[0][0])""")
mating_scheme = sp.RandomMating()
```

We start by creating the meta population with a single deme of the required size and 10 independent loci. Then, we create two operators to initialize the population; one operator initializes the sex of all individuals (the default is two sexes with a probability of 50 percent to be assigned to either male or female). The other operator initializes all the loci with two alleles (maybe of a SNP) with a frequency probability of 50 percent for each allele.

We also create two post-mating operators: one to calculate allele frequencies for all loci and another to just print the allele frequency of loci 0 and allele 0. These will be executed in all generations. We are not using pre-operators here—just initialization, post-operators, and a mating scheme. Finally, we specify the standard random mating. This is a very basic model.

Let's run the simulator for our basic scenario with a single replicate:

```
sim = sp.Simulator(pops, rep=1)
sim.evolve(initOps=list(init_ops.values()),
preOps=list(pre_ops.values()), postOps=list(post_ops.values()),
matingScheme=mating_scheme, gen=num_gens)
```

We will create a simulator object that will be responsible for evolving our population. We will specify that we just want a single replicate. Having many replicates is a more common situation, which we will address in future recipes.



Note that we are dealing with a stochastic process, so you will get different results. This is especially important if the population size is small. Indeed, one of the most important results in population genetics is that in small populations, stochastic drift is a very strong factor. All results in this chapter are stochastic in nature, so expect to see different results from the ones presented throughout this chapter. If you want deterministic results, you can use a predetermined random seed, but do not let this trick you into thinking that these are deterministic processes in nature.

The output will be like the following:

```
0 0.54  
1 0.60  
2 0.64  
3 0.62  
4 0.62  
5 0.57  
6 0.53  
7 0.55  
8 0.52  
9 0.49
```

Let's perform a simple population genetic analysis using a new simulation model, and research the impact of population size on the loss of heterozygosity over time. We will start by developing a mini-framework to store and easily access variables of interest:

```
from copy import deepcopy  
  
def init_accumulators(pop, param):  
    accumulators = param  
    for accumulator in accumulators:  
        pop.vars()[accumulator] = []  
    return True  
  
def update_accumulator(pop, param):  
    accumulator, var = param  
    pop.vars()[accumulator].append(deepcopy(pop.vars()[var]))  
    return True
```

This code is the complex part of this recipe; it's comprised of two operators. One is to be used as an initialization operator, which will add a variable to the population (as simuPOP allows you to maintain extra variables at the population level). Another function will append the results in a pre-operator or post-operator to the variable. This may seem abstract now, but we will make this process clear soon.

We will use deepcopy to make sure that we have our own copy of the variable, because it can be changed by a future operator.

We will ~~need~~ to compute the expected heterozygosity from the allelic frequency, as follows:

```
def calc_exp_he(pop):
    #assuming bi-allelic markers coded as 0 and 1
    pop.dvars().expHe = {}
    for locus, freqs in pop.dvars().alleleFreq.items():
        f0 = freqs[0]
        pop.dvars().expHe[locus] = 1 - f0**2 - (1 - f0)**2
    return True

init_ops['accumulators'] = sp.PyOperator(init_accumulators,
param=['num_males', 'exp_he'])
post_ops['Stat-males'] = sp.Stat(numOfMales=True)
post_ops['ExpHe'] = sp.PyOperator(calc_exp_he)
post_ops['male_accumulation'] = sp.PyOperator(update_accumulator,
param=('num_males', 'numOfMales'))
post_ops['expHe_accumulation'] = sp.PyOperator(update_accumulator,
param=('exp_he', 'expHe'))
del post_ops['Stat-freq-eval']
```

Note that we are still using the operators that were specified in the previous execution to initialize sex, genotype, and so on; we will just be adding our ordered dictionaries to them.

First, we will develop a function to compute our expected heterozygosity from allele frequency for all available loci. Then, we will add two accumulators (num\_males and exp\_he) using an initialization operator.

We will then add four post-operators: one will compute the number of males, another will compute the expected heterozygosity, and the third postoperator will transfer the computation from each generation to a variable that stores the result over time. `numOfMales` is the result of a `simuPOP` operator that computes the number of males for the current generation, whereas `num_males` maintains a list of all `numOfMales` across the whole execution. `numOfMales` is recomputed and lost on each `sim.evolve` step, whereas `num_of_males` is appended.

The previous strategy cannot be used to store very large variables over the whole simulation, but it works for small variables.

Note that the expected heterozygosity operator depends on the existing operator to compute allele frequencies that already exist in the dictionary.

This has to be executed before the one computing expected heterozygosity (the ordered dictionary assures this).

We will now compare two populations with population sizes of 40 and 500:

```
num_gens = 100
pops_500 = sp.Population(500, loci=[1] * num_loci)
sim = sp.Simulator(pops_500, rep=1)
sim.evolve(initOps=list(init_ops.values()),
           preOps=list(pre_ops.values()), postOps=list(post_ops.values()),
           matingScheme=mating_scheme, gen=num_gens)
pop_500_after = deepcopy(sim.population(0))
pops_40 = sp.Population(40, loci=[1] * num_loci)
sim = sp.Simulator(pops_40, rep=1)
sim.evolve(initOps=init_ops.values(), preOps=pre_ops.values(),
           postOps=post_ops.values(), matingScheme=mating_scheme,
           gen=num_gens)
pop_40_after = deepcopy(sim.population(0))
```

Let's plot the loss of heterozygosity and the distribution of the number of males:

```
def calc_loci_stat(var, fun):
    stat = []
    for gen_data in var:
        stat.append(fun(list(gen_data.values())))
    return stat

sns.set_style('white')
fig, axs = plt.subplots(1, 2, figsize=(16, 9), sharey=True,
                      squeeze=False)

def plot_pop(ax1, pop):
```

```

for locus in range(num_loci):
    ax1.plot([x[locus] for x in pop.dvars().exp_he],
            color=(0.75, 0.75, 0.75))
    mean_exp_he = calc_loci_stat(pop.dvars().exp_he, np.mean)
    ax1.plot(mean_exp_he, color='r')

axs[0, 0].set_title('PopSize: 40')
axs[0, 1].set_title('PopSize: 500')
axs[0, 0].set_ylabel('Expected heterozygosity')
plot_pop(axs[0, 0], pop_40_after)
plot_pop(axs[0, 1], pop_500_after)
ax = fig.add_subplot(4, 4, 6)
ax.set_title('Distribution of number of males')
ax.boxplot(pop_40_after.dvars().num_males)
ax = fig.add_subplot(4, 4, 16)
ax.set_title('Distribution of number of males')
ax.boxplot(pop_500_after.dvars().num_males)
fig.tight_layout()

```

We get the following graph as the output:

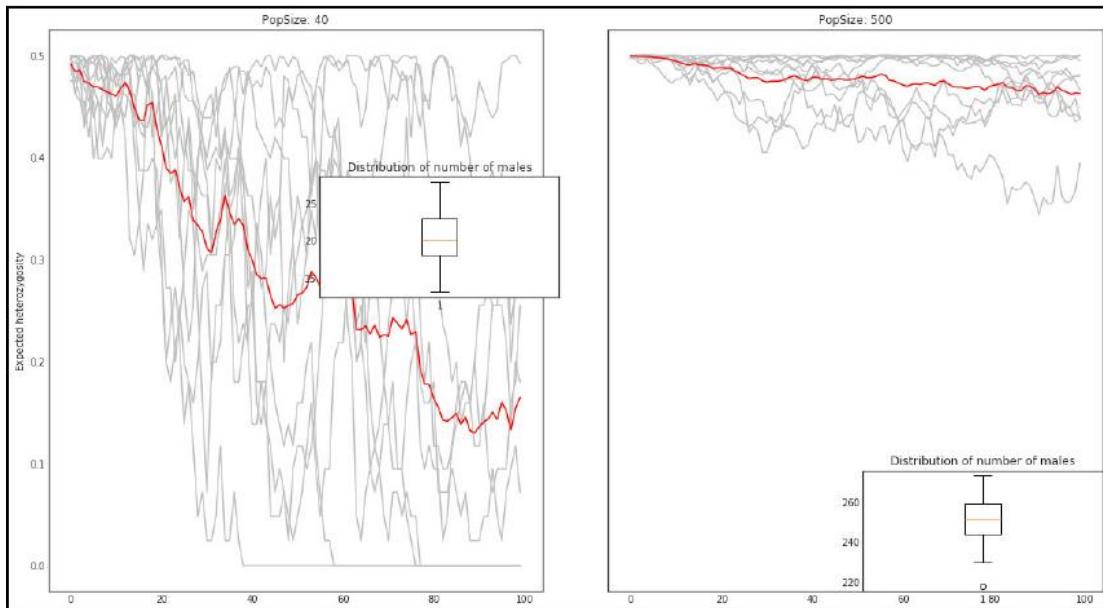


Figure 1: The decline in heterozygosity over time in a population of 40 (left) and 500 (right); the gray lines are individual markers, whereas the red lines show the mean; the box plots represent the distribution of the number of males in both scenarios

Major plots show the decrease in expected heterozygosity. They behave exactly as expected: bigger loss in the smaller population with bigger variance. Gray lines depict individual loci, whereas the red line shows the mean. Note how easy it is to extract the heterozygosity from the preceding code (this is the advantage of using the accumulator framework).

Box plots show the distribution of the number of males produced in each generation. Remember that the probability of each individual of being male is 50 percent, so the number of males will vary in each generation. If you have a very small population, there is a possibility that no males or females are generated in a single cycle. In such cases, the simulator will raise an exception and stop. Try performing a simulation with just 10 individuals, and this will eventually happen.

## There's more...

simuPOP is a very powerful simulator. Although we will address some of its features in the following recipes, it's impossible to go through all of them. If you want to simulate linked loci, non-autosomal chromosomes, complex demographies, different sex ratios and models, or mutation, simuPOP will accommodate you. Do not forget to check its website at <http://simupop.sourceforge.net/> and check out its great documentation. The user and reference manuals are fantastic. For example, in the documentation, you will find widely used demographic models, such as the cosi model of human demographics.

## Simulating selection

We will now perform an example of simulating selection with simuPOP. We will perform a simple case with dominant mutation on a single locus, along with a complex case with two loci using epistatic effects. The epistatic effect will have a required mutation on a main SNP. While this mutation is required to confer advantage, another mutation on another SNP adds up to the previous one (but does nothing on its own); this was inspired by the very real case of malarial resistance to the Sulfadoxine/Pyrimethamine drug, which always requires a mutation on the Dihydrofolate Reductase (DHFR) gene, which can be enhanced with a mutation on the Deoxyhypusine Synthase (DHPS) gene. If you are interested in finding out more, refer to the Origin and Evolution of Sulfadoxine Resistant Plasmodium falciparum article from Vinayak et al. on PLOS Pathogens at <http://journals.plos.org/plospathogens/article?id=10.1371/journal.ppat.1000830>.

## Getting ready

Read the previous recipe (Introducing forward-time simulations) as it will introduce the basic programming framework. If you are using the Jupyter Notebook files, this content is in Chapter05/Selection.ipynb.

## How to do it...

Take a look at the following steps:

Let's start with the boilerplate variable initialization:

```
from collections import OrderedDict
from copy import deepcopy
import simuPOP as sp

num_loci = 10
pop_size = 1000
num_gens = 101

init_ops = OrderedDict()
pre_ops = OrderedDict()
post_ops = OrderedDict()

def init_accumulators(pop, param):
    accumulators = param
    for accumulator in accumulators:
        pop.vars()[accumulator] = []
    return True

def update_accumulator(pop, param):
    accumulator, var = param
    pop.vars()[accumulator].append(deepcopy(pop.vars()[var]))
    return True

pops = sp.Population(pop_size, loci=[1] * num_loci,
infoFields=['fitness'])
```

Most of this code was explained in the previous recipe, but look at the very last line. There is now an infoFields parameter in the population. While you can have population variables, you can also have variables for each individual; these are limited to floating point numbers, which are specified in the infoFields parameter. We will simulate a fairly big population (1000) to avoid drift effects overpowering selection.

To ensure that we can control the number of selected alleles at initialization, let's create a function to perform it:

```
def create_derived_by_count(pop, param):
    #Assumes everything is autosomal and that derived is low (<0.5)
    locus, cnt = param
    for i, ind in enumerate(pop.individuals()):
        for marker in range(pop.totNumLoci()):
            if i < cnt and locus == marker:
                ind.setAllele(1, marker, 0)
            else:
                ind.setAllele(0, marker, 0)
                ind.setAllele(0, marker, 1)
    return True
```

We will use this to initialize our loci under selection (instead of `InitGenotype`, which we will still use for neutral markers).

Let's add all of the operators except for selection:

```
init_ops['Sex'] = sp.InitSex()
init_ops['Freq-sel'] = sp.PyOperator(create_derived_by_count,
param=(0, 10))
#careful above
init_ops['Freq-neutral'] = sp.InitGenotype(freq=[0.5, 0.5],
loci=range(1, num_loci))
post_ops['Stat-freq'] = sp.Stat(alleleFreq=sp.ALL_AVAIL)
post_ops['Stat-freq-eval'] = sp.PyEval(r'''%d %.3f\n' % (gen,
alleleFreq[0][1]), reps=[0], step=10)
mating_scheme = sp.RandomMating()
```

There are selected and neutral loci in this simulation. We will not be using the neutral loci anymore, but it's quite common to simulate a handful of selected loci among many neutral loci in order to compare their behavior.

Let's add the code for dominant selection at a single locus, store the frequency of the derived (selected) allele, and run the simulation using several replicates:

```
ms = sp.MapSelector(loci=0, fitness={
    (0, 0): 0.90,
    (0, 1): 1,
    (1, 1): 1})
pre_ops['Selection'] = ms

def get_freq_deriv(pop, param):
    marker, name = param
    expHe = {}
```

```

pop.vars()[name] = pop.dvars().alleleFreq[marker][1]
return True

init_ops['accumulators'] = sp.PyOperator(init_accumulators,
param=['freq_sel'])
post_ops['FreqSel'] = sp.PyOperator(get_freq_deriv, param=(0,
'freqDeriv'))
post_ops['freq_sel_accumulation'] =
sp.PyOperator(update_accumulator, param=('freq_sel', 'freqDeriv'))
sim = sp.Simulator(pops, rep=100)
sim.evolve(initOps=list(init_ops.values()),
preOps=list(pre_ops.values()), postOps=list(post_ops.values()),
matingScheme=mating_scheme, gen=num_gens)

```

We will create a fitness operator that will compute the fitness parameter for each individual with a dominant encoding. If you have the selected (coded with 1) mutation, you are fitter than if you are homozygous ~~0 or 1~~. Note that it's quite easy to model a recessive mutation (only benefits derived from homozygous individuals) or even a heterozygote advantage (only benefits derived from heterozygous individuals).

In this case, we will run 100 replicates (as specified on the simulator initialization). So, there will be 100 independent runs with independent results.

For each run, we will store the frequency of the derived (selected) allele. This is the purpose of `get_freq_deriv` and its related operators.

We can now plot the change in frequency of derived alleles in all 100 independent replicates, as follows:

```

sns.set_style('white')
fig = plt.figure(figsize=(16, 9))
ax = fig.add_subplot(111)
ax.set_title('Frequency of selected alleles in 100 replicates over time')
ax.set_xlabel('Generation')
ax.set_ylabel('Frequency of selected allele')
for pop in sim.populations():
    ax.plot(pop.vars()['freq_sel'])

```

The following is the output:

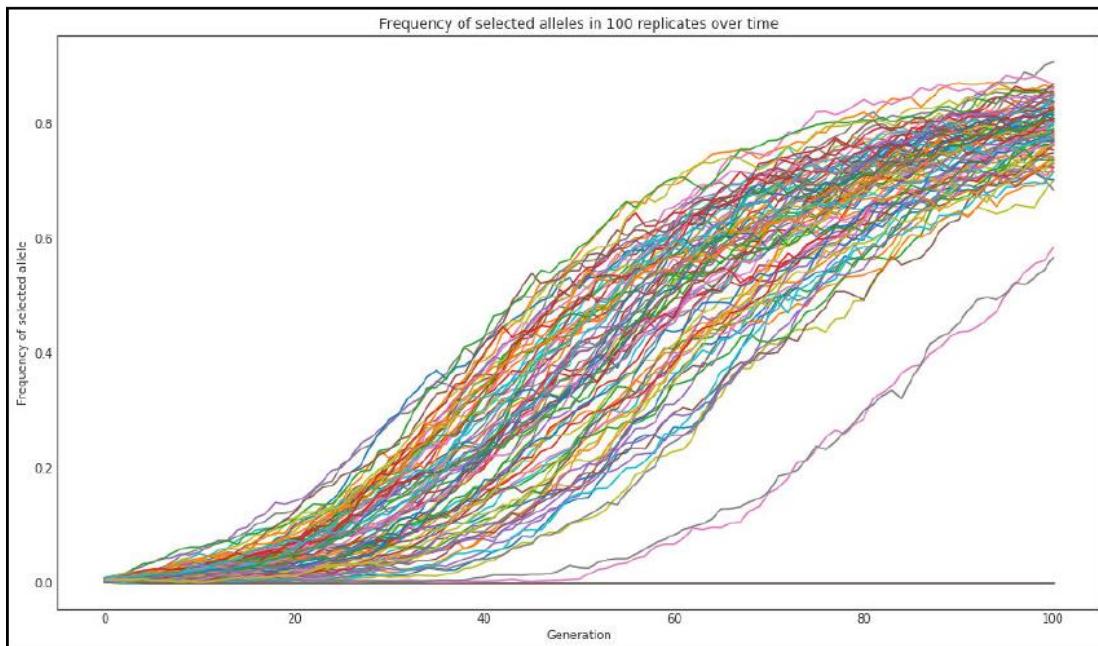


Figure 2: The increase in frequency of the selected allele over time in 100 independent replicates

Note that each line represents a trajectory in different replicates. This was performed with a large population size; if you try this with a smaller value (say 50), you will get quite a different pattern due to drift.

**N6w**, let's look at a complex example involving epistasis between two loci under selection. Here, we will perform 15 replicates, as shown in the following code:

```
pop_size = 5000
num_gens = 100
pops = sp.Population(pop_size, loci=[1] * num_loci,
infoFields=['fitness'])

def example_epistasis(geno):
    if geno[0] + geno[1] == 0:
        return 0.7
    elif geno[2] + geno[3] == 0:
        return 0.8
    else:
```

```

        return 0.9 + 0.1 * (geno[2] + geno[3] - 1)

init_ops = OrderedDict()
pre_ops = OrderedDict()
post_ops = OrderedDict()
init_ops['Sex'] = sp.InitSex()
init_ops['Freq-sel'] = sp.InitGenotype(freq=[0.99, 0.01], loci=[0,
1])
init_ops['Freq-neutral'] = sp.InitGenotype(freq=[0.5, 0.5],
loci=range(2, num_loci))
pre_ops['Selection'] = sp.PySelector(loci=[0, 1],
func=example_epistasis)
init_ops['accumulators'] = sp.PyOperator(init_accumulators,
param=['freq_sel_major', 'freq_sel_minor'])
post_ops['Stat-freq'] = sp.Stat(alleleFreq=sp.ALL_AVAIL)
post_ops['FreqSelMajor'] = sp.PyOperator(get_freq_deriv, param=(0,
'FreqSelMajor'))
post_ops['FreqSelMinor'] = sp.PyOperator(get_freq_deriv, param=(1,
'FreqSelMinor'))
post_ops['freq_sel_major_accumulation'] =
sp.PyOperator(update_accumulator, param=('freq_sel_major',
'FreqSelMajor'))
post_ops['freq_sel_minor_accumulation'] =
sp.PyOperator(update_accumulator, param=('freq_sel_minor',
'FreqSelMinor'))
sim = sp.Simulator(pops, rep=15)
sim.evolve(initOps=list(init_ops.values()),
preOps=list(pre_ops.values()), postOps=list(post_ops.values()),
matingScheme=mating_scheme, gen=num_gens)

```

The example\_epistasis function will take the first two loci to compute the fitness of the individual: 0.7 if it does not have the derived allele at the main loci; 0.8 if it has the derived allele at the main loci, but no derived allele at the secondary loci; 0.9 if it has one derived secondary loci (adding to the main loci); and 1 if it's homozygous for the derived allele in the secondary loci (again adding to the main one). So, having a mutation just on the secondary is irrelevant (it will stay at 0.7). The main loci is dominant, but the secondary loci is coded as additive (it's better to have two alleles that are derived, than just one).

We initialize the selected loci separately with a derived frequency of 1 percent, whereas the neutral loci starts at 50 percent. We also track both our selected alleles using the usual framework.

Let's plot the frequencies of both selected alleles (the main and the secondary loci):

```
fig = plt.figure(figsize=(16, 9))
ax1 = fig.add_subplot(111)
ax1.set_xlabel('Generation')
ax1.set_ylabel('Frequency of selected allele')
ax1.set_title('Frequency of selected alleles (principal and supporting) over time in 15 replicates')
for pop in sim.populations():
    ax1.plot(pop.vars()['freq_sel_major'])
    ax1.plot(pop.vars()['freq_sel_minor'], '--')
```

The output is shown in the following graph:

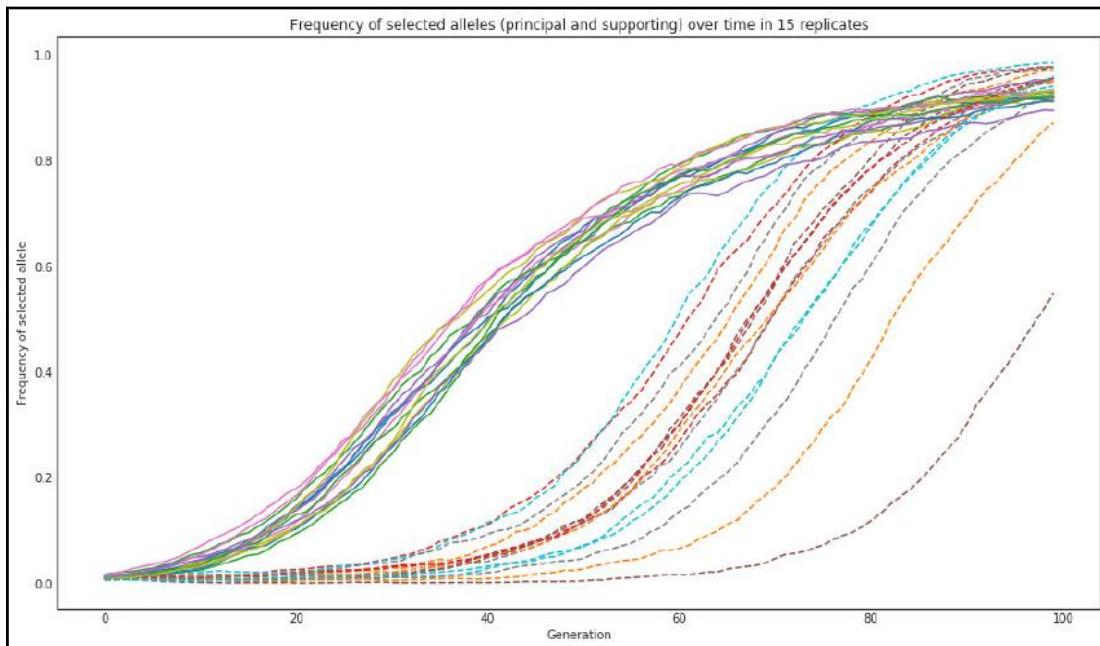


Figure 3: Frequency of selected alleles (principal and supporting loci) over time in 15 replicates; the main allele is drawn with a straight line and the secondary allele is drawn with a dashed line

In the preceding graph, you can see the dynamic of both alleles: the main allele (shown as a straight line) and the secondary allele (shown as a dashed line).

## There's more...

If you run multiple replicates, simuPOP allows you to take full advantage of a multicore computer because it can be configured to run multithreaded (check the documentation). In this case, the more you depend on simuPOP native operators, the better. Python-coded operators will be single-threaded because of Python Global Interpreter Lock (GIL). If you want to know more about the GIL refer to <http://www.dabeaz.com/python/UnderstandingGIL.pdf>.

While performing multiple replicates of complex models, I prefer to use a different strategy: I perform a single replicate per process, but run multiple processes. This has the advantage of scaling on a cluster, whereas simuPOP multithreaded code can only be used on a single computer. For very complex simulations, I do not compute any statistics at all with simuPOP; I just save the results to a file (simuPOP has operators to dump data, for example, in the Genepop format) and then use external applications to compute any statistics. Again, this strategy is just for very complex simulations with many replicates. If your requirements are simpler, multithreaded simuPOP with its built-in statistical methods might be enough.

## Simulating population structure using island and stepping-stone models

We will now simulate population structure. Let's start with an island model and then create a one-dimensional stepping-stone model. We will study behavior on these models and distinguish between deme-level statistics and meta-population level statistics. Strictly speaking, we will simulate fragmentation models by splitting into islands or stepping-stones.

## Getting ready

Read the first recipe (Introducing forward-time simulations) as it introduces the basic programming framework. If you are using the Notebook files, the content is in Chapter05/Pop\_Structure.ipynb.

## How to do it...

Take a look at the following steps:

Let's start with some basic code from the first recipe:

```
from collections import defaultdict, OrderedDict
from copy import deepcopy
import simuPOP as sp
from simuPOP import demography

num_loci = 10
pop_size = 50
num_gens = 101
num_pops = 10
migs = [0, 0.005, 0.01, 0.02, 0.05, 0.1]
init_ops = OrderedDict()
pre_ops = OrderedDict()
post_ops = OrderedDict()
pops = sp.Population([pop_size] * num_pops, loci=[1] * num_loci,
infoFields=['migrate_to'])
```

We will simulate an island model with 10 islands (num\_pops). Also, we will introduce a new infoFields, migrate\_to, which is necessary to implement migration. We will try out several migration rates (includAlso, when we create a population, the population size is now a list of 10 values (we could have demes with different sizes, but we will use the same ones here).

We will include a variation of the previous functions to accumulate values, as follows:

```
def init_accumulators(pop, param):
    accumulators = param
    for accumulator in accumulators:
        if accumulator.endswith('_sp'):
            pop.vars()[accumulator] = defaultdict(list)
        else:
            pop.vars()[accumulator] = []
    return True

def update_accumulator(pop, param):
    accumulator, var = param
    if var.endswith('_sp'):
        for sp in range(pop.numSubPop()):
            pop.vars()[accumulator][sp].append(deepcopy(pop.vars(sp)[var[:-3]]))
    else:
        pass
```

---

```
pop.vars()[accumulator].append(deepcopy(pop.vars()[var]))
return True
```

simuPOP allows you to compute statistics per sub-population. For example, if you have an island model with 10 populations, you can actually compute 11 allele frequencies per locus: 10 for each deme, plus one for the meta-population (that is, all 10 demes are considered a single population). The preceding functions cater to this (as simuPOP variables for subpopulations are suffixed with \_sp).

Let's add some operators and run a simulation, as follows:

```
init_ops['accumulators'] = sp.PyOperator(init_accumulators,
param=['fst'])
init_ops['Sex'] = sp.InitSex()
init_ops['Freq'] = sp.InitGenotype(freq=[0.5, 0.5])
for i, mig in enumerate(migs):
    post_ops['mig-%d' % i] =
        sp.Migrator(demography.migrIslandRates(mig, num_pops), reps=[i])
post_ops['Stat-fst'] = sp.Stat(structure=sp.ALL_AVAIL)
post_ops['fst_accumulation'] = sp.PyOperator(update_accumulator,
param=('fst', 'F_st'))
mating_scheme = sp.RandomMating()
sim = sp.Simulator(pops, rep=len(migs))
sim.evolve(initOps=list(init_ops.values()),
preOps=list(pre_ops.values()), postOps=list(post_ops.values()),
matingScheme=mating_scheme, gen=num_gens)
```

We will compute  $F_{ST}$  over all loci here; there is nothing special about how this is done. We just add its statistical operator and support functions to accumulate its result over the generations. simuPOP supports many other statistic operators; be sure to check the manual.

There is an operator to perform the island migration (`migrIslandRates`). This requires the number of demes and the migration rate (that is, the fraction of individuals that migrate).

As you saw in the previous recipe, simuPOP allows you to execute replicates with shared parameters. However, you can also vary the parameters per replicate. This is what we do in this case, that is, we replicate 0 with a migration of 0, replicate 1 of 0.005, and so on. So, different replicates will simulate different things.

Let's plot F over time for all different migration rates, as follows:

```
import seaborn as sns
sns.set_style('white')
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(16, 9))
ax = fig.add_subplot(111)
for pop in sim.populations():
    ax.plot(pop.dvars().fst, label='mig rate %.4f' % mig)
ax.legend(loc=2)
ax.set_ylabel('F ST ')
ax.set_xlabel('Generation')
```

The output is obtained in the form of the following graph:

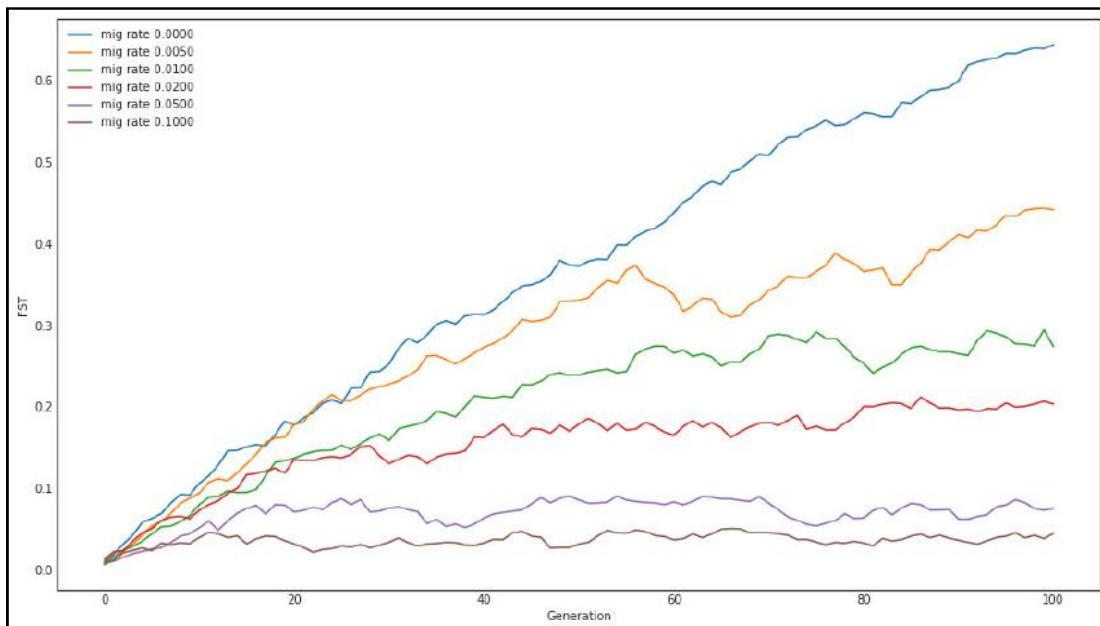


Figure 4: Evolution over time with different migration rates in an island model with 10 demes, each with 50 individuals

We take the result from each replicate and plot it; each line represents a different migration rate. As expected from the theory, FST increases faster with less migration.

Now, let's look at the stepping-stone model to study the behavior of population genetic statistics in the meta-population; on each deme, let's start with some standard code:

```

num_gens = 400
num_loci = 5
init_ops = OrderedDict()
pre_ops = OrderedDict()
post_ops = OrderedDict()
init_ops['Sex'] = sp.InitSex()
init_ops['Freq'] = sp.InitGenotype(freq=[0.5, 0.5])
post_ops['Stat-freq'] = sp.Stat(alleleFreq=sp.ALL_AVAIL,
vars=['alleleFreq', 'alleleFreq_sp'])
init_ops['accumulators'] = sp.PyOperator(init_accumulators,
param=['allele_freq', 'allele_freq_sp'])
post_ops['freq_accumulation'] = sp.PyOperator(update_accumulator,
param=('allele_freq', 'alleleFreq'))
post_ops['freq_sp_accumulation'] =
sp.PyOperator(update_accumulator, param=('allele_freq_sp',
'alleleFreq_sp'))
for i, mig in enumerate(migs):
    post_ops['mig-%d' % i] =
sp.Migrator(demography.migrSteppingStoneRates(mig, num_pops),
reps=[i])
pops = sp.Population([pop_size] * num_pops, loci=[1] * num_loci,
infoFields=['migrate_to'])
sim = sp.Simulator(pops, rep=len(migs))
sim.evolve(initOps=list(init_ops.values()),
preOps=list(pre_ops.values()), postOps=list(post_ops.values()),
matingScheme=mating_scheme, gen=num_gens)

```

There are two differences from the preceding code. One is that we are using a function to generate a migration based on the stepping-stone model, instead of the island model. Also, note that we are computing and storing allele frequencies per subpopulation (`alleleFreq_sp`) and at the meta-population level (`alleleFreq`).

Let's plot the minimum allele frequency for every locus on the meta-population level and on each deme, as follows:

```

def get_maf(var):
    locus_data = [gen[locus] for gen in var]
    maf = [min(freq.values()) for freq in locus_data]
    maf = [v if v != 1 else 0 for v in maf]
    return maf

fig, axs = plt.subplots(3, num_pops // 2 + 1, figsize=(16, 9),

```

```

sharex=True, sharey=True, squeeze=False)
fig.suptitle('Minimum allele frequency at the meta-population and 5
demes', fontsize='xx-large')
for line, pop in enumerate([sim.population(0), sim.population(1),
sim.population(len(migs) - 1)]):
    for locus in range(num_loci):
        maf = get_maf(pop.dvars().allele_freq)
        axs[line, 0].plot(maf)
        axs[line, 0].set_facecolor('black')
        for nsp in range(num_pops // 2):
            for locus in range(num_loci):
                maf =
                    get_maf(pop.dvars().allele_freq_sp[nsp *
                    2])
                axs[line, nsp + 1].plot(maf)
fig.subplots_adjust(hspace=0, wspace=0)

```

We get the following output:

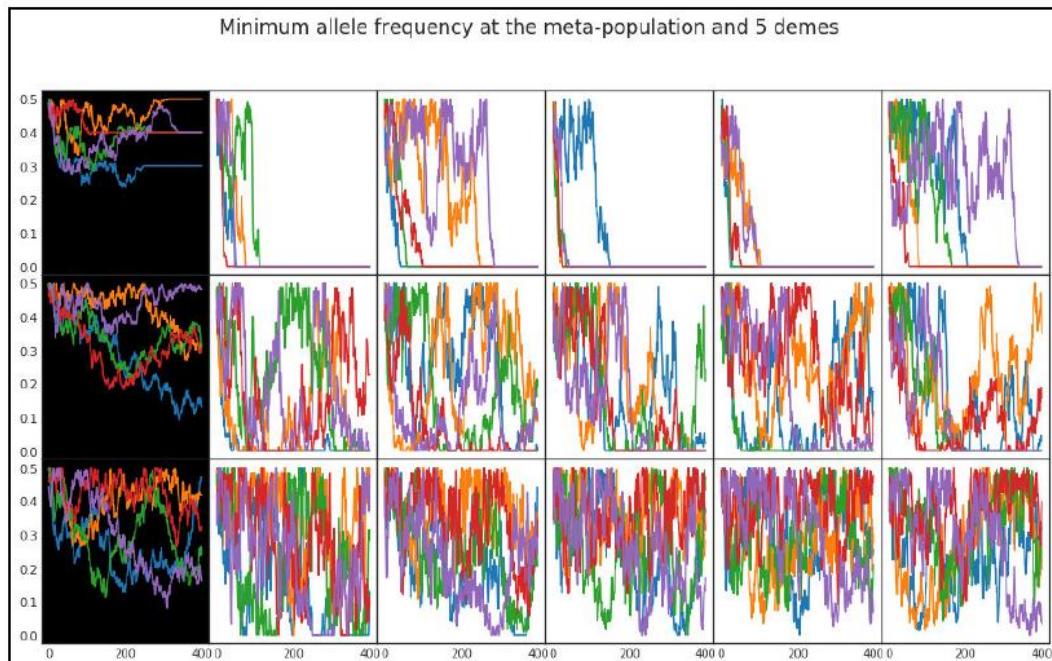


Figure 5: MAF in 5 loci. The left column (black background) shows the MAF in the meta-population. The others show the value in each individual deme. The top line is without migration, the middle line is with a migration of 0.005, and the bottom line is with a migration 0.1.

On the top line (no migration), note how different alleles can fixate on different demes, whereas at the meta level, the frequency is maintained at intermediate levels. This is not possible with migration, as alleles might jump from deme to deme.

## Modeling complex demographic scenarios

Here, we will show you how simuPOP can be extremely flexible for demographic modeling. We will simulate an age-structured population with different fecundity per age for males and different maximum litter sizes for females. In the middle of the simulation, we will remove all older females. We will study the effective population size across the simulation. Furthermore, we will simulate multiallelic loci this time (for example, for the simulation of microsatellites).

The removal of a part of the population can model many things; for example, the management of a conserved population in a national park, or the usage of insecticides in vector populations, or modeling the illegal poaching of animals. The applications of this kind of modeling are plenty.

## Getting ready

We will have three age groups. The first age group will not be able to reproduce (modeling infants). The two older age groups can. Males of age two have twice the chance of mating than males of age three. Females of age two can have many offspring in a cycle, whereas females of age three can only have one. Males of age one have 80 percent chance of surviving to age two and again 80 percent chance of surviving to age three. For females, the value is 90 percent for both.

Read the first recipe (Introducing forward-time simulations), as it introduces the basic programming framework. If you are using the Notebook files, this content is in Chapter05/Complex.ipynb.

## How to do it...

Take a look at the following steps:

Let's start by defining a function that will cull individuals according to their age and policy:

```
def kill(pop):
    kills = []
    for i in pop.individuals():
        if i.sex() == 1:
            cut = pop.dvars().survival_male[int(i.age)]
        else:
            cut = pop.dvars().survival_female[int(i.age)]
        if pop.dvars().gen > pop.dvars().cut_gen and i.age ==
2:
            cut = 0
        if random.random() > cut:
            kills.append(i.ind_id)
    pop.removeIndividuals(IDs=kills)
    return True
```

This function assumes that there are a couple of population variables (that we will create later) with the survival rate per sex. Also, there is a provision to kill all females of age 2 after a certain generation.

We need to have a function to choose the parents, since mating is far from random:

```
def choose_parents(pop):
    #name convention required
    fathers = []
    mothers = []
    for ind in pop.individuals():
        if ind.sex() == 1:
            fathers.extend([ind] *
pop.dvars().male_age_fecundity[int(ind.age)])
        else:
            ind.num_kids = 0
            mothers.append(ind)
    while True:
        father = random.choice(fathers)
        mother_ok = False
        while not mother_ok:
            mother = random.choice(mothers)
            if mother.num_kids <
pop.dvars().max_kids[int(mother.age)]:
```

```

        mother.num_kids += 1
        mother_ok = True
        yield father, mother

def calc_demo(gen, pop):
    if gen > pop.dvars().cut_gen:
        add_females = len([ind for ind in pop.individuals([0, 2])
            if ind.sex() == 2])
    else:
        add_females = 0
    return pop_size + pop.subPopSize([0, 3]) + add_females

```

The choose\_parents function will choose the father from a list that will include all males of age two and three. Males of age two get two entries (coming from a male\_age\_fecundity variable, which we will define later). Females will be allowed to have a maximum number of offspring according to their age.

There is also a function to determine the next population size. This is mostly to maintain the population at a constant level. If we cull females that are of an old age (as per the previous specification), we will use a virtual subpopulation (see step 4) that contains only old-aged individuals and get females.

The Mating function is now a bit more complex than usual, as shown in the following code:

```

mating_scheme = sp.HeteroMating([
    sp.HomoMating(
        sp.PyParentsChooser(choose_parents),
        sp.OffspringGenerator(numOffspring=1, ops=[
            sp.MendelianGenoTransmitter(), sp.IdTagger()],
            weight=1),
        sp.CloneMating(weight=-1)],
    subPopSize=calc_demo)

```

The Mating function is now much more complex than the standard random Mating function; the CloneMating part will copy all individuals to the next cycle (that is, individuals, age), whereas the HomoMating function will add a few extra individuals according to the choice of parents in the preceding function.

Let's add some necessary boilerplate code:

```

pop_size = 300
num_loci = 50
num_alleles = 10
num_gens = 90
cut_gen = 50

```

```

max_kids = [0, 0, float('inf'), 1]
male_age_fecundity = [0, 0, 2, 1]
survival_male = [1, 0.8, 0.8, 0]
survival_female = [1, 0.9, 0.9, 0]
pops = sp.Population(pop_size, loci=[1] * num_loci,
infoFields=['age', 'ind_id', 'num_kids'])
pops.setVirtualSplitter(sp.InfoSplitter(field='age', cutoff=[1, 2,
3]))

```

Note the fecundity and survival variables and the new infoFields as well. However, the most important novelty is the creation of virtual subpopulations; simuPOP allows you to split your population into virtual subgroups according to many criteria. In our case, we will have virtual subpopulations divided by age. For example, we already used this to get older females of the population in the culling stage. Check simuPOP's documentation on this concept, because it's very powerful.

Let's create all of the operators and run the simulation, as follows:

```

init_ops = OrderedDict()
pre_ops = OrderedDict()
post_ops = OrderedDict()

def init_age(pop):
    pop.dvars().male_age_fecundity = male_age_fecundity
    pop.dvars().survival_male = survival_male
    pop.dvars().survival_female = survival_female
    pop.dvars().max_kids = max_kids
    pop.dvars().cut_gen = cut_gen
    return True

def init_accumulators(pop, param):
    accumulators = param
    for accumulator in accumulators:
        pop.vars()[accumulator] = []
    return True

def update_pyramid(pop):
    pyr = defaultdict(int)
    for ind in pop.individuals():
        pyr[(int(ind.age), int(ind.sex()))] += 1
        pop.vars()['age_pyramid'].append(pyr)
    return True

def update_ldne(pop):
    pop.vars()['ldne'].append(pop.dvars().Ne_LD[0.05])
    return True

```

```

init_ops['Sex'] = sp.InitSex()
init_ops['ID'] = sp.IdTagger()
init_ops['accumulators'] = sp.PyOperator(init_accumulators,
param=['ldne', 'age_pyramid'])
init_ops['Freq'] = sp.InitGenotype(freq=[1 / num_alleles] *
num_alleles)
init_ops['Age-prepare'] = sp.PyOperator(init_age)
init_ops['Age'] = sp.InitInfo(lambda: random.randint(0,
len(survival_male) - 1), infoFields='age')
pre_ops['Kill'] = sp.PyOperator(kill)
pre_ops['Age'] = sp.InfoExec('age += 1')
pre_ops['pyramid_accumulator'] = sp.PyOperator(update_pyramid)
post_ops['Ne'] = sp.Stat(effectiveSize=sp.ALL_AVAIL, subPops=[[0,
0]], vars=['Ne_LD'])
post_ops['Ne_accumulator'] = sp.PyOperator(update_ldne)
sim = sp.Simulator(pops, rep=1)
sim.evolve(initOps=list(init_ops.values()),
preOps=list(pre_ops.values()), postOps=list(post_ops.values()),
matingScheme=mating_scheme, gen=num_gens)

```

We run the simulation for 90 generations and start killing females of age 2 at generation 50. We will consider the first 10 generations as burn-in.

We will use markers with 10 alleles (microsatellite-like). Age-pyramid functions, that is, `ae`, are used to store the evolution of age structure over time. More generally, note all code is used to manipulate age. We compute an effective population size (`Nestimator` based on linkage disequilibrium. Note that we compute this only on a virtual 0 subpopulation (that is, newborns). This is because the `Nestimation` with this method only makes sense if you use a single cohort of individuals.

We can now extract all values and plot the `Ne` estimation over time, along with the `age_pyramid` function:

```

ld_ne = sim.population(0).dvars().ldne
pyramid = sim.population(0).dvars().age_pyramid

fig = plt.figure(figsize=(16, 9))
ax_ldne = fig.add_subplot(211)
ax_ldne.plot([x[0] for x in ld_ne[10:]], 'pink')
ax_ldne.plot([x[1] for x in ld_ne[10:]], 'k--')
ax_ldne.plot([x[2] for x in ld_ne[10:]], 'k--')
ax_ldne.set_xticks(range(0, 81, 10))
ax_ldne.set_xticklabels([str(x) for x in range(10, 91, 10)])
ax_ldne.axvline(cut_gen - 10)
ax_ldne.set_xlabel('Cycle')

```

```
ax_ldne.set_ylabel('Effective population size (Est)')

def plot_pyramid(ax_bp, pyramids):
    bp_data = [[] for group in range(3 * 2)]
    for my_pyramid in pyramids:
        for (age, sex), cnt in my_pyramid.items():
            bp_data[(age - 1) * 2 + (sex - 1)].append(cnt)
    skip = 0
    for group in range(6):
        if group in [2, 4]:
            skip += 1
        bp = ax_bp.boxplot([bp_data[group]], positions = [skip +
            group + 1], notch=True, patch_artist=True)
        bp['boxes'][0].set_facecolor('yellow' if group % 2 == 1
        else 'red')
        for group in range(3):
            ax_bp.text(1 + 3 * group, 90, 'M', va='top', ha='center')
            ax_bp.text(2 + 3 * group, 90, 'F', va='top', ha='center')
        ax_bp.set_xlim(0, 9)
        ax_bp.set_ylim(20, 90)
        ax_bp.set_xticklabels(['1', '2', '3'])
        ax_bp.set_xticks([1.5, 4.5, 7.5])
        ax_bp.legend()

pre_decline = pyramid[10:50]
post_decline = pyramid[51:]
ax_bp = fig.add_subplot(2, 2, 3)
plot_pyramid(ax_bp, pre_decline)
ax_bp = fig.add_subplot(2, 2, 4)
plot_pyramid(ax_bp, post_decline)
```

The bottom charts shows the distribution of individuals per age group (1 to 3) and sex. The left chart shows the version before the cull of old age females, whereas the right chart shows the version after the cull:

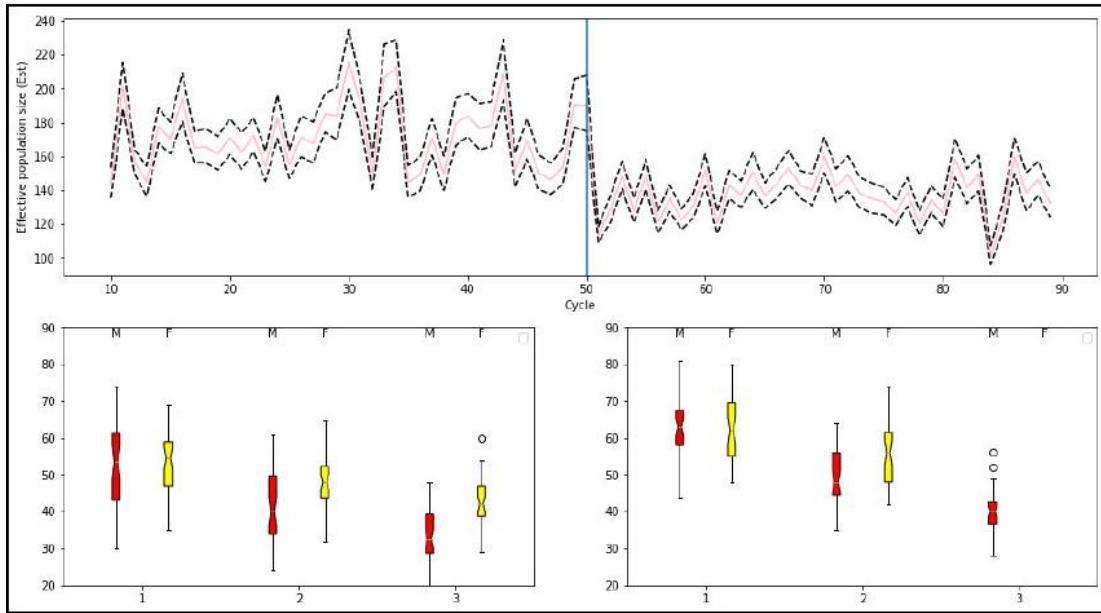


Figure 6: The top chart shows estimation (including 5 and 95 percent confidence intervals in dashed lines), number of individuals per age group (1,2 and 3 years) and sex (males in red and females in yellow)

In the preceding diagrams, the top chart depicts the change in effective population size estimation over time (the blue line shows where old-aged females start to get culled). The bottom chart shows the distribution of individuals per age group (1, 2, or 3 years) and sex (males in red and females in yellow). The left chart is before the cull and the right chart is after the cull.

# 6

# Phylogenetics

In this chapter, we will cover the following recipes:

- Preparing a dataset for phylogenetic analysis
- Aligning genetic and genomic data
- Comparing sequences
- Reconstructing phylogenetic trees
- Playing recursively with trees
- Visualizing phylogenetic data

## Introduction

Phylogenetics is the application of molecular sequencing to study the evolutionary relationship among organisms. The typical way to illustrate this process is through the use of phylogenetic trees. The computation of these trees from genomic data is an active field of research with many real-world applications.

We will take the practical approach mentioned in this book to a new level: most of the recipes here are inspired by a recent study on the Ebola virus, researching the recent Ebola outbreak in Africa. This study is called Genomic surveillance elucidates Ebola virus origin and transmission during the 2014 outbreak, by Gire et al., published on Science, and is available at <http://www.sciencemag.org/content/345/6202/1369.short>. Here, we will try to follow a similar methodology to arrive at similar results from the paper.

In this chapter, we will use DendroPy (a phylogenetics library) and Biopython.

# Preparing a dataset for phylogenetic analysis

In this recipe, we will download and prepare the dataset to be used for our analysis. The dataset contains complete genomes of the Ebola virus. We will use DendroPy to download and prepare the data.

## Getting ready

We will download complete genomes from GenBank; these genomes were collected from various Ebola outbreaks, including several from the 2014 outbreak. Note that there are several virus species that cause the Ebola virus disease; the species involved in the 2014 outbreak (the EBOV virus, formally known as the Zaire Ebola virus) is the most common, but this disease is caused by more species of the genus Ebolavirus; four others are also available in sequenced form. You can read more at <https://en.wikipedia.org/wiki/Ebolavirus>.

If you have already gone through the previous chapters, you may panic looking at the potential data sizes involved here; this is not a problem at all because these are genomes of viruses that are around 19 kbp in size each. So, our approximately 100 genomes are actually quite light.

As usual, this information is available in the corresponding Jupyter Notebook file, which is available at Chapter06/Exploration.ipynb.

## How to do it...

Take a look at the following steps:

First, let's start specifying our data sources using DendroPy, as follows:

```
import dendropy
from dendropy.interop import genbank

def get_ebov_2014_sources():
    #EBOV_2014
    #yield 'EBOV_2014', genbank.GenBankDna(id_range=(233036,
233118), prefix='KM')
    yield 'EBOV_2014', genbank.GenBankDna(id_range=(34549, 34563),
prefix='KM0')
```

```

def get_other_ebov_sources():
    #EBOV other
    yield 'EBOV_1976', genbank.GenBankDna(ids=['AF272001',
'KC242801'])
    yield 'EBOV_1995', genbank.GenBankDna(ids=['KC242796',
'KC242799'])
    yield 'EBOV_2007', genbank.GenBankDna(id_range=(84, 90),
prefix='KC2427')

def get_other_ebolavirus_sources():
    #BDBV
    yield 'BDBV', genbank.GenBankDna(id_range=(3, 6),
prefix='KC54539')
    yield 'BDBV', genbank.GenBankDna(ids=['FJ217161']) #RESTV
    yield 'RESTV', genbank.GenBankDna(ids=['AB050936', 'JX477165',
'JX477166', 'FJ621583', 'FJ621584', 'FJ621585'])
    #SUDV
    yield 'SUDV', genbank.GenBankDna(ids=['KC242783', 'AY729654',
'EU338380', 'JN638998', 'FJ968794', 'KC589025', 'JN638998'])
    #yield 'SUDV', genbank.GenBankDna(id_range=(89, 92),
prefix='KC5453')
    #TAFV
    yield 'TAFV', genbank.GenBankDna(ids=['FJ217162'])

```

We have three functions: one to retrieve data from the most recent EBOV outbreak, another from the previous EBOV outbreaks, and one from the outbreaks of other species.

Note that the DendroPy GenBank interface provides several different ways to specify lists or ranges of records to retrieve. Some lines are commented out. These include code to download more genomes. For our purpose, the subset that we will download is enough.

Now, we will create a set of FASTA files; we will use these files here and in future recipes:

```

other = open('other.fasta', 'w')
sampled = open('sample.fasta', 'w')

for species, recs in get_other_ebolavirus_sources():
    tn = dendropy.TaxonNamespace()
    char_mat = recs.generate_char_matrix(taxon_namespace=tn,
        gb_to_taxon_fn=lambda gb: tn.require_taxon(label='%s_%s' %
(species, gb.accession)))
    char_mat.write_to_stream(other, 'fasta')
    char_mat.write_to_stream(sampled, 'fasta')
other.close()

```

```

ebov_2014 = open('ebov_2014.fasta', 'w')
ebov = open('ebov.fasta', 'w')
for species, recs in get_ebov_2014_sources():
    tn = dendropy.TaxonNamespace()
    char_mat = recs.generate_char_matrix(taxon_namespace=tn,
                                         gb_to_taxon_fn=lambda gb:
                                         tn.require_taxon(label='EBOV_2014_%s' % gb.accession))
    char_mat.write_to_stream(ebov_2014, 'fasta')
    char_mat.write_to_stream(sampled, 'fasta')
    char_mat.write_to_stream(ebov, 'fasta')
ebov_2014.close()

ebov_2007 = open('ebov_2007.fasta', 'w')
for species, recs in get_other_ebov_sources():
    tn = dendropy.TaxonNamespace()
    char_mat = recs.generate_char_matrix(taxon_namespace=tn,
                                         gb_to_taxon_fn=lambda gb: tn.require_taxon(label='%s_%s' %
(species, gb.accession)))
    char_mat.write_to_stream(ebov, 'fasta')
    char_mat.write_to_stream(sampled, 'fasta')
    if species == 'EBOV_2007':
        char_mat.write_to_stream(ebov_2007, 'fasta')

ebov.close()
ebov_2007.close()
sampled.close()

```

We will generate several different FASTA files, which include either all genomes, just EBOV, or just EBOV samples from the 2014 outbreak. In this chapter, we will mostly use the sample.fasta file with all genomes.

Note the use of dendropy functions to create FASTA files that are retrieved from GenBank records through conversion. The ID of each sequence on the FASTA file is produced by a lambda function that uses species and year, apart from the GenBank accession number.

Let's extract four (of the total seven) genes in the virus, as follows:

```

my_genes = ['NP', 'L', 'VP35', 'VP40']
def dump_genes(species, recs, g_dls, p_hdls):
    for rec in recs:
        for feature in rec.feature_table:
            if feature.key == 'CDS':
                gene_name = None
                for qual in feature.qualifiers:
                    if qual.name == 'gene':
                        if qual.value in my_genes:

```

```

        gene_name = qual.value
    elif qual.name == 'translation':
        protein_translation = qual.value
    if gene_name is not None:
        locs = feature.location.split('.')
        start, end = int(locs[0]), int(locs[-1])
        g_hdls[gene_name].write('>%s_%s\n' % (species,
rec.accession))
        p_hdls[gene_name].write('>%s_%s\n' % (species,
rec.accession))
        g_hdls[gene_name].write('%s\n' %
rec.sequence_text[start - 1 : end])
        p_hdls[gene_name].write('%s\n' %
protein_translation)
    g_hdls = {}
    p_hdls = {}
for gene in my_genes:
    g_hdls[gene] = open('%s.fasta' % gene, 'w')
    p_hdls[gene] = open('%s_P.fasta' % gene, 'w')
for species, recs in get_other_ebolavirus_sources():
    if species in ['RESTV', 'SUDV']:
        dump_genes(species, recs, g_hdls, p_hdls)
for gene in my_genes:
    g_hdls[gene].close()
    p_hdls[gene].close()

```

We start by searching the first GenBank record for all gene features (refer to Chapter 2, Next Generation Sequencing, or the National Center for Biotechnology Information (NCBI) documentation for further details; although we will use DendroPy and not Biopython here, the concepts are similar) and write to FASTA files in order to extract the genes. We put each gene in a different file and only take two virus species. We also get translated proteins, which are available on the records for each gene.

Let's create a function to get the basic statistical information from the alignment, as follows:

```

def describe_seqs(seqs):
    print('Number of sequences: %d' % len(seqs.taxon_namespace))
    print('First 10 taxon sets: %s' % ', '.join([taxon.label for
taxon in seqs.taxon_namespace[:10]]))
    lens = []
    for tax, seq in seqs.items():
        lens.append(len([x for x in seq.symbols_as_list() if x !=
'']))
    print('Genome length: min %d, mean %.1f, max %d' % (min(lens),
sum(lens) / len(lens), max(lens)))

```

Our function takes a DnaCharacterMatrix DendroPy class and counts the number of taxons. We then extract all amino acids per sequence (we exclude gaps identified by -) to compute the length, and report the minimum, mean, and maximum sizes. Take a look at the DendroPy documentation for details on the API.

Let's inspect the sequence of the EBOV genome and compute the basic statistics, as shown earlier:

```
ebov_seqs = dendropy.DnaCharacterMatrix.get_from_path('ebov.fasta',
schema='fasta', data_type='dna')
print('EBOV')
describe_seqs(ebov_seqs)
del ebov_seqs
```

We then call a function and get 25 sequences with a minimum size of 18,700, a mean of 18,925.2, and a maximum of 18,959. This is a small genome when compared to eukaryotes.

Note that at the very end, the memory structure is deleted. This is because the memory footprint is still quite big (DendroPy is a pure Python library and has some costs in terms of speed and memory). Be careful with your memory usage when you load full genomes.

Now, let's inspect the other Ebola virus genome file and count the number of different species:

```
print('ebolavirus sequences')
ebolav_seqs =
dendropy.DnaCharacterMatrix.get_from_path('other.fasta',
schema='fasta', data_type='dna')
describe_seqs(ebolav_seqs)
from collections import defaultdict
species = defaultdict(int)
for taxon in ebolav_seqs.taxon_namespace:
    toks = taxon.label.split('_')
    my_species = toks[0]
    if my_species == 'EBOV':
        ident = '%s (%s)' % (my_species, toks[1])
    else:
        ident = my_species
    species[ident] += 1
for my_species, cnt in species.items():
    print("%20s: %d" % (my_species, cnt))
del ebolav_seqs
```

The name prefix of each taxon is indicative of the species, and we leverage that to fill a dictionary of counts.

The output for species and the EBOV breakdown is as follows (with the legend as Bundibugyo virus=BDBV, Tai Forest virus=TAFV, Sudan virus=SUDV, and Reston virus=RESTV: We have 1 TAFV, 6 SUDV, 6 RESTV, and 5 BDBV).

Let's extract the basic statistics of a gene on the virus:

```
gene_length = {}
my_genes = ['NP', 'L', 'VP35', 'VP40']
for name in my_genes:
    gene_name = name.split('.')[0]
    seqs =
        dendropy.DnaCharacterMatrix.get_from_path('%s.fasta' % name,
        schema='fasta', data_type='dna')
    gene_length[gene_name] = []
    for tax, seq in seqs.items():
        gene_length[gene_name].append(len([x for x in
            seq.symbols_as_list() if x != '-']))
for gene, lens in gene_length.items():
    print ('%6s: %d' % (gene, sum(lens) / len(lens)))
```

This allows you to have an overview of the basic gene information (name and mean size), as follows:

```
NP: 2218
L: 6636
VP35: 990
VP40: 988
```

## There's more...

Most of the work here can probably be performed with Biopython, but DendroPy has additional functionalities that will be explored in later recipes. Furthermore, as you will see, it's more robust with certain tasks (such as file parsing). Most importantly, there is another Python library to perform phylogenetics that you should consider. It's called ETE, and is available at <http://etetoolkit.org/>.

## See also

- The US Center for Disease Control (CDC) has a good introductory page on the Ebola virus disease at <https://www.cdc.gov/vhf/ebola/history/summaries.html>
- The reference application in phylogenetics is Joe Felsenstein's PhyloXML, which can be found at <http://evolution.genetics.washington.edu/phyloxml.html>
- We will use the Nexus and Newick formats in future recipes (<http://evolution.genetics.washington.edu/phyloxml/newicktree.html>), but also check out the PhyloXML format (<http://en.wikipedia.org/wiki/PhyloXML>)

## Aligning genetic and genomic data

Before we can perform any phylogenetic analysis, we need to align our genetic and genomic data. Here, we will use MAFFT (<http://mafft.cbrc.jp/alignment/software/>) to perform the genome analysis. The gene analysis will be performed using MUSCLE (<http://www.drive5.com/muscle/>).

## Getting ready

To perform the genomic alignment, you will need to install MAFFT, and to perform the genic alignment, MUSCLE will be used. Also, we will use trimAl (<http://trimal.cgenomics.org/>) to remove spurious sequences and poorly aligned regions in an automated manner. All packages are available from Bioconda. As usual, this information is available in the corresponding Jupyter Notebook file at Chapter06/Alignment.ipynb. You will need to have run the previous Notebook as it will generate the files that are required here. In this chapter, we will use Biopython.

## How to do it...

Take a look at the following steps:

We will now run MAFFT to align the genomes, as shown in the following code. This task is CPU-intensive and memory-intensive, and will take quite some time:

```
from Bio.Align.Applications import MafftCommandline  
mafft_cline = MafftCommandline(input='sample.fasta', ep=0.123,  
reorder=True, maxiterate=1000, localpair=True)
```

```

print(mafft_cline)
stdout, stderr = mafft_cline()
with open('align.fasta', 'w') as w:
    w.write(stdout)

```

The parameters are the same as the ones specified in the supplementary material of the paper. We will use the Biopython interface to call MAFFT.

Let's use trimAl to trim sequences, as follows:

```
os.system('trimal -automated1 -in align.fasta -out trim.fasta -
fasta')
```

Here, we just call the application using `os.system`. The `-automated1` parameter is from the supplementary material.

We can also run MUSCLE to align proteins:

```

from Bio.Align.Applications import MuscleCommandline
my_genes = ['NP', 'L', 'VP35', 'VP40']
for gene in my_genes:
    muscle_cline = MuscleCommandline(input='%s_P.fasta' % gene)
    print(muscle_cline)
    stdout, stderr = muscle_cline()
    with open('%s_P_align.fasta' % gene, 'w') as w:
        w.write(stdout)

```

We use Biopython to call an external application. Here, we will align a set of proteins.

Note that to make some analysis of molecular evolution, we have to compare aligned genes, not proteins (for example, comparing synonymous and nonsynonymous mutations). However, we just have aligned proteins. Therefore, we have to convert the alignment to the gene sequence form.

Let's align the genes by finding three nucleotides that correspond to each amino acid:

```

from Bio import SeqIO
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord
from Bio.Alphabet import generic_protein

for gene in my_genes:
    gene_seqs = {}
    unal_gene = SeqIO.parse('%s.fasta' % gene, 'fasta')
    for rec in unal_gene:

```

```
    gene_seqs[rec.id] = rec.seq
al_prot = SeqIO.parse('%s_P_align.fasta' % gene, 'fasta')
al_genes = []
for protein in al_prot:
    my_id = protein.id
    seq = ""
    pos = 0
    for c in protein.seq:
        if c == '-':
            seq += '---'
        else:
            seq += str(gene_seqs[my_id][pos:pos + 3])
            pos += 3
    al_genes.append(SeqRecord(Seq(seq), id=my_id))
SeqIO.write(al_genes, '%s_align.fasta' % gene, 'fasta')
```

The code gets the protein and the gene coding; if a gap is found in a protein, three gaps are written; if an amino acid is found, the corresponding nucleotides of the gene are written.

## Comparing sequences

Here, we will compare aligned sequences. We will perform gene and genome-wide comparisons.

## Getting ready

We will use DendroPy and will require the results from the previous two recipes. As usual, this information is available in the corresponding Notebook at Chapter06/Comparison.ipynb.

## How to do it...

Take a look at the following steps:

Let's start analyzing the gene data. For simplicity, we will only use the data from two other species of the genus Ebola virus that are available in the extended dataset: the Reston virus (RESTV) and the Sudan virus (SUDV):

```
import os
from collections import OrderedDict
import dendropy
```

```

from dendropy.calculate import popgenstat

genes_species = OrderedDict()
my_species = ['RESTV', 'SUDV']
my_genes = ['NP', 'L', 'VP35', 'VP40']

for name in my_genes:
    gene_name = name.split('.')[0]
    char_mat =
        dendropy.DnaCharacterMatrix.get_from_path('%s_align.fasta' % name,
        'fasta')
    genes_species[gene_name] = {}
    for species in my_species:
        genes_species[gene_name][species] =
            dendropy.DnaCharacterMatrix()
        for taxon, char_map in char_mat.items():
            species = taxon.label.split('_')[0]
            if species in my_species:
                genes_species[gene_name][species].taxon_namespace.add_taxon(taxon)
                genes_species[gene_name][species][taxon] = char_map

```

We get four genes that we stored in the first recipe and aligned in the second.

We load all the files (which are FASTA formatted) and create a dictionary with all the genes. Each entry will be a dictionary itself with the RESTV or SUDV species, including all reads. This is not a lot of data, just a handful of genes.

Let's print some basic information for all four genes, such as the number of segregating sites (seg\_sites), nucleotide diversity (nuc\_div), Tajima's D (taj\_d), and Waterson's theta (wat\_theta) (check the See also section of this recipe for links on these statistics):

```

import numpy as np
import pandas as pd
summary = np.ndarray(shape=(len(genes_species), 4 *
len(my_species)))
stats = ['seg_sites', 'nuc_div', 'taj_d', 'wat_theta']
for row, (gene, species_data) in enumerate(genes_species.items()):
    for col_base, species in enumerate(my_species):
        summary[row, col_base * 4] =
            popgenstat.num_segregating_sites(species_data[species])
        summary[row, col_base * 4 + 1] =
            popgenstat.nucleotide_diversity(species_data[species])
        summary[row, col_base * 4 + 2] =
            popgenstat.tajimas_d(species_data[species])
        summary[row, col_base * 4 + 3] =
            popgenstat.wattersons_theta(species_data[species])

```

```

columns = []
for species in my_species:
    columns.extend(['%s (%s)' % (stat, species) for stat in stats])
df = pd.DataFrame(summary, index=genes_species.keys(),
columns=columns)
df # vs print(df)

```

Let's look at the output first, and then we'll explain how to build it:

	seg_sites (RESTV)	nuc_div (RESTV)	taj_d (RESTV)	wat_theta (RESTV)	seg_sites (SUDV)	nuc_div (SUDV)	taj_d (SUDV)	wat_theta (SUDV)
NP	113.0	0.020659	-0.482275	49.489051	118.0	0.029630	1.203522	56.64
L	288.0	0.018143	-0.295386	126.131387	282.0	0.024193	1.412350	135.36
VP35	43.0	0.017427	-0.553739	18.832117	50.0	0.027761	1.069061	24.00
VP40	61.0	0.026155	-0.188135	26.715328	41.0	0.023517	1.269160	19.68

I used a pandas DataFrame to print the results because it's really tailored to deal with an operation like this. We will initialize our DataFrame with a NumPy multidimensional array with four rows (genes) and four statistics times the two species.

Statistics, such as the number of segregating sites, nucleotide diversity, Tajima's D, and Watterson's theta, are computed by DendroPy. Note the placement of individual data points in the array (the coordinate computation).

Look at the very last line: if you are on the IPython Notebook, just putting the df at the end will render the DataFrame and cell output as well. If you are not on a Notebook, use print(df) (you can also perform this in a Notebook, but it will not look as pretty).

Now, let's extract similar information, but genome-wide instead of only gene-wide. In this case, we will use a subsample of two EBOV outbreaks (from 2007 and 2014). We will perform a function to display basic statistics, as follows:

```

def do_basic_popgen(seqs):
    num_seg_sites = popgenstat.num_segregating_sites(seqs)
    avg_pair =
        popgenstat.average_number_of_pairwise_differences(seqs)
    nuc_div = popgenstat.nucleotide_diversity(seqs)
    print('Segregating sites: %d, Avg pairwise diffs: %.2f,
Nucleotide diversity %.6f % (num_seg_sites, avg_pair, nuc_div)')
    print("Watterson's theta: %s"
        popgenstat.wattersons_theta(seqs))
    print("Tajima's D: %s" % popgenstat.tajimas_d(seqs))

```

By now, this function should be easy to understand, given the preceding examples.

Now, let's extract a subsample of the data properly, and output the statistical information:

```
ebov_seqs = dendropy.DnaCharacterMatrix.get_from_path(
    'trim.fasta', schema='fasta', data_type='dna')
sl_2014 = []
drc_2007 = []
ebov2007_set = dendropy.DnaCharacterMatrix()
ebov2014_set = dendropy.DnaCharacterMatrix()
for taxon, char_map in ebov_seqs.items():
    print(taxon.label)
    if taxon.label.startswith('EBOV_2014') and len(sl_2014) < 8:
        sl_2014.append(char_map)
        ebov2014_set.taxon_namespace.add_taxon(taxon)
        ebov2014_set[taxon] = char_map
    elif taxon.label.startswith('EBOV_2007'):
        drc_2007.append(char_map)
        ebov2007_set.taxon_namespace.add_taxon(taxon)
        ebov2007_set[taxon] = char_map
        #ebov2007_set.extend_map({taxon: char_map})
del ebov_seqs

print('2007 outbreak:')
print('Number of individuals: %s' % len(ebov2007_set.taxon_set))
do_basic_popgen(ebov2007_set)
print('\n2014 outbreak:')
print('Number of individuals: %s' % len(ebov2014_set.taxon_set))
do_basic_popgen(ebov2014_set)
```

Here, we will construct two versions of two datasets: the 2014 outbreak and the 2007 outbreak. We generate one version as DnaCharacterMatrix, and another as a list. We will use this list version at the end of this recipe.

As the dataset for the EBOV outbreak of 2014 is large, we subsample it with just eight individuals, which is a comparable sample size to the dataset of the 2007 outbreak.

Again, we delete the ebov\_seqs data structure to conserve memory (these are genomes, not only genes).

If you perform this analysis on the complete dataset for the 2014 outbreak available on GenBank (99 samples), be prepared to wait for quite some time.

The output is shown here:

```
2007 outbreak:  
Number of individuals: 7  
Segregating sites: 25, Avg pairwise diffs: 7.71, Nucleotide  
diversity 0.000412  
Watterson's theta: 10.204081632653063  
Tajima's D: -1.383114157484101  
  
2014 outbreak:  
Number of individuals: 8  
Segregating sites: 6, Avg pairwise diffs: 2.79, Nucleotide  
diversity 0.000149  
Watterson's theta: 2.31404958677686  
Tajima's D: 0.9501208027581887
```

Finally, we perform some statistical analysis on the two subsets of 2007 and 2014, as follows:

```
pair_stats = popgenstat.PopulationPairSummaryStatistics(sl_2014,  
drc_2007)  
print('Average number of pairwise differences irrespective of  
population: %.2f '%  
pair_stats.average_number_of_pairwise_differences)  
print('Average number of pairwise differences between populations:  
%.2f' % pair_stats.average_number_of_pairwise_differences_between)  
print('Average number of pairwise differences within populations:  
%.2f' % pair_stats.average_number_of_pairwise_differences_within)  
print('Average number of net pairwise differences : %.2f '%  
pair_stats.average_number_of_pairwise_differences_net)  
print('Number of segregating sites: %d' %  
pair_stats.num_segregating_sites)  
print("Watterson's theta: %.2f" % pair_stats.wattersons_theta)  
print("Wakeley's Psi: %.3f" % pair_stats.wakeleys_psi)  
print("Tajima's D: %.2f" % pair_stats.tajimas_d)
```

Note that we will perform something slightly different here; we will ask DendroPy (`popgenstat.PopulationPairSummaryStatistics`) to directly compare two populations so that we get the following results:

```
Average number of pairwise differences irrespective of population:  
284.46  
Average number of pairwise differences between populations: 535.82  
Average number of pairwise differences within populations: 10.50
```

Average number of net pairwise differences : 525.32  
Number of segregating sites: 549  
Watterson's theta: 168.84  
Wakeley's Psi: 0.308  
Tajima's D: 3.05

The Number of segregating sites is now much bigger because we are dealing with data from two different populations that are reasonably diverged. The average number of pairwise differences among populations is quite large. As expected, this is much larger than the average number for the population, irrespective of the population information.

## There's more...

If you want to get many population genetics formulas, including the ones used here, I strongly recommend that you get the manual for the Arlequin software suite (<http://cmpg.unibe.ch/software/Arlequin35/>). If you do not use Arlequin to perform data analysis, its manual is probably the best reference to implement formulas. This free document probably has more relevant details on formula implementation than any book that I can remember.

## Reconstructing phylogenetic trees

Here, we will construct phylogenetic trees for the aligned dataset for all Ebola species. We will follow a procedure that's quite similar to the one used in the paper.

## Getting ready

This recipe requires RAxML, a program for maximum likelihood-based inference of large phylogenetic trees, which you can check out at <http://sco.h-its.org/exelixis/software.html>. Bioconda includes it, where it is named raxml. Note that the binary is called raxmlHPC.

The code here is simple, but it will take time to execute because it will call RAxML (which is computationally intensive). If you opt to use the DendroPy interface, it may also become memory-intensive. We will interact with RAxML, DendroPy, and Biopython, leaving you with a choice of which interface to use; DendroPy gives you an easy way to access results, whereas Biopython is less memory-intensive. Although there is a recipe for visualization later in this chapter, we will nonetheless plot one of our generated trees here.

As usual, this information is available in the corresponding Notebook at Chapter06/Reconstruction.ipynb. You will need the output of the previous recipe to complete this one.

## How to do it...

Take a look at the following steps:

For DendroPy, we will load the data first and then reconstruct the genus dataset, as follows:

```
import os
import shutil
import dendropy
from dendropy.interop import raxml
ebola_data =
dendropy.DnaCharacterMatrix.get_from_path('trim.fasta', 'fasta')
rx = raxml.RaxmlRunner()
ebola_tree = rx.estimate_tree(ebola_data, ['-m', 'GTRGAMMA', '-N',
'10'])
print('RAxML temporary directory %s:' % rx.working_dir_path)
del ebola_data
```

Remember that the size of the data structure for this is quite big; therefore, ensure that you have enough memory to load this.

Be prepared to wait some time. Depending on your computer, this could take more than one hour. If it takes much longer, consider restarting the process, as a RAxML bug may sometimes occur.

We will run RAxML with the GTR $\Gamma$  nucleotide substitution model, as specified in the paper. We will only perform 10 replicates to speed up the results, but you should probably do a lot more, say 100. At the end, we delete the genome data from memory as it takes up a lot of memory.

The ebola\_data variable will have the best RAxML tree, with distances included. The RaxmlRunner object will have access to other information generated by RAxML. Let's print the directory where DendroPy will execute RAxML. If you inspect this directory, you will find a lot of files. As RAxML returns the best tree, you may want to ignore all of these files, but we will discuss this a little in the alternative Biopython step.

We will save trees for future analysis; in our case, it will be visualization, as shown in the following code:

```
ebola_tree.write_to_path('my_ebola.nex', 'nexus')
```

We will write sequences to a NEXUS file because we need to store the topology information. FASTA is not enough here.

Let's visualize our genus tree, as follows:

```
import matplotlib.pyplot as plt
from Bio import Phylo
my_ebola_tree = Phylo.read('my_ebola.nex', 'nexus')
my_ebola_tree.name = 'Our Ebolavirus tree'
fig = plt.figure(figsize=(16, 18))
ax = fig.add_subplot(1, 1, 1)
Phylo.draw(my_ebola_tree, axes=ax)
```

We will defer the explanation of this code until the proper recipe further down the road, but if you look at the following diagram and compare it with the results from the paper, you will see clearly that it looks like a step in the right direction. For example, all individuals from the same species are clustered together.

You will notice that trimAl changed the names of its sequences, for example, for adding their sizes. This is easy to solve; we will deal with this in the Visualizing phylogenetic data recipe:

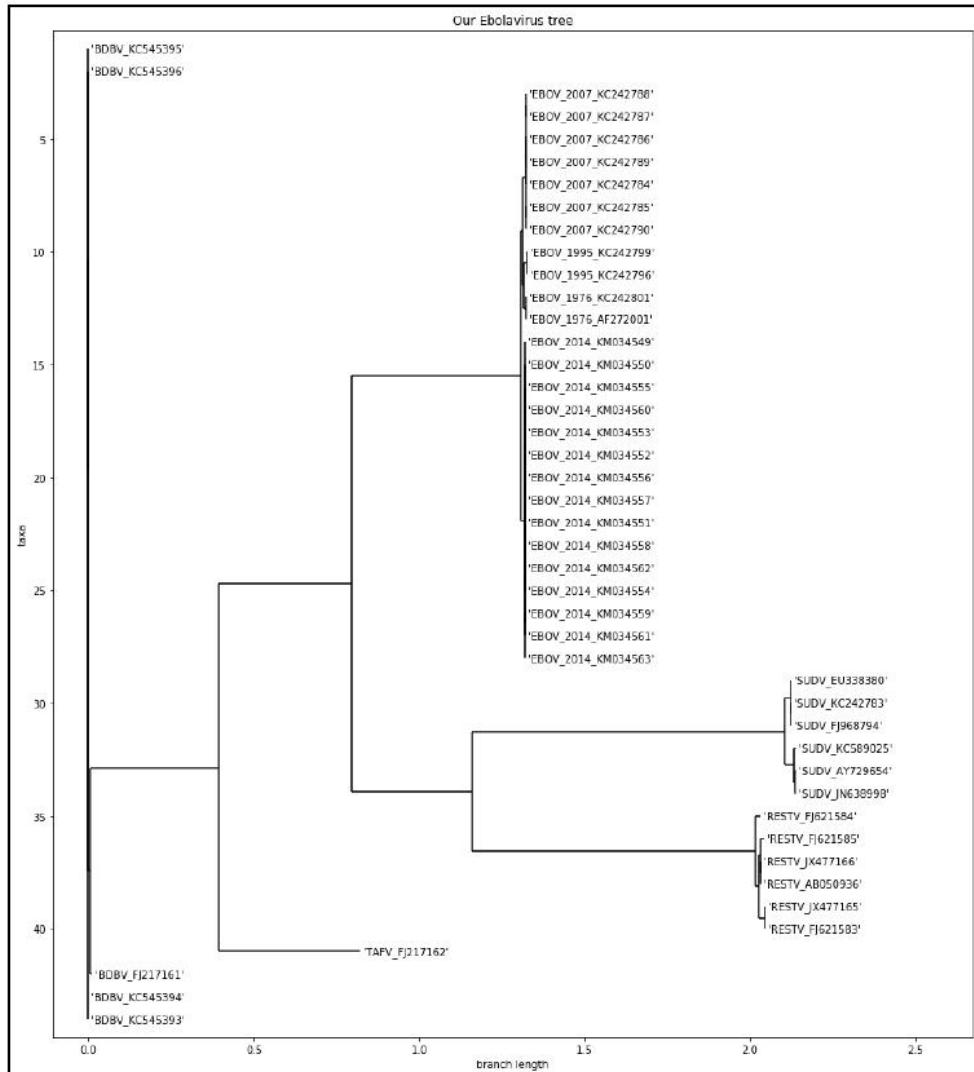


Figure 1: The phylogenetic tree that we generated with RAxML for all Ebola viruses

Let's reconstruct the phylogenetic tree with RAxML via Biopython. The Biopython interface is less declarative, but much more memory-efficient than DendroPy, so after running it, it will be your responsibility to process the output, whereas DendroPy automatically returns the best tree, as shown in the following code:

```
import random
import shutil
from Bio.Phylo.Applications import RaxmlCommandline

raxml_cline = RaxmlCommandline(sequences='trim.fasta',
model='GTRGAMMA', name='biopython', num_replicates='10',
parsimony_seed=random.randint(0, sys.maxsize),
working_dir=os.getcwd() + os.sep + 'bp_rx')
print(raxml_cline)
try:
    os.mkdir('bp_rx')
except OSError:
    shutil.rmtree('bp_rx')
    os.mkdir('bp_rx')
out, err = raxml_cline()
```

DendroPy has a more declarative interface than Biopython, so you can take care of a few extra things. You should specify the seed (Biopython will put a fixed default of 10000 if you do not do so) and the working directory. With RAxML, the working directory specification requires the absolute path.

Let's inspect the outcome of the Biopython run. While the RAxML output is the same (save for stochasticity) for DendroPy and Biopython, DendroPy abstracts away a few things. With Biopython, you need to take care of the results yourself. You can also perform this with DendroPy, but in this case, it is optional:

```
from Bio import Phylo
biopython_tree = Phylo.read('bp_rx/RAxML_bestTree.biopython',
'newick')
```

The preceding code will read the best tree from the RAxML run. The name of the file was appended with the project name that you specified in the previous step (in this case, biopython).

Take a look at the content of the bp\_rx directory; here, you will find all the outputs from RAxML, including all ten alternative trees.

## There's more...

Although the purpose of this book is not to teach phylogenetic analysis, it's important to know why we do not inspect consensus and support information on the tree topology. You should research this in your dataset. For more information, refer to <http://www.geol.umd.edu/~tholtz/G331/lectures/cladistics5.pdf>.

## Playing recursively with trees

This is not a book about programming on Python, as the topic is vast. Having said that, it's not common for introductory Python books to discuss recursive programming at length. Recursive programming techniques are usually well tailored to deal with trees. It is also a required programming strategy with functional programming dialects, which can be quite useful when you perform concurrent processing. This is common when processing very large datasets.

The phylogenetic notion of a tree is slightly different from that in computer science. Phylogenetic trees can be rooted (if so, then they are normal tree data structures) or unrooted, making them undirected acyclic graphs. Phylogenetic trees can also have weights in their edges. Thus, be mindful of this when you read the documentation; if text is written by a phylogeneticist, you can expect the tree (rooted and unrooted), while most other documents will use undirected acyclic graphs for unrooted trees. In this recipe, we will assume that all of the trees are rooted.

Finally, note that while this recipe is devised mostly to help you understand recursive algorithms and tree-like structures, the final part is actually quite practical and fundamental for the next recipe to work.

## Getting ready

You will need to have the files from the previous recipe. As usual, you can find this content in the Chapter06/Trees.ipynb Notebook file. We will use DendroPy's tree representations here. Note that most of this code is easily generalizable compared to other tree representations and libraries (phylogenetic or not).

## How to do it...

Take a look at the following steps:

First, let's load the RAxML-generated tree for all Ebola viruses, as follows:

```
import dendropy
ebola_raxml = dendropy.Tree.get_from_path('my_ebola.nex', 'nexus')
```

Then, we need to compute the level of each node (the distance to the root node):

```
def compute_level(node, level=0):
    for child in node.child_nodes():
        compute_level(child, level + 1)
    if node.taxon is not None:
        print("%s: %d %d" % (node.taxon, node.level(), level))

compute_level(ebola_raxml.seed_node)
```

DendroPy's node representation has a level method (which is used for comparison), but the point here is to introduce a recursive algorithm, so we will implement it anyway.

Note how the function works; it's called with the seed\_node (which is the root node, since the code works under the assumption that we are dealing with rooted trees). The default level for the root node ~~is~~ the function will then call itself for all its children nodes, increasing the level by one. Then, for each node that is not a leaf (as in it's internal to the tree), the calling will be repeated, and this will recurse until we get to the leaf nodes.

For the leaf nodes, we then print the level (we could have done the same for the internal nodes) and show the same information computed by DendroPy's internal function.

Now, let's compute the height of each node. The height of the node is the number of edges of the maximum downward path (going to the leaves), starting on that node, as follows:

```
def compute_height(node):
    children = node.child_nodes()
    if len(children) == 0:
        height = 0
    else:
        height = 1 + max(map(lambda x: compute_height(x), children))
    desc = node.taxon or 'Internal'
    print("%s: %d %d" % (desc, height, node.level()))
```

```

    return height

    compute_height(ebola_raxml.seed_node)

```

Here, we will use the same recursive strategy, but each node will return its height to its parent; if the node is a leaf, then the ~~height~~, then it's 1 plus the maximum of the height of its entire offspring.

Note that we use a map over a lambda function to get all the heights of all the children of the current node. We then choose the maximum (the max function performs a reduce operation here because it summarizes all the values that are reported). If you are relating this to MapReduce frameworks, you are correct; these are inspired by functional programming dialects like these.

Now, let's compute the number of offspring for each node; this should be quite easy to understand by now:

```

def compute_nofs(node):
    children = node.child_nodes()
    nofs = len(children)
    map(lambda x: compute_nofs(x), children)
    desc = node.taxon or 'Internal'
    print("%s: %d %d" % (desc, nofs, node.level()))

compute_nofs(ebola_raxml.seed_node)

```

We will now print all the leaves (this is apparently trivial):

```

def print_nodes(node):
    for child in node.child_nodes():
        print_nodes(child)
    if node.taxon is not None:
        print('%s (%d)' % (node.taxon, node.level()))

print_nodes(ebola_raxml.seed_node)

```

Note that all the functions that we have developed until now impose a very clear traversal pattern on the tree. It calls its first offspring, then that offspring will call their offspring, and so on; only after this will the function be able to call its next offspring in a depth-first pattern. However, we can do things differently.

Now, let's print the leaf nodes in a breadth-first manner, that is, we will print the leaves with the lowest level (closer to the root) first, as follows:

```
from collections import deque

def print_breadth(tree):
    queue = deque()
    queue.append(tree.seed_node)
    while len(queue) > 0:
        process_node = queue.popleft()
        if process_node.taxon is not None:
            print('%s (%d)' % (process_node.taxon,
process_node.level()))
        else:
            for child in process_node.child_nodes():
                queue.append(child)
    print_breadth(ebola_raxml)
```

Before we explain this algorithm, let's look at how different the result from this run will be compared to the previous one. For starters, take a look at the following diagram. If you print the nodes by depth-first order, you will get Y, A, X, B, and C, but if you perform a breath-first traversal, you will get X, B, C, Y, and A. Tree traversal will have an impact on how the nodes are visited; more often than not, this is important.

Regarding the preceding code here, we will use a completely different approach, as we will perform an iterative algorithm. We will use a first-in, first-out (FIFO) queue to help order our nodes. Note that Python's deque can be used as efficiently as FIFO, as well as in last-in, first-out (LIFO), because it implements an efficient data structure when you operate at both extremes.

The algorithm starts by putting the root node on the queue. While the queue is not empty, we will take the node out front. If it's an internal node, we will put all its children on the queue.

We will iterate the preceding step until the queue is empty. I encourage you to take a pen and paper and see how this works by performing the example on the following diagram, the code is small, but not trivial:

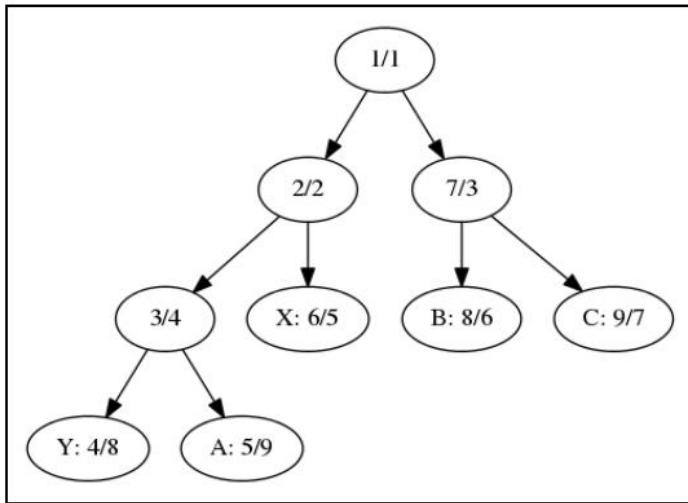


Figure 2: An example tree; the first number of each node indicates the order in which that node is visited with a depth-first algorithm; the second number indicates the order with a breadth-first algorithm

Let's get back to the real dataset. As we have a bit too much data to visualize, we will generate a trimmed-down version, where we remove the subtrees that have single species (in the case of EBOV, they have the same outbreak). We will also ladderize the tree, that is, sort the child nodes in the order of the number of children:

```

from copy import deepcopy
simple_ebola = deepcopy(ebola_raxml)

def simplify_tree(node):
    prefs = set()
    for leaf in node.leaf_nodes():
        my_toks = leaf.taxon.label.split(' ')
        if my_toks[0] == 'EBOV':
            prefs.add('EBOV' + my_toks[1])
        else:
            prefs.add(my_toks[0])
    if len(prefs) == 1:
        print(prefs, len(node.leaf_nodes()))
        node.taxon = dendropy.Taxon(label=list(prefs)[0])
        node.set_child_nodes([])
  
```

```
else:  
    for child in node.child_nodes():  
        simplify_tree(child)  
  
    simplify_tree(simple_ebola.seed_node)  
    simple_ebola.ladderize()  
    simple_ebola.write_to_path('ebola_simple.nex', 'nexus')
```

We will perform a deepcopy of the tree structure. As our function and the ladderization is destructive (it will change the tree), we will want to maintain the original tree.

DendroPy is able to enumerate all the leaf nodes (at this stage, a good exercise would be to write a function to perform this). With this functionality, we will get all the leaves for a certain node. If they share the same species and the outbreak year as in the case of EBOV, we remove all the child nodes, leaves, and internal subtree nodes.

If they do not share the same species, we recurse down until that happens. The worst case is that when you are already at a leaf node, the algorithm trivially resolves to the species of the current node.

## There's more...

There is a massive amount of computer science literature on the topic of trees and data structures; if you want to read more, Wikipedia provides a great introduction at [http://en.wikipedia.org/wiki/Tree\\_data\\_structure](http://en.wikipedia.org/wiki/Tree_data_structure).

Note that the use of lambda functions and map is not encouraged as a Python dialect; you can read some (old) opinions on the subject from Guido van Rossum at <http://www.artima.com/weblogs/viewpost.jsp?thread=98196>. I presented it here because it's a very common dialect with functional and recursive programming. The more common dialect will be based on a list comprehensions.

In any case, the functional dialect based on using map and reduce is the conceptual base for MapReduce frameworks, and you can use frameworks such as Hadoop, Disco, or Spark to perform high-performance bioinformatics computing.

## Visualizing phylogenetic data

In this recipe, we will discuss how to visualize phylogenetic trees. DendroPy has only simple visualization mechanisms based on drawing textual ASCII trees, but Biopython has quite a rich infrastructure, which we will leverage here.

## Getting ready

This will require you to have completed all of the previous recipes. Remember that we have the files for the whole genus Ebola virus, including the RAxML tree. Furthermore, a simplified genus version will have been produced in the previous recipe. As usual, you can find this content in the Chapter06/Visualization.ipynb Notebook file.

## How to do it...

Take a look at the following steps:

Let's load all the phylogenetic data:

```
from copy import deepcopy
from Bio import Phylo
ebola_tree = Phylo.read('my_ebola.nex', 'nexus')
ebola_tree.name = 'Ebolavirus tree'
ebola_simple_tree = Phylo.read('ebola_simple.nex', 'nexus')
ebola_simple_tree.name = 'Ebolavirus simplified tree'
```

For all of the trees that we read, we will change the name of the tree, since the name will be printed later.

Now we can draw ASCII representations of the trees:

```
Phylo.draw_ascii(ebola_simple_tree)
Phylo.draw_ascii(ebola_tree)
```

The ASCII representation of the simplified genus tree is shown in the following diagram. Here, we will not print the complete version because it will take several pages, but if you run the preceding code, you will be able to see that it's actually quite readable:

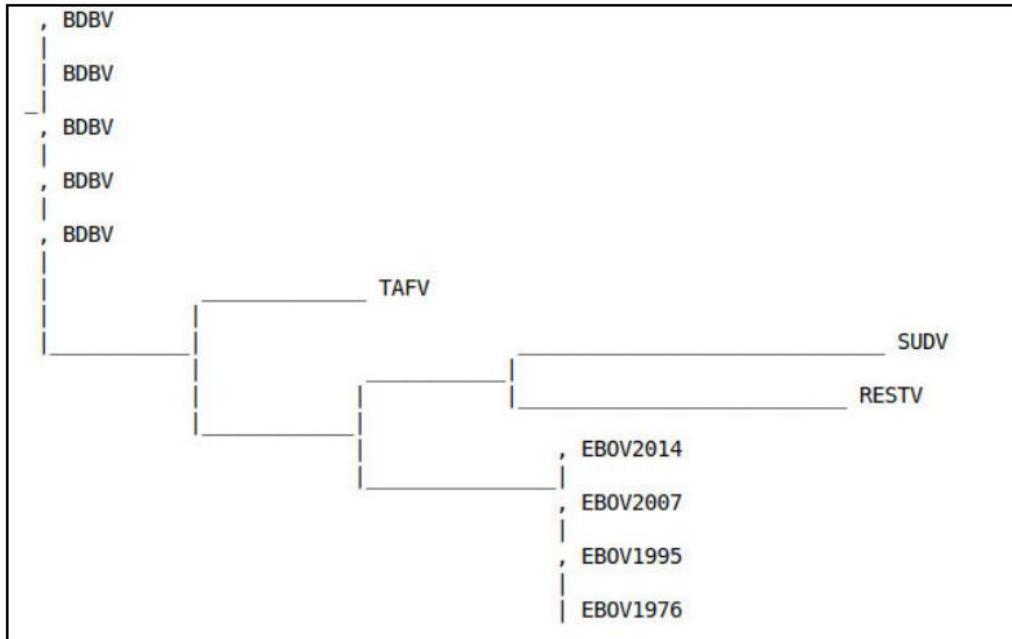


Figure 3: ASCII representation of a simplified Ebola virus dataset

Bio3Phylo allows for the graphical representation of trees by using matplotlib as a backend:

```
import matplotlib.pyplot as plt
fig = plt.figure(figsize=(16, 22))
ax = fig.add_subplot(111)
Phylo.draw(ebola_simple_tree, branch_labels=lambda c:
c.branch_length if c.branch_length > 0.02 else None, axes=ax)
```

In this case, we will print the branch lengths on the edges, but we will remove all of the lengths that are less than 0.02 to avoid clutter. The result of doing this is shown in the following diagram:

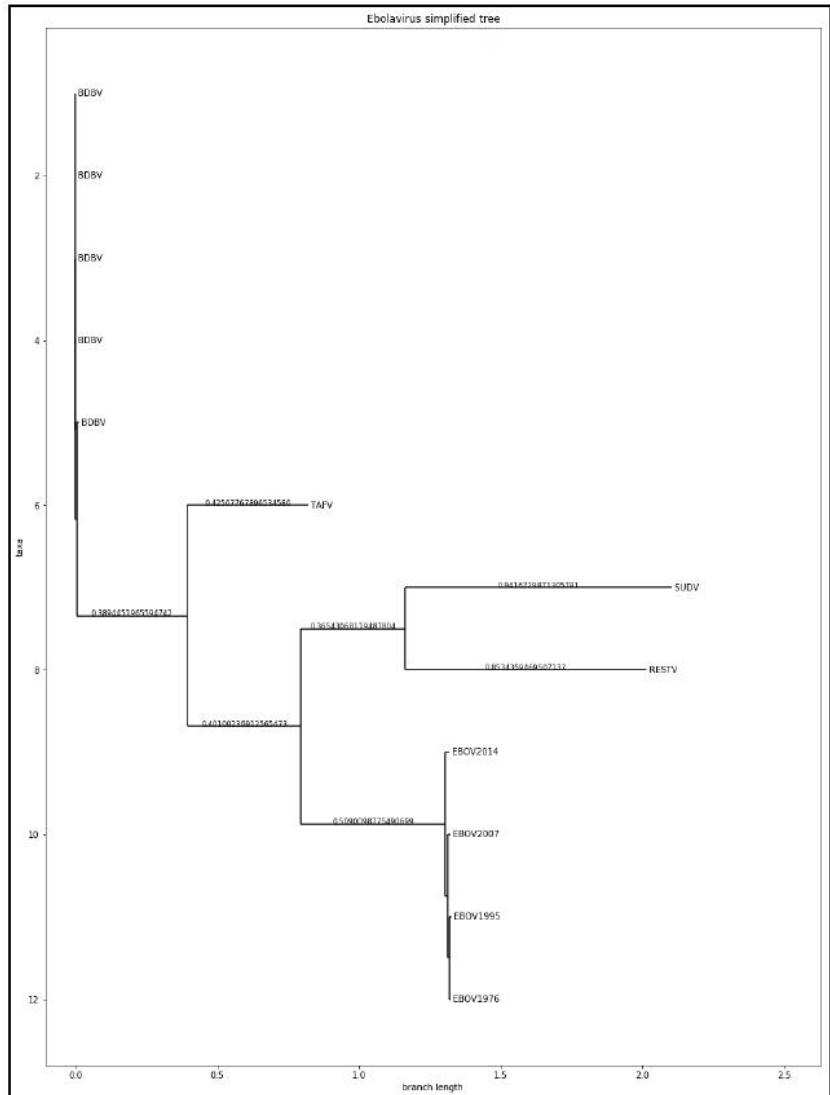


Figure 4: A matplotlib-based version of the simplified dataset with branch-lengths added

We will now plot the complete dataset, but we will color each bit of the tree differently. If a subtree has only a single virus species, it will get its own color. EBOV will have two colors; one for the 2014 outbreak and one for the others, as follows:

```
fig = plt.figure(figsize=(16, 22))
ax = fig.add_subplot(111)
from collections import OrderedDict
my_colors = OrderedDict({
    'EBOV_2014': 'red',
    'EBOV': 'magenta',
    'BDBV': 'cyan',
    'SUDV': 'blue',
    'RESTV' : 'green',
    'TAFV' : 'yellow'
})

def get_color(name):
    for pref, color in my_colors.items():
        if name.find(pref) > -1:
            return color
    return 'grey'

def color_tree(node, fun_color=get_color):
    if node.is_terminal():
        node.color = fun_color(node.name)
    else:
        my_children = set()
        for child in node.clades:
            color_tree(child, fun_color)
            my_children.add(child.color.to_hex())
        if len(my_children) == 1:
            node.color = child.color
        else:
            node.color = 'grey'

ebola_color_tree = deepcopy(ebola_tree)
color_tree(ebola_color_tree.root)
Phylo.draw(ebola_color_tree, axes=ax, label_func=lambda x:
x.name.split(' ')[0][1:] if x.name is not None else None)
```

This is a tree traversing algorithm, not unlike the ones presented in the previous recipe. As a recursive algorithm, it works as follows. If the node is a leaf, it will get a color based on its species (or the EBOV outbreak year). If it's an internal node and all the descendant nodes below it are of the same species, it will get the color of that species; if there are several species after that, it will be colored in grey. Actually, the color function can be changed, and will be changed later. Only the edge colors will be used (the labels will be printed in black).

Note that ladderization (performed in the previous recipe with DendroPy) helps quite a lot in terms of a clear visual appearance.

We also deepcopy the genus tree to color a copy; remember from the previous recipe that some tree traversal functions can change the state, and in this case, we want to preserve a version without any coloring.

Note the usage of the lambda function to clean up the name that was changed by trimAl, as shown in the following diagram:

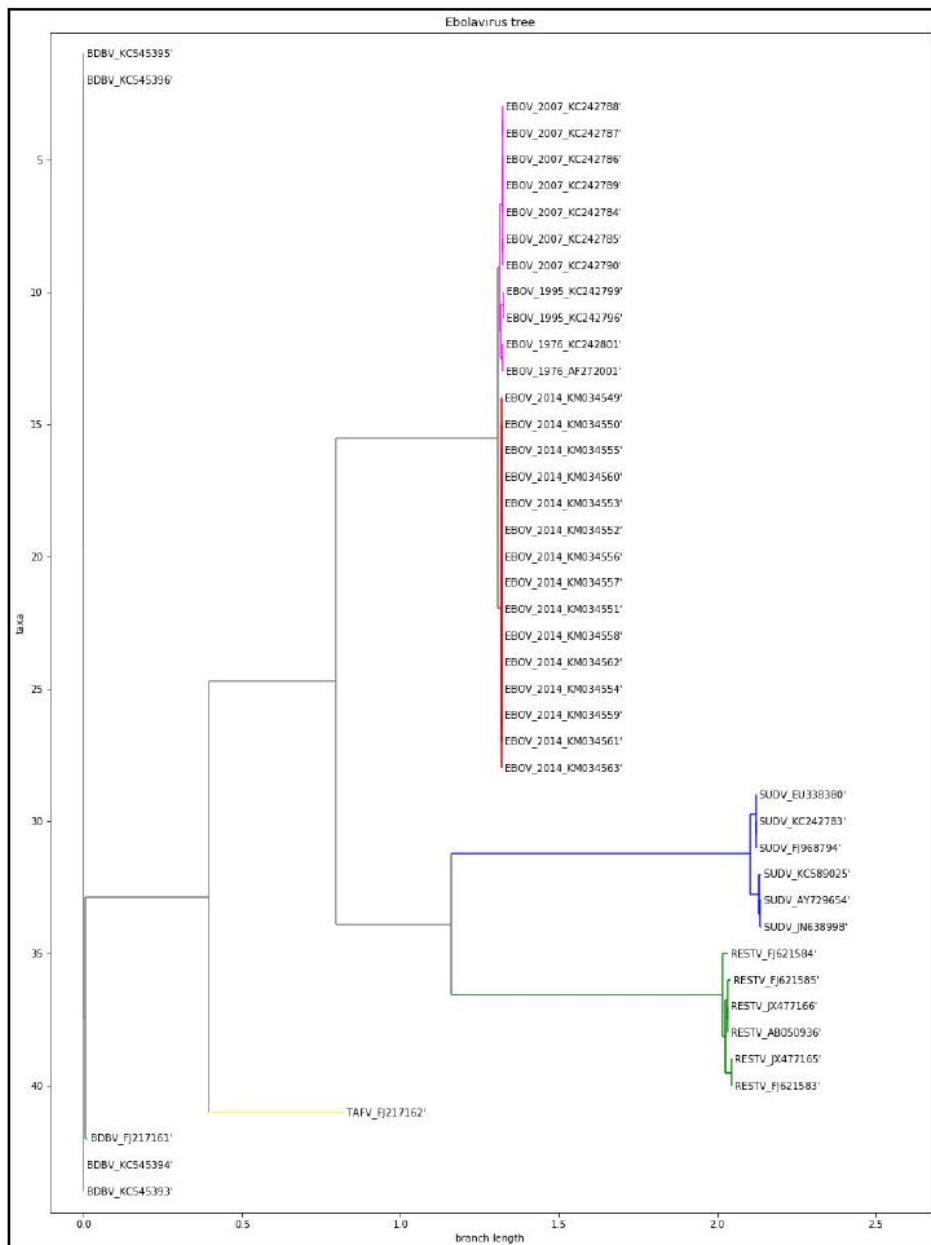


Figure 5: A ladderized and colored phylogenetic tree with the complete Ebola virus dataset

## There's more...

Tree and graph visualization is a complex topic; we will revisit this subject in Chapter 10, Other Topics in Bioinformatics, when we interface with Cytoscape. One interesting free software alternative (Java-based) for graph visualization is Gephi (<http://gephi.github.io/>). If you want to know more about the algorithms for rendering trees and graphs, check the Wikipedia page at [http://en.wikipedia.org/wiki/Graph\\_drawing](http://en.wikipedia.org/wiki/Graph_drawing) for an introduction to this fascinating topic.

Be careful not to trade style for substance, though. For example, the previous edition of this book had a pretty rendering of a phylogenetic tree using a graph-rendering library. While it was clearly the most beautiful image in that chapter, it was misleading in terms of branch lengths.

# 7

# Using the Protein Data Bank

In this chapter, we will cover the following recipes:

- Finding a protein in multiple databases
- Introducing Bio.PDB
- Extracting more information from a PDB file
- Computing molecular distances on a PDB file
- Performing geometric operations
- Animating with PyMOL
- Parsing the mmCIF files with Biopython

## Introduction

Proteomics is the study of proteins, including their function and structure. One of the main objectives of this field is to characterize the 3D structure of proteins. One of the most widely known computational resources in the proteomics field is the Protein Data Bank (PDB), a repository with structural data of large biomolecules. Of course, there are also many databases that focus instead on protein primary structure; these are somewhat similar to the genomic databases that we saw in Chapter 2, Next-Generation Sequencing.

In this chapter, we will mostly focus on processing data from the PDB. We will look at how to parse PDB files, perform some geometric computations, and visualize molecules. We will use the old PDB file format because, conceptually, it allows you to perform most necessary operations in a stable environment. Having said that, the newer mmCIF slated to replace the PDB format will also be presented in the Parsing the mmCIF files with Biopython recipe. We will use Biopython and introduce PyMOL for visualization. We will not discuss molecular docking here because this is probably more suitable for a chemoinformatics book.

Throughout this chapter, we will use a classic example of a protein: tumor protein p53, a protein involved in the regulation of the cell cycle (for example, apoptosis). This protein is highly related to cancer. There is plenty of information available about this protein on the web.

Let's start with something that you should be more familiar with right now: accessing databases, especially for a protein's primary structure (sequences of amino acids).

## Finding a protein in multiple databases

Before we start performing some more structural biology, we will look at how we can access existing proteomic databases, such as UniProt. We will query UniProt for our gene of interest, TP53, and take it from there.

## Getting ready

To access data, we will use Biopython and the REST API (we used a similar approach in Chapter 3, Working with Genomes) with the requests library to access web APIs. The requests API is an easy-to-use wrapper for web requests that can be installed using standard Python mechanisms (for example, pip and conda). You can find this content in the Chapter07/Intro.ipynb Notebook file.

## How to do it...

Take a look at the following steps:

First, let's define a function to perform REST queries on UniProt, as follows:

```
import requests
server = 'http://www.uniprot.org/uniprot'
def do_request(server, ID='', **kwargs):
    params = ''
    req = requests.get('%s/%s%s' % (server, ID, params),
    params=kwargs)
    if not req.ok:
        req.raise_for_status()
    return req
```

We can now query all p53 genes that have been reviewed:

```
req = do_request(server, query='gene:p53 AND reviewed:yes',
format='tab',
columns='id,entry name,length,organism,organism-
id,database(PDB),database(HGNC)',
limit='50')
```

We will query the p53 gene and request all entries that are reviewed (manually curated). The output will be in a tabular format. We will request a maximum of 50 results, specifying the desired columns.

We could have restricted the output to just human data, but for this example, let's include all available species.

Let's check the results, as follows:

```
import pandas as pd
import io

uniprot_list = pd.read_table(io.StringIO(req.text))
uniprot_list.rename(columns={'Organism ID': 'ID'}, inplace=True)
print(uniprot_list)
```

We will use pandas for easily processing the tab-delimited list and pretty printing. The output of the Notebook is as follows:

	Entry	Entry name	Length	Organism	ID	Cross-reference (PDB)	Cross-reference (HGNC)
0	Q9WG78	P53_BARBU	369	Barbus barbus (Barbel) ( <i>Cyprinus barbus</i> )	40830	NaN	NaN
1	Q29537	P53_CANFA	381	Canis familiaris (Dog) ( <i>Canis lupus familiaris</i> )	9615	NaN	NaN
2	O09185	P53_CRIGR	383	Cricetulus griseus (Chinese hamster) ( <i>Cricetul...</i> )	10029	NaN	NaN
3	Q8SPZ3	P53_DELLE	387	Delphinapterus leucas (Beluga whale)	9749	NaN	NaN
4	P79882	P53_HORSE	280	Equus caballus (Horse)	9796	NaN	NaN
5	P04637	P53_HUMAN	393	Homo sapiens (Human)	9606	1A1U;1AIE;1C26;1DT7;1GZH;1H26;1HS5;1JSP;1KZY;1...	11998;
6	O93379	P53_ICTPU	376	Ictalurus punctatus (Channel catfish) ( <i>Silurus...</i> )	7998	NaN	NaN
7	P56423	P53_MACFA	393	Macaca fascicularis (Crab-eating macaque) ( <i>Cyn...</i> )	9541	NaN	NaN
8	P61260	P53_MACFU	393	Macaca fuscata fuscata (Japanese macaque)	9543	NaN	NaN
9	P56424	P53_MACMU	393	Macaca mulatta (Rhesus macaque)	9544	NaN	NaN
10	P02340	P53_MOUSE	387	Mus musculus (Mouse)	10090	1HUB;2GEQ;2IOI;2IDM;2IOO;2P52;3EXJ;3EXL;	NaN
11	P25035	P53_ONCMY	396	Oncorhynchus mykiss (Rainbow trout) ( <i>Salmo gai...</i> )	8022	NaN	NaN

Now, we can get the human p53 ID and use Biopython to retrieve and parse the SwissProt record:

```
from Bio import ExPASy, SwissProt
```

```
p53_human = uniprot_list[uniprot_list.ID == 9606]['Entry'].tolist()[0]
handle = ExPASy.get_sprot_raw(p53_human)
sp_rec = SwissProt.read(handle)
```

We then use Biopython's SwissProt module to parse the record. 9606 is the NCBI taxonomic code for humans.

As usual, if you get an error with network services, it may be a network or server problem. If this is the case, just retry at a later date.

Let's take a look at the p53 record, as follows:

```
print(sp_rec.entry_name, sp_rec.sequence_length, sp_rec.gene_name)
print(sp_rec.description)
print(sp_rec.organism, sp_rec.seqinfo)
```

```
print(sp_rec.sequence)
print(sp_rec.comments)
print(sp_rec.keywords)
```

The output is as follows:

```
P53_HUMAN 393 Name=TP53; Synonyms=P53;
RecName: Full=Cellular tumor antigen p53; AltName: Full=Antigen
NY-CO-13; AltName: Full=Phosphoprotein p53; AltName: Full=Tumor
suppressor p53;
Homo sapiens (Human). (393, 43653, 'AD5C149FD8106131')
MEEPQSDPSVEPLSQETFSDLWKLLENVLSPLPSQAMDDMLSPDDIEQWFTEDPGPDEAPRMP
EAAPPVAPAPAAPTAAAPAPAPSPLSSVPSQKTYQGSYGFRLGFLHSGTAKSVTCTYSPALNKM
CQLAKTCPVQLWVDSTPPGTRVRAMAIYKQSQHMTEVVRCPHHERCSDSLAPPQHLIRVEGNL
RVEYLDDRNTFRHSVVVPYEPPEVGSDCTTIHYNYMCNSSCMGGMNRRPILTITLEDSSGNLLGRN
SFEVRVCACPGDRRTEENLRKKGEPHHELPPGSTKRALPNNTSSSPQPKKKPLDGEYFTLQIRGR
ERFEMFRELNEALELKDAQAGKEPGGSRAHSSHILSKKGQSTSRRHKKLMFKTEGPDS
```

A deeper look at the preceding record reveals a lot of really interesting information, especially on features, Gene Ontology (GO), and database cross\_references:

```
from collections import defaultdict
done_features = set()
print(len(sp_rec.features))
for feature in sp_rec.features:
    if feature[0] in done_features:
        continue
    else:
        done_features.add(feature[0])
        print(feature)
print(len(sp_rec.cross_references))
per_source = defaultdict(list)
for xref in sp_rec.cross_references:
    source = xref[0]
    per_source[source].append(xref[1:])
print(per_source.keys())
done_GOs = set()
print(len(per_source['GO']))
for annot in per_source['GO']:
    if annot[1][0] in done_GOs:
        continue
    else:
        done_GOs.add(annot[1][0])
        print(annot)
```

Note that we are not even printing the whole information here, just a summary of it. We print a number of features on the sequence with one example per type, a number of external database references plus databases that are referred, and a number of GO entries, along with three examples. Currently, there are 1493 features, 604 external references, and 133 GO terms just for this protein:

```

1493
('CHAIN', 1, 393, 'Cellular tumor antigen p53.', 'PRO_0000105703')
('DNA_BIND', 102, 292, '', '')
('REGION', 1, 83, 'Interaction with HRMT1L2.', '')
('MOTIF', 17, 25, 'TADI.', '')
('METAL', 176, 176, 'Zinc.', '')
('SITE', 120, 120, 'Interaction with DNA.', '')
('MOD_RES', 9, 9, 'Phosphoserine; by HTPK4. {ECO:0000269|PubMed:18022393}.', '')
('CROSSLINK', 291, 291, 'Glycyl lysine isopeptide (Lys-Gly) (interchain with G-Cter in ubiquitin). {ECO:0000269|PubMed:19536131}.', '')
('VAR_SEQ', 1, 132, 'Missing (in Isoform 7, Isoform 8 and isoform 9). {ECO:0000303|PubMed:16131611}.', 'VSP_040833')
('VARIANT', 5, 5, 'Q->H (in a sporadic cancer; somatic mutation; abolishes strongly phosphorylation).', 'VAR_044543')
('MUTAGEN', 15, 15, 'S->A: Loss of interaction with PPP2R5C, PPP2CA AND PPP2R1A. {ECO:0000269|PubMed:17967874}.', '')
('HELIX', 19, 23, '{ECO:0000244|PDB:3DAC}.', '')
('STRAND', 27, 29, '{ECO:0000244|PDB:2K8F}.', '')
('TURN', 105, 106, '{ECO:0000244|PDB:3D06}.', '')
604
('GeneReviews', 'DNASU', 'MIM', 'SUPFAM', 'Genevestigator', 'HOVERGEN', 'ExpressionAtlas', 'MaxQB', 'GeneWiki', 'SMR', 'Orphanet',
'CTD', 'GO', 'PhylomeDB', 'CCDS', 'nextProt', 'BindingDB', 'RefSeq', 'PRIDE', 'DMDM', 'Reactome', 'PROSITE', 'TreeFam', 'SWISS-2DPA
GE', 'NextBio', 'DIP', 'PRO', 'PANTHER', 'TCDB', 'Gene3D', 'DrugBank', 'PMAP-CutDB', 'Bgee', 'EvolutionaryTrace', 'ChEMBL', 'PIR',
'InParanoid', 'GeneCards', 'Pfam', 'PDBsum', 'KEGG', 'eggNOG', 'EMBL', 'PaxDb', 'DisProt', 'Proteomes', 'ProteinModelPortal', 'Bise
mbl', 'ChitaRS', 'SignalLink', 'HPA', 'IntAct', 'MINT', 'PDB', 'UniGene', 'OMA', 'InterPro', 'PharmGKB', 'PhosphoSite', 'GenomeRNai',
'KO', 'Biogrid', 'UCSC', 'HGNC', 'PRINTS', 'GeneTree', 'GeneID')
133
('GO:0000785', 'G:chromatin', 'IBA:GO_Central')
('GO:0005524', 'F:ATP binding', 'IDA:UniProtKB')
('GO:0006915', 'P:apoptotic process', 'TAS:Reactome')

```

## There's more...

There are many more databases with information on proteins—some of these are referred to in the preceding record. You can explore its result to try and find data elsewhere. For detailed information about UniProt's REST interface, refer to [http://www.uniprot.org/help/programmatic\\_access](http://www.uniprot.org/help/programmatic_access).

## Introducing Bio.PDB

Here, we will introduce Biopython's PDB module for working with the Protein Data Bank. We will use three models that represent part of the p53 protein. You can read more about these files and p53 at <http://www.rcsb.org/pdb/101/motm.do?momID=31>.

## Getting ready

You should be aware of the basic PDB data model of model/chain/residue/atom objects. A good explanation on Biopython's Structural Bioinformatics FAQ can be found at [http://biopython.org/wiki/The\\_Biopython\\_Structural\\_Bioinformatics\\_FAQ](http://biopython.org/wiki/The_Biopython_Structural_Bioinformatics_FAQ).

You can find this content in the Chapter07/PDB.ipynb Notebook file.

Of the three models that we will download, the 1TUP model will be used in the remaining recipes. Take some time to study this model, as it will help you later on.

## How to do it...

Take a look at the following steps:

First, let's retrieve our models of interest, as follows:

```
from Bio import PDB

repository = PDB.PDBList()
repository.retrieve_pdb_file('1TUP', pdir='.', file_format='pdb')
repository.retrieve_pdb_file('1OLG', pdir='.', file_format='pdb')
repository.retrieve_pdb_file('1YCQ', pdir='.', file_format='pdb')
```

Note that Bio.PDB will take care of downloading files for you. Moreover, these downloads will only occur if no local copy is already present.

Let's parse our records, as shown in the following code:

```
parser = PDB.PDBParser()
p53_1tup = parser.get_structure('P 53 - DNA Binding',
'pdb1tup.ent')
p53_1olg = parser.get_structure('P 53 - Tetramerization',
'pdb1olg.ent')
p53_1ycq = parser.get_structure('P 53 - Transactivation',
'pdb1ycq.ent')
```

You may get some warnings about the content of the file. These are usually not problematic.

Let's inspect our headers, as follows:

```
def print_pdb_headers(headers, indent=0):
    ind_text = ' ' * indent
    for header, content in headers.items():
```

```

if type(content) == dict:
    print('\n%s%20s:' % (ind_text, header))
    print_pdb_headers(content, indent + 4)
    print()
elif type(content) == list:
    print('%s%20s:' % (ind_text, header))
    for elem in content:
        print('%s%21s %s' % (ind_text, '->', elem))
else:
    print('%s%20s: %s' % (ind_text, header, content))

print_pdb_headers(p53_1tup.header)

```

Headers are parsed as a dictionary of dictionaries. As such, we will use a recursive function to parse them. This function will increase indentation for ease of reading, and annotate lists of elements with the → prefix. For more details on recursive functions, refer to the previous chapter, Chapter 6, Phylogenetics. The output is as follows:

```

structure_method: x-ray diffraction
    head: antitumor protein/dna
    journal: AUTH Y.CHO,S.GORINA,P.D.JEFFREY,N.P.PAVLETICH TITL CRYSTAL STRUCTURE OF A P53 TUMOR SUPPRESSOR-DNA TITL 2 C
OMPLEX: UNDERSTANDING TUMORIGENIC MUTATIONS.REF SCIENCE V. 265 346 1994REFN ISSN 0036-
8075PMID 8023157
    journal_reference: y.cho,s.gorina,p.d.jeffrey,n.p.pavletich crystal structure of a p53 tumor suppressor-dna complex: understanding tumorigenic mutations. science v. 265 346 1994 issn 0036-8075 8023157

compound:

    1:
        molecule: dna (5'-d(*tp*tp*tp*cp*cp*tp*ap*gp*ap*cp*tp*tp*gp*cp*cp*cp*a p*ap*tp*tp*a)-3')
        misc:
        engineered: yes
        chain: e

    3:
        molecule: protein (p53 tumor suppressor )
        misc:
        engineered: yes
        chain: a, b, c

    2:
        molecule: dna (5'-d(*ap*tp*ap*ap*tp*tp*gp*gp*gp*cp*ap*ap*gp*tp*cp*tp*a p*gp*gp*ap*a)-3')
        misc:
        engineered: yes
        chain: f

keywords: antigen p53, antitumor protein/dna complex
name: tumor suppressor p53 complexed with dna
author: Y.Cho,S.Gorina,P.D.Jeffrey,N.P.Pavletich
deposition_date: 1995-07-11
release_date: 1995-07-11

```

We want to know the content of each chain on these files; for this, let's take a look at the COMPND records:

```
print(p53_1tup.header['compound'])
print(p53_1olg.header['compound'])
print(p53_1ycq.header['compound'])
```

This will return all compound headers printed in the preceding code. Unfortunately, this is not the best way to get information on chains. An alternative would be to get DBREF records, but Biopython's parser is currently not able to access these. Having said that, using a tool like grep will easily extract this information.

Note that for the 1TUP model, A, B, and C chains are from the protein, and E and F chains are from the DNA. This information will be useful in the future.

Let's do a top-down analysis of each PDB file. For now, let's just get all of the chains, the number of residues, and atoms per chain, as follows:

```
def describe_model(name, pdb):
    print()
    for model in pdb:
        for chain in model:
            print('%s - Chain: %s. Number of residues: %d. Number of
atoms: %d.' %
                  (name, chain.id, len(chain),
                   len(list(chain.get_atoms()))))

    describe_model('1TUP', p53_1tup)
    describe_model('1OLG', p53_1olg)
    describe_model('1YCQ', p53_1ycq)
```

We will perform a bottom-up approach in a later recipe. Here is the output for 1TUP:

```
1TUP - Chain: E. Number of residues: 43. Number of atoms: 442.
1TUP - Chain: F. Number of residues: 35. Number of atoms: 449.
1TUP - Chain: A. Number of residues: 395. Number of atoms: 1734.
1TUP - Chain: B. Number of residues: 265. Number of atoms: 1593.
1TUP - Chain: C. Number of residues: 276. Number of atoms: 1610.

1OLG - Chain: A. Number of residues: 42. Number of atoms: 698.
1OLG - Chain: B. Number of residues: 42. Number of atoms: 698.
1OLG - Chain: C. Number of residues: 42. Number of atoms: 698.
1OLG - Chain: D. Number of residues: 42. Number of atoms: 698.

1YCQ - Chain: A. Number of residues: 123. Number of atoms: 741.
1YCQ - Chain: B. Number of residues: 16. Number of atoms: 100.
```

Let's get all non-standard residues (HETATM), with the exception of Water, in the 1TUP model, as shown in the following code:

```
for residue in p53_1tup.get_residues():
    if residue.id[0] in [' ', 'W']:
        continue
    print(residue.id)
```

We have three zincs, one on each of the protein chains.

Let's take a look at a residue:

```
res = next(p53_1tup[0]['A'].get_residues())
print(res)
for atom in res:
    print(atom, atom.serial_number, atom.element)
p53_1tup[0]['A'][94]['CA']
```

This will print all atoms on a certain residue:

```
<Residue SER het= resseq=94 icode= >
<Atom N> 858 N
<Atom CA> 859 C
<Atom C> 860 C
<Atom O> 861 O
<Atom CB> 862 C
<Atom OG> 863 O
<Atom CA>
```

Note the last statement. It is there just to show you that you can directly access an Atom by resolving model, chain, Residue, and finally, the Atom.

Finally, let's export the protein fragment to a FASTA file, as follows:

```
from Bio.SeqIO import PdbIO, FastaIO

def get_fasta(pdb_file, fasta_file, transfer_ids=None):
    fasta_writer = FastaIO.FastaWriter(fasta_file)
    fasta_writer.write_header()
    for rec in PdbIO.PdbSeqresIterator(pdb_file):
        if len(rec.seq) == 0:
            continue
        if transfer_ids is not None and rec.id not in transfer_ids:
            continue
        print(rec.id, rec.seq, len(rec.seq))
        fasta_writer.write_record(rec)
get_fasta(open('pdb1tup.ent'), open('1tup.fasta', 'w'),
transfer_ids=['1TUP:B'])
```

```
get_fasta(open('pdb1olg.ent'), open('1olg.fasta', 'w'),
transfer_ids=['1OLG:B'])
get_fasta(open('pdb1ycq.ent'), open('1ycq.fasta', 'w'),
transfer_ids=['1YCQ:B'])
```

If you inspect the protein chain, you will see that they are equal in each model, so we export a single one. In the case of 1YCQ, we export the smallest one, because the biggest one is not p53-related. As you can see, here, we are using Bio.SeqIO, not Bio.PDB.

## There's more...

The PDB parser is incomplete. It's not very likely that a complete parser will be seen soon, as the community is migrating to the mmCIF format.

Although the future is the mmCIF format (<http://mmcif.wwpdb.org/>), PDB files are still around. Conceptually, many operations are similar after you have parsed the file.

# Extracting more information from a PDB file

Here, we will continue our exploration of the record structure produced by Bio.PDB from PDB files.

## Getting ready

For general information about the PDB models that we are using, refer to the previous recipe.

You can find this content in the Chapter07/Stats.ipynb Notebook file.

## How to do it...

Take a look at the following steps:

First, let's retrieve 1TUP, as follows:

```
from Bio import PDB
repository = PDB.PDBList()
parser = PDB.PDBParser()
repository.retrieve_pdb_file('1TUP', pdir='.', file_format='pdb')
#XXX
p53_1tup = parser.get_structure('P 53', 'pdb1tup.ent')
```

Then, extract some atom-related statistics:

```
from collections import defaultdict

atom_cnt = defaultdict(int)
atom_chain = defaultdict(int)
atom_res_types = defaultdict(int)
for atom in p53_1tup.get_atoms():
    my_residue = atom.parent
    my_chain = my_residue.parent
    atom_chain[my_chain.id] += 1
    if my_residue.resname != 'HOH':
        atom_cnt[atom.element] += 1
    atom_res_types[my_residue.resname] += 1
print(dict(atom_res_types))
print(dict(atom_chain))
print(dict(atom_cnt))
```

This will print information on the atom's residue type, number of atoms per chain, and quantity per element, as follows:

```
{'DT': 257, 'DC': 152, 'DA': 270, 'DG': 176, 'HOH': 384, 'SER': 323, 'VAL': 315, 'PRO': 294, 'GLN': 189, 'LYS': 135, 'THR': 294, 'TYR': 288, 'GLY': 156, 'PHE': 165, 'ARG': 561, 'LEU': 336, 'HIS': 210, 'ALA': 105, 'CYS': 180, 'ASN': 216, 'MET': 144, 'TRP': 42, 'ASP': 192, 'ILE': 144, 'GLU': 297, 'ZN': 3}
{'E': 442, 'F': 449, 'A': 1734, 'B': 1593, 'C': 1610}
{'O': 1114, 'C': 3238, 'N': 1001, 'P': 40, 'S': 48, 'ZN': 3}
```

Note that the preceding number of residues is not the proper number of residues, but the amount of times that a certain residue type is referred (it adds up to the number of atoms, not residues).

Note the water (W), nucleotide (DA, DC, DG, and DT), and zinc (ZN) residues, which add to the amino acid ones.

Now, let's count the instance per residue and the number of residues per chain:

```
res_types = defaultdict(int)
res_per_chain = defaultdict(int)
for residue in p53_1tup.get_residues():
    res_types[residue.resname] += 1
    res_per_chain[residue.parent.id] +=1
print(dict(res_types))
print(dict(res_per_chain))
```

The following is the output:

```
{'DT': 13, 'DC': 8, 'DA': 13, 'DG': 8, 'HOH': 384, 'SER': 54,
'VAL': 45, 'PRO': 42, 'GLN': 21, 'LYS': 15, 'THR': 42, 'TYR': 24,
'GLY': 39, 'PHE': 15, 'ARG': 51, 'LEU': 42, 'HIS': 21, 'ALA': 21,
'CYS': 30, 'ASN': 27, 'MET': 18, 'TRP': 3, 'ASP': 24, 'ILE': 18,
'GLU': 33, 'ZN': 3}
{'E': 43, 'F': 35, 'A': 395, 'B': 265, 'C': 276}
```

We can also get the bounds of a set of atoms:

```
import sys

def get_bounds(my_atoms):
    my_min = [sys.maxsize] * 3
    my_max = [-sys.maxsize] * 3
    for atom in my_atoms:
        for i, coord in enumerate(atom.coord):
            if coord < my_min[i]:
                my_min[i] = coord
            if coord > my_max[i]:
                my_max[i] = coord
    return my_min, my_max

chain_bounds = []
for chain in p53_1tup.get_chains():
    print(chain.id, get_bounds(chain.get_atoms()))
    chain_bounds[chain.id] = get_bounds(chain.get_atoms())
print(get_bounds(p53_1tup.get_atoms()))
```

A set of atoms can be a whole model, a chain, a residue, or any subset you are interested in. In this case, we will print boundaries for all chains and the whole model. Numbers convey little intuition, so we will get a little bit more graphical.

To have a notion of the size of each chain, a plot is probably more informative than the numbers in the preceding code:

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure(figsize=(16, 9))
ax3d = fig.add_subplot(111, projection='3d')
ax_xy = fig.add_subplot(331)
ax_xy.set_title('X/Y')
ax_xz = fig.add_subplot(334)
ax_xz.set_title('X/Z')
ax_zy = fig.add_subplot(337)
ax_zy.set_title('Z/Y')
color = {'A': 'r', 'B': 'g', 'C': 'b', 'E': '0.5', 'F': '0.75'}
zx, zy, zz = [], [], []
for chain in p53_1tup.get_chains():
    xs, ys, zs = [], [], []
    for residue in chain.get_residues():
        ref_atom = next(residue.get_iterator())
        x, y, z = ref_atom.coord
        if ref_atom.element == 'ZN':
            zx.append(x)
            zy.append(y)
            zz.append(z)
            continue
        xs.append(x)
        ys.append(y)
        zs.append(z)
    ax3d.scatter(xs, ys, zs, color=color[chain.id])
    ax_xy.scatter(xs, ys, marker='.', color=color[chain.id])
    ax_xz.scatter(xs, zs, marker='.', color=color[chain.id])
    ax_zy.scatter(zs, ys, marker='.', color=color[chain.id])
    ax3d.set_xlabel('X')
    ax3d.set_ylabel('Y')
    ax3d.set_zlabel('Z')
    ax3d.scatter(zx, zy, zz, color='k', marker='v', s=300)
    ax_xy.scatter(zx, zy, color='k', marker='v', s=80)
    ax_xz.scatter(zx, zz, color='k', marker='v', s=80)
    ax_zy.scatter(zz, zy, color='k', marker='v', s=80)
    for ax in [ax_xy, ax_xz, ax_zy]:
        ax.get_yaxis().set_visible(False)
        ax.get_xaxis().set_visible(False)
```

There are plenty of molecular visualization tools. Indeed, we will discuss PyMOL later. However, matplotlib is enough for some simple visualization. The most important point about matplotlib is that it's stable and very easy to integrate into reliable production code.

In the following chart, we performed a 3D plot of chains, with the DNA in grey and the protein chains in different colors. We also plot planar projections (X/Y, X/Z, and Z/Y) on the left-hand side in the following graph:

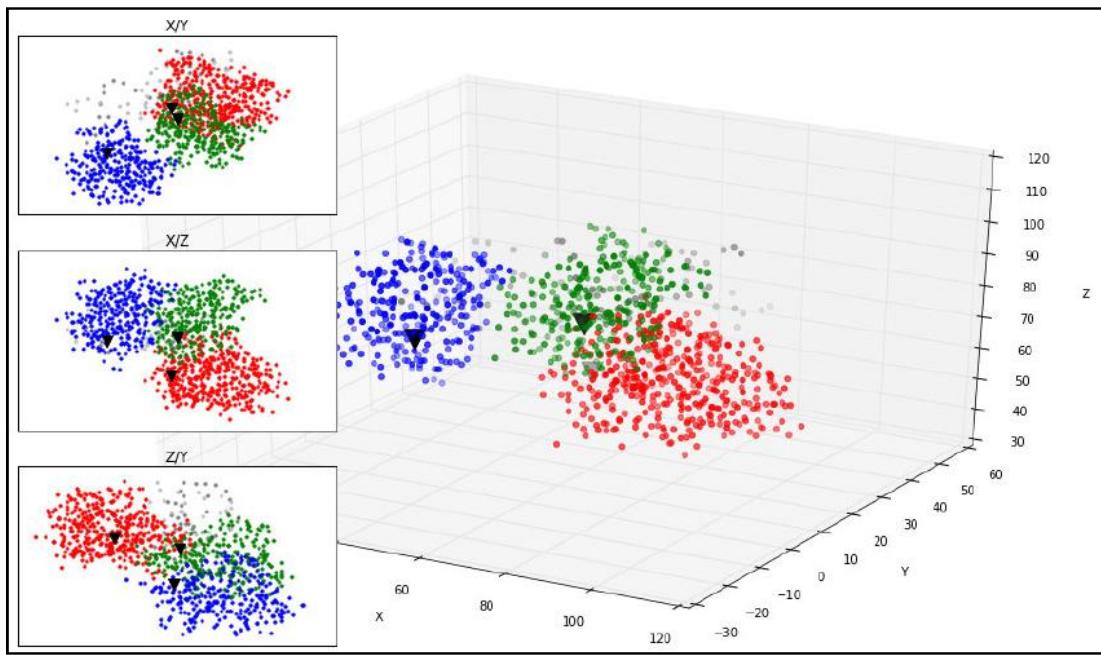


Figure 1: The spatial distribution of the protein chains; the main figure is a 3D plot and the left subplots are planar views (X/Y, X/Z, and Z/Y)

## Computing molecular distances on a PDB file

Here, we will find atoms closer to three zincs in the 1TUP model. We will consider several distances to these zincs. We will take this opportunity to discuss the performance of algorithms.

## Getting ready

You can find this content in the Chapter07/Distance.ipynb Notebook file. Take a look at the Introducing Bio.PDB recipe in this chapter for more information.

## How to do it...

Take a look at the following steps:

Let's load our model, as follows:

```
from Bio import PDB
repository = PDB.PDBList()
parser = PDB.PDBParser()
repository.retrieve_pdb_file('1TUP', pdir='.', file_format='pdb')
p53_1tup = parser.get_structure('P 53', 'pdb1tup.ent')
```

We will now get our zincs, against which we will perform later comparisons:

```
zns = []
for atom in p53_1tup.get_atoms():
    if atom.element == 'ZN':
        zns.append(atom)
for zn in zns:
    print(zn, zn.coord)
```

You should see three zinc atoms.

Now, let's define a function to get the distance between one atom and a set of other atoms, as follows:

```
import math

def get_closest_atoms(pdb_struct, ref_atom, distance):
    atoms = []
    rx, ry, rz = ref_atom.coord
    for atom in pdb_struct.get_atoms():
        if atom == ref_atom:
            continue
        x, y, z = atom.coord
        my_dist = math.sqrt((x - rx)**2 + (y - ry)**2 + (z - rz)**2)
        if my_dist < distance:
            atoms[atom] = my_dist
    return atoms
```

We get coordinates for our reference atom and then iterate over our desired comparison list. If an atom is close enough, it's added to the return list.

We now compute atoms near our zinCs, the distance of which is up to 4 Ångströms for our model:

```
for zn in zns:  
    print()  
    print(zn.coord)  
    atoms = get_closest_atoms(p53_1tup, zn, 4)  
    for atom, distance in atoms.items():  
        print(atom.element, distance, atom.coord)
```

Here, we show the result for the first zinc, including the element, distance, and coordinates:

```
[58.108 23.242 57.424]  
C 3.4080117696286854 [57.77 21.214 60.142]  
S 2.3262243799594877 [57.065 21.452 58.482]  
C 3.4566537492335123 [58.886 20.867 55.036]  
C 3.064120559761192 [58.047 22.038 54.607]  
N 1.9918273537290707 [57.755 23.073 55.471]  
C 2.9243719601324525 [56.993 23.943 54.813]  
C 3.857729198122736 [61.148 25.061 55.897]  
C 3.62725094648044 [61.61 24.087 57.001]  
S 2.2789209624943494 [60.317 23.318 57.979]  
C 3.087214470667822 [57.205 25.099 59.719]  
S 2.2253158446520818 [56.914 25.054 57.917]
```

We only have three zinCs, so the number of computations is quite reduced. However, imagine that we had more, or that we were doing a pairwise comparison among all atoms in the set (remember that the number of comparisons grows quadratically with the number of atoms in a pairwise case). Although our case is small, it's not difficult to forecast use cases, with more comparisons taking a lot of time. We will get back to this soon.

Let's see how many atoms we get as we increase the distance:

```
for distance in [1, 2, 4, 8, 16, 32, 64, 128]:  
    my_atoms = []  
    for zn in zns:  
        atoms = get_closest_atoms(p53_1tup, zn, distance)  
        my_atoms.append(len(atoms))  
    print(distance, my_atoms)
```

The result is as follows:

```
1 [0, 0, 0]
2 [1, 0, 0]
4 [11, 11, 12]
8 [109, 113, 106]
16 [523, 721, 487]
32 [2381, 3493, 2053]
64 [5800, 5827, 5501]
128 [5827, 5827, 5827]
```

As we have seen before, this specific case is not very expensive, but let's time this anyway:

```
import timeit
nexecs = 10
print(timeit.timeit('get_closest_atoms(p53_1tuple, zns[0], 4.0)',
    'from __main__ import get_closest_atoms, p53_1tuple, zns',
    number=nexecs) / nexecs * 1000)
```

Here, we will use the timeit module to execute this function 10 times and then print the result in milliseconds. We pass the function as a string and pass yet another string with the necessary imports to make this function work. On a Notebook, you are probably aware of the %timeit magic and how it makes your life much easier in this case. This takes roughly 40 milliseconds on the machine where the code was tested. Obviously, on your computer, you will get somewhat different results.

Can we do better? Let's consider a different distance function, as shown in the following code:

```
def get_closest_alternative(pdb_struct, ref_atom, distance):
    atoms = {}
    rx, ry, rz = ref_atom.coord
    for atom in pdb_struct.get_atoms():
        if atom == ref_atom:
            continue
        x, y, z = atom.coord
        if abs(x - rx) > distance or abs(y - ry) > distance or
            abs(z - rz) > distance:
            continue
        my_dist = math.sqrt((x - rx)**2 + (y - ry)**2 + (z -
            rz)**2)
        if my_dist < distance:
            atoms[atom] = my_dist
    return atoms
```

So, we take the original function and add a very simplistic if with distances. The rationale for this is that the computational cost of the square root, and maybe the float power operation, is very expensive, so we will try to avoid it. However, for all atoms that are closer than the target distance in any dimension, this function will be more expensive.

Now, let's time against it:

```
print(timeit.timeit('get_closest_alternative(p53_1tuple, zns[0],  
4.0)',  
      'from __main__ import get_closest_alternative, p53_1tuple,  
zns',  
      number=nexecs) / nexecs * 1000)
```

On the same machine that we used in the preceding example, we have 16 milliseconds, that is, it is roughly three times faster.

However, is it always better? Let's compare the cost with different distances, as follows:

```
print('Standard')  
for distance in [1, 4, 16, 64, 128]:  
    print(timeit.timeit('get_closest_atoms(p53_1tuple, zns[0],  
distance)',  
      'from __main__ import get_closest_atoms, p53_1tuple, zns,  
distance',  
      number=nexecs) / nexecs * 1000)  
print('Optimized')  
for distance in [1, 4, 16, 64, 128]:  
    print(timeit.timeit('get_closest_alternative(p53_1tuple, zns[0],  
distance)',  
      'from __main__ import get_closest_alternative, p53_1tuple,  
zns, distance',  
      number=nexecs) / nexecs * 1000)
```

The result is shown in the following output:

```
Standard  
85.08649739999328  
86.50681579999855  
86.79630599999655  
96.95437099999253  
96.21982420001132  
Optimized  
30.253444099980698  
32.69531210000878  
52.965772600009586
```

```
142.53310030001103  
141.26269519999823
```

Note that the cost of the Standard version is mostly constant, whereas the Optimized version varies with the distance to get the closest atoms; the larger the distance, the more cases will be computed using the extra if, plus the square root, making the function more expensive.

The larger point here is that you can probably code functions that are more efficient using smart computation shortcuts, but the complexity cost may change qualitatively. In the preceding case, I suggest that the second function is more efficient for all realistic and interesting cases when you're trying to find the closest atoms. However, you have to be careful while designing your own versions of optimized algorithms.

## Performing geometric operations

We will now perform computations with geometry information, including computing the center of the mass of chains and of whole models.

### Getting ready

You can find this content in the Chapter07/Mass.ipynb Notebook file.

### How to do it...

Let's take a look at the following steps:

First, let's retrieve the data:

```
from Bio import PDB  
repository = PDB.PDBList()  
parser = PDB.PDBParser()  
repository.retrieve_pdb_file('1TUP', pdir='.', file_format='pdb')  
p53_1tup = parser.get_structure('P 53', 'pdb1tup.ent')
```

Then, let's recall the type of residues that we have with the following code:

```
my_residues = set()  
for residue in p53_1tup.get_residues():  
    my_residues.add(residue.id[0])  
print(my_residues)
```

So, we have H\_ZN (zinc) and W (water), which are HETATMs; the vast majority are standard PDB atoms.

Let's compute the masses for all chains, zinCs, and waters using the following code:

```
def get_mass(atoms, accept_fun=lambda atom: atom.parent.id[0] != 'W'):
    return sum([atom.mass for atom in atoms if accept_fun(atom)])

chain_names = [chain.id for chain in p53_1tup.get_chains()]
my_mass = np.ndarray(len(chain_names), 3)
for i, chain in enumerate(p53_1tup.get_chains()):
    my_mass[i, 0] = get_mass(chain.get_atoms())
    my_mass[i, 1] = get_mass(chain.get_atoms(),
                           accept_fun=lambda atom: atom.parent.id[0] not in [' ', 'W'])
    my_mass[i, 2] = get_mass(chain.get_atoms(),
                           accept_fun=lambda atom: atom.parent.id[0] == 'W')
masses = pd.DataFrame(my_mass, index=chain_names, columns=['No Water', 'ZinCs', 'Water'])
print(masses) # masses
```

The get\_mass function returns the mass of all atoms in the list that pass an acceptance criterion function. The default acceptance criterion is not to be a water residue.

We then compute the mass for all chains. We have three versions: just amino acids, zinCs, and waters. Zinc does nothing more than detect a single atom per chain in this model. The output is as follows:

	No Water	ZinCs	Water
E	6068.04412	0.00	351.9868
F	6258.20442	0.00	223.9916
A	20548.26300	65.39	3167.8812
B	20368.18840	65.39	1119.9580
C	20466.22540	65.39	1279.9520

Let's compute the geometric center and center of mass of the model, as follows:

```
def get_center(atoms,
    weight_fun=lambda atom: 1 if atom.parent.id[0] != 'W' else 0):
    xsum = ysum = zsum = 0.0
    acum = 0.0
    for atom in atoms:
        x, y, z = atom.coord
        weight = weight_fun(atom)
        acum += weight
        xsum += weight * x
        ysum += weight * y
        zsum += weight * z
    return xsum / acum, ysum / acum, zsum / acum
print(get_center(p53_1tup.get_atoms()))
print(get_center(p53_1tup.get_atoms(),
    weight_fun=lambda atom: atom.mass if atom.parent.id[0] != 'W'
else 0))
```

First, we define a weighted function to get the coordinates of the center. The default function will treat all atoms as equal, as long as they are not a water residue.

We then compute the geometric center and the center of mass by redefining the weight function with a value of each atom equal to its mass. The geometric center is computed, irrespective of its molecular weights.

For example, you may want to compute the center of mass of the protein without DNA chains.

Let's compute the center of mass and the geometric center of each chain, as follows:

```
my_center = np.ndarray((len(chain_names), 6))
for i, chain in enumerate(p53_1tup.get_chains()):
    x, y, z = get_center(chain.get_atoms())
    my_center[i, 0] = x
    my_center[i, 1] = y
    my_center[i, 2] = z
    x, y, z = get_center(chain.get_atoms(),
        weight_fun=lambda atom: atom.mass if atom.parent.id[0] !=
    'W' else 0)
    my_center[i, 3] = x
    my_center[i, 4] = y
    my_center[i, 5] = z
weights = pd.DataFrame(my_center, index=chain_names,
    columns=['X', 'Y', 'Z', 'X (Mass)', 'Y (Mass)', 'Z (Mass)'])
print(weights) # weights
```

The result is shown here:

	X	Y	Z	X (Mass)	Y (Mass)	Z (Mass)
E	49.727231	32.744879	81.253417	49.708513	32.759725	81.207395
F	51.982368	33.843370	81.578795	52.002223	33.820064	81.624394
A	72.990763	28.825429	56.714012	72.822668	28.810327	56.716117
B	67.810026	12.624435	88.656590	67.729100	12.724130	88.545659
C	38.221565	-5.010494	88.293141	38.169364	-4.915395	88.166711

## There's more...

Although this is not a book based on the protein structure determination technique, remember that X-ray crystallography methods cannot detect hydrogens, so computing the mass of residues might be based on very inaccurate models; refer to [http://www.umass.edu/microbio/chime/pe\\_beta/pe/protepl/help\\_hyd.htm](http://www.umass.edu/microbio/chime/pe_beta/pe/protepl/help_hyd.htm) for more information.

## Animating with PyMOL

Here, we will create a video of the p53 1TUP model. We will start our animation by going around the p53 1TUP model and then zooming in; as we zoom in, we change the render strategy so that you can see what is deep in the model. You can find the YouTube version of the video that you will generate at <https://www.youtube.com/watch?v=CuEP40Id9O8>.

## Getting ready

This recipe will be presented as a Python script, not as a Notebook. This is mostly because the output is not interactive, but a set of image files that will need further post-processing.

You will need to install PyMOL (<http://www.pymol.org>). On Debian/Ubuntu/Linux, you can use the apt-get install pymol command.

PyMOL is more of an interactive program than a Python library, so I strongly encourage that you play with it before moving on to the recipe. This can be fun! The code for this recipe is available on the GitHub repository as a script, along with this chapter's Notebook file at Chapter07. We will use the PyMol\_Movie.py file in this recipe.

## How to do it...

Take a look at the following steps:

Let's initialize and retrieve our PDB model and prepare the rendering, as follows:

```
import pymol
from pymol import cmd
#pymol.pymol_argv = ['pymol', '-qc'] # Quiet / no GUI
pymol.finish_launching()

cmd.fetch('1TUP', async=False)
cmd.disable('all')
cmd.enable('1TUP')
cmd.hide('all')
cmd.show('sphere', 'name zn')
```

Note that the `pymol_argv` line makes the code silent. In your first execution, you may want to comment this out and see the user interface.

For movie rendering, this will come in handy (as we will see soon). As a library, PyMOL is quite tricky to use. For instance, after the import, you have to call `finish_launching`. We then fetch our PDB file.

What then follows is a set of PyMOL commands. Many web guides for interactive usage can be quite useful to understand what is going on. Here, we will enable all of the models for viewing purposes, hiding all (because the default view is lines and this is not good enough), then making zinCs visible as spheres.

At this stage, barring zinCs, everything else is invisible.

To render our model, we will use three scenes, as follows:

```
cmd.show('surface', 'chain A+B+C')
cmd.show('cartoon', 'chain E+F')
cmd.scene('S0', action='store', view=0, frame=0, animate=-1)
cmd.show('cartoon')
cmd.hide('surface')
cmd.scene('S1', action='store', view=0, frame=0, animate=-1)
cmd.hide('cartoon', 'chain A+B+C')
```

```
cmd.show('mesh', 'chain A')
cmd.show('sticks', 'chain A+B+C')
cmd.scene('S2', action='store', view=0, frame=0, animate=-1)
```

We need to define two scenes. One scene is for the time when we go around the protein (surface-based, thus opaque) and the other is for the time when we dive in (cartoon-based). The DNA is always rendered as a cartoon.

We also define a third scene for when we zoom out at the end. The protein will get rendered as sticks, and we add a mesh to chain A so that the relationship becomes clearer with the DNA.

Let's define the basic parameter of our video, as follows:

```
cmd.set('ray_trace_frames', 0)
cmd.mset(1, 500)
```

We define the default ray-tracing algorithm. This line does not need to be there, but try to increase the number to 1, 2, or 3 and be ready to wait a lot.

You can only use it if you have the OpenGL interface on (with the GUI), so, for this fast version, you will need to have the GUI on (pymol\_argv should be commented as it is).

We then inform PyMOL that we will have 500 frames.

In the first 150 frames, we move around using the initial scene. We go around the model a bit, and go near the DNA using the following code:

```
cmd.frame(0)
cmd.scene('S0')
cmd.mview()
cmd.frame(60)
cmd.set_view((-0.175534308, -0.331560850, -0.926960170,
              0.541812420, 0.753615797, -0.372158051,
              0.821965039, -0.567564785, 0.047358301,
              0.000000000, 0.000000000, -249.619018555,
              58.625568390, 15.602619171, 77.781631470,
              196.801528931, 302.436492920, -20.000000000))

cmd.mview()
cmd.frame(90)
cmd.set_view((-0.175534308, -0.331560850, -0.926960170,
              0.541812420, 0.753615797, -0.372158051,
              0.821965039, -0.567564785, 0.047358301,
              -0.000067875, 0.000017881, -249.615447998,
              54.029174805, 26.956727982, 77.124832153,
```

```
196.801528931, 302.436492920, -20.0000000000))
cmd.mview()
cmd.frame(150)
cmd.set_view((-0.175534308, -0.331560850, -0.926960170,
              0.541812420, 0.753615797, -0.372158051,
              0.821965039, -0.567564785, 0.047358301,
              -0.000067875, 0.000017881, -55.406421661,
              54.029174805, 26.956727982, 77.124832153,
              2.592475891, 108.227416992, -20.0000000000))
cmd.mview()
```

We define three points; the first two align with the DNA and the last goes in. We get coordinates (all of these numbers) by using PyMOL in interactive mode, navigating using the mouse and keyboard, and using the get\_view command, which will return coordinates that you can cut and paste.

The first frame is as follows:

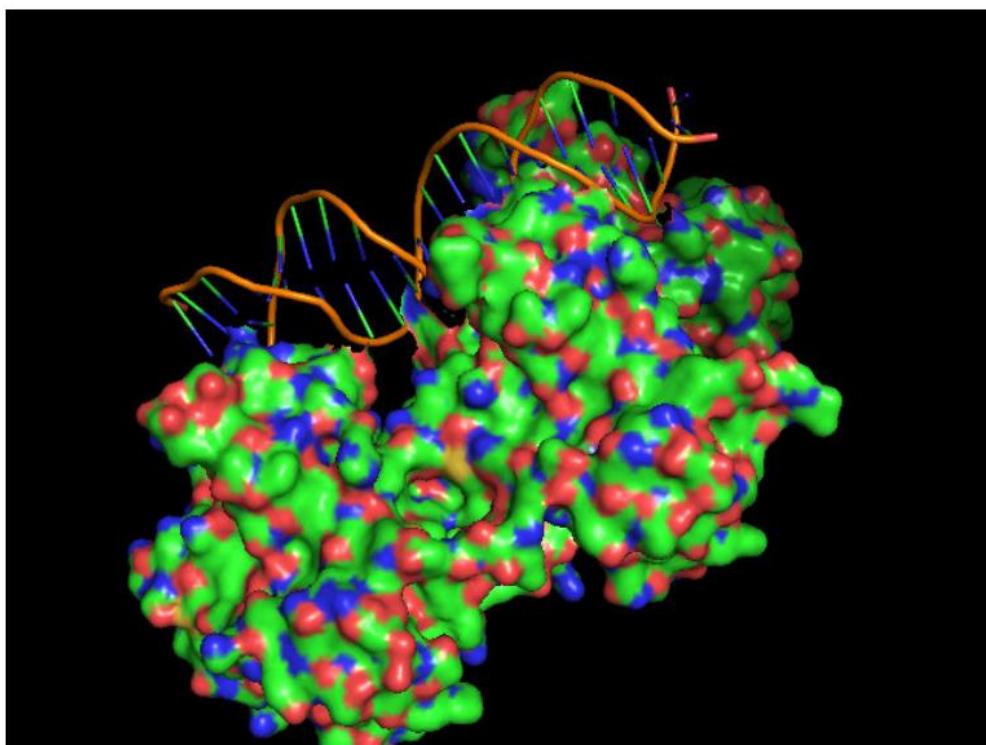


Figure 2: Frame 0 and scene S0

We now change the scene, in preparation to go inside the protein:

```
cmd.frame(200)
cmd.scene('S1')
cmd.mview()
```

The following screenshot shows the current position:

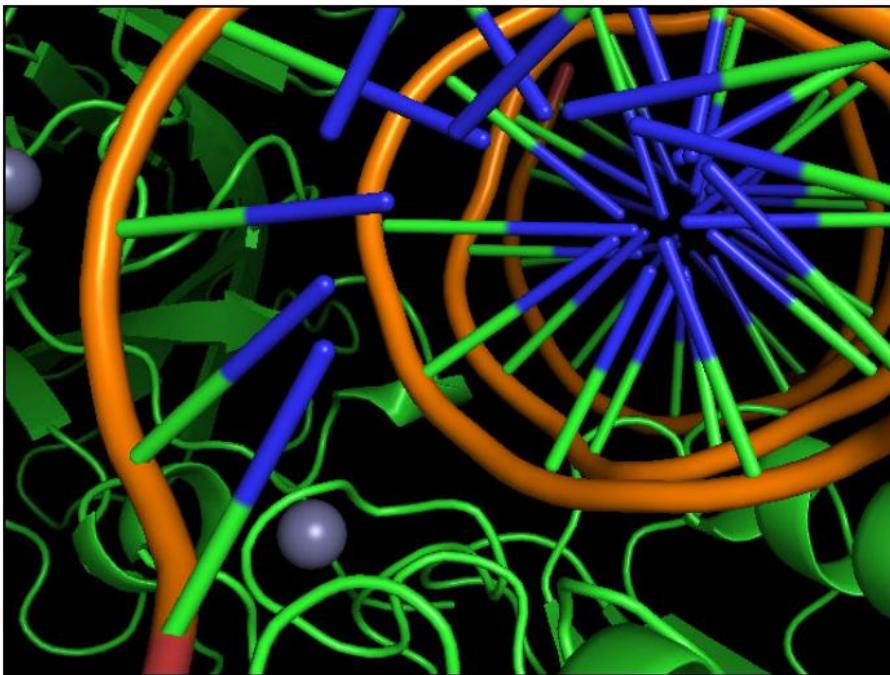


Figure 3: Frame 200 near the DNA molecule and scene S1

We move inside the protein and change the scene at the end using the following code:

```
cmd.frame(350)
cmd.scene('S1')
cmd.set_view((0.395763457, -0.173441306, 0.901825786,
              0.915456235, 0.152441502, -0.372427106,
              -0.072881661, 0.972972929, 0.219108686,
              0.000070953, 0.000013039, -37.689743042,
              57.748500824, 14.325904846, 77.241867065,
              -15.123448372, 90.511535645, -20.000000000))
```

```
cmd.mview()  
cmd.frame(351)  
cmd.scene('S2')  
cmd.mview()
```

We are now fully inside, as shown in the following screenshot:

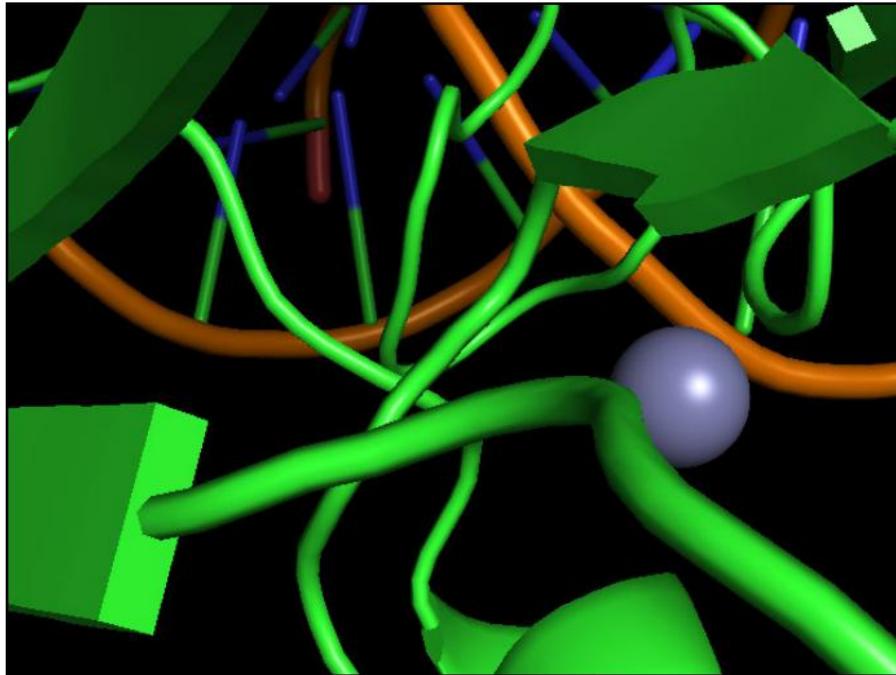


Figure 4: Frame 350, scene S1 on the verge of changing to S2

Finally,<sup>7</sup>we let PyMOL return to its original position, and then play, save, and quit:

```
cmd.frame(500)  
cmd.scene('S2')  
cmd.mview()  
cmd.mplay()  
cmd.mpng('p53_1tup')  
cmd.quit()
```

This will generate 500 PNG files with the p53\_1tup prefix.

Here is a frame approaching the end (450):

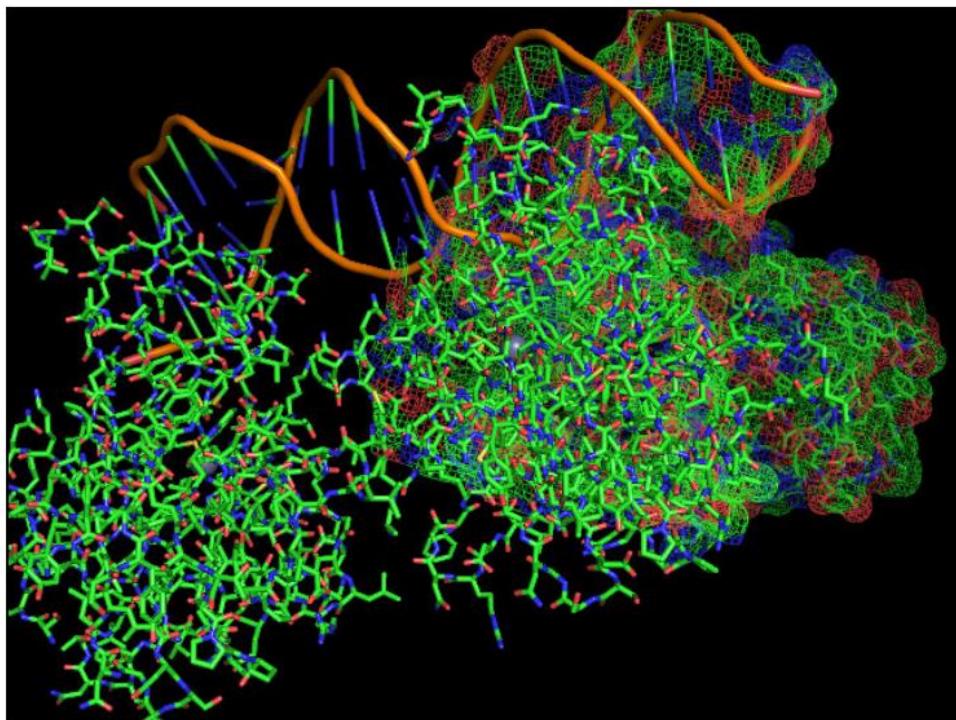


Figure 5: Frame 450 and scene S2

## There's more...

The YouTube video was generated using ffmpeg on Linux at 15 frames per second, as follows:

```
ffmpeg -r 15 -f image2 -start_number 1 -i "p53_1tup%04d.png" example.mp4
```

There are plenty of applications being used to generate videos from images. PyMOL can generate an MPEG, but this means we have to install extra libraries.

PyMOL was created to be used interactively from its console (which can be extended in Python). Using it the other way around (importing from Python with no GUI) can be complicated and frustrating; PyMOL starts a separate thread to render images that work asynchronously.

For example, this means that your code may be in a different position from where the renderer is. I have put another script called PyMol\_Intro.py on the GitHub repository; you will see that the second PNG call will start before the first one has finished. Try the script code and see how you expect it to behave, and how it actually behaves.

There is plenty of good documentation for PyMOL from a GUI perspective at <http://www.pymolwiki.org/index.php/MovieSchool>. This is a great starting point if you want to make movies, and <http://www.pymolwiki.org> is a treasure trove of information.

## Parsing mmCIF files using Biopython

The mmCIF file format is probably the future. Biopython doesn't have full functionality to work with it yet, but we will take a look at what is here now.

### Getting ready

As Bio.PDB is not able to automatically download mmCIF files, you need to get your protein file and rename it to 1tup.cif. This can be found at <https://github.com/PacktPublishing/Bioinformatics-with-Python-Cookbook-Second-Edition/blob/master/Datasets.ipynb> under the 1TUP.cif name.

You can find this content in the Chapter07/mmCIF.ipynb Notebook file.

### How to do it...

Take a look at the following steps:

Let's parse the file. We just use the MMCIF parser instead of the PDB parser:

```
from Bio import PDB
parser = PDB.MMCIFParser()
p53_1tup = parser.get_structure('P53', '1tup.cif')
```

Let's inspect the following chains:

```
def describe_model(name, pdb):
    print()
    for model in p53_1tup:
        for chain in model:
            print('%s - Chain: %s. Number of residues: %d. Number
of atoms: %d.' %
```

```
(name, chain.id, len(chain),
len(list(chain.get_atoms())))
describe_model('1TUP', p53_1tup)
```

The output will be as follows:

```
1TUP - Chain: E. Number of residues: 43. Number of atoms: 442.
1TUP - Chain: F. Number of residues: 35. Number of atoms: 449.
1TUP - Chain: A. Number of residues: 395. Number of atoms: 1734.
1TUP - Chain: B. Number of residues: 265. Number of atoms: 1593.
1TUP - Chain: C. Number of residues: 276. Number of atoms: 1610.
```

Many of the fields are not available on the parsed structure, but the fields can still be retrieved by using a lower-level dictionary, as follows:

```
mmcif_dict = PDB.MMCIF2Dict.MMCIF2Dict('1tup.cif')
for k, v in mmcif_dict.items():
    print(k, v)
print()
```

Unfortunately, this list is large and requires some post-processing to make some sense of it, but this is available.

## There's more...

You still have all the model information from the mmCIF file made available by Biopython, so the parser is still quite useful. You can expect more developments with the mmCIF parser than with the PDB parser.

There is a Python library for this that's been made available by the developers of PDB at <http://mmcif.wwpdb.org/docs/sw-examples/python/html/index.html>.

# 8

# Bioinformatics Pipelines

In this chapter, you will find recipes on the following:

- Introducing Galaxy servers
- Accessing Galaxy using the API
- Developing a Galaxy tool
- Using generic pipelines with bioinformatics data
- Deploying a variant analysis pipeline with Airflow

## Introduction

Pipelines are fundamental in any data science environment. Data processing is never a single task. Many pipelines are implemented via ad hoc scripts. This can be done in a useful way, but in many cases, they fail many fundamental viewpoints: reproducibility, maintainability, and extensibility.

In bioinformatics, you can find three main types of pipeline systems:

- Frameworks like Galaxy (<https://usegalaxy.org>), which are geared toward users, that is, they expose easy-to-use user interfaces, hiding most of the underlying machinery
- Frameworks like Script of Scripts (SoS) (<https://svatlab.github.io/sos-docs/>), which are geared toward data analysis, with a focus on with programming knowledge
- Finally generic workflow systems like Apache Airflow (<https://airflow.incubator.apache.org/>), which take a less data-centered approach to workflow management

In this chapter, we will discuss Galaxy, which is especially important for bioinformaticians who are supporting users that are less inclined to code their own solutions. We will also be discussing Apache Airflow as a generic workflow tool. We will not be focusing on a specific tool that is data-centric and programmer only, but if you look at Galaxy internals (which you will do in this chapter) away from the UI, you actually have a very good example of a great data-centric pipeline that can be programmed.

The code for these recipes is presented not as notebooks, but as Python scripts available in the Chapter08/pipelines directory of the book repository.

## Introducing Galaxy servers

Galaxy (<https://galaxyproject.org/tutorials/g101/>) is an open source system that empowers non-computational users to do computational biology. It is the most widely used, user-friendly pipeline system available. Galaxy can be installed on a server by any user, but there are also plenty of other servers on the web with public access, the flagship being <http://usegalaxy.org>.

Our focus in the following recipes will be the programming side of Galaxy: interfacing using the Galaxy API, and developing a Galaxy tool to extend its functionality. Before we start, you are strongly advised to approach Galaxy as a user. You can do this by creating a free account on <http://usegalaxy.org>, and playing around with it a bit. Getting a level of understanding that includes workflows is recommended.

## Getting ready

In this recipe, we will carry out a local installation of a Galaxy server using Docker. As such, a local Docker installation is required. This will vary in complexity across operating systems: easy on Linux, medium on macOS, and medium-to-hard on Windows.

This installation is recommended for the next two recipes, but you might be able to do them by using existing public servers. Note that public servers' interfaces might change over time, so what works today might not work tomorrow. Instructions on how to use public servers for the next two recipes are available in the There's more... section.

## How to do it...

Take a look at the following steps. These assume that you have a Docker-enabled command line:

First, we pull the Galaxy Docker image with the following command:

```
docker pull bgruening/galaxy-stable:18.05
```

This will pull Björn Grüning's amazing Docker Galaxy image. Use the 18.05 label, as shown in the preceding command; anything more recent might break this and the next recipe.

Create a directory on your system. This directory will hold the persistent output of the Docker container across runs.



Docker containers are transient with regard to disk space. This means that when you stop the container, all disk changes will be lost. This can be solved by mounting volumes from the host on Docker, as in the next step. All content in the mounted volumes will persist.

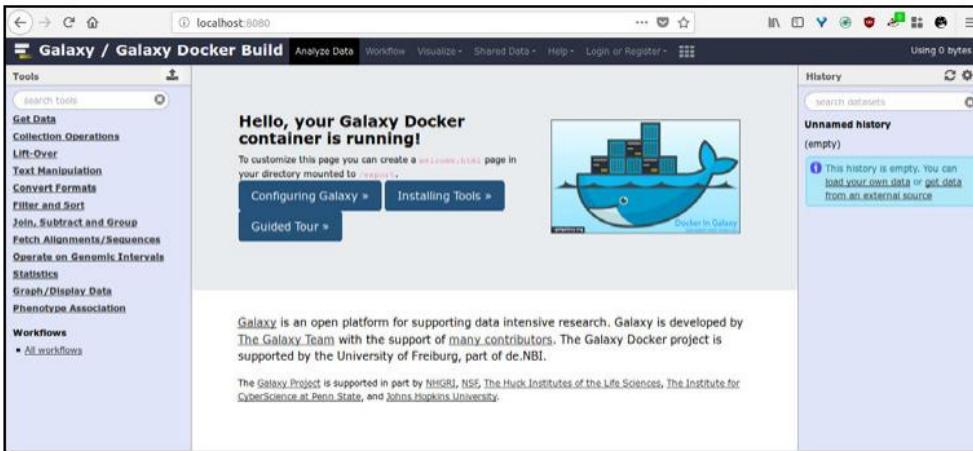
We can now run the image with the following command:

```
docker run -d -v YOUR_DIRECTORY:/export -p 8080:80 -p 8021:21  
bgruening/galaxy-stable:18.05
```

Replace YOUR\_DIRECTORY with the full path to the directory you created in Step 2. If the preceding command fails, make sure you have permission to run Docker. This will vary across operating systems.

Check the content of YOUR\_DIRECTORY. The first time the image runs, it will create all of the files needed for persistent execution across Docker runs. That means maintaining user databases, datasets, workflows, and so on.

Point your browser to <http://localhost:8080>. If you get some errors, wait a few seconds. You should see the following screenshot:



Now log in (see the top bar) with the default user/password combination: admin@galaxy.org/admin.

On the top menu, choose User, and inside, choose Preferences.

Now choose Manage API Key.

Do not change the API key. The purpose of the preceding exercise is for you to know where the API key is. In real scenarios, you will have to go to this screen to get your key. Just note the API key: admin. In normal situations, this will be an MD5 hash, by the way.

So, at this stage, we have our server installed with the following (default) credentials: User: admin@galaxy.org, password: admin, API key: admin. The access point is localhost:8080.

## There's more...

The way Björn Grüning's image is going to be used throughout this chapter is quite simple; after all, this is not a system administration or DevOps book, but a programming one. If you visit <https://github.com/bgruening/docker-galaxy-stable>, you will see that there is an infinite amount of ways to configure the image, and all are well-documented. Our simple approach here works for our development purposes.

If you don't want to install Galaxy on your local computer, you can use a public server such as <https://usegalaxy.org> to do the next recipe. It is not 100% foolproof, as services change over time, but it will probably be very close. Take the following steps:

- 1 Create an account on a public server (<https://usegalaxy.org> or other)
- 2 Follow the instructions above to access your API key
- 3 In the next recipe, you will have to replace the host, user, password, and API key

## Accessing Galaxy using the API

While Galaxy's main use case is via an easy-to-use web interface, it also provides a REST API for programmatic access. There are interfaces provided in several languages, for example, Python support is available from BioBlend (<https://bioblend.readthedocs.io>).

Here, we are going to develop a script that will load a BED file into Galaxy and call a tool to convert it to GFF format. We will load the file via Galaxy's FTP server.

## Getting ready

If you did not go through the previous recipe, please read its There's more... section. The code was tested in a local server, as prepared in the preceding recipe, but it might require some adaptations if you run it against a public server.

Our code will need to authenticate itself against the Galaxy server in order to perform the necessary operations. Because security is an important issue, this recipe will not be totally naive with regards to it. Our script will be configured via a YAML file, for example:

```
rest_protocol: http
server: localhost
rest_port: 8080
ftp_port: 8021
user: admin@galaxy.org
password: admin
api_key: admin
```

Our script will not accept this file as plaintext, but it will require it to be encrypted. That being said, there is a big hole in our security plan: we will be using HTTP (instead of HTTPS) and FTP (instead of SFTP), which means that passwords will pass in the clear over the network. Obviously this is a bad solution, but space considerations put a limit on what we can do (especially in the preceding recipe). Really secure solutions will require HTTPS and SFTP.

We will need a script that takes a YAML file and generates an encrypted version:

```
import base64
import getpass
from io import StringIO
import os

from ruamel.yaml import YAML

from cryptography.fernet import Fernet
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC

password = getpass.getpass('Please enter the password:').encode()

salt = os.urandom(16)
kdf = PBKDF2HMAC(algorithm=hashes.SHA256(), length=32, salt=salt,
                  iterations=100000, backend=default_backend())
key = base64.urlsafe_b64encode(kdf.derive(password))
fernet = Fernet(key)

with open('salt', 'wb') as w:
    w.write(salt)

yaml = YAML()

content = yaml.load(open('galaxy.yaml', 'rt', encoding='utf-8'))
```

```
print(type(content), content)
output = StringIO()
yaml.dump(content, output)
print ('Encrypting:\n%s' % output.getvalue())

enc_output = fernet.encrypt(output.getvalue().encode())

with open('galaxy.yaml.enc', 'wb') as w:
    w.write(enc_output)
```

The preceding file can be found on Chapter08/pipelines/galaxy/encrypt.py in the GitHub repository at <https://github.com/PacktPublishing/Bioinformatics-with-Python-Cookbook-Second-Edition>.

You will need to input a password for encryption.

The preceding code is not Galaxy-related: it reads a YAML file and encrypts it with a password supplied by the user. It uses the cryptography module encryption and ruamel.yaml for YAML processing. Two files are output: the encrypted YAML file and the salt file for encryption. For security reasons, the salt file should not be public.

This approach to securing credentials is far from being sophisticated; it is mostly illustrative that you have to be careful with your code when dealing with authentication tokens. There are far more instances on the web of hardcoded security credentials.

## How to do it...

Take a look at the following steps, which can be found on Chapter8/pipelines/galaxy/api.py:

We start by decrypting our configuration file. We need to supply a password:

```
import base64
from collections import defaultdict
import ftplib
import getpass
import pprint
import warnings

from ruamel.yaml import YAML

from cryptography.fernet import Fernet
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
```

```
import pandas as pd

from bioblend.galaxy import GalaxyInstance

pp = pprint.PrettyPrinter()
warnings.filterwarnings('ignore')
# explain above, and warn

with open('galaxy.yaml.enc', 'rb') as f:
    enc_conf = f.read()

password = getpass.getpass('Please enter the password:').encode()
with open('salt', 'rb') as f:
    salt = f.read()
kdf = PBKDF2HMAC(algorithm=hashes.SHA256(), length=32, salt=salt,
                  iterations=100000, backend=default_backend())
key = base64.urlsafe_b64encode(kdf.derive(password))
fernet = Fernet(key)
yaml = YAML()
conf = yaml.load(fernet.decrypt(enc_conf).decode())
```

The last line summarizes it all: the YAML module will load the configuration from a decrypted file. Note that we also read salt in order to be able to decrypt the file.

We'll now get all configuration variables, prepare the server URL, and specify the name of the Galaxy history that we will be creating (bioinf\_example):

```
server = conf['server']
rest_protocol = conf['rest_protocol']
rest_port = conf['rest_port']
user = conf['user']
password = conf['password']
ftp_port = int(conf['ftp_port'])
api_key = conf['api_key']

rest_url = '%s://%s:%d' % (rest_protocol, server, rest_port)
history_name = 'bioinf_example'
```

Finally, we are able to connect to the Galaxy server:

```
gi = GalaxyInstance(url=rest_url, key=api_key)
gi.verify = False
```

4We will now list all histories available:

```
histories = gi.histories
print('Existing histories:')
for history in histories.get_histories():
    if history['name'] == history_name:
        histories.delete_history(history['id'])
    print(' - ' + history['name'])
print()
```

On the first execution, you will get an unnamed history, but on the other executions you will also get bioinf\_example, which we will delete at this stage, so that we start with a clean slate.

Afterward, we create the history bioinf\_example:

```
ds_history = histories.create_history(history_name)
```

If you want, you can check on the web interface, and you will find the new history there.

We are going to upload the file now; this requires an FTP connection. The file is supplied with the code:

```
print('Uploading file')
ftp = ftplib.FTP()
ftp.connect(host=server, port=ftp_port)
ftp.login(user=user, passwd=password)
f = open('LCT.bed', 'rb')
ftp.storbinary('STOR LCT.bed', f)
f.close()
ftp.close()
```

We will now tell Galaxy to load the file on the FTP server into its internal database:

```
gi.tools.upload_from_ftp('LCT.bed', ds_history['id'])
```

Let's summarize the contents of our history:

```
def summarize_contents(contents):
    summary = defaultdict(list)
    for item in contents:
        summary['id'].append(item['id'])
        summary['hid'].append(item['hid'])
        summary['name'].append(item['name'])
        summary['type'].append(item['type'])
```

```
summary['extension'].append(item['extension'])
return pd.DataFrame.from_dict(summary)

print('History contents:')
pd_contents = summarize_contents(contents)
print(pd_contents)
print()
```

We only have one entry:

```
  id hid name type extension
0 f2db41e1fa331b3e 1 LCT.bed file auto
```

Let's inspect the metadata for our BED file:

```
print('Metadata for LCT.bed')
bed_ds = contents[0]
pp.pprint(bed_ds)
print()
```

The result is:

```
{'create_time': '2018-11-28T21:27:28.952118',
'dataset_id': 'f2db41e1fa331b3e',
'deleted': False,
'extension': 'auto',
'hid': 1,
'history_content_type': 'dataset',
'history_id': 'f2db41e1fa331b3e',
'id': 'f2db41e1fa331b3e',
'name': 'LCT.bed',
'purged': False,
'state': 'queued',
'tags': [],
'type': 'file',
'type_id': 'dataset-f2db41e1fa331b3e',
'update_time': '2018-11-28T21:27:29.149933',
'url':
'/api/histories/f2db41e1fa331b3e/contents/f2db41e1fa331b3e',
'veisible': True}
```

Let's turn our attention to the existing tools on the server and get metadata about them:

```
print('Metadata about all tools')
all_tools = gi.tools.get_tools()
pp pprint(all_tools)
print()
```

The this will print a long list of tools.

Now let's get some information about our tool:

```
bed2gff = gi.tools.get_tools(name='Convert BED to GFF')[0]
print("Converter metadata:")
pp pprint(gi.tools.show_tool(bed2gff['id'], io_details=True,
link_details=True))
print()
```

The tool name was available in the preceding step. Note that we get the first element of a list as, in theory, there can be more than one version of the tool installed. The abridged output is:

```
{'config_file': '/galaxy-central/lib/galaxy/datatypes/convertisers/bed_to_gff_converter.xml',
'id': 'CONVERTER_bed_to_gff_0',
'inputs': [{ 'argument': None,
    'edam': { 'edam_data': ['data_3002'],
        'edam_formats': ['format_3003']},
    'extensions': ['bed'],
    'label': 'Choose BED file',
    'multiple': False,
    'name': 'input1',
    'optional': False,
    'type': 'data',
    'value': None}],
'labels': [],
'link': '/tool_runner?tool_id=CONVERTER_bed_to_gff_0',
'min_width': -1,
'model_class': 'Tool',
'name': 'Convert BED to GFF',
'outputs': [{ 'edam_data': 'data_1255',
    'edam_format': 'format_2305',
    'format': 'gff',
    'hidden': False,
    'model_class': 'ToolOutput',
    'name': 'output1'}],
'panel_section_id': None,
'panel_section_name': None,
```

```
'target': 'galaxy_main',
'version': '2.0.0'}
```

Finally, let's run a tool to convert our BED file into GFF:

```
def dataset_to_param(dataset):
    return dict(src='hda', id=dataset['id'])

tool_inputs = {
    'input1': dataset_to_param(bed_ds)
}

gi.tools.run_tool(ds_history['id'], bed2gff['id'],
    tool_inputs=tool_inputs)
```

The parameters of the tool can be inspected in the preceding step. If you go to the web interface, you will see something like this:

Seqname	Source	Feature	Start	End	Score	Strand	Frame	Group
##bed_to_gff_converter.py								
2	bed2gff	ENSE00002202258	135836530	135837180	0	-	.	ENSE00002202258
2	bed2gff	ENSE00001660765	135833111	135833190	0	-	.	ENSE00001660765
2	bed2gff	ENSE00001731451	135829593	135829676	0	-	.	ENSE00001731451
2	bed2gff	ENSE00001659962	135823901	135824003	0	-	.	ENSE00001659962
2	bed2gff	ENSE0000177620	135822020	135822098	0	-	.	ENSE0000177620
2	bed2gff	ENSE00001602826	135817341	135818061	0	-	.	ENSE00001602826
2	bed2gff	ENSE00000776576	135812311	135812366	0	-	.	ENSE00000776576
2	bed2gff	ENSE00001008768	135808443	135809993	0	-	.	ENSE00001008768
2	bed2gff	ENSE00000776573	135807128	135807396	0	-	.	ENSE00000776573
2	bed2gff	ENSE00000776572	135804767	135805057	0	-	.	ENSE00000776572
2	bed2gff	ENSE00000776571	135803930	135804128	0	-	.	ENSE00000776571
2	bed2gff	ENSE00000776570	135800907	135800909	0	-	.	ENSE00000776570
2	bed2gff	ENSE00003515081	135798029	135798138	0	-	.	ENSE00003515081
2	bed2gff	ENSE00001630333	135794641	135794775	0	-	.	ENSE00001630333
2	bed2gff	ENSE00001667885	135790658	135790881	0	-	.	ENSE00001667885
2	bed2gff	ENSE00001728878	135789571	135789798	0	-	.	ENSE00001728878
2	bed2gff	ENSE00001653704	135787840	135788544	0	-	.	ENSE00001653704
2	bed2gff	ENSE00001745158	135812311	135812989	0	-	.	ENSE00001745158
2	bed2gff	ENSE00001608768	135808443	135809993	0	-	.	ENSE00001608768
2	bed2gff	ENSE00000776573	135807128	135807396	0	-	.	ENSE00000776573
2	bed2gff	ENSE00000776572	135804767	135805057	0	-	.	ENSE00000776572
2	bed2gff	ENSE00000776571	135803930	135804128	0	-	.	ENSE00000776571

Checking the results of our script via Galaxy's web interface

# Developing a Galaxy tool

Galaxy is mostly a very advanced workflow engine with an easy-to-use interface. Concrete work is delegated to tools. Here, we will learn how to develop a basic tool. The fundamental objective of this chapter is to demonstrate how easy it is to develop a tool; it might seem daunting, but really it is not. The Galaxy developers made sure that all the boilerplate work is done by a tool called Planemo (<https://planemo.readthedocs.io>). Planemo is very well-documented, and this recipe should be seen as a starting point; Planemo's documentation is the end point. Again, our main objective here is to gain confidence that tool development is accessible for all who want to do it.

## Getting ready

Strictly speaking, tool development is language-agnostic. Many tools are actually just wrappers for existing applications; no Python code involved. But Python is well-suited to tool development, especially because Galaxy is implemented in Python, so it's the best choice for tool development. Planemo is actually a Python tool, so you will need Python in any case.

We are going to develop a very basic tool here: we will take a BED file, count the number of entries, and compute some basic statistics. The point here is not the tool itself, but all the artifacts needed to develop the tool.

At this stage, installing Planemo with conda is not stable. Also, Planemo is Python 2, not 3. As such, we will go back in time and use Python 2, and virtualenv for the installation,. Let's create a virtual environment on the command line:

```
virtualenv .pln  
. .pln/bin/activate
```

You should now be inside the virtual environment .pln. Now perform the following command:

```
pip install planemo
```

After a bit of time, all should be installed. If some dependencies fail, check that you have all the necessary development tools for your environment.

A good part of our work will be to create an XML file describing the tool. A final version of this file is provided in Chapter08/pipelines/galaxy/tool/bioinf\_text.xml.final.

## How to do it...

Take a look at the following steps:

We start by asking planemo to create a scaffold configuration for our tool:

```
planemo tool_init --id bioinf_test --name "Basic BED analyzer"
```

This will generate a file called `bioinf_text.xml`. Please inspect it, as most of the next steps will require changing it.

Start by linting the file:

```
planemo lint # You can also do only planemo l
```

This is the Planemo linting tool, and you will see that there are some problems with the file.

Let's add some help text. Replace the `<help>...</help>` section with:

```
<help><![CDATA[  
A Basic BED analyzer.  
  
Counts entries and optionally reports basic stats  
]]></help>
```

Lint again, just to make sure no more errors were introduced. It still fails, but no new error should appear.

Let's add some inputs and outputs, replace the `<inputs>`, and `<outputs>` sections with:

```
<inputs>  
  <param type="boolean" name="stats"  
        truevalue="--stats=yes" falsevalue="--stats=no"  
        label="Compute statistics?"/>  
  <param type="data" name="input1" format="bed"  
        label="BED file"/>  
</inputs>  
<outputs>  
  <data name="output1" format="txt" />  
</outputs>
```

We have one input parameter, a boolean, which specifies whether we want extra statistics or not (this will become clear in the Python code next). We have one input file of type bed and one output file of type txt.

Let's add the command to be executed:

```
<command detect_errors="exit_code"><![CDATA[  
    python ${_tool_directory}_/bed_analyzer.py '$stats' '$input1'  
    '$output1';  
]]></command>
```

The Python script will be defined in a later step. For now, the important point is `$_tool_directory_`, as it specifies the directory where we can find our script. Lint again to be on the safe side.

Let's add some citation, as that is inspected by lint. Add the following after `</help>`:

```
<citations>  
    <citation type="bibtex">  
        @book{antao2019bioinformatics,  
            title={Bioinformatics with Python cookbook, Second edition},  
            author={Antao, Tiago},  
            year={2015},  
            publisher={Packt Publishing Ltd}  
        }  
    </citation>  
</citations>
```

7. On the repository Chapter08/pipelines/galaxy/tool/test-data directory, you will find some files. These are to be used for a couple of test cases. The XML code for them is:

```
<tests>  
    <test>  
        <param name="stats" value="true" />  
        <param name="input1" value="LCT.bed" ftype="bed"/>  
        <output name="output1" file="output2.txt" ftype="txt"/>  
    </test>  
    <test>  
        <param name="stats" value="false" />  
        <param name="input1" value="LCT.bed" ftype="bed"/>  
        <output name="output1" file="output1.txt" ftype="txt"/>  
    </test>  
</tests>
```

The input file is the same, but in one case we compute statistics, while in the other we do not. As usual, do not forget to lint as you progress.

Finally, this is the Python (2) code to process the file. The code is not the most important part of the recipe; it's just an example to help you understand the rest of the infrastructure:

```
from __future__ import print_function
import sys

stats = sys.argv[1] == '--stats=yes'
input_file = sys.argv[2]
output_file = sys.argv[3]

num_recs = 0
positions = set()
with open(input_file) as f:
    f.readline()
    for l in f:
        num_recs += 1
        toks = l.rstrip().split('\t')
        positions.add(int(toks[1]))
        positions.add(int(toks[2]))

with open(output_file, 'w') as w:
    w.write('Records: %d\n' % num_recs)
    if stats:
        w.write('Minimum position: %d\n' % min(positions))
        w.write('Maximum position: %d\n' % max(positions))
```

Finally, let's do the following:

```
planemo test # or planemo t
```

The first execution will take a long time, as a complete environment will have to be downloaded.

## There's more...

Hopefully, at this stage, you have seen that tool development is nothing to be afraid of. Check the Planemo documentation for more complex examples. If you end up developing a new tool, consider submitting it to the Galaxy toolshed so that the whole community can benefit from your work.

# Using generic pipelines with bioinformatics data

Galaxy is mostly geared toward users who are less inclined to program. Knowing how to deal with it, even if you prefer a more programmer-friendly environment, is important because of its pervasiveness. It is reassuring that an API exists to interact with Galaxy. But if you want a more programmer-friendly pipeline, there are many alternatives available.

Here, we will explore Airflow, originally from Airbnb, and currently incubating under the Apache umbrella. Airflow is somewhat at the other end of the pipeline world: it is completely subject-agnostic (actually, its development has nothing to do with bioinformatics), and it is completely geared toward programming.

## Getting ready

Be careful with the sources for the installation of Airflow, as some might not be up to date. At this stage, this applies to some conda packages that are available.

At the time of writing, the best approach is probably to use a standard Python installation that either comes with your OS or from <http://www.python.org>. Anaconda Python is not recommended at this stage.

That being said, with time, things might change. Please check Airflow's documentation. Only Step 1 in the following section is dependent on the assumption that you are using "old-fashioned" tools for installation.

More detailed installation instructions can be found in Airflow's documentation.

## How to do it...

Take a look at the following steps:

Create a virtual environment and install Airflow along with matplotlib (which we will use in the next recipe):

```
virtualenv .aflow
. .aflow/bin/activate
export AIRFLOW_GPL_UNIDECODE=yes
pip install apache-airflow
pip install matplotlib
```

With virtualenv, you might want to specify the Python interpreter with the `-p` parameter, like this: `virtualenv -p /usr/bin/python3 .aflow`. Do not forget to activate the environment before starting Airflow.

Let's now initialize Airflow's database:

```
airflow initdb
```

This will initialize an SQLite database. This is only useful for very basic testing purposes: with an SQLite database, no concurrency will be possible, defeating a lot of the purpose of using a workflow system. In production systems, you should use another database. But not only in production systems; a sequential system can have some different semantics from a concurrent one with Airflow, as we will see in the next recipe.

All initialization will take place on `~/airflow`, have a look inside the directory.

We now start Airflow's web interface and scheduler:

```
airflow webserver -p 8080  
airflow schedule
```

Let's create two connections for our next recipe, one with FTP to NCBI and another that's a local filesystem connection. Fire up a browser to `http://localhost:8080`. On the Admin menu, choose Connections. There, click Create, and for NCBI, use the following parameters:

```
Conn Id: ftp_ncbi  
Conn Type: FTP  
Host: ftp.ncbi.nlm.nih.gov  
Login: anonymous  
Password: your@email
```

For the file connection, use:

```
Conn Id: fs_bioinf  
Conn Type: File (path)  
Extra: {"path": "/PATH_TO_WHERE_YOU_PUT_THE_FILES"}
```

Replace `PATH_TO_WHERE_YOU_PUT_THE_FILES` with a reference to an empty directory with some disk space.

Your connection list should look something like this:

	Conn Id	Conn Type	Host	Port	Is Encrypted	Is Extra Encrypted
1	airflow_ci	mysql	localhost		●	○
2	airflow_db	mysql	localhost		●	●
3	aws_default	aws			●	○
4	azure_data_lake_default	azure_data_lake			●	○
5	beeline_default	beeline	localhost	10000	●	○
6	bigrquery_default	google_cloud_platform			●	●
7	cassandra_default	cassandra	localhost	9042	●	●
8	databricks_default	databricks	localhost		●	●
9	druid_broker_default	druid	druid-broker	8082	●	○
10	druid_ingest_default	druid	druid-overlord	8081	●	○
11	emr_default	emr			●	○
12	fs_biolinf	fs			●	○
13	fs_default	fs			●	○
14	ftp_ncbi		ftp.ncbi.nlm.nih.gov		○	●

Finally, we create a directory in which to put our workflow code:

```
mkdir ~/airflow/dags
```

## Deploying a variant analysis pipeline with Airflow

Here, we will develop a mini variant analysis pipeline with Airflow. The objective here is not to get the scientific part right—we cover that in other chapters—but to see how to create components with Airflow. Our mini-pipeline will download HapMap data, subsample at 1% and 10%, do a simple PCA, and draw it.

## Getting ready

You will need PLINK installed. Remember that we are not using a conda environment, so you have to make sure it is available for Airflow. We will define the following tasks:

- 1 Downloading data
- 2 Uncompressing it
- 3 Sub-sampling at 10%
- 4 Sub-sampling at 1%
- 5 Computing PCA on the 1% sub-sample
- 6 Charting the PCA

Our pipeline recipe will have two parts: the actual coding of the pipeline and making the pipeline actually execute.

The code for this can be found on Chapter08/pipelines/airflow/create\_tasks.py.

## How to do it...

With Airflow, a pipeline specification is written in Python, which is quite convenient for us! Let's start with the import part:

```
from datetime import datetime, timedelta
import os
from os.path import isfile

from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from airflow.contrib.hooks.ftp_hook import FTPHook
from airflow.contrib.hooks.fs_hook import FSHook
from airflow.contrib.sensors.file_sensor import FileSensor
```

Our main object will be a Directed Acyclic Graph (DAG), which will include all the tasks and their dependencies. For now, we will specify a dictionary with some properties:

```
dag_args = {
    'owner': 'airflow',
    'description': 'Bioinformatics with Python Cookbook pipeline',
    'depends_on_past': False,
    'start_date': datetime(2016, 1, 18),
    'email': ['your@email.here'],
    'email_on_failure': False,
```

```
'email_on_retry': False,
'retries': 1,
'retry_delay': timedelta(minutes=1)
}
dag = DAG('bioinf', default_args=dag_args, schedule_interval=None)
```

Many of these properties are related to backfills, which we won't be using here.

Now let's specify the download part:

```
ftp_directory = '/hapmap/genotypes/hapmap3/plink_format/draft_2/'
ftp_files = {
    'hapmap3_r2_b36_fwd.consensus.qc.poly.map.bz2':
    'hapmap.map.bz2',
    'hapmap3_r2_b36_fwd.consensus.qc.poly.ped.bz2':
    'hapmap.ped.bz2',
    'relationships_w_pops_121708.txt': 'relationships.txt',
}

def download_files(ds, **kwargs):
    fs = FSHook('fs_bioinf')
    force = kwargs['params'].get('force', 'false') == 'true'
    with FTPHook('ftp_ncbi') as ftp:
        for ftp_name, local_name in ftp_files.items():
            local_path = fs.get_path() + '/' + local_name
            uncompressed_local_path = local_path[:-4]
            if (isfile(local_path) or
                isfile(uncompressed_local_path)) and not force:
                continue
            if not isfile(local_name):
                ftp.retrieve_file(ftp_directory + ftp_name,
                                  local_path)
            open(fs.get_path() + '/done.txt', 'wb')
    return True
```

We have three files to download. We will use both connectors here: the FTP connector to retrieve the files, and the FS connector to store them. We will also touch a file called done.txt.

While it's not strictly needed, we will create FileSensor (just as a pedagogical example). This sensor will be the trigger for the uncompression:

```
s_fs_sensor = FileSensor(  
    task_id='download_sensor',  
    fs_conn_id='fs_bioinf',  
    filepath='done.txt',  
    dag=dag)
```

The sensor will be looking for done.txt.

Next, let's uncompress the files:

```
def uncompress_files(ds, **kwargs):  
    fs = FSHook('fs_bioinf')  
    if isfile(fs.get_path() + '/hapmap.ped'):   
        return True  
    os.system('bzip2 -d  
{fs_path}/hapmap.map.bz2'.format(fs_path=fs.get_path()))  
    os.system('bzip2 -d  
{fs_path}/hapmap.ped.bz2'.format(fs_path=fs.get_path()))  
    return True
```

The code for sub-sampling will be:

```
def subsample_10p(ds, **kwargs):  
    fs = FSHook('fs_bioinf')  
    os.system('/home/tra/anaconda3/bin/plink --recode --file  
{fs_path}/hapmap --noweb --out {fs_path}/hapmap10 --thin 0.1 --geno  
0.1'.format(fs_path=fs.get_path()))  
    return True  
  
def subsample_1p(ds, **kwargs):  
    fs = FSHook('fs_bioinf')  
    os.system('/home/tra/anaconda3/bin/plink --recode --file  
{fs_path}/hapmap --noweb --out {fs_path}/hapmap1 --thin 0.01 --geno  
0.1'.format(fs_path=fs.get_path()))  
    return True
```

We will compute PCA with plink. This is intended to be an example computation. We did not LD prune, or even remove the sexual chromosomes:

```
def compute_pca(ds, **kwargs):
    fs = FSHook('fs_bioinf')
    os.system('/home/tra/anaconda3/bin/plink --pca --file
{fs_path}/hapmap1 -out
{fs_path}/pca'.format(fs_path=fs.get_path()))
    return True
```

Finally, the code to produce a rough chart is as follows:

```
def plot_pca(ds, **kwargs):
    import matplotlib
    matplotlib.use('svg')
    import pandas as pd
    fs = FSHook('fs_bioinf')
    pca_df = pd.read_csv(fs.get_path() + '/pca.eigenvec', sep=' ',
    header=None)
    ax = pca_df.plot.scatter(x=2, y=3)
    ax.figure.savefig(fs.get_path() + '/pca.png')
```

We specify a backend that is least likely to cause problems with Airflow.

Now, we will inject this code as tasks in our DAG. Firstly, the data download:

```
t_download_files = PythonOperator(
    task_id='download_files',
    provide_context=True,
    python_callable=download_files,
    params={'force': 'Force download (boolean)'},
    dag=dag)
```

Now file uncompression and its dependency on the file sensor:

```
t_uncompress_files = PythonOperator(
    task_id='uncompress_files',
    provide_context=True,
    python_callable=uncompress_files,
    dag=dag)
t_uncompress_files.set_upstream(s_fs_sensor)
```

Next! **Sub-sampling**. Both operators depend on uncompression:

```
t_subsample_10p = PythonOperator(  
    task_id='subsample_10p',  
    provide_context=True,  
    python_callable=subsample_10p,  
    dag=dag)  
t_subsample_10p.set_upstream(t_uncompress_files)  
  
t_subsample_1p = PythonOperator(  
    task_id='subsample_1p',  
    provide_context=True,  
    python_callable=subsample_1p,  
    dag=dag)  
t_subsample_1p.set_upstream(t_uncompress_files)
```

**PCA1Computation** comes next and depends on 1% sub-sampling:

```
t_compute_pca = PythonOperator(  
    task_id='compute_pca',  
    provide_context=True,  
    python_callable=compute_pca,  
    dag=dag)  
t_compute_pca.set_upstream(t_subsample_1p)
```

And finally, drawing the PCA chart:

```
t_plot_pca = PythonOperator(  
    task_id='plot_pca',  
    provide_context=True,  
    python_callable=plot_pca,  
    dag=dag)  
t_plot_pca.set_upstream(t_compute_pca)
```

We'll also introduce a dependency on the sensor based on the download. This is only needed with the sequential executor to avoid a race:

```
s_fs_sensor.set_upstream(t_download_files)
```

15We can now copy the preceding `create_tasks.py` file to `~/airflow/dags`.

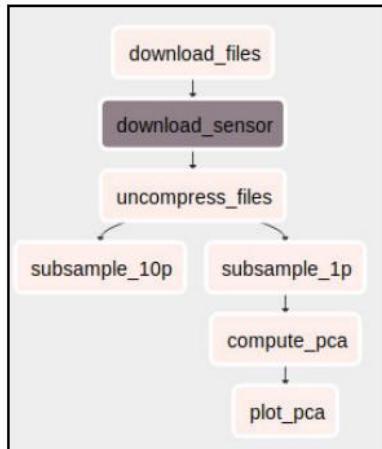
Let's **inject** it into airflow (from this point onward, the code is to be executed on the shell):

```
python ~/airflow/dags/create_tasks.py
```

Let's get airflow to do some sanity checking for us:

```
airflow list_dags  
airflow list_tags bioinf --tree
```

Now go to the web interface at <http://localhost:8080>, click on bioinf and you will get a DAG representation like the following:



A representation of our pipeline

Let's run a single task, the download:

```
airflow test bioinf download_files now
```

After the download has ended, check on the FS directory to see if the files are there.

Tasks can actually have parameters, and in our case we coded the possibility to force the download even if the files are there:

```
airflow test -tp '{ "force": "true"}' bioinf download_files now
```

This will redownload the files, even if they are there.

Remove the files from the FS directory, in preparation for running the whole pipeline.

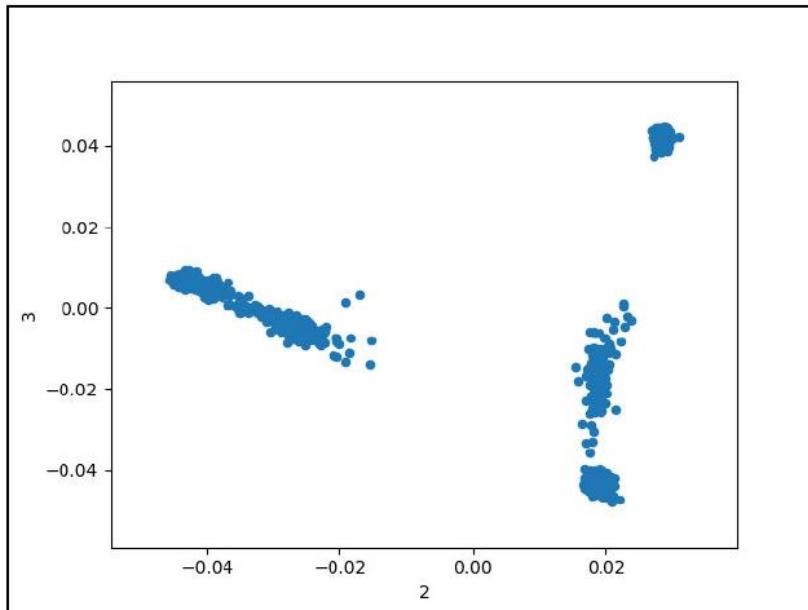
Most probably, you will need to unpause the DAG:

```
airflow unpause bioinf
```

You can now trigger the DAG:

```
airflow trigger_dag -e now bioinf
```

You can follow the progress of the pipeline via the web interface or the CLI, and when it is done you should find a PCA PNG on the FS directory. It is a rough PCA, but should be similar to the one in Chapter 4, Population Genetics (save for symmetry issues, which are OK with PCA). My rough version was this:



A very rough PCA produced by the Airflow pipeline

## There's more...

The preceding recipe was made to run on a simple configuration of Airflow, with a default SQLite database, which implies sequential execution (only one task at a time). The explicit dependency of the FS sensor on the download is only there to avoid a race condition with the basic executor. Furthermore, with other executors (that is, supported by another backend database), the sub-samplings could occur in parallel.

# 9

# Python for Big Genomics Datasets

In this chapter, we will cover the following recipes:

- Using high-performance data formats – HDF5
- Doing parallel computing with Dask
- Using high-performance data formats – Parquet
- Computing sequencing statistics using Spark
- Optimizing code with Cython and Numba

## Introduction

In this chapter, we will discuss high-performance computing techniques for large computational biology datasets. We will talk about efficient data storage, code parallelization, running software in clusters, and code optimization. We will try to avoid any solution to a specific proprietary technology (for example, Amazon EC2) and will instead design code that will be applicable in a wide range of scenarios.

The previous edition of this book had some recipes that compared lazy and eager data structures. This made sense, as Python 2 was mostly eager and Python 3 is mostly lazy. As Python 2 is behind us, that content has been dropped. That being said, make sure that you understand the difference and that your code is mostly lazy. Be sure to check generators in Python. Use them.

As the sizes of the datasets is constantly increasing, in this edition, we cannot evade discussing the efficient storage of bioinformatics data, and so we will discuss the Hierarchical Data Format (HDF5) and Apache Parquet. Parquet is almost required if you plan on using Spark for the processing of large datasets.

Many of the topics in the following recipes deserve a whole book in their own right. The objective here is not to be exhaustive, but to give you a taste of the possibilities available. You are strongly encouraged to research further if you find any of the following topics interesting.

## Using high-performance data formats – HDF5

VCF processing is very slow: if you do an empty for loop over a big VCF file, it can easily take days just to parse it. This is because text parsing is very demanding. Alternatively, using NumPy arrays is fast, but you are limited to whatever fits in memory. There are several alternatives to deal with both of these problems (and we will explore more than one in this chapter). Here, we will consider representing our data in HDF5 format.

We will use an existing HDF5 file that was exported from a VCF file and do some basic extraction of data.

### Getting ready

We will revisit the *Anopheles gambiae* dataset that we used in previous chapters. There is a HDF5 version of the VCF file that we used previously. You can download chromosome arm 3L from <ftp://ngs.sanger.ac.uk/production/ag1000g/phase1/ar3/variation/main/hdf5/ag1000g.phase1.ar3.pass.3L.h5>. Remember that we are dealing with a VCF representation of 765 mosquitoes that can be carriers of *Plasmodium falciparum*, the parasite responsible for malaria.

The file is 19 GB in size, so I recommend installing a tool such as HDF Compass at (<https://support.hdfgroup.org/projects/compass/>, available on Debian/Ubuntu Linux with `apt-get install hdf-compass`) to graphically inspect the file before proceeding. HDF5 is mostly a key-value store, where keys are represented hierarchically.

You can also use command-line tools to inspect the file. For example, we will use the HDF5 Tools (`apt-get install hdf5-tools`) to browse the content of the file:

```
h5ls -r ag1000g.phase1.ar3.pass.3L.h5
```

Here is an abridged version of the output:

```

/
/3L
/3L/calldata
/3L/calldata/DP
/3L/calldata/GQ
/3L/calldata/genotype
/3L/calldata/is_called
/3L/samples
/3L/variants
[...]
/3L/variants/DP
/3L/variants/MQ
/3L/variants/POS
/3L/variants/is_snp

```

Group  
Group  
Group  
Dataset {9643193/Inf, 765}  
Dataset {9643193/Inf, 765}  
Dataset {9643193/Inf, 765, 2}  
Dataset {9643193/Inf, 765}  
Dataset {765}  
Group  
Dataset {9643193/Inf}  
Dataset {9643193/Inf}  
Dataset {9643193/Inf}  
Dataset {9643193/Inf}

We can see the root group, which contains only another group inside, /3L (remember that we are inspecting chromosome arm 3L). Inside /3L, we have a dataset called samples with 765 entries, and two more groups, calldata and variants. These groups correspond to the VCF equivalents. For example, we have 9,643,193 entries for all annotations – one for each entry in the VCF file. Also, the genotype data is quite big; 9,643,193 entries times 765 samples times 2 (diploid data).

In our example, we will split the chromosome arm in windows of 50,000 bp and extract some basic statistics, such as number of polymorphic sites, missingness, and number of non-biallelic loci.

## How to do it...

Take a look at the following steps:

We will start by importing the necessary libraries. We will be using h5py to read the file. We will then access the data using the keys:

```

from math import ceil
import numpy as np
import h5py

h5_3L = h5py.File('ag1000g.phase1.ar3.pass.3L.h5', 'r')
samples = h5_3L['/3L/samples']
calldata_genotype = h5_3L['/3L/calldata/genotype']
positions = h5_3L['/3L/variants/POS']

```

```
alt_alleles = h5_3L['/3L/variants/ALT']
is_snp = h5_3L['/3L/variants/is_snp']
num_samples = len(samples)
```

There are alternatives to h5py, but be careful as they might impose constraints on keys and data (for instance, the read methods of pandas might do this). While we are referring to the objects, they are not being loaded in memory.

Now, let's get some information about the datasets, as follows:

```
print(calldata_genotype)
print(samples)
```

You will get some output about size and type:

```
<HDF5 dataset "genotype": shape (9643193, 765, 2), type "|i1">
<HDF5 dataset "samples": shape (765,), type "|S8">
```

Now, we have some initialization code to gather statistics on our windows. There should be nothing new here:

```
window_size = 50000
last_position = positions[-1]
num_windows = ceil(last_position / window_size)
limits = np.full((num_windows, 2), -1)
curr_window = positions[0] // window_size
limits[curr_window, 0] = 0
```

We will now do a traverse of positions to find out the boundaries of each window (that is, how many SNPs there will be per window):

```
for index, position in enumerate(positions):
    my_window = position // window_size
    if index % 1000000 == 0:
        print(index, position)
    if my_window != curr_window:
        limits[my_window, 0] = index
        limits[curr_window, 1] = index - 1
        curr_window = my_window
limits[num_windows - 1, 1] = len(positions)
```

Let's do some sanity checking:

```
print(limits[0], limits[-1])
print(positions[-1] // window_size, num_windows)
```

The output is as follows:

```
[ 0 43] [9642606 9643193]
839 840
```

The first window includes loci 0 to 43 and the last window includes loci 9642606 to 9643193. The last position falls in the window indexed as 839. There are 840 windows.

This is the function that's used for compute statistics over a single window. There is nothing HDF5-specific here:

```
def calc_statistics(v):
    start, end = v[0], v[1]
    min_maf = 0.5
    max_maf = 0.0
    non_bi = 0
    missing = 0
    non_snp = 0
    num_samples = len(samples)
    print(v)
    for pos in range(start, end + 1):
        if not is_snp[pos]:
            non_snp += 1
            continue
        if np.max(calldata_genotype[pos]) > 1:
            non_bi += 1
            continue
        if np.min(calldata_genotype[pos]) < 0:
            missing += 1
            continue
        num_alt = np.sum(calldata_genotype[pos]) # Because they
are coded as 1
        num_ref = num_samples * 2 - num_alt # (because all are
called)
        min_called = min(num_ref, num_alt)
        maf = min_called / (2 * num_samples)
        if maf < min_maf:
            min_maf = maf
        if maf > max_maf:
            max_maf = maf
    return {'total': end - start + 1, 'missing': missing,
            'non_snp': non_snp, 'non_bi': non_bi,
            'min_maf': min_maf, 'max_maf': max_maf}
```

We return a dictionary with our observations.

Now we will create a vectorized version of the function:

```
calc_statistics_v = np.vectorize(calc_statistics,  
signature='(m)->()')
```

This creates a version of the function that will run on individual elements of the array. So, if you pass 10 elements of the array, this will run 10 times and collect the results.

Let's do exactly that, and call the vectorized version on the first 10 elements of the array:

```
stats = calc_statistics_v(limits[:10])
```

You can run this on all the elements of the array, but it will still take a lot of time. We will revisit this in the following recipes.

Let's print our collected data, using pandas for better-looking visualization:

```
import pandas as pd  
stats_to_df = {  
    'total': [],  
    'missing': [],  
    'non_snp': [],  
    'non_bi': [],  
    'min_maf': [],  
    'max_maf': []  
}  
for stat in stats:  
    for k, v in stat.items():  
        stats_to_df[k].append(v)  
pd.DataFrame(stats_to_df)
```

An abridged version of the output is as follows:

	<code>total</code>	<code>missing</code>	<code>non.snp</code>	<code>non.bi</code>	<code>min.maf</code>	<code>max.maf</code>
<b>0</b>	44	3	0	4	0.000654	0.019608
<b>1</b>	922	153	0	32	0.000654	0.486928
<b>2</b>	947	178	0	37	0.000654	0.451634
<b>3</b>	1508	77	0	49	0.000654	0.394771
<b>4</b>	16	0	0	2	0.000654	0.323529
<b>5</b>	367	8	0	10	0.000000	0.450327
<b>6</b>	1235	8	0	54	0.000000	0.395425
<b>7</b>	1570	32	0	68	0.000654	0.393464
<b>8</b>	193	0	0	4	0.000654	0.254902
<b>9</b>	255	39	0	16	0.000654	0.411765

## There's more...

In the next recipe, we will discuss how to accelerate processing with parallel processing. Parallel processing of HDF5 is a complex topic, especially parallel writing, which is mostly impossible. Check out <https://www.stackoverflow.com/questions/41367568/dask-and-parallel-hdf5-writing> for more information.

To create HDF5 from VCF files, check Alistair Miles' blog post (<https://alimanfoo.github.io/2017/06/14/read-vcf.html>) and his library, scikit-allel (<https://scikit-allel.readthedocs.io/en/latest/>). His other library, Zarr, can be a great choice for dealing with persistent data (<https://zarr.readthedocs.io/en/stable/>).

## Doing parallel computing with Dask

The previous code is still quite slow, so now, we will use parallel processing to accelerate our data analysis. Our first approach will be using Dask, a Python-based library that provides scalable parallelism: most of the code that scales on your laptop will be able to scale on a large cluster. Dask is a fairly low-level and Python-related approach. Later in this chapter, we will discuss an alternative approach that is more high-level and language-agnostic.

## Getting ready

We will make a parallel version of the previous code, so you will need to have the same dataset available. We will be using HDF5 processing, so you should be acquainted with the previous recipe anyway.

## How to do it...

Take a look at the following steps:

We will start by doing the necessary imports and checking Dask's version:

```
from multiprocessing.pool import Pool
from math import ceil

import numpy as np

import h5py

import dask
import dask.array as da
import dask.multiprocessing
print(dask.__version__)
```

Make sure that Dask is at least version 0.19.2, as we will be using fairly recent features.

Now load some HDF5 data for processing:

```
h5_3L = h5py.File('ag1000g.phase1.ar3.pass.3L.h5', 'r')
samples = h5_3L['3L/samples']
positions = h5_3L['3L/variants/POS']
num_samples = len(samples)
del samples
```

While this recipe is a Dask version of the previous one, there will be slight differences imposed by the Dask programming model. At this stage, notice that we are only accessing some of the HDF5 fields that we used before.

The following code is lifted verbatim from the previous recipe, and creates the list of loci per window:

```
window_size = 50000
last_position = positions[-1]
num_windows = ceil(last_position / window_size)
limits = np.full((num_windows, 2), -1)
curr_window = positions[0] // window_size
limits[curr_window][0] = 0
for index, position in enumerate(positions):
    my_window = position // window_size
    if index % 1000000 == 0:
        print(index, position)
    if my_window != curr_window:
        limits[my_window, 0] = index
        limits[curr_window, 1] = index - 1
        curr_window = my_window
limits[num_windows - 1, 1] = len(positions)
```

Now let's convert our NumPy array into a Dask array:

```
limits = da.from_array(limits, chunks=(60, 2))
```

If you print the limits, you will get something different from the previous recipe:

```
print(limits[0], limits[-1])
```

The output is as follows:

```
dask.array<getitem, shape=(2,), dtype=int64, chunksize=(2,)>
dask.array<getitem, shape=(2,), dtype=int64, chunksize=(2,)>
```

This hasn't been executed yet, as the results will only be computed when explicitly asked.

Now, we will create a function to load the necessary data to compute statistics:

```
def get_hdf5():
    global calldata_genotype, alt_alleles, is_snp, num_samples

    try:
        calldata_genotype
    except NameError:
        import os
        print('Open', os.getpid())
        h5_3L = h5py.File('ag1000g.phase1.ar3.pass.3L.h5', 'r')
        samples = h5_3L['3L/samples']
```

```

calldata_genotype = h5_3L['/3L/calldata/genotype']
alt_alleles = h5_3L['/3L/variants/ALT']
is_snp = h5_3L['/3L/variants/is_snp']
return calldata_genotype, is_snp, alt_alleles, num_samples

```

The preceding code will set some global variables that will be accessed by our statistics function. At this stage, it is important to notice that our Dask code will be using multi-processing (there are many alternatives here, from multi-threading to cluster scheduling). This is due to how the HDF5 library works – concurrent read access will have to be initiated on each process (that is, the HDF5 file will have to be open on each process). The purpose of this code is to do exactly that. How it is called will be made clear later.

In any case, be careful with concurrent read access to HDF5 files. If the library that you use has some kind of lock-system, you might end up with code that is not parallel in practice. You can detect this by inspecting the CPU usage of your code.

7. Now, we will code the statistics function:

```

@da.as_gufunc(signature="(i)->()", output_dtypes=dict,
vectorize=True)
def calc_statistics(v):
    calldata_genotype, is_snp, alt_alleles, num_samples =
get_hdf5()
    start, end = v[0], v[1]
    min_maf = 0.5
    max_maf = 0.0
    non_bi = 0
    missing = 0
    non_snp = 0
    print(v)
    for pos in range(start, end + 1):
        if not is_snp[pos]:
            non_snp += 1
            continue
        if np.max(calldata_genotype[pos]) > 1:
            non_bi += 1
            continue
        if np.min(calldata_genotype[pos]) < 0:
            missing += 1
            continue
        num_alt = np.sum(calldata_genotype[pos]) # Because they
are coded as 1
        num_ref = num_samples * 2 - num_alt # (because all are
called)

```

```
min_called = min(num_ref, num_alt)
maf = min_called / (2 * num_samples)
if maf < min_maf:
    min_maf = maf
if maf > max_maf:
    max_maf = maf
return {'total': end - start + 1, 'missing': missing,
        'non.snp': non_snp, 'non.bi': non_bi,
        'min.maf': min_maf, 'max.maf': max_maf}
```

The code is similar to the equivalent from the previous recipe, save for the decorator at the very top. It mostly vectorizes the function for Dask. Here, we are using an experimental feature of Dask that allows us to use NumPy vectorizing.

We can now do the computing of the statistics:

```
stats = None
with dask.config.set(scheduler='multiprocessing'):
    stats = calc_statistics(limits).compute()
```

We will be using the multiprocessing scheduler for this. Dask is lazy by default, so we need to call the compute method.

We can now print the result:

```
print(stats[:10]) # just the first 10
```

The result will be the same as in the previous recipe.

## There's more...

There is a lot more to be discovered in Dask, for example, support for pandas' DataFrames, and more ad hoc structures, in addition to the NumPy arrays that we used here. You are strongly encouraged to read a bit about scheduling. Here, we used the multi-processing scheduler, but a lot of Dask's flexibility comes from the fact that schedulers can be swapped while maintaining the analysis code.

# Using high-performance data formats – Parquet

In the previous recipes, we used HDF5 as a format for the storage of genomic data. In this recipe, we will consider another format: Parquet, from the Apache Project. There are not, as far as I know, many use cases of Bioinformatics in Parquet (<https://parquet.apache.org/>), but there are several reasons why this format should be considered. For one, it can be used natively with Apache Spark (see the next recipe), and it can also be far more intelligent than HDF5 in terms of storage of data. Think, for example, faster indexing of data.

In this recipe, we will convert a subset of the HDF5 file that we used in the previous two recipes.

## Getting ready

You will need to download the same dataset as in the previous two recipes. At the very least, you are recommended to browse the HDF5 dataset (see the Getting ready section of the first recipe). There is no need to get acquainted with the rest of the code.

We will use Dask-native support for Parquet conversion.

## How to do it...

Take a look at the following steps:

We will start by doing all the necessary imports. Dask will be responsible for doing the conversion:

```
from math import ceil  
  
import numpy as np  
  
import h5py  
  
import dask.array as da  
import dask.dataframe as dd
```

We then read all the HDF5 datasets that we want to convert. For the sake of our example, we will use positions. If the position is an SNP, the qual and mq0 annotations will be used:

```
h5_3L = h5py.File('ag1000g.phase1.ar3.pass.3L.h5', 'r')
positions = h5_3L['/3L/variants/POS']
is_snp = h5_3L['/3L/variants/is_snp']
qual = h5_3L['/3L/variants/QUAL']
mq0 = h5_3L['/3L/variants/MQ0']
```

We will now create a Dask DataFrame:

```
all_ddf = dd.from_array(positions, columns=['POS'])
is_snp_dseries = dd.from_array(is_snp)
qual_dseries = dd.from_array(qual)
mq0_dseries = dd.from_array(mq0)
```

We start by creating a DataFrame (notice the use of dd and not da). The first one will actually be the DataFrame for conversion to Parquet, while the other ones will have an intermediary step.

We will now add the remaining data to the Dask DataFrame for conversion:

```
all_ddf['is_snp'] = is_snp_dseries
all_ddf['QUAL'] = qual_dseries
all_ddf['MQ0'] = mq0_dseries
```

Finally, we do the conversion:

```
all_ddf.to_parquet('geno.parquet')
```

## There's more...

This is by far the more experimental recipe of this book, but for very demanding datasets, Parquet is probably one of the more efficient formats available and has native support in Spark. In the medium to long-term, we will probably see developments in this space. You should expect other ways of converting data into the Parquet format to appear. If you decide to use Parquet, be sure to check all the different indexing strategies that the format supports.

# Computing sequencing statistics using Spark

If you need to use parallel computing, then Spark is one alternative to Dask. Its abstraction level is slightly higher. This gives you less granular control over the computation, but is more declarative to code. Spark is also somewhat language agnostic (it is actually Java/Scala-based). Here, we will compute some very basic statistics over the Parquet dataset that we generated in the previous recipe.

## Getting ready

Preparing for this recipe can be quite tricky. First, we will have to start a Spark server. At the time of writing this book, the conda packages for accessing Spark were quite immature. We will still use conda here, but we will not install any Spark packages from conda. Follow these steps to prepare the environment:

Make sure that you have Java 8 installed. Be careful with the Java version, as an older version will not work, but a newer might also be problematic.

Download Spark (<https://spark.apache.org/downloads.html>). This code was tested with version 2.3.2. Do not use an older version, although you might want to try a newer one.

3. Unzip it and enter the directory. Run `./sbin/start-all.sh`.

With your browser, connect to `http://localhost:8080`. You should see some output from Spark.

5. Use conda install `py4j` at the command line.

Note the directory where you have Spark installed.

It bears repeating: do not install any conda package related to Spark. At least at the time of writing of this book, this will just mean trouble.

We will be using PySpark, a Python interface for Spark, but we will use the version that comes with your Spark download.

Finally, you will need to have run the previous recipe, as it will provide the input for this one.

## How to do it...

Let's take a look at the following steps:

Let's start by making sure that we can access PySpark:

```
import sys
sys.path.append('/PATH_TO/spark-2.3.2-bin-hadoop2.7/python/') # Not
conda
#Careful with Java version
#conda install py4j
```

Be sure to change PATH\_TO to whatever path you have for your Spark installation.

Now, let's import pyspark:

```
import pyspark as spark
from pyspark.sql.functions import col,round as round_
```

We will be using the round function, but we will rename it to round\_ to avoid clashes with the builtin round function.

Let's connect to our Spark server:

```
sc = spark.SparkContext('spark://127.0.1.1:7077')
```

Now, we will create SQLcontext:

```
sqlc = spark.SQLContext(sc)
```

There are other contexts for Spark, and we will discuss them in the next section of this recipe.

We can now read the DataFrame from Parquet:

```
all_df = sqlc.read.parquet('geno.parquet')
```

If the code crashes here, then most probably, you have the wrong Java version. You will want Java 8, but this might change over time.

Let's inspect its type:

```
print(type(all_df))
```

We now have a Spark DataFrame:

```
<class 'pyspark.sql.dataframe.DataFrame'>
```

Finally, we can do DataFrame operations on it:

```
all_df.select(col("*"), round_(col("POS") /  
50000).alias('window')).groupBy('window').count().orderBy('window')  
.show()
```

This is a fairly complex data frame query (check pandas for more information). It computes the number of observations per 50,000 bp window. The abridged output is as follows:

window	count
0.0	43
1.0	475
2.0	858
3.0	1146
4.0	915
6.0	746
7.0	1420
8.0	1167
9.0	32
[...]	

## There's more...

Do not be demotivated by the complex setup. This is a one-off operation. Instead, check whether the computing model underlying Spark is one that you like or not.

Spark abstracts away lots of issues related to parallelization, to the point that you actually wonder what's going on. If you follow the Java processes that are doing the work, you will see that many of them are using several cores to perform computation. If you are comfortable with that level of distance to the implementation, then Spark is for you. If not, check the Dask recipe.

If you read the Spark documentation, you will find lots of examples related to resilient distributed datasets (RDDs). This is an interesting avenue to explore, but here we are using one that should be quite familiar to pandas users. RDDs and yet another API, datasets, are both worth checking out. DataFrames and datasets are more recent in Spark than RDDs, so expect them to be slightly less mature.

# Optimizing code with Cython and Numba

In this recipe, we will look at a short introduction on how to optimize code with Cython and Numba. These are competitive approaches; Cython is a superset of Python that allows you to call C functions and specify C types. Numba is a just-in-time compiler that optimizes the Python code.

As an example, we will reuse the distance recipe from Chapter 7, Using the Protein Data Bank. We will compute the distance between all of the atoms in a PDB file.

## Getting ready

Cython normally requires specifying your optimized code in a separate .pyx file (Numba is a more declarative solution without this requirement). As IPython provides a magic to hide this, we will use IPython here. However, note that if you are on plain Python, the Cython development will be a bit more cumbersome. As usual, this is available in the Chapter09/Cython\_Numba.ipynb Notebook file.

## How to do it...

Take a look at the following steps:

Let's load our PDB structure with the following code:

```
import math  
  
%load_ext Cython  
  
from Bio import PDB  
repository = PDB.PDBList()  
parser = PDB.PDBParser()  
repository.retrieve_pdb_file('1TUP', pdir='.')  
p53_1tup = parser.get_structure('P 53', 'pdb1tup.ent')
```

Here is our pure Python distance function:

```
def get_distance(atoms):  
    atoms = list(atoms)  
    natoms = len(atoms)  
    for i in range(natoms - 1):  
        xi, yi, zi = atoms[i].coord  
        for j in range(i + 1, natoms):  
            xj, yj, zj = atoms[j].coord
```

```
my_dist = math.sqrt((xi - xj)**2 + (yi - yj)**2 + (zi -  
zj)**2)
```

This will compute the distance between all of the atoms in the PDB file. We do not care about the result here, just the time cost, so we do not return anything.

We can get the time cost as follows:

```
%time get_distance(p53_1tup.get_atoms())
```

The timing will vary from computer to computer, but where this code was tested, the output was as follows:

```
CPU times: user 3min 34s, sys: 3.61 ms, total: 3min 34s  
Wall time: 3min 34s
```

We use `%time` just because the function is very slow. As a general rule, this is not recommended: you should use `%timeit`, which will report statistics across several runs.

Let's take a look at the first Cython version, which is nothing more than an attempt to compile this with Cython, and see how much time we gain:

```
%%cython  
import math  
def get_distance_cython_0(atoms):  
    atoms = list(atoms)  
    natoms = len(atoms)  
    for i in range(natoms - 1):  
        xi, yi, zi = atoms[i].coord  
        for j in range(i + 1, natoms):  
            xj, yj, zj = atoms[j].coord  
            my_dist = math.sqrt((xi - xj)**2 + (yi - yj)**2 + (zi -  
zj)**2)  
  
%time get_distance_cython_0(p53_1tup.get_atoms())
```

The output on my computer was as follows:

```
CPU times: user 3min 25s, sys: 628 ms, total: 3min 25s  
Wall time: 3min 24s
```

We gained nothing here. Indeed, we were not hoping for much. We know that with Cython, the code requires some changes.

Let's rewrite the function for Cython and see how much time it takes here:

```
%%cython
cimport cython
from libc.math cimport sqrt, pow

cdef double get_dist_cython(double xi, double yi, double zi,
                           double xj, double yj, double zj):
    return sqrt(pow(xi - xj, 2) + pow(yi - yj, 2) + pow(zi - zj,
2))

def get_distance_cython_1(object atoms):
    natoms = len(atoms)
    cdef double x1, xj, yi, yj, zi, zj
    for i in range(natoms - 1):
        xi, yi, zi = atoms[i]
        for j in range(i + 1, natoms):
            xj, yj, zj = atoms[j]
            my_dist = get_dist_cython(xi, yi, zi, xj, yj, zj)

%timeit get_distance_cython_1([atom.coord for atom in
p53_1tup.get_atoms()])
```

So, we took the expensive arithmetic computation that sits in the inner loop and optimized it.

We use libc (the fast C code) and make sure that Cython has all the necessary typing information.

The result changes dramatically:

19.3 s ± 303 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

It's slightly better than an order of magnitude!

As a side note, we used %timeit here, just to show an example of its output. Remember that you should prefer using %timeit to %time. Note that we only optimized the inner loop, which was highly number crunching. You probably don't want to perform more than this because over-optimizing your algorithms tends to make them difficult to read and manage.

Also, you get no sizable advantage from optimizing the non-inner loop code.

We will now switch to numba and use a decorator to create an optimized version of the original function, and time it with the following code:

```
from numba import float_
from numba.decorators import jit

get_distance_numba_0 = jit(get_distance)

%time get_distance_numba_0(p53_1tup.get_atoms())
```

Again, the result is not great:

```
CPU times: user 3min 40s, sys: 280 ms, total: 3min 40s
Wall time: 3min 40s
```

Here, we had neither positive or negative expectations because in theory, numba can optimize lots of code. Maybe future versions will be able to deal with this code automatically.

We can refactor this code to see whether we can get a better result with Numba:

```
@jit
def get_dist_numba(xi, yi, zi, xj, yj, zj):
    return math.sqrt((xi - xj)**2 + (yi - yj)**2 + (zi - zj)**2)

def get_distance_numba_1(atoms):
    natoms = len(atoms)
    for i in range(natoms - 1):
        xi, yi, zi = atoms[i]
        for j in range(i + 1, natoms):
            xj, yj, zj = atoms[j]
            my_dist = get_dist_numba(xi, yi, zi, xj, yj, zj)

%timeit get_distance_numba_1([atom.coord for atom in
p53_1tup.get_atoms()])
```

Our timing is now as follows:

```
35.3 s ± 402 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

This is almost double the Cython version, but is still way better than the pure Python code.

Note that refactoring is also performed as in Cython, but the code is 100 percent Python (whereas the Cython code is close to Python, but not Python). Also, there was no need to decorate the function extensively. In theory, you could annotate the type of the function and parameters, but Numba does a great job at discovering this for you. Lastly, you get no improvements (in this case, at least) with more annotations.

## There's more...

This is just a very small taste of what both libraries can do. For example, Numba can work with NumPy and generate code for GPUs.

Note that the performance comparison will vary from problem to problem. You can find cases on the web where Numba outperformed Cython. Refer to <https://jakevdp.github.io/blog/2012/08/24/numba-vs-cython/> for examples.

It should be clear that Numba is less intrusive than Cython because you end up with 100 percent Python code (although you may still have to refactor for performance). Do not over-optimize; find the most critical parts of your code and concentrate your efforts there.

# 10

## Other Topics in Bioinformatics

In this chapter, we will cover the following recipes:

- Doing metagenomics with the QIIME 2 Python API
- Inferring shared chromosomal segments with Germline
- Accessing the Global Biodiversity Information Facility via REST
- Georeferencing GBIF datasets
- Plotting protein interactions with Cytoscape the hard way

### Introduction

In this chapter, we will address some topics that are well within the remit of computational biology and deserve at least some reference. We will start with a recipe on metagenomics using QIIME 2. Then, we will infer shared chromosomal segments across individuals using Germline. After that, we will have a look at some recipes using the Global Biodiversity Information Facility (GBIF), a database of worldwide scientific data on biodiversity. Then, we will interface Python with Cytoscape, a powerful software platform that's used for visualizing genomic and proteomic interaction networks.

We will take the opportunity to indirectly introduce other topics, such as more bioinformatics databases, graph processing, and geo-referencing, that are relevant to our context (one way or another). To interface, we will only use REST APIs for all databases and services, making the code in all of the recipes here quite streamlined. You may want to refresh your knowledge of REST architectures. We will use the Requests library for REST interfacing. If you've never used it before, don't worry; it's actually quite easy to use.

# Doing metagenomics with the QIIME 2 Python API

Wikipedia says that metagenomics is the study of genetic material that's recovered directly from environmental samples. Note that "environment" here should be interpreted broadly: in the case of our example, we will deal with gastrointestinal microbiomes on a study of a fecal microbiome transplant in children with gastrointestinal problems. The study is one of the tutorials of QIIME 2, which is one of the most widely used applications for data analysis in metagenomics. QIIME 2 has several interfaces: a GUI, a command line, and a Python API called the Artifact API.

Tomasz Kościółek has an outstanding tutorial for using the Artifact API based on the most well-developed (client-based, not artifact-based) tutorial on QIIME 2, the "Moving Pictures" tutorial (<http://nbviewer.jupyter.org/gist/tkoscio/29de5198a4be81559a075756c2490fde>). Here, we will create a Python version of the Fecal Microbiota Transplant study that's available, as with the client interface, at <https://docs.qiime2.org/2018.8/tutorials/fmt/>. You should get familiar with it as we won't go into the details of the biology here. I do follow a more convoluted route than Tomasz: this will allow you to get a bit more acquainted with QIIME 2 Python internals. After you get this experience, you will probably want to follow Tomasz's route, not mine. However, the experience you get here will make you more comfortable and self-confident with QIIME's internals.

## Getting ready

This recipe is slightly more complicated to set up. We will have to create a conda environment where packages from QIIME 2 are segregated from packages from all other applications. The steps that you need to follow are simple.

On OS X, use the following code to create a new conda environment:

```
wget https://data.qiime2.org/distro/core/qiime2-2018.8-py35-osx-conda.yml  
conda env create -n qiime2-2018.8 --file qiime2-2018.8-py35-osx-conda.yml
```

On Linux, use the following code to create the environment:

```
wget https://data.qiime2.org/distro/core/qiime2-2018.8-py35-linux-conda.yml  
conda env create -n qiime2-2018.8 --file qiime2-2018.8-py35-linux-conda.yml
```

If these instructions do not work, check the QIIME 2 website for an updated version (<https://docs.qiime2.org/2018.8/install/native>). QIIME 2 is updated regularly.

At this stage, you need to enter the QIIME 2 conda environment by using source activate qiime2-2018.8. If you want to get to the standard conda environment, use source deactivate instead. You might want to install other packages you like inside QIIME 2's environment using conda install.

To prepare for Jupyter execution, you should install the QIIME 2 extension, as follows:

```
jupyter serverextension enable --py qiime2 --sys-prefix
```



The extension is highly interactive and allows you to look at data from different view points that cannot be captured in this book. The downside is that it won't work in nbviewer (some cell outputs won't be visible with the static viewer). For that reason, we will supply a PDF of an execution. Remember to interact with the outputs from the extension, since many are dynamic.

You can now start Jupyter. The Notebook can be found in the Chapter10/QIIME2\_Metagenomics.ipynb file.



This recipe will not work in the Docker container, as the environment available there is the standard environment. This means that you are working from our Docker installation. In this case, you will have to download the recipe and install the packages manually.

You can find the instructions to get the data of both the Notebook files and the QIIME 2 tutorial.

## How to do it...

Let's take a look at the following steps:

Let's start by checking what plugins are available:

```
import pandas as pd

from qiime2.metadata.metadata import Metadata
from qiime2.metadata.metadata import CategoricalMetadataColumn
from qiime2.sdk import Artifact
from qiime2.sdk import PluginManager
from qiime2.sdk import Result

pm = PluginManager()
demux_plugin = pm.plugins['demux']
#demux_emp_single = demux_plugin.actions['emp_single']
demux_summarize = demux_plugin.actions['summarize']
print(pm.plugins)
```

We are also accessing the demultiplexing plugin and its summarize action.

Let's take a peek at the summarize action, namely inputs, outputs, and parameters:

```
print(demux_summarize.description)
demux_summarize_signature = demux_summarize.signature
print(demux_summarize_signature.inputs)
print(demux_summarize_signature.parameters)
print(demux_summarize_signature.outputs)
```

The output will be as follows:

```
Summarize counts per sample for all samples, and generate
interactive positional quality plots based on `n` randomly selected
sequences.
OrderedDict([('data',
ParameterSpec(qiime_type=SampleData[JoinedSequencesWithQuality | PairedEndSequencesWithQuality | SequencesWithQuality],
view_type=<class 'q2_demux._summarize._visualizer._PlotQualView'\>, default=NOVALUE, description='The demultiplexed sequences to be summarized.'))])
OrderedDict([('n', ParameterSpec(qiime_type=Int, view_type=<class 'int'\>, default=10000, description='The number of sequences that should be selected at random for quality score plots. The quality plots will present the average positional qualities across all of the sequences selected. If input sequences are paired end, plots will be generated for both forward and reverse reads for the same'))])
```

```
`n` sequences.'))])  
OrderedDict([('visualization',  
ParameterSpec(qiime_type=Visualization, view_type=None,  
default=NOVALUE, description=NOVALUE))])
```

We will now load the first dataset, demultiplex it, and visualize some demultiplexing statistics:

```
seqs1 = Result.load('fmt-tutorial-demux-1-10p.qza')  
sum_data1 = demux_summarize(seqs1)  
  
sum_data1.visualization
```

The original data for this recipe is supplied in QIIME 2 format. Obviously, you will have your own original data in some other format (probably FASTQ)—see the There's more... section for a way to load a standard format.



QIIME 2's .qza and .qzv formats are simply zipped files. You can have a look at the content with unzip.

The chart will be similar to in the QIIME CLI tutorial, but be sure to check the interactive quality plot of our output:

Let's do the same for the second dataset:

```
seqs2 = Result.load('fmt-tutorial-demux-2-10p.qza')  
sum_data2 = demux_summarize(seqs2)  
  
sum_data2.visualization
```

Let's use the DADA2 (<https://github.com/benjineb/dada2>) plugin for quality control:

```
dada2_plugin = pm.plugins['dada2']  
dada2_denoise_single = dada2_plugin.actions['denoise_single']  
qual_control1 = dada2_denoise_single(demultiplexed_seqs=seqs1,  
trunc_len=150, trim_left=13)  
qual_control2 = dada2_denoise_single(demultiplexed_seqs=seqs2,  
trunc_len=150, trim_left=13)
```

Let's extract some statistics from denoising (first set):

```
metadata_plugin = pm.plugins['metadata']
metadata_tabulate = metadata_plugin.actions['tabulate']
stats_meta1 =
    metadata_tabulate(input=qual_control1.denoising_stats.view(Metadata
))
stats_meta1.visualization
```

Again, the result can be found online in the QIIME 2 CLI version of the tutorial.

Now, let's do the same for the second set:

```
stats_meta2 =
metadata_tabulate(input=qual_control2.denoising_stats.view(Metadata
))
stats_meta2.visualization
```

Now, merge the denoised data:

```
ft_plugin = pm.plugins['feature-table']
ft_merge = ft_plugin.actions['merge']
ft_merge_seqs = ft_plugin.actions['merge_seqs']
ft_summarize = ft_plugin.actions['summarize']
ft_tab_seqs = ft_plugin.actions['tabulate_seqs']

table_merge = ft_merge(tables=[qual_control1.table,
qual_control2.table])
seqs_merge =
ft_merge_seqs(data=[qual_control1.representative_sequences,
qual_control2.representative_sequences])
```

Then, gather some quality statistics from the merge:

```
ft_sum = ft_summarize(table=table_merge.merged_table)
ft_sum.visualization
```

Finally, let's get some information about the merged sequences:

```
tab_seqs = ft_tab_seqs(data=seqs_merge.merged_data)
tab_seqs.visualization
```

## There's more...

The preceding code does not show you how to import data. The actual code will vary from case to case (single-end data, paired-end data, or already demultiplexed data), but for the main QIIME 2 tutorial, Moving Pictures, assuming that you have downloaded the single-end, non-demultiplexed data, and barcodes into a directory called data, you can do the following:

```
data_type = 'EMPSingleEndSequences'  
conv = Artifact.import_data(data_type, 'data')  
conv.save('out.qza')
```

As stated in the preceding code, if you look on GitHub for this Notebook, the static nbviewer system will not be able to render the Notebook correctly (you have to run it yourself). There is a PDF on the directory that includes a print version of the Notebook with the content. This is far from perfect; it is not interactive, since the quality is not great, but at least it lets you get an idea of the output without running the code.

## Inferring shared chromosomal segments with Germline

The discovery of shared chromosomal segments among individuals can have many applications: finding relationships across individuals, estimating strong bottlenecks, or possible signals of selection.

To execute this recipe, we will use Germline, which is an efficient tool for performing the inference of shared chromosomal segments. It requires phased data. Phased data allows you to assign genotyped data to a specific chromosome, that is, instead of having a list of genotype calls per position, we end up with reconstructed haplotypes.

## Getting ready

The preparation of data requires some work. First, we must provide the final phased haplotypes for our dataset in the notebook directory so that you can skip the following process, save for the trivial download of integrated\_call\_samples.20101123.ped.

The directory from the repository is Chapter10. The Notebook is Germline.ipynb, the data file is good.match.gz, and the code support files are merge.py and clean\_sample.py.

As an example, we will use a phased dataset (chromosome 21) from the 1000 Genomes Project. You can download it with the following command:

```
wget
ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/phase1/analysis_results/shapeit2_p
hased_haplotypes/ALL.chr21.SHAPeIT2_integrated_phase1_v3.20101123.snps_inde
ls_svs.genotypes.all.vcf.gz

wget
ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/phase1/analysis_results/integrated
_call_sets/integrated_call_samples.20101123.ped
```

We will now convert the VCF file into a format that is more common with phased data. For that, we will use a Perl script from the impute package:

```
wget https://mathgen.stats.ox.ac.uk/impute/scripts/vcf2impute_legend_haps
perl vcf2impute_legend_haps -vcf
ALL.chr21.SHAPeIT2_integrated_phase1_v3.20101123.snps_indels_svs.genotypes.
all.vcf.gz -leghap 21 -chr 21 -snps_only
```

We will now merge the locations with the actual phased data (which is provided separately) by the Perl script. You can find this code in the merge.py file in the Chapter10 directory:

```
import gzip
import sys

hap = sys.argv[1]
legend = sys.argv[2]
chrom = sys.argv[3]

hap_f = gzip.open(hap, 'rt', encoding='utf-8')
legend_f = gzip.open(legend, 'rt', encoding='utf-8')

legend_f.readline() # header
snp = legend_f.readline()
while snp != '':
    haps = hap_f.readline()
    sys.stdout.write('%s %s %s' % (chrom, snp.rstrip(), haps))
    snp = legend_f.readline()
```

We will also reduce some of the information available on samples; this code can be found in the clean\_sample.py file:

```
import sys

sys.stdout.write('ID_1 ID_2 missing\n0 0 \n')
for line in sys.stdin:
```

```
ind = line.rstrip()
sys.stdout.write('%s %s 0\n' % (ind, ind))
```

The preceding two bits are used as follows:

```
python merge.py 21.hap.gz 21.legend.gz 21 > 21.merge
python clean_sample.py < 21.sample_list > 21.sample
```

Now, we do a final conversion with the Germline conversion tool and call germline properly:

```
impute_to_ped 21.merge 21.sample g21
germline -input g21.ped 21.map -output good
```

It is beyond the scope of this book to discuss all the germline options, but it is worth noting that you can supply a genetic map.

Finally, we gzip the file to save disk space:

```
gzip good.match
```

## How to do it...

Let's take a look at the following steps:

The first thing that we do is select an individual per family at most. This step is not strictly necessary (as opposed to many other recipes where having related individuals will bias the analysis), but we will do it anyway:

```
from collections import defaultdict
import gzip

import scipy
import scipy.stats as stats

germline_file = 'good.match.gz'
sample_file = 'integrated_call_samples.20101123.ped'

inds = set()
ind_pop = {}
selected_inds = {}
pop_inds = defaultdict(list)

with gzip.open(germline_file, 'rt', encoding='utf-8') as f:
    for l in f:
        toks = l.rstrip().split()
```

```

inds.add(toks[1])
inds.add(toks[3])

with open(sample_file, 'rt', encoding='utf-8') as f:
    f.readline() # header
    for l in f:
        toks = l.rstrip().split('\t')
        fam = toks[0]
        ind = toks[1]
        pop = toks[6]
        if ind not in inds:
            continue
        selected_inds[fam] = ind # We just want one per family
        ind_pop[ind] = pop

    for ind, pop in ind_pop.items():
        pop_inds[pop].append(ind)

```

This code is slightly more complicated than expected. We are mostly trying to ensure that any inconsistency between the metadata file (integrated\_call\_samples.20101123.ped) and the germline file is removed.

We will now traverse the germline output while extracting the following bits of information: the largest shared haplotype, and basic information about the shared haplotypes. The basic information includes the two individuals involved, the start and end position, and the size in Mbp (if you supply a genetic map, the size will be in cM):

```

larger_shared = (0, None, None)
sizes = []
all_sizes = []
shared = defaultdict(int)
with gzip.open(germline_file, 'rt', encoding='utf-8') as f:
    for l in f:
        toks = l.rstrip().split()
        ind1 = toks[1]
        ind2 = toks[3]
        start = int(toks[5])
        end = int(toks[6])
        size = float(toks[10])
        all_sizes.append(size)
        if start > 10000000 and end < 15000000:
            continue
        if size > larger_shared[0]:
            larger_shared = (size, ind1, ind2)
        shared[tuple(sorted((ind1, ind2)))] += size

```

```
sizes.append(size)
```

We actually store two lists of basic information: one complete and one excluding positions that start and end between the 10 MB and 14 MB positions. Can you guess why we are doing this? Think about how chromosomes are organized... We will answer this in the next step, so read on.

We will now use SciPy to extract basic statistics from both datasets:

```
print(stats.describe(sizes))
print(stats.describe(all_sizes))
```

The output is as follows:

```
DescribeResult(nobs=43736, minmax=(3.008, 24.508),
mean=3.4114706191695627, ....
DescribeResult(nobs=173, minmax=(3.008, 24.508),
mean=4.792404624277457, ...
```

Here, there are 43736 observations, but only 173 are outside the 10 MB to 14 MB region! What makes this region in the chromosome so special that there are so many shared haplotypes there? This is where the centromere is located.

Let's print some information about the largest shared haplotype in our dataset:

```
print(larger_shared)
print(ind_pop[larger_shared[1]], ind_pop[larger_shared[2]])
```

The output is as follows:

```
(24.508, 'HG00501', 'HG00512')
CHS CHS
```

Finally, let's print some population statistics:

```
pop_shared = defaultdict(list)
for inds, total_size in shared.items():
    ind1, ind2 = inds
    pop1 = ind_pop[ind1]
    pop2 = ind_pop[ind2]
    if pop1 == pop2:
        pop_shared[pop1].append(total_size)

for pop, total_sizes in pop_shared.items():
    print("%s %3d %3d %*.2f %*.2f" % (pop,
                                         len(pop_inds[pop]),
                                         len(total_sizes),
```

```
    6, scipy.mean(total_sizes),
    6, scipy.median(total_sizes)))
print(pop_inds.keys())
```

The output is as follows:

```
PUR 55 28 4.05 3.82
LWK 86 15 9.10 4.95
GBR 89 7 5.66 5.99
TSI 98 3 3.63 3.53
FIN 91 38 3.97 3.63
CHS 93 14 11.98 11.87
CLM 59 12 3.89 3.43
MXL 65 11 9.48 8.35
ASW 50 2 16.61 16.61
CEU 83 2 7.77 7.77
CHB 88 1 3.82 3.82
dict_keys(['CEU', 'ASW', 'MXL', 'CLM', 'GBR', 'FIN', 'IBS', 'YRI',
'CHB', 'JPT', 'LWK', 'TSI', 'PUR', 'CHS'])
```

Note that three populations do not seem to have shared haplotypes (outside the centromere region, that is).

## There's more...

If you use germline, be sure to check all its parameters. In terms of analysis, there is plenty that could be done with shared haplotype information. From a programming perspective, there is some interesting work to be done with regards to the visualization of shared haplotypes across a chromosome/genome.

## Accessing the Global Biodiversity Information Facility via REST

The GBIF (<http://www.gbif.org>) gives the available information about biodiversity in a programmatic friendly way using a REST API.

In GBIF, we will find evidence of the occurrence of species across the planet, and much of this information is geo-referenced.

In this recipe, we will concentrate on two types of GBIF information: species and occurrences.

Species are actually a more general taxonomic framework, and occurrences record the observations of species.

In this recipe, we will try to extract biodiversity information related to bears. You can find this content in the Chapter10/GBIF.ipynb Notebook.

## How to do it...

Let's take a look at the following steps:

First, let's define a function to get the data on REST, as shown in the following code:

```
import requests

def do_request(service, a1=None, a2=None, a3=None, **kwargs):
    server = 'http://api.gbif.org/v1'
    params = ""
    for a in [a1, a2, a3]:
        if a is not None:
            params += '/' + a
    req = requests.get('%s/%s%s' % (server, service, params),
                       params=kwargs, headers={'Content-Type':
                       'application/json'})
    if not req.ok:
        req.raise_for_status()
    return req.json()
```

Then, look at how many species records refer to the word bear. Remember that this is actually more general than species. You will also get records for all kinds of taxonomic ranks with the following code:

```
req = do_request('species', 'search', q='bear')
print(req['count'])
```

A typical record contains information about taxonomic rank, the relevant taxonomic information (kingdom, family, genus, and so on), and the scientific rank. The following screenshot is an example of a record:

```
{'key': 9701974,
'datasetKey': 'd7dddbf4-2cf0-4f39-9b2a-bb099caae36c',
'constituentKey': 'e01b0cbb-a10a-420c-b5f3-a3b20cc266ad',
'parentKey': 9661213,
'parent': 'Mammarenavirus',
'kingdom': 'Viruses',
'family': 'Arenaviridae',
'genus': 'Mammarenavirus',
'species': 'Bear Canyon mammarenavirus',
'kingdomKey': 8,
'familyKey': 6230,
'genusKey': 9661213,
'speciesKey': 9701974,
'scientificName': 'Bear Canyon mammarenavirus',
'authorship': '',
'nameType': 'VIRUS',
'taxonomicStatus': 'ACCEPTED',
'rank': 'SPECIES',
'origin': 'SOURCE',
'numDescendants': 0,
'numOccurrences': 0,
'habitats': [],
'nomenclaturalStatus': [],
'threatStatuses': [],
'descriptions': [],
'vernacularNames': [],
'higherClassificationMap': {'8': 'Viruses',
'6230': 'Arenaviridae',
'9661213': 'Mammarenavirus'},
'synonym': False}
```

RECORD

There are ~~300~~ many records to inspect; let's restrict ourselves to the ones that are on the taxonomic rank of a family, as shown in the following code:

```
req_short = do_request('species', 'search', q='bear',
rank='family')
print(req_short['count'])
bear = req_short['results'][0]
```

These records are ordered by relevance to the search term; according to GBIF's algorithms, we will take the very first record to continue our work.

As GBIF limits the number of records that you can get per call, let's use the following function to get all records (of course, this is to be used with care):

```
import time

def get_all_records(rec_field, service, a1=None, a2=None, a3=None,
                    **kwargs):
    records = []
    all_done = False
    offset = 0
    num_iter = 0
    while not all_done and num_iter < 100: # arbitrary
        req = do_request(service, a1=a1, a2=a2, a3=a3,
                          offset=offset, **kwargs)
        all_done = req['endOfRecords']
        if not all_done:
            time.sleep(0.1)
            offset += req['limit']
            records.extend(req[rec_field])
            num_iter += 1
    return records
```



Many REST services offer paging APIs, that is, you can take results in parts by issuing multiple calls, specifying the starting point (in this case, offset), and a limit for the number of results. Since we want to be good citizens, we will include a sleep of 0.1 seconds between calls to the server so that there is no excessive burden on it.

Now, given a certain internal node in the taxonomic tree, let's get all the leaves down from that node, as shown in the following code:

```
def get_leaves(nub):
    leaves = []
    recs = get_all_records('results', 'species', str(nub),
                          'children')
    if len(recs) == 0:
        return None
    for rec in recs:
        if 'nubKey' not in rec: # XXX why?
            continue
        rec_leaves = get_leaves(rec['nubKey'])
        if rec_leaves is None:
            leaves.append(rec)
```

```

else:
    leaves.extend(rec_leaves)
return leaves

```

Here, nub is GBIF's taxonomy identifier.

Finally, let's get the leaves for the first bear node that we got on the search by the family rank. We will mostly have the name and taxonomy information. We print the scientific name, rank of the record, and the vernacular name, if it exists, for all the leaves:

```

records = get_all_records('results', 'species',
str(bear['nubKey']), 'children')
leaves = get_leaves(bear['nubKey'])
for rec in leaves:
    print(rec['scientificName'], rec['rank'], end=' ')
    vernaculars = do_request('species', str(rec['nubKey']),
'vernacularNames', language='en')['results']
    for vernacular in vernaculars:
        if vernacular['language'] == 'eng':
            print(vernacular['vernacularName'], end='')
            break
    print()

```

The vernacular name comes from another service. GBIF has vernacular names in many languages; here, we will choose the English version. This call will take a few seconds because of the sleep code that we introduced previously.

For all of the leaves, we will now summarize the source of all records, the country of observation, the number of extinct references, and the species with no occurrences at all (remember that occurrence is another fundamental GBIF concept), as shown in the following code:

```

basis_of_record = defaultdict(int)
country = defaultdict(int)
zero_occurrences = 0
count_extinct = 0
for rec in leaves:
    #print(rec['scientificName'], rec['rank'], rec['taxonID'])
    occurrences = get_all_records('results', 'occurrence',
'search', taxonKey=rec['nubKey'])
    for occurrence in occurrences:
        basis_of_record[occurrence['basisOfRecord']] += 1
        country[occurrence.get('country', 'NA')] += 1
        #there is also publishingCountry
    if len(occurrences) > 0:
        zero_occurrences += 1

```

```

profiles = do_request('species', str(rec['nubKey']),
'speciesProfiles')['results']
for profile in profiles:
    if profile.get('extinct', False):
        count_extinct += 1
        break

```

We maintain two dictionaries. One is basis\_of\_record, with a count per different record origin, and the second is country, with a count per country of observation (there is also the country that published the results, and it is different many times). We also check the speciesProfiles service to check whether a record is labeled as extinct.

Let's plot this, as follows:

```

import numpy as np
import matplotlib.pyplot as plt

countries, obs_countries = zip(*sorted(country.items(), key=lambda
x: x[1]))
basis_name, basis_cnt = zip(*sorted(basis_of_record.items(),
key=lambda x: x[1]))
fig = plt.figure(figsize=(16, 9))
ax = fig.add_subplot(1, 2, 1)
ax.barh(np.arange(10) - 0.5, obs_countries[-10:])
ax.set_title('Top 10 countries per occurrences')
ax.set_yticks(range(10))
ax.set_ylim(0.5, 9.5)
ax.set_yticklabels(countries[-10:])
#refer metadata problems

ax = fig.add_subplot(2, 2, 2)
ax.set_title('Basis of record')
ax.bar(np.arange(len(basis_name)), basis_cnt, color='g')
basis_name = [x.replace('OBSERVATION', 'OBS').replace('_SPECIMEN',
'') for x in basis_name]
ax.set_xticks(0.5 + np.arange(len(basis_name)))
ax.set_xticklabels(basis_name, size='x-small')

ax = fig.add_subplot(2, 2, 4)
other = len(leaves) - zero_occurrences - count_extinct
pie_values = [zero_occurrences, count_extinct, other]
labels = ['No occurrence (%d)' % zero_occurrences,
          'Extinct (%d)' % count_extinct, 'Other (%d)' % other]
ax.pie(pie_values, labels=labels,
       colors=['cyan', 'magenta', 'yellow'])
ax.set_title('Status for each species')

```

The following is the output:

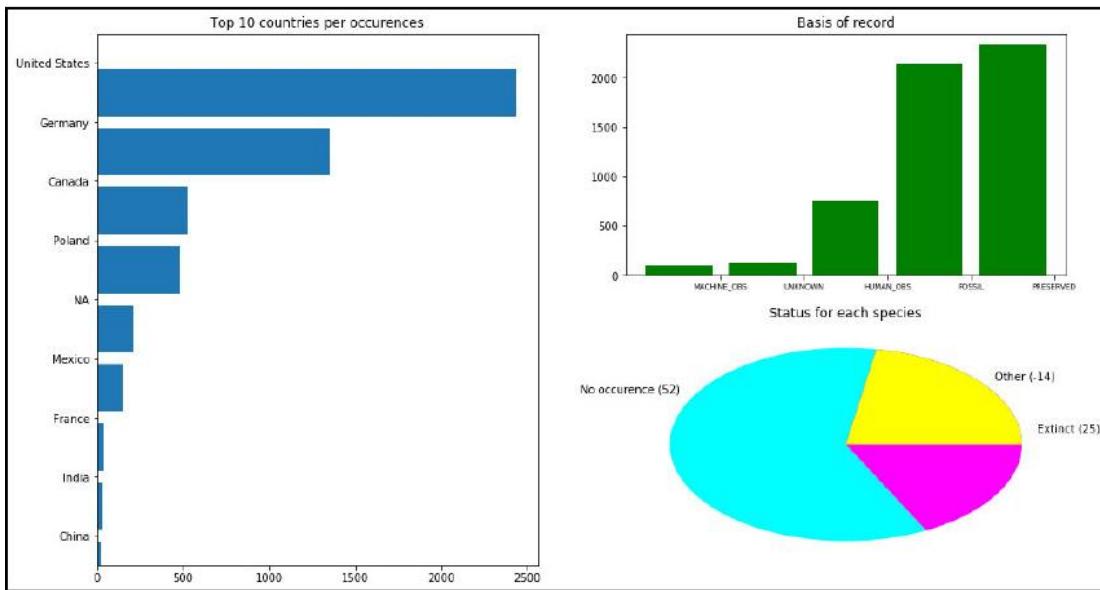


Figure 1: This diagram shows the referred countries of occurrence, the distribution of the origin of records, and the status of the records in terms of extinction and occurrence

Be careful with pie charts because they are seen by many as difficult to interpret. Here, we explicitly added the number of observations per group to make ours clearer. If you prefer, you can import seaborn with Matplotlib to get a more modern look. Some records do not have country information (the NA entry on the chart).

## There's more...

The GBIF database doesn't seem to be totally consistent in terms of data. The sources of the occurrences and species differ. Therefore, not too much effort is put into standardization. From species names to information about countries (such as the preceding NA case), you can see this in different parts of the database. Also, many records do not have the complete geo-referenced data, so be careful when analyzing the data. The REST API is documented at <http://www.gbif.org/developer/summary>.

# Georeferencing GBIF datasets

In this recipe, we will work with the geo-referenced data from the GBIF dataset. We will take this opportunity to look at how to interface Python with OpenStreetMap (<https://www.openstreetmap.org>), a freely available mapping service. We will also use a Python image processing library called Pillow (<http://python-pillow.github.io/>), which is based on PIL. You may want to read a little bit about both before starting. Tile Map Services (<http://wiki.openstreetmap.org/wiki/TMS>) is quite an important concept to get a basic grasp of. The GBIF and OpenStreetMap tiles are available behind REST services.

In our example, we will try to extract information from GBIF using the geographic coordinates of the Galápagos archipelago.

## Getting ready

You will need to install Pillow by using conda install pillow or pip install pillow.

You can find this content in the Chapter10/GBIF\_extra.ipynb Notebook.

## How to do it...

Let's take a look at the following steps:

First, let's define a function to get a map tile (a PNG image of size 256 x 256) from OpenStreetMap. We will also define a function to convert geographical coordinate systems to tile indexes, as shown in the following code:

```
import math
import requests

def get_osm_tile(x, y, z):
    url = 'http://tile.openstreetmap.org/%d/%d/%d.png' % (z, x, y)
    req = requests.get(url)
    if not req.ok:
        req.raise_for_status()
    return req

def deg_xy(lat, lon, zoom):
    lat_rad = math.radians(lat)
    n = 2 ** zoom
    x = int((lon + 180) / 360 * n)
```

```
y = int((1 - math.log(math.tan(lat_rad) + (1 /
math.cos(lat_rad))) / math.pi) / 2 * n)
return x, y
```

This code is responsible for getting a tile from the OpenStreetMap server, which is a REST service. The most complicated part to understand in this recipe is the conversion between geographical coordinates and the tiling system, such as using `deg_xy`, which converts latitude, longitude, and zoom into tile coordinates. Let's use this to make this clear.

Let's get four tiles at different zoom levels around our coordinates of interest. We will then compose a single image with Pillow (including all tiles), as follows:

```
from io import BytesIO

lat, lon = -0.666667, -90.55

pils = []
for zoom in [0, 1, 5, 8]:
    x, y = deg_xy(lat, lon, zoom)
    print(x, y, zoom)
    osm_tile = get_osm_tile(x, y, zoom)
    pil_img = PIL.Image.open(BytesIO(osm_tile.content))
    #StringIO(osm_tile.content)
    #pil_img = PIL.Image.open(osm_tile.content)
    pils.append(pil_img)
composite = PIL.Image.new('RGBA', (520, 520))
composite.paste(pils[0], (0, 0, 256, 256))
composite.paste(pils[1], (264, 0, 520, 256))
composite.paste(pils[2], (0, 264, 256, 520))
composite.paste(pils[3], (264, 264, 520, 520))
```

We will use the latitude and longitude extracted from Wikipedia for the Galápagos.

We will work at four zoom levels; `None` means the whole world. In this case, there is only a single tile with  $x = 0$  and  $y = 0$  coordinates. We then take the zoom level of 1, which has a total of 4 tiles ( $2 \times 2$ ) for the planet, and we use the tile with the coordinates  $x=0$  and  $y = 1$ . We then go to level 5, where we have 1,024 tiles ( $32 \times 32$  or  $2^{**5}$ ) with the coordinates  $x = 7$  and  $y = 16$ . Finally, we take the level 8 zoom with 65,536 tiles ( $256 \times 256$ ) with the coordinates  $x = 63$  and  $y = 128$ .

We take the four tiles, one for each zoom level, which are of  $256 \times 256$  resolution, and then use Pillow to compose an image.

Now, we will define a function to convert Pillow images to Jupyter Notebook images and display them:

```
def convert_pil(img):
    b = BytesIO()
    img.save(b, format='png')
    return display.Image(data=b.getvalue())

convert_pil(composite)
```

The following is the output:

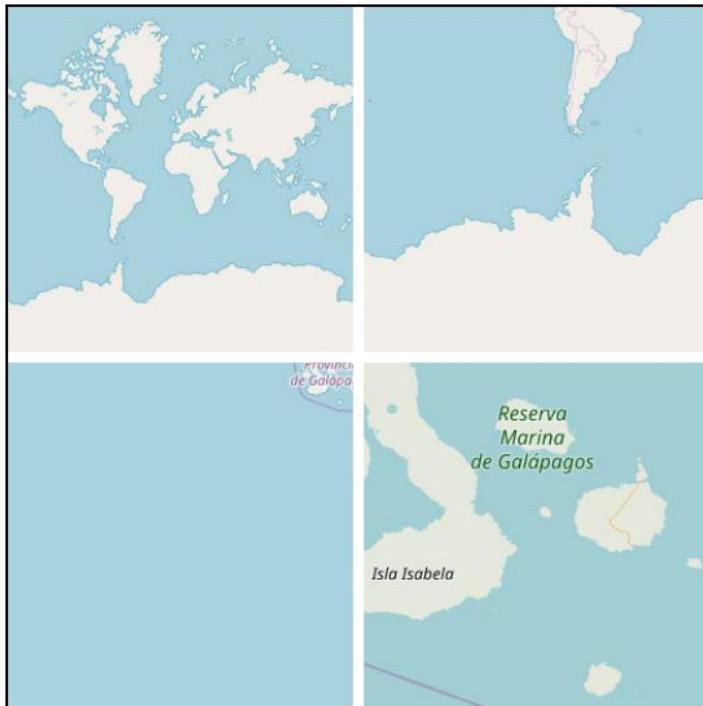


Figure 2: Zooming in on the Galápagos archipelago

Note that at zoom levels 5 and 8, it's impossible to get the whole archipelago within a single tile.



Let's define a function to get the surrounding tiles. This function will abstract away the source of all tiles. With this, we can use the same function with OpenStreetMap and other servers (such as the GBIF server). Again, we will use Pillow to join several tiles, as shown in the following code:

```
def get_surrounding(x, y, z, tile_fun):
    composite = PIL.Image.new('RGBA', (768, 768))
    for xi, x_ in enumerate([x - 1, x, x + 1]):
        for yi, y_ in enumerate([y - 1, y, y + 1]):
            tile_req = tile_fun(x_, y_, z)
            pos = (xi * 256, yi * 256, xi * 256 + 256, yi * 256 +
256)
            img = PIL.Image.open(BytesIO(tile_req.content))
            composite.paste(img, pos)
    return composite
```

Let's get the tiles for the area around the Galápagos. Therefore, we get 3 x 3 tiles in a single 768 x 768 image:

```
zoom = 8
x, y = deg_xy(lat, lon, zoom)
osm_big = get_surrounding(x, y, zoom, get_osm_tile)
```

Finally, let's get the GBIF tile. We will start by getting the worldwide tile (zoom of 0) for our bears from the preceding recipe. We will not plot these here, but you can easily see the result from the preceding code in the corresponding Notebook:

```
def get_gbif_tile(x, y, z, **kwargs):
    server = 'http://api.gbif.org/v1'
    kwargs['x'] = str(x)
    kwargs['y'] = str(y)
    kwargs['z'] = str(z)
    req = requests.get('%s/map/density/tile' % server,
                       params=kwargs,
                       headers={})
    if not req.ok:
        req.raise_for_status()
    return req

gbif_tile = get_gbif_tile(0, 0, 0, resolution='4', type='TAXON',
key='6163845')
img = PIL.Image.open(StringIO(gbif_tile.content))
```

This code is very similar to the OpenStreetMap code because they are both REST-based.

The img variable contains a Pillow representation of the 256 x 256 zoom level 0 for the GBIF of our bears. The GBIF interface allows you to constrain the result in many ways; here, we want our bear tax on ID.

It turns out that it is quite easy to query GBIF for all of the occurrences and species in our Galápagos tiles, as shown in the following code:

```
import functools
zoom = 8
x, y = deg_xy(lat, lon, zoom)
gbif_big = get_surrounding(
    x, y, zoom,
    functools.partial(get_gbif_tile, hue='0.1',
                      resolution='2', saturation='True'))
```

Note that the code here is remarkably similar to the previous application of `get_surrounding`. Indeed, the most complex piece of code has nothing to do with GBIF; we perform a partial function application to instantiate some GBIF parameters regarding visual parameters.

Now, we will use Pillow again to join the images from OpenStreetMap and GBIF:

```
compose = PIL.Image.alpha_composite(osm_big, gbif_big)
convert_pil(compose)
```

The following is the output:

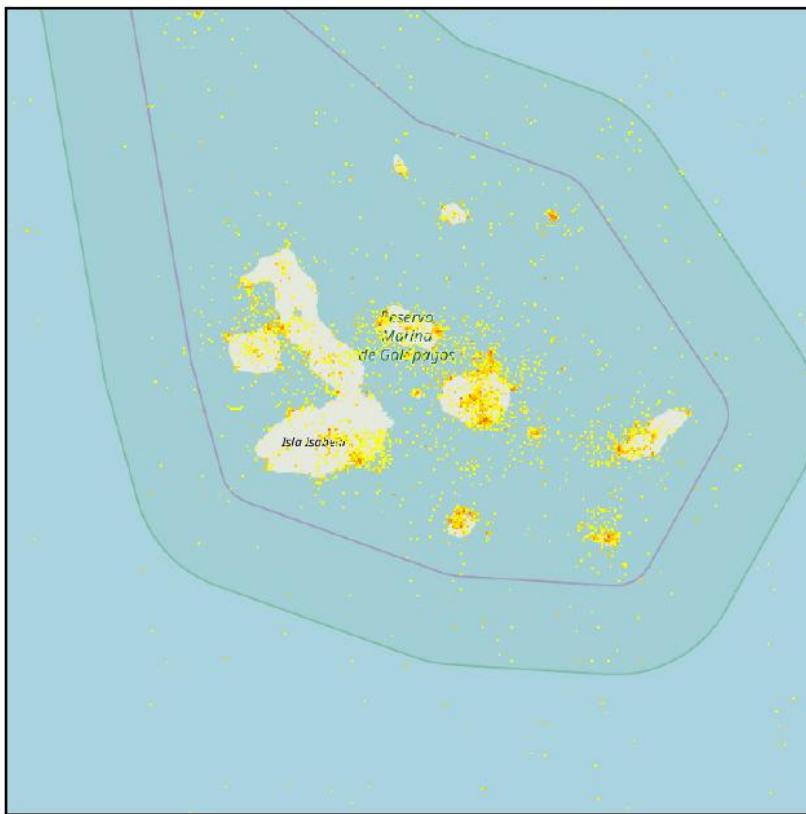


Figure 3: Overlaying species information from GBIF on top of OpenStreetMap tiles

## There's more...

It is possible to extract the occurrence records based on geographical coordinates. In the previous recipe, add a parameter geometry to your do\_request/get\_all\_records call. This should be a textual representation in a subset of well-known text ([http://en.wikipedia.org/wiki/Well-known\\_text](http://en.wikipedia.org/wiki/Well-known_text)). For details of the supported subset, refer to <http://www.gbif.org/developer/occurrence>. As an example, a rectangular area near the Galápagos area can be represented as follows:

```
start = 2, -93  
end = 1, -91
```

```
geom = 'POLYGON(({xi} {yi}, {xf} {yi}, {xf} {yf}, {xi} {yf}, {xi} {yi}))'.format(xi=start[1], xf=end[1], yi=start[0], yf=end[0])
```

If you want to find out more about tiling coordinates, refer to [http://wiki.openstreetmap.org/w/index.php?title=Slippy\\_map\\_tilenames](http://wiki.openstreetmap.org/w/index.php?title=Slippy_map_tilenames). The GBIF REST interface is documented at <http://www.gbif.org/developer/summary>. There is also a lot of documentation about OpenStreetMap; refer to its wiki link at <https://wiki.openstreetmap.org/>.

## Plotting protein interactions with Cytoscape the hard way

In this recipe, we will use Cytoscape (<http://cytoscape.org/>), which is a platform that's used for visualizing molecular interaction networks. Here, we will interact with Cytoscape using a REST interface. There are easier ways to perform this recipe, but we will take this opportunity to continue interacting with the PSICQUIC service. Also, we will exercise the NetworkX graph processing library (<https://networkx.github.io/>), which will be useful on its own.

Taking a page from Chapter 7, Using the Protein Data Bank, we will plot p53 interactions that are stored in the UniProt database.

### Getting ready

You will need to install Cytoscape version 3.6.1 (or higher), which will require Java 8 (currently, it does not work with 9). You will also need the cyREST application in Cytoscape (see the Apps main menu in Cytoscape for this). The code will use a REST interface to communicate with Cytoscape, so it will run outside it, but it will require Cytoscape to be running, so start Cytoscape with cyREST before running the following recipe.

You should also install py2cytoscape (via pip) and NetworkX. You can find this content in the Chapter10/Cytoscape.ipynb Notebook.

As this will require a massive download of software, the Chapter10/Cytoscape.ipynb Notebook will not work in our Docker implementation.

## How to do it...

Let's take a look at the following steps:

First, let's access the PSICQUIC service (its version of the UniProt database) via its REST interface, as shown in the following code:

```
import requests

def get_psicquic_uniprot(query, **kwargs):
    kwargs['format'] = kwargs.get('format', 'tab27')
    server =
        'http://www.ebi.ac.uk/Tools/webservices/psicquic/uniprot/webservice'
        's/current/search/query'
    req = requests.get('%s/%s' % (server, query), params=kwargs)
    return req.content.decode('utf-8')
```

Then, get all the genes referred to (along with their respective species) and the interactions:

```
from collections import defaultdict

genes_species = defaultdict(set)
interactions = {}

def get_gene_name(my_id, alt_names):
    toks = alt_names.split('|')
    for tok in toks:
        if tok.endswith('(gene name)'):
            return tok[tok.find(':') + 1:tok.find('(')]
    return my_id + '?' # no name...

def get_vernacular_tax(tax):
    return tax.split('|')[0][tax.find('(') + 1:-1]

def add_interactions(species):
    for rec in species.split('\n'):
        toks = rec.rstrip().split('\t')
        if len(toks) < 15:
            continue # empty line at the end
        id1 = toks[0][toks[0].find(':') + 1:]
        id2 = toks[1][toks[1].find(':') + 1:]gene1, gene2 =
get_gene_name(id1, toks[4]), get_gene_name(id2, toks[5])
        tax1, tax2 = get_vernacular_tax(toks[9]),
        get_vernacular_tax(toks[10])
        inter_type = toks[11][toks[11].find('(') + 1:-1]
        miscore = float(toks[14].split(':')[1])
```

```
genes_species[tax1].add(gene1)
genes_species[tax2].add(gene2)
interactions[((tax1, gene1), (tax2, gene2))] = {'score':
miscore, 'type': inter_type}
```

We will create a dictionary with species as a key, with a set of genes referred.

We will also create a dictionary of interactions with the key being a tuple with the genes involved, the value, the iteration type, and miscore. `add_interactions` changes a couple of external dictionaries that are in place. This dialect is not scalable in a very complex program because it is bug-prone. It works well for our short script, but be sure to adapt the code if you are using this in a larger infrastructure.

The result is output in the PSI-MI TAB 2.7 format (<https://code.google.com/p/psimi/wiki/PsimiTab27Format>). This is an easy to parse and tab-delimited format, which includes the identifiers for both iterators (including the database), aliases (for example, gene names), the interaction method, taxonomy identifiers, and so on. Here, we take aliases, species, and the confidence score for each interaction.

Let's extract interactions with the human p53 protein, which is also homologous in rats and mice. You can discover the following IDs using UniProt:

```
human = get_psiquic_uniprot('uniprotkb:P04637')
add_interactions(human)
rat = get_psiquic_uniprot('uniprotkb:P10361')
add_interactions(rat)
mouse = get_psiquic_uniprot('uniprotkb:P02340')
add_interactions(mouse)
```

With this information, we can now start drawing on Cytoscape. Remember that Cytoscape must be running on the local machine and that the cyREST plugin must be installed. First, we will construct a NetworkX graph. This graph will have extra annotations for genes and species added to the nodes and interaction types, and confidence scores added to the edges, as follows:

```
import networkx as nx

server = 'http://localhost:1234/v1'

def get_node_id(species, gene):
    if species == 'human':
        return gene
    elif species in ['mouse', 'rat']:
        return '%s (%s)' % (gene, species[0])
```

```

else:
    return '%s (%s)' % (gene, species)

graph = nx.Graph()
for species, genes in genes_species.items():
    for gene in genes:
        name = get_node_id(species, gene)
        graph.add_node(get_node_id(species, gene), species=species,
                      gene=gene)
for (i1, i2), attribs in interactions.items():
    tax1, gene1 = i1
    tax2, gene2 = i2
    graph.add_edge(get_node_id(tax1, gene1),
                  get_node_id(tax2, gene2), interaction=attribs['type'],
                  score=attribs['score'])

```

Now, let's now convert the NetworkX graph into a Cytoscape representation and plot it, as shown in the following code:

```

import json
from IPython.display import Image
from py2cytoscape.util import from_networkx

server = 'http://localhost:1234/v1'

p53_interactions = from_networkx(graph)
p53_net = requests.post(server + '/networks',
                        data=json.dumps(p53_interactions),
                        headers={'Content-Type':
                        'application/json'})
net_id = p53_net.json()['networkSUID']
requests.get('%s/apply/layouts/circular/%d' % (server, net_id))
requests.get('%s/apply/styles/Gradient1/%d' % (server, net_id))
Image('%s/networks/%d/views/first.png' % (server, net_id))

```

Here, we will use a py2cytoscape utility function to convert the NetworkX representation into a JSON version that can be sent to Cytoscape. Remember to have Cytoscape running, and the cyREST plugin installed. We will use the default network interface of cyREST here.

We will create our graph (network service with the HTTP POST method), specify a circular layout, and draw the image that Cytoscape renders, as shown in the following diagram:

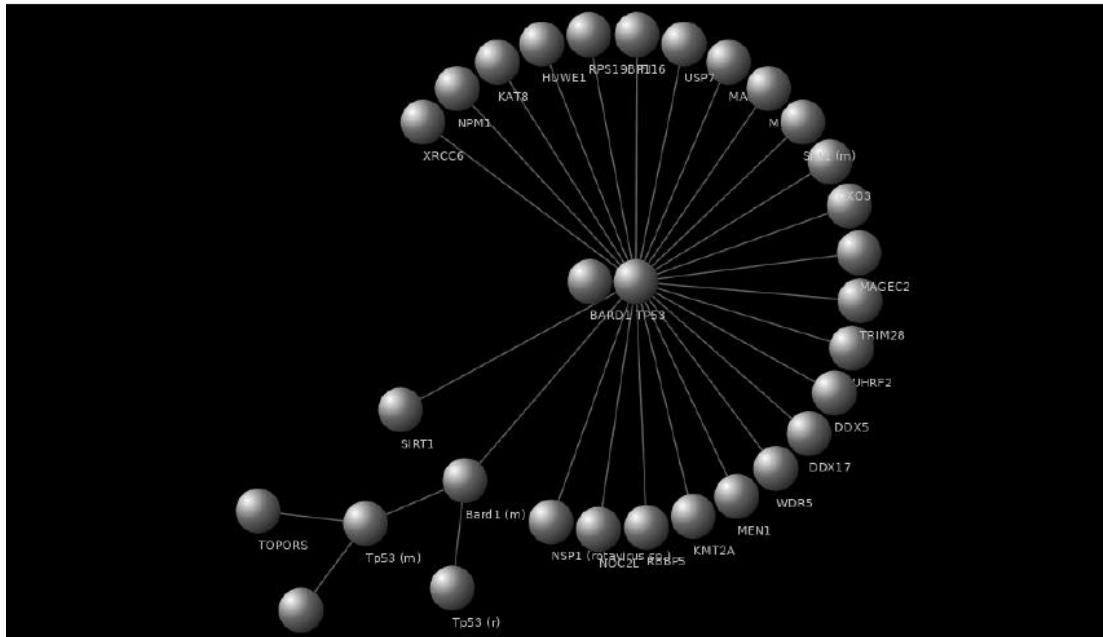


Figure 4: A first approach with Cytoscape for a p53 query on UniProt

Next, we want a different style for our plot; that is, we want the gene name (currently, it has the name and the species indicator, if not human) in the node. To color each node by species, we define a style, as follows:

```
ustyle = {
    'title': 'Color style',
    'mappings': [
        {'mappingType': 'discrete',
         'map': [
             {'key': 'human', 'value': '#00FF00'},
             {'key': 'rat', 'value': '#FF00FF'},
             {'key': 'mouse', 'value': '#00FFFF'}],
         'visualProperty': 'NODE_FILL_COLOR',
         'mappingColumnType': 'String',
         'mappingColumn': 'species'},
        {
            'mappingType': 'passthrough',
            'visualProperty': 'NODE_LABEL',
            'mappingColumnType': 'String',
            'mappingColumn': 'gene'},
        {
            'mappingType': 'passthrough'}
```

```

    'visualProperty': 'EDGE_TOOLTIP',
    'mappingColumnType': 'String',
    'mappingColumn': 'interaction'
  ],
  'defaults': [ {"visualProperty": "NODE_FILL_COLOR",
    "value": "#FFFFFF"} ]}

```

There are quite a few existing styles, but here, we will color our nodes as a function of the species, labeling a node with the gene name.

Finally, we will apply our style, change the layout (do not confuse the graph layout with the graph style), and plot our new version, as follows:

```

res = requests.post(server + "/styles", data=json.dumps(ustyle),
                    headers={'Content-Type': 'application/json'})

requests.get('%s/apply/layouts/force-directed/%d' % (server,
net_id))
res = requests.get('%s/apply/styles/Color style/%d' % (server,
net_id),
headers={'Content-Type': 'application/json'})
Image('%s/networks/%s/views/first.png' % (server, net_id))

```

The following is the output:

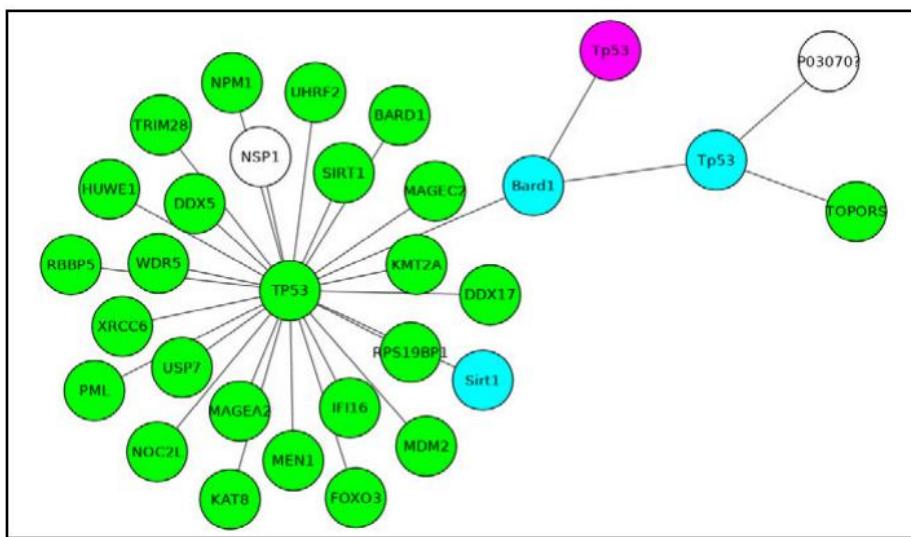


Figure 5: Our p53 network with a tailored style and a force-directed layout

## There's more...

The preceding example was designed to exercise more than the REST interface to Cytoscape; we interfaced with the PSICQUIC server and used the NetworkX graph library, a functionality that can be useful without Cytoscape. To start with, there is plenty of documentation for NetworkX and Cytoscape available on their web pages (<https://cytoscape.org/>, <https://networkx.github.io/>). The Kyoto Encyclopedia of Genes and Genomes (KEGG) is one of the most useful resources that you can use with Cytoscape; as this is very well-documented, we opted for a less common example based on UniProt. A fantastic piece of documentation for interacting with Cytoscape via REST interfaces using IPython, including a KEGG example, is available at <http://nbviewer.ipython.org/github/kidekerlab/cy-rest-python/blob/develop/index.ipynb>.

If drawing KEGG pathways is all that you need to , there is a much lighter solution than having to install and use Cytoscape, which is Biopython. You can find the KEGG module documentation at [http://nbviewer.ipython.org/github/widdowquinn/notebooks/blob/master/Biopython\\_KGML\\_intro.ipynb](http://nbviewer.ipython.org/github/widdowquinn/notebooks/blob/master/Biopython_KGML_intro.ipynb).

# 11

# Advanced NGS Processing

In this chapter, we will cover the following recipes:

- Preparing the dataset for analysis
- Using Mendelian error information for quality control
- Using decision trees to further explore the data
- Exploring the data with standard statistics
- Finding genomic features from sequencing annotations

## Introduction

If you work with next-generation sequencing (NGS) data, you know that quality analysis and processing are two of the great time-sinks in getting results.

In this chapter, we will delve deeper into NGS analysis by using a dataset that includes information about relatives; in our case, a mother, a father, and around 20 offspring. This is a common technique for performing quality analysis, as pedigree information will allow us to make inferences on the amount of errors that our filtering rules might produce. We will be using HDF5 representing VCF files. We also introduce a bit more of NumPy and pandas in this chapter.

## Preparing the dataset for analysis

Our starting point will be a VCF file (or equivalent), with calls made by a genotyper (Genome Analysis Toolkit (GATK) in our case), including the annotations. As we will be filtering NGS data, we need reliable decision criteria to call a site. So, how do we get that information? Generally, we can't, but if we need to do it, there are three basic approaches:

- Using a more robust sequencing technology for comparison; for example, using Sanger sequencing to verify NGS datasets. This is cost-prohibitive and can only be done for a few loci.
- Sequencing closely related individuals, for example, two parents and their offspring. In this case, we use Mendelian inheritance rules to devise if a certain call is acceptable or not. This was the strategy used by both the human and Anopheles 1,000 Genomes Projects.
- Finally, we can use simulations. This setup is not only quite complex, but also of dubious reliability. It's more of a theoretical option.

In this chapter, we will use the second option based on the Anopheles 1,000 Genomes Project. This project makes available information based on crosses between mosquitoes. A cross will include the parents (mother and father) and up to 20 offspring.

In this recipe, we are going to prepare our data for usage in the next recipes.

## Getting ready

We will download our files in HDF5 format for faster processing. Please be advised that these files are quite big; you will need a good network connection and plenty of disk space:

```
wget -c  
ftp://ngs.sanger.ac.uk/production/ag1000g/phase1/AR3/variation/main/hdf5/ag  
1000g.phase1.ar3.pass.3L.h5  
wget -c  
ftp://ngs.sanger.ac.uk/production/ag1000g/phase1/AR3/variation/main/hdf5/ag  
1000g.phase1.ar3.pass.2L.h5
```

The files have four crosses with around 20 offspring each. We will use chromosome arms 3L and 2L. At this stage, we also compute Mendelian errors (a subject of the next recipe, so we will delay a detailed discussion until then).

The relevant notebook is Chapter11/Preparation.ipynb. There is also a local sample metadata file in the directory called samples.tsv.

## How to do it...

After downloading the data, follow these steps:

First, start with a few imports:

```
import pickle
import gzip
import random
import numpy as np
import h5py
import pandas as pd
```

Let's get the sample metadata:

```
samples = pd.read_csv('samples.tsv', sep='\t')
print(len(samples))
print(samples['cross'].unique())
print(samples[samples['cross'] == 'cross-29-2'][['id',
'function']])
print(len(samples[samples['cross'] == 'cross-29-2']))
print(samples[samples['function'] == 'parent'])
```

We also print some basic information about the cross we are going to use, and all the parents.

We prepare to deal with chromosome arm 3L based on its HDF5 file:

```
h5_3L = h5py.File('ag1000g.crosses.phase1.ar3sites.3L.h5', 'r')
samples_hdf5 = list(map(lambda sample: sample.decode('utf-8'),
h5_3L['/3L/samples']))

calldata_genotype = h5_3L['/3L/calldata/genotype']

MQ0 = h5_3L['/3L/variants/MQ0']
MQ = h5_3L['/3L/variants/MQ']
QD = h5_3L['/3L/variants/QD']
Coverage = h5_3L['/3L/variants/Coverage']
CoverageMQ0 = h5_3L['/3L/variants/CoverageMQ0']
HaplotypeScore = h5_3L['/3L/variants/HaplotypeScore']
QUAL = h5_3L['/3L/variants/QUAL']
FS = h5_3L['/3L/variants/FS']
DP = h5_3L['/3L/variants/DP']
HRun = h5_3L['/3L/variants/HRun']
ReadPosRankSum = h5_3L['/3L/variants/ReadPosRankSum']
my_features = {
    'MQ': MQ,
    'QD': QD,
```

```

'Coverage': Coverage,
'HaplotypeScore': HaplotypeScore,
'QUAL': QUAL,
'FS': FS,
'DP': DP,
'HRun': HRun,
'ReadPosRankSum': ReadPosRankSum
}

num_features = len(my_features)
num_alleles = h5_3L['3L/variants/num_alleles']
is_snp = h5_3L['3L/variants/is_snp']
POS = h5_3L['3L/variants/POS']

```

The code to compute Mendelian errors is the following:

```

#compute mendelian errors (biallelic)
def compute_mendelian_errors(mother, father, offspring):
    num_errors = 0
    num_ofs_problems = 0
    if len(mother.union(father)) == 1:
        # Mother and father are homo and the same
        for ofs in offspring:
            if len(ofs) == 2:
                # Offspring is het
                num_errors += 1
                num_ofs_problems += 1
            elif len(ofs.intersection(mother)) == 0:
                # Offspring is homo, but opposite from parents
                num_errors += 2
                num_ofs_problems += 1
            elif len(mother) == 1 and len(father) == 1:
                # Mother and father are homo and different
                for ofs in offspring:
                    if len(ofs) == 1:
                        # Homo, should be het
                        num_errors += 1
                        num_ofs_problems += 1
            elif len(mother) == 2 and len(father) == 2:
                # Both are het, individual offspring can be anything
                pass
    else:
        # One is het, the other is homo
        homo = mother if len(mother) == 1 else father
        for ofs in offspring:
            if len(ofs) == 1 and ofs.intersection(homo):
                # homo, but not including the allele from parent
                that is homo

```

```

        num_errors += 1
        num_ofs_problems += 1
    return num_errors, num_ofs_problems

```

We will discuss this in the next recipe, Using Mendelian error information for quality control.

We now define a support generator and function to select acceptable positions and accumulate basic data:

```

def acceptable_position_to_genotype():
    for i, genotype in enumerate(calldata_genotype):
        if is_snp[i] and num_alleles[i] == 2:
            if len(np.where(genotype == -1)[0]) > 1:
                # Missing data
                continue
            yield i

def accumulate(fun):
    accumulator = {}
    for res in fun():
        if res is not None:
            accumulator[res[0]] = res[1]
    return accumulator

```

We now need to find the indexes of our cross (mother, father and 20 offspring) on the HDF5 file:

```

def get_family_indexes(samples_hdf5, cross_pd):
    offspring = []
    for i, individual in cross_pd.T.iteritems():
        index = samples_hdf5.index(individual.id)
        if individual.function == 'parent':
            if individual.sex == 'M':
                father = index
            else:
                mother = index
        else:
            offspring.append(index)
    return {'mother': mother, 'father': father, 'offspring': offspring}

cross_pd = samples[samples['cross'] == 'cross-29-2']
family_indexes = get_family_indexes(samples_hdf5, cross_pd)

```

Finally, we will actually compute Mendelian errors and save them to disk:

```

mother_index = family_indexes['mother']
father_index = family_indexes['father']
offspring_indexes = family_indexes['offspring']
all_errors = {}

def get_mendelian_errors():
    for i in acceptable_position_to_genotype():
        genotype = calldata_genotype[i]
        mother = set(genotype[mother_index])
        father = set(genotype[father_index])
        offspring = [set(genotype[ofs_index]) for ofs_index in
                    offspring_indexes]
        my_mendelian_errors = compute_mendelian_errors(mother,
                                                        father, offspring)
        yield POS[i], my_mendelian_errors

mendelian_errors = accumulate(get_mendelian_errors)

pickle.dump(mendelian_errors,
            gzip.open('mendelian_errors.pickle.gz', 'wb'))

```

We will now generate an efficient NumPy array with annotations and Mendelian error information:

```

ordered_positions = sorted(mendelian_errors.keys())
ordered_features = sorted(my_features.keys())
num_features = len(ordered_features)
feature_fit = np.empty((len(ordered_positions), len(my_features) +
2), dtype=float)

for column, feature in enumerate(ordered_features): # 'Strange'
    order
        print(feature)
        current_hdf_row = 0
        for row, genomic_position in enumerate(ordered_positions):
            while POS[current_hdf_row] < genomic_position:
                current_hdf_row += 1
                feature_fit[row, column] =
my_features[feature][current_hdf_row]

for row, genomic_position in enumerate(ordered_positions):
    feature_fit[row, num_features] = genomic_position
    feature_fit[row, num_features + 1] = 1 if
mendelian_errors[genomic_position][0] > 0 else 0

```

```
np.save(gzip.open('feature_fit.npy.gz', 'wb'), feature_fit,
allow_pickle=False, fix_imports=False)
pickle.dump(ordered_features, open('ordered_features', 'wb'))
```

Buried in this code is one of the most important decisions of the whole chapter: how do we weigh Mendelian errors? In our case, we only store a 1 if there is any kind of error, and 0 if there is none.

Changing gears, let's extract some information from chromosome arm 2L now:

```
h5_2L = h5py.File('ag1000g.crosses.phase1.ar3sites.2L.h5', 'r')
samples_hdf5 = list(map(lambda sample: sample.decode('utf-8'),
h5_2L['/2L/samples']))
calldata_DP = h5_2L['/2L/calldata/DP']
POS = h5_2L['/2L/variants/POS']
```

Here, we are only interested in the parents:

```
def get_parent_indexes(samples_hdf5, parents_pd):
    parents = []
    for i, individual in parents_pd.T.iteritems():
        index = samples_hdf5.index(individual.id)
        parents.append(index)
    return parents

parents_pd = samples[samples['function'] == 'parent']
parent_indexes = get_parent_indexes(samples_hdf5, parents_pd)
```

We'll extract the sample DP for each parent:

```
all_dps = []
for i, pos in enumerate(POS):
    if random.random() > 0.01:
        continue
    pos_dp = calldata_DP[i]
    parent_pos_dp = [pos_dp[parent_index] for parent_index in
parent_indexes]
    all_dps.append(parent_pos_dp + [pos])
all_dps = np.array(all_dps)
np.save(gzip.open('DP_2L.npy.gz', 'wb'), all_dps,
allow_pickle=False, fix_imports=False)
```

# Using Mendelian error information for quality control

So, how can we infer the quality of calls using Mendelian inheritance rules? Let's look at expectations for different genotypical configurations of the parents:

- For a certain potential bi-allelic SNP, if the mother is AA and the father is also AA, then all offspring will be AA.
- If the mother is AA and the father TT, then all offspring will have to be heterozygous (AT). They always get an A from the mother and they always get a T from the father.
- If the mother is AA and the father is AT, then offspring can be either AA or AT. They always get the A from the mother, but they can get either an A or a T from the father.
- If the both the mother and the father are heterozygous (AT), then the offspring can be anything. In theory, there is not much we can do here.

In practice, we can ignore mutations, which is safe to do with most eukaryotes. The number of mutations (noise, from our perspective) is several orders of magnitude lower than the signal we are looking for.

In this recipe, we are going to do a small theoretical study of the distribution and Mendelian errors, and further process the data for downstream analysis based on errors. The relevant Notebook file is Chapter11/Mendel.ipynb.

## How to do it...

We will need a few imports:

```
import random
import matplotlib.pyplot as plt
%matplotlib inline
```

Before we do any empirical analysis, let's try to understand what information we can extract from the case where the mother is AA and the father is AT. Let's answer the question, "If we have 20 offspring, what is the probability of all of them being heterozygous?":

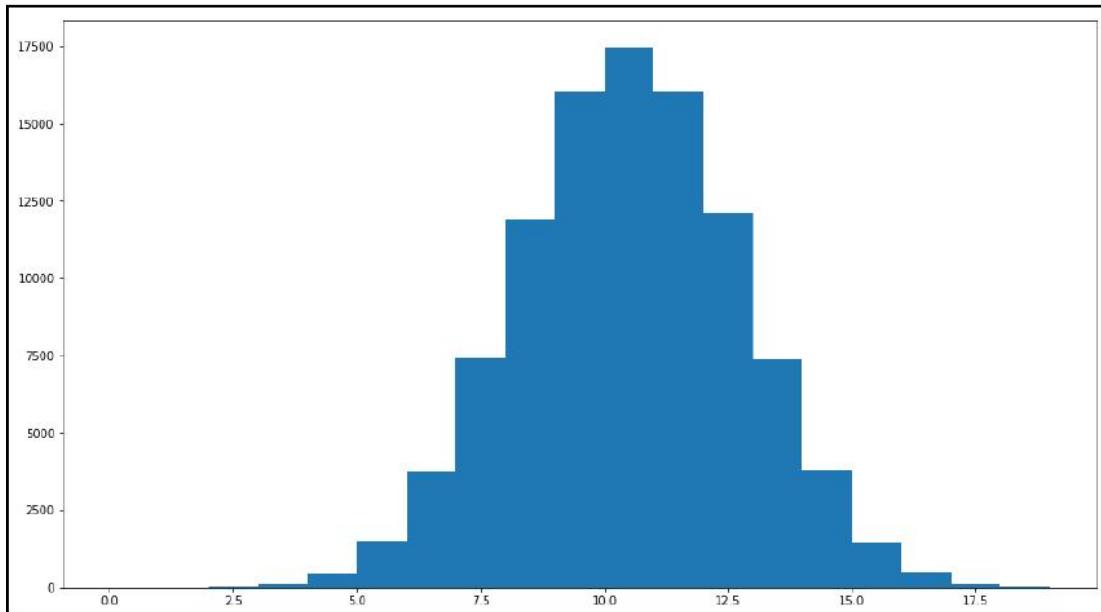
```
num_sims = 100000
num_ofs = 20
num_hets_AA_AT = []
```

```

for sim in range(num_sims):
    sim_hets = 0
    for ofs in range(20):
        sim_hets += 1 if random.choice([0, 1]) == 1 else 0
    num_hets_AA_AT.append(sim_hets)
fig, ax = plt.subplots(1, 1, figsize=(16,9))
ax.hist(num_hets_AA_AT, bins=range(20))
print(len([num_hets for num_hets in num_hets_AA_AT if num_hets==20]))

```

We get the following output:



Results from 100,000 simulations: the number of offspring that are heterozygous for a certain loci where the mother is AA and the father is heterozygous

Here, we have done 100,000 simulations. In my case (this is stochastic, so your result might vary), I got exactly zero simulations where all offspring were heterozygous. Indeed, these are permutations with repetition, so the probability of all being heterozygous is  $\frac{1}{2^{20}}$  or  $9.5367431640625e-07$ —not very likely. So, even if for a single offspring, we can have AT or AA; for twenty, it is very unlikely that all of them are of the same type. This is information we can use for a less naive interpretation of Mendelian errors.

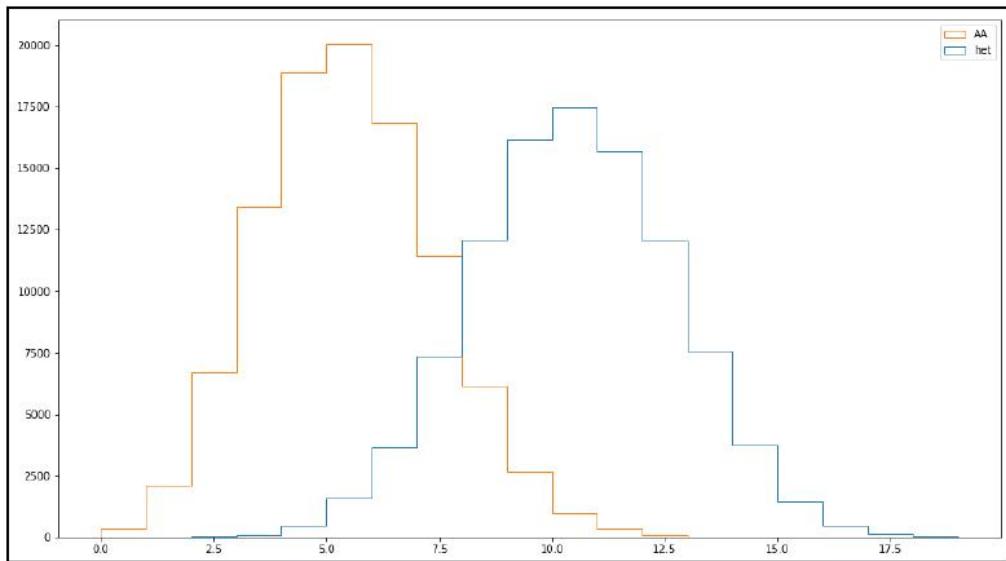
Let's repeat the analysis where the mother and the father are both AT:

```

num_AAs_AT_AT = []
num_hets_AT_AT = []
for sim in range(num_sims):
    sim_AAs = 0
    sim_hets = 0
    for ofs in range(20):
        derived_cnt = sum(random.choices([0, 1], k=2))
        sim_AAs += 1 if derived_cnt == 0 else 0
        sim_hets += 1 if derived_cnt == 1 else 0
    num_AAs_AT_AT.append(sim_AAs)
    num_hets_AT_AT.append(sim_hets)
fig, ax = plt.subplots(1,1, figsize=(16,9))
ax.hist([num_hets_AT_AT, num_AAs_AT_AT], histtype='step',
fill=False, bins=range(20), label=['het', 'AA'])
plt.legend()

```

The output is as follows:



Results from 100,000 simulations: the number of offspring that are AA or heterozygous for a certain loci where both parents are also heterozygous

In this case, we have also permutations with repetition, but we have four possible values, not two: AA, AT, TA, and TT. We end up with the same probability for all individuals being AT: 9.5367431640625e-07. It's even worse (twice as bad, in fact) for all of them being homozygous of the same type (all TT or all AA).

OK, after this probabilistic prelude, let's get down to more data-moving stuff. The first thing that we will do is check how many errors we have. Let's load the data from the previous recipe:

```
import gzip
import pickle
import random

import numpy as np

mendelian_errors =
pickle.load(gzip.open('mendelian_errors.pickle.gz', 'rb'))
feature_fit = np.load(gzip.open('feature_fit.npy.gz', 'rb'))
ordered_features = np.load(open('ordered_features', 'rb'))
num_features = len(ordered_features)
```

Let's see how many errors we have:

```
print(len(mendelian_errors), len(list(filter(lambda x: x[0] >
0,mendelian_errors.values()))))
```

Not many of the calls have Mendelian errors—only around 5%, great.

Let's create a balanced set where roughly half of the set has errors. For that, we will randomly drop a lot of good calls. First, we compute the fraction of errors:

```
total_observations = len(mendelian_errors)
error_observations = len(list(filter(lambda x: x[0] >
0,mendelian_errors.values())))
ok_observations = total_observations - error_observations
fraction_errors = error_observations/total_observations
print (total_observations, ok_observations, error_observations,
100*fraction_errors)
del mendelian_errors
```

We use that information to get a set of accepted entries: all the errors plus an approximately equal quantity of ok calls. We print the number of entries at the end (this will vary as the ok list is stochastic):

```
prob_ok_choice = error_observations / ok_observations

def accept_entry(row):
    if row[-1] == 1:
        return True
    return random.random() <= prob_ok_choice

accept_entry_v = np.vectorize(accept_entry, signature='(i)->()')
```

```
accepted_entries = accept_entry_v(feature_fit)
balanced_fit = feature_fit[accepted_entries]
del feature_fit
balanced_fit.shape
len([x for x in balanced_fit if x[-1] == 1]), len([x for x in
balanced_fit if x[-1] == 0])
```

Finally, we save it:

```
np.save(gzip.open('balanced_fit.npy.gz', 'wb'), balanced_fit,
allow_pickle=False, fix_imports=False)
```

## There's more...

With regards to Mendelian errors and their impact on cost functions, let's think about the following case: the mother is AA, the father is AT, and all offspring are AA. Does this mean that the father is wrongly called, or that we failed to detect a few heterozygous offspring? From this reasoning, it's probably the father that is wrongly called. This has an impact in terms of some more refined Mendelian error estimation functions: it's probably more costly to have a few offspring wrong, than just a single sample (the father) wrong. In this case, you might think it's trivial (the probability of having no heterozygous offspring is so low that it's probably the father), but if you have 18 offspring AA and two AT, is it still "trivial"? This is not just a theoretical problem, because it severely impacts the design of a proper cost function.

Our function in a previous recipe, Preparing the dataset for analysis, is naive, but is enough for the level of refinement that will allow us to have some interesting results further down the road.

## Using decision trees to explore the data

We are now ready to start exploring the data with the objective of finding some rules on how to filter it. Because we have a lot of annotations to explore (in our case, we reduced them, but generally, that would be the case), we need to find a place to start. It can be daunting to go out on a blind fishing expedition. My personal preference for a first approach is using a machine learning technique called decision trees. Decision trees will suggest what the fundamental annotations segregating the data in correct and error calls are. Another advantage of decision trees is that they barely need any data preparation, as opposed to many other machine learning techniques.

## How to do it...

We start with a few imports, most notably of scikit-learn:

```
import gzip
import pickle

import numpy as np
import graphviz
from sklearn import tree
```

Let's load the data and split it into inputs and outputs:

```
balanced_fit = np.load(gzip.open('balanced_fit.npy.gz', 'rb'))
ordered_features = pickle.load(open('ordered_features', 'rb'))

train_X = balanced_fit[:, :-2] #POS and errors
train_Y = balanced_fit[:, -1]
```

We call the decision tree algorithm:

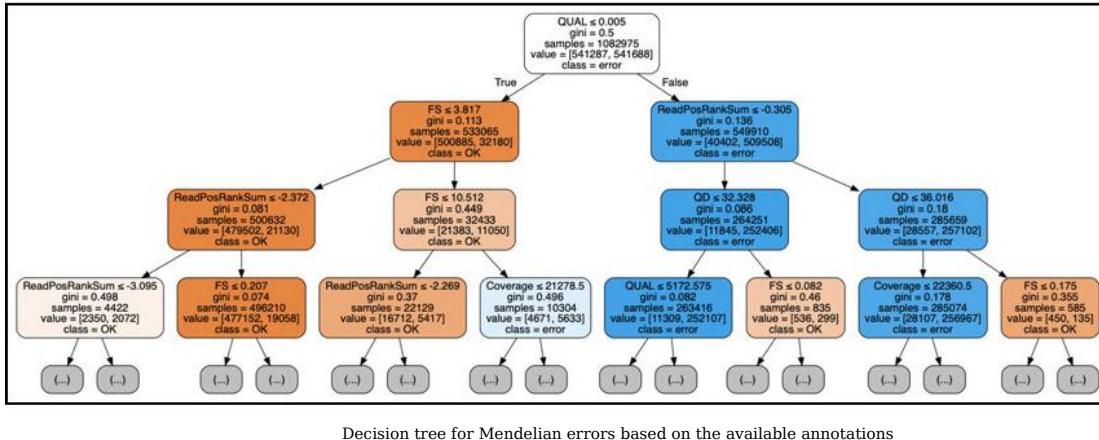
```
estimator = tree.DecisionTreeClassifier(max_depth=4)
tree_fit = estimator.fit(train_X, train_Y)
print(tree_fit.feature_importances_)
```

We also print the feature\_importances\_, which is suggesting that QUAL is the most important.

Let's draw a tree:

```
graphviz_representation = tree.export_graphviz(tree_fit,
out_file=None,
          max_depth=3,
feature_names=ordered_features,
          class_names = ['OK',
'error'],
          filled=True,
rounded=True, special_characters=True)
graph = graphviz.Source(graphviz_representation)
graph
```

The following is the output generated:



What would happen if we remove QUAL? Let's rerun without it:

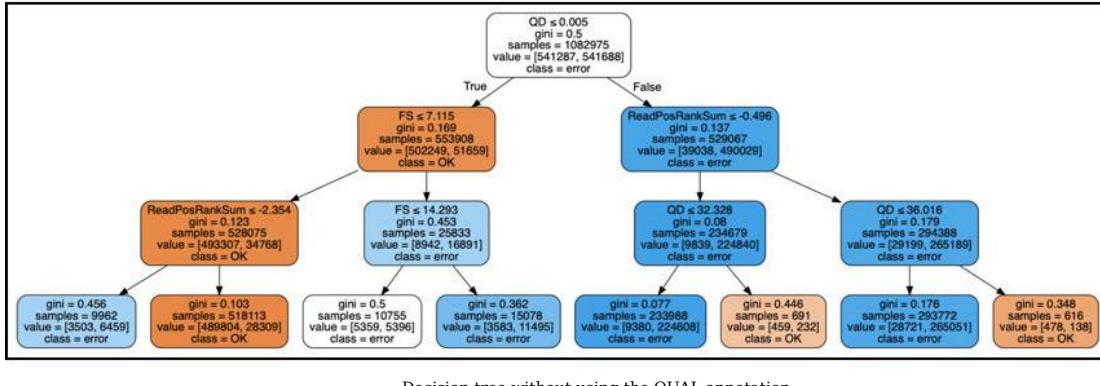
```
ordered_features2 = list(ordered_features)
del ordered_features2[-2]
train_X2 = balanced_fit[:, :-4]
train_X2 = np.concatenate((train_X2, balanced_fit[:, -3:-2]), 1)
ordered_features2
```

Then call the algorithm again:

```
estimator = tree.DecisionTreeClassifier(max_depth=3)
tree_fit2 = estimator.fit(train_X2, train_Y)
```

And the output is as follows:

```
graphviz_representation = tree.export_graphviz(tree_fit2,
out_file=None,
max_depth=3,
feature_names=ordered_features2,
class_names=['OK',
'error'],
filled=True,
rounded=True, special_characters=True)
graph = graphviz.Source(graphviz_representation)
graph
```



QD now becomes important where it wasn't before. When you remove an important variable, some other variable that was "hidden" might become important. This actually also suggests a correlation between QAL and QD.

## Exploring the data with standard statistics

Now that we have a compass from the decision tree, let's explore the data in order to get more insights that might help us to better filter the data. You can find this content in Chapter11/Exploration.ipynb.

## How to do it...

We start, as usual, with the necessary imports:

```

import gzip
import pickle
import random

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from pandas.plotting import scatter_matrix

%matplotlib inline
  
```

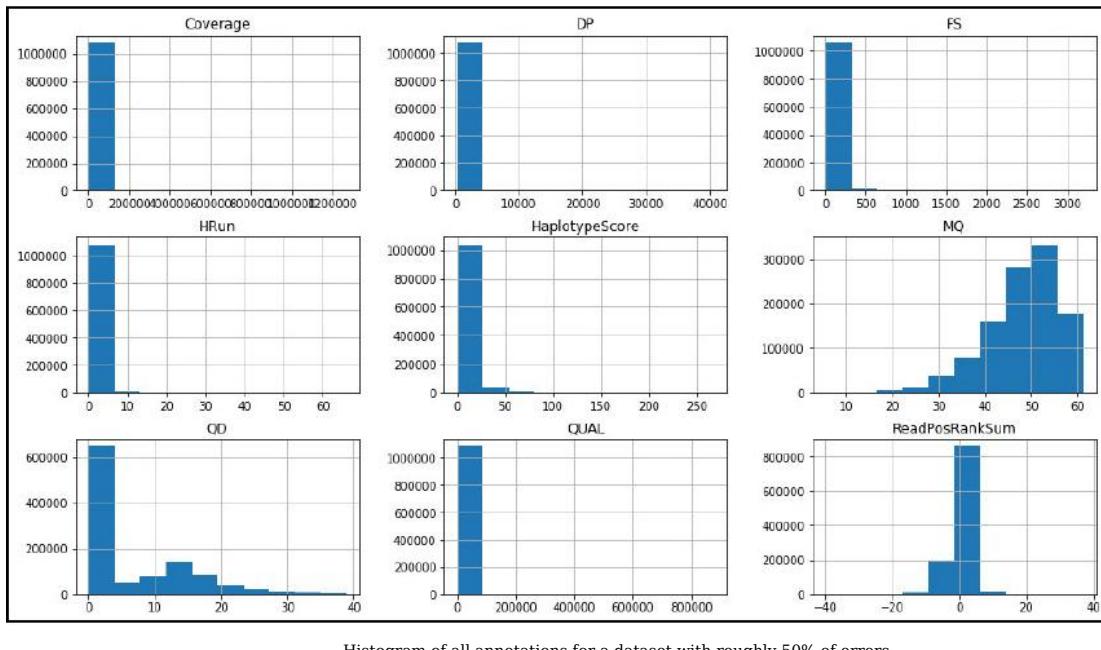
Then we load the data. We will use pandas to navigate it:

```
fit = np.load(gzip.open('balanced_fit.npy.gz', 'rb'))
ordered_features = np.load(open('ordered_features', 'rb'))
num_features = len(ordered_features)
fit_df = pd.DataFrame(fit, columns=ordered_features + ['pos',
'error'])
num_samples = 80
del fit
```

Let's ask pandas to show an histogram of all annotations:

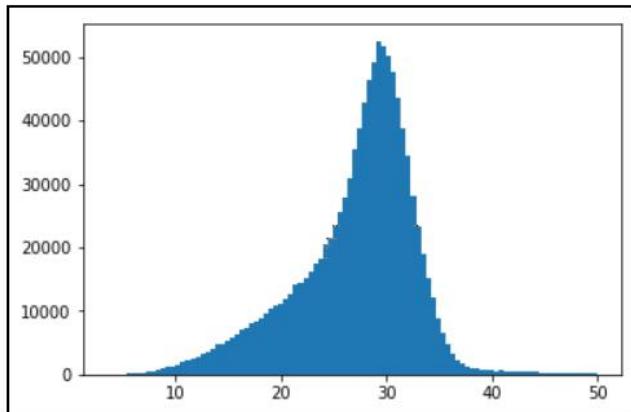
```
fig,ax = plt.subplots(figsize=(16,9))
fit_df.hist(column=ordered_features, ax=ax)
```

The following histogram is generated:



For some annotations, we do not get interesting information. We can try to zoom in, for example, with DP:

```
fit_df['MeanDP'] = fit_df['DP'] / 80
fig, ax = plt.subplots()
_ = ax.hist(fit_df[fit_df['MeanDP'] < 50]['MeanDP'], bins=100)
```



Histogram zooming in on an area of interest for DP

We are actually dividing DP by the number of samples in order to get a more meaningful number.

We will split the dataset in two, one for the errors and the other for the positions with no Mendelian errors:

```
errors_df = fit_df[fit_df['error'] == 1]
ok_df = fit_df[fit_df['error'] == 0]
```

From the decision tree, which suggests QUAL is important, let's split on 0.005 (the value the tree suggests), and check how we get errors and correct calls split:

```
ok_qual_above_df = ok_df[ok_df['QUAL'] > 0.005]
errors_qual_above_df = errors_df[errors_df['QUAL'] > 0.005]
print(ok_df.size, errors_df.size, ok_qual_above_df.size,
      errors_qual_above_df.size)
print(ok_qual_above_df.size / ok_df.size, errors_qual_above_df.size
      / errors_df.size)
```

Clearly, ['QUAL'] > 0.005 gets lots of errors, while not getting lots of ok positions. This is positive, as we have hope to filter it.

From the second decision tree, let's do the same with QD:

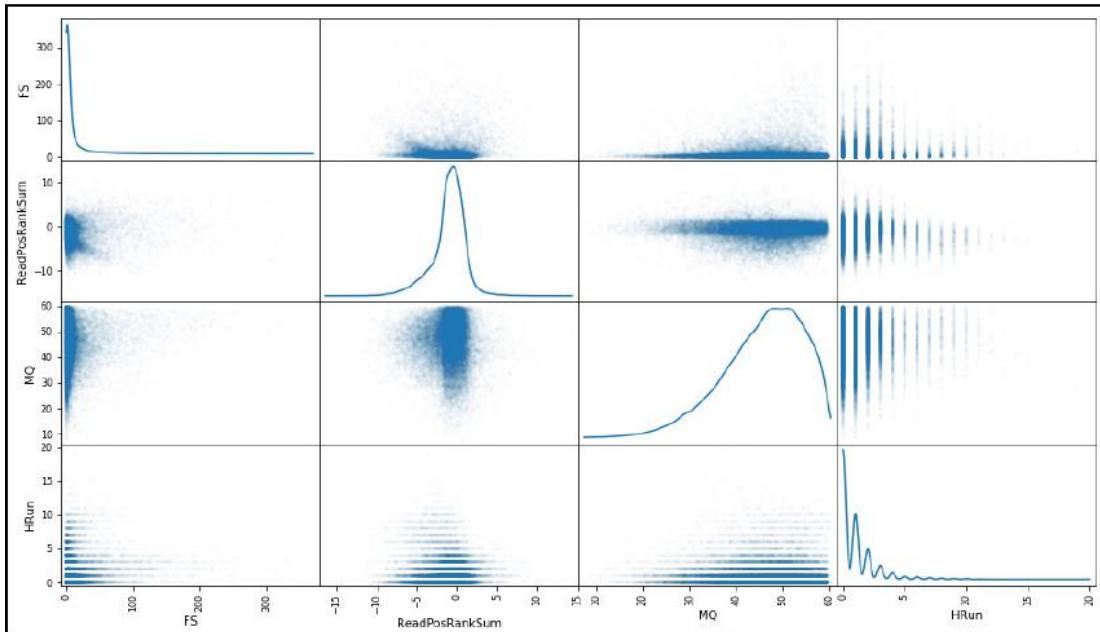
```
ok_qd_above_df = ok_df[ok_df['QD'] > 0.05]
errors_qd_above_df = errors_df[errors_df['QD'] > 0.05]
print(ok_df.size, errors_df.size, ok_qd_above_df.size,
      errors_qd_above_df.size)
print(ok_qd_above_df.size / ok_df.size, errors_qd_above_df.size /
      errors_df.size)
```

Again, we have some interesting results.

Let's take an area where there are less errors, and study the relationships between annotations on errors. We will plot annotations pairwise:

```
not_bad_area_errors_df =
    errors_df[(errors_df['QUAL'] < 0.005) & (errors_df['QD'] < 0.05)]
    = scatter_matrix(not_bad_area_errors_df[['FS', 'ReadPosRankSum',
'MQ', 'HRun']], diagonal='kde', figsize=(16, 9), alpha=0.02)
```

The preceding code generates the following output:



Scatter matrix of annotations of errors for an area of the search space

And now do the same on the good calls:

```
not_bad_area_ok_df =
ok_df[(ok_df['QUAL']<0.005)&(ok_df['QD']<0.05)]
= scatter_matrix(not_bad_area_ok_df[['FS', 'ReadPosRankSum',
'MQ', 'HRun']], diagonal='kde', figsize=(16, 9), alpha=0.02)
```

The output is as follows:

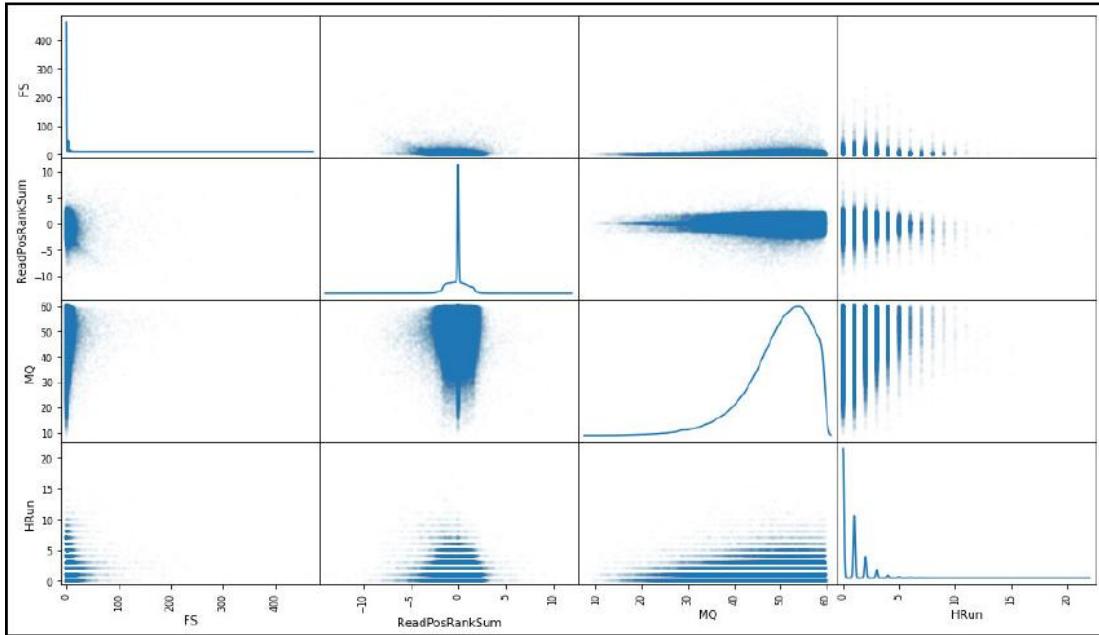


Figure: Scatter matrix of annotations of good calls for an area of the search space

Finally, let's see how our rules would perform on the complete dataset (remember that we are using a dataset roughly composed of 50% errors and 50% ok calls):

```
all_fit_df = pd.DataFrame(np.load(gzip.open('feature_fit.npy.gz',
'rb')), columns=ordered_features + ['pos', 'error'])
potentially_good_corner_df =
all_fit_df[(all_fit_df['QUAL']<0.005)&(all_fit_df['QD']<0.05)]
all_errors_df=all_fit_df[all_fit_df['error'] == 1]
print(len(all_fit_df), len(all_errors_df), len(all_errors_df) /
len(all_fit_df))
```

We also remind ourselves that there are roughly 10.9 million markers in our full dataset, with around 5% errors.

Let's get some statistics on our good\_corner:

```
potentially_good_corner_errors_df =  
potentially_good_corner_df[potentially_good_corner_df['error'] ==  
1]  
print(len(potentially_good_corner_df),  
len(potentially_good_corner_errors_df),  
len(potentially_good_corner_errors_df) /  
len(potentially_good_corner_df))  
print(len(potentially_good_corner_df)/len(all_fit_df))
```

So we reduce the error rate to 0.33% (from 5%), while having only 9.6 million markers.

## There's more...

Is a reduction in error from 5% to 0.3% while losing 12% of markers good or bad? Well, it depends on what analysis you want to do next. Maybe your method is resilient to loss of markers but not to many errors, in which case this might help. But if it is the other way around, maybe you prefer to have the complete dataset even if it has more errors. If you apply different methods, maybe you will use different datasets from method to method. In the specific case of this Anopheles dataset, there is so much data that reducing the size will probably be fine for almost anything. But if you have fewer markers, you will have to assess your needs in terms of markers and quality.

## Finding genomic features from sequencing annotations

We will finalize this chapter and this book with a simple recipe that suggests that sometimes you can learn important things from simple unexpected results, and that apparent quality issues might mask important biological questions.

We will plot read depth (DP) across chromosome arm 2L for all the parents on our crosses. The recipe can be found on Chapter11/2L.ipynb.

## How to do it...

Let's start with the usual imports:

```
%matplotlib inline  
from collections import defaultdict  
import gzip  
  
import numpy as np  
import matplotlib.pyplot as plt
```

Let's load the data that we saved on the first recipe:

```
num_parents = 8  
dp_2L = np.load(gzip.open('DP_2L.npy.gz', 'rb'))  
print(dp_2L.shape)
```

And let's print median DP for the whole chromosome arm, and a part of it in the middle for all parents:

```
for i in range(num_parents):  
    print(np.median(dp_2L[:,i]), np.median(dp_2L[50000:150000,i]))
```

Interestingly, the median for the whole chromosome sometimes does not hold for that big region in the middle, so let's dig further.

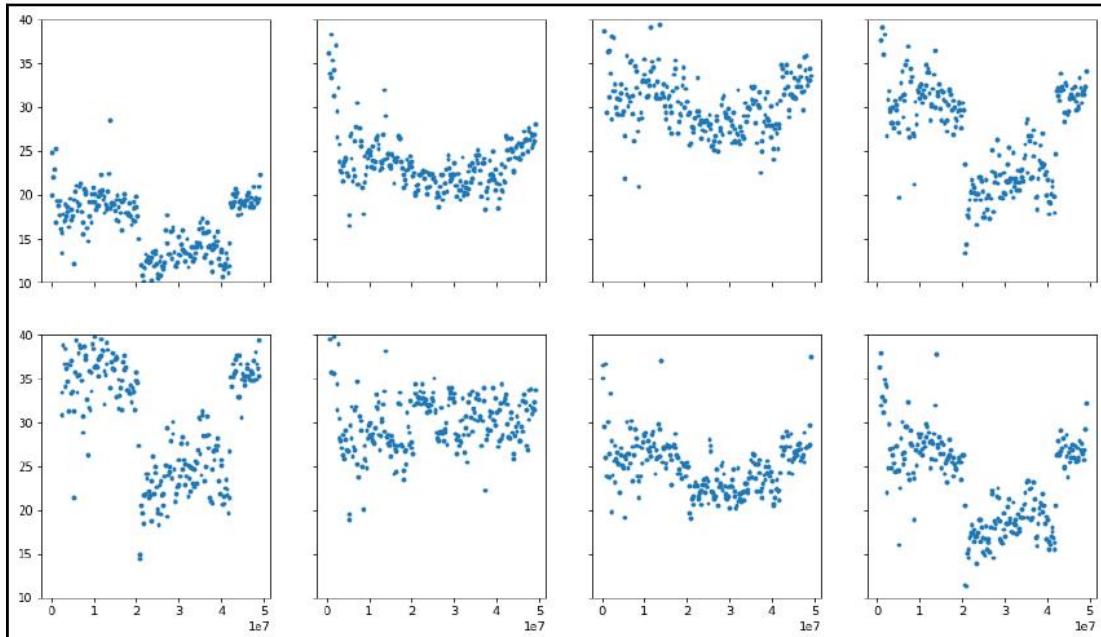
We will print the median DP for 200,000 kbp windows across the chromosome arm. Let's start with the window code:

```
window_size = 200000  
parent_DP_windows = [defaultdict(list) for i in range(num_parents)]  
  
def insert_in_window(row):  
    for parent in range(num_parents):  
        parent_DP_windows[parent][row[-1] //  
        window_size].append(row[parent])  
  
insert_in_window_v = np.vectorize(insert_in_window,  
signature='(n)->()')  
_ = insert_in_window_v(dp_2L)
```

Let's plot it:

```
fig, axs = plt.subplots(2, num_parents // 2, figsize=(16, 9),
sharex=True, sharey=True, squeeze=True)
for parent in range(num_parents):
    ax = axs[parent // 4][parent % 4]
    parent_data = parent_DP_windows[parent]
    ax.set_xlim(0, 5e7)
    ax.set_ylim(10, 40)
    ax.plot(*zip(*[(win*window_size, np.mean(lst)) for win, lst in
parent_data.items()]), ':')
```

The following plot shows the output:



You will notice that for some mosquitoes, for example, the ones on the first and last columns, there is a clear drop of DP in the middle of the chromosome arm. In some of them, like the third column, there is a bit of drop—not so pronounced. And on the bottom parent of the second column, there is no drop at all.

## There's more...

The preceding pattern has a biological cause that ends up having consequences on sequencing: Anopheles mosquitoes might have a big chromosomal inversion in the middle of arm 2L. Karyotypes that are not the same as the one on the reference genome used to make the calls are harder to call due to evolutionary divergence. These make the number of sequencer reads in that area lower. This is very specific to this species, but you might expect other kinds of features to appear on other organisms.

A more widely-known case is Copy Number Variation (CNV): if a reference genome has only a copy of a feature, but the individual that you are sequencing has n, then you can expect to see a DP of n times the median across the genome.

But, in the general case, it is a good idea to be on the lookout for strange results throughout the analysis. Sometimes, that is the hallmark of an interesting biological feature, as it here. Either that, or it's a pointer to a mistake: for example, Principal Components Analysis (PCA) can be used to find mislabeled samples (as they might cluster in the wrong group).

# Index

## 1

1,000 Genomes Project  
reference 44

## A

admixture  
download link 136  
population structure, investigating 135, 136, 138, 140  
reference 135

Airflow  
reference 232  
variant analysis pipeline, deploying 250, 251, 253, 254, 255, 257

alignment data  
working with 45, 47, 48, 50

AmiGO  
reference 103

Anaconda distribution  
download link 12

Anaconda  
reference 8  
required software, installing 9, 10, 12

analysis  
dataset, preparing for 311, 312, 313, 314, 315

annotations  
genes, extracting from reference 90, 91, 92

API  
used, for accessing Galaxy 236, 237, 238, 239, reference 107  
240, 242

Arlequin  
reference 183

B

Basic local alignment search tool (BLAST) 32

basic sequence analysis

bioinformatics data  
generic pipelines, using with 248, 250

bioinformatics  
pipeline systems 232

Biopython SeqIO page  
reference 36

Biopython  
mmCIF files, parsing 230  
reference 33, 36

Biostars  
reference 33, 52

Browser Extensible Data (BED) 69, 90

Burrows-Wheeler Aligner (bwa)  
about 10, 45, 87  
reference 52

bzip2

## C

chloroquine resistance transporter (CRT) 29

chromosome arm 3L  
download link 259

code  
optimizing, with Cython 274, 275, 277

optimizing, with Numba 274, 276, 277  
Complete Genomics data  
reference 52  
complex demographic scenarios  
modeling 162, 163, 164, 166, 167, 168  
Comprehensive R Archive Network (CRAN)  
reference 22  
Copy Number Variation (CNV) 332  
copy number variations (CNVs) 53, 104  
Cython  
code, optimizing 274, 275, 276  
Cytoscape  
protein interactions, plotting 303, 304, 305, 306  
reference 303

## D

DADA2 plugin  
reference 283  
Dask  
parallel computing, performing 264, 265, 266, 268  
data  
analyzing, in VCF 53, 54  
exploring, decision trees used 321, 322, 323  
exploring, with standard statistics 324, 326, 328  
329  
databases  
protein, finding in 202, 203, 204, 206  
dataset  
preparing, for analysis 316  
datasets  
exploring, with Bio.PopGen 117, 118, 119, 120, 122  
managing, with PLINK 105, 107, 108, 110, 111  
preparing, for analysis 311, 312, 313, 314, 316  
preparing, for phylogenetic analysis 170, 172, 173, 175  
decision trees  
used, for exploring data 321, 322, 323  
Deoxyhypusine Synthase (DHPS) 149  
Dihydrofolate Reductase (DHFR) 149  
Directed Acyclic Graph (DAG) 251  
Docker  
reference 8, 15  
required software, installing 13, 14, 15

## E

Ebolavirus  
reference 170  
Ensembl REST API  
orthologues, finding 94, 96, 97  
Ensembl  
gene ontology information, retrieving from 98, 99, 101  
reference 32  
ETE  
reference 175  
extensions, IPython  
reference 25

## F

F-statistics  
computing 123, 124, 126, 128, 129  
reference 129  
FASTQ  
reference 44  
fastStructure  
reference 135  
first-in, first-out (FIFO) 191  
fixation index (FST) 104  
FlyBase  
reference 81  
forward-time simulations 143, 144, 145, 147, 149

## G

Galaxy tool  
Galaxy  
about 233  
accessing, API used 236, 237, 238, 239, 240, 242  
installing 234  
reference 232, 233  
GBIF datasets  
georeferencing 297, 298, 300, 301  
GBIF REST interface  
reference 303  
GenBank  
accessing 27, 29, 31  
gene ontologies

reference 103  
Gene Ontology (GO) 74, 205  
gene ontology information  
  retrieving, from Ensembl 98, 99, 101  
Genepop  
  about 112, 113, 114, 115, 116, 117  
  reference 105  
General Transfer Format (GTF) 87  
Generic Feature Format (GFF) 87  
generic pipelines  
  using, with bioinformatics data 248, 250  
genes  
  extracting, from reference 91, 92  
genetic data  
  aligning 176, 177  
genome accessibility  
  studying 56, 58, 60, 62, 64, 65, 67  
Genome Analysis Toolkit (GATK)  
  about 10, 46  
  reference 56  
genome annotations  
  traversing 87, 88, 89  
genome-wide association studies (GWAS) 105  
genomes, National Center for Biotechnology  
  Information (NCBI)  
  reference 81  
genomes  
  high-quality reference genomes 75  
  low-quality genome references 81  
  organism genomes 75  
genomic data  
  aligning 176, 177  
genomic features  
  finding, from sequencing annotations 329, 339  
geometric operations  
  performing 220, 221, 222  
Gephi  
  reference 200  
Germline  
  shared chromosomal segments, inferring 285, 286, 287, 288, 289  
GFF spec  
  reference 90  
ggplot  
  reference 22

Global Biodiversity Information Facility  
  accessing, via REST 291, 292, 293, 294  
  reference 290  
graphs  
  reference 200

**H**

HapMap  
  reference 106  
hard filters  
  reference 57  
HDF Compass  
  reference 259  
Hierarchical Data Format (HDF5)  
  about 258  
  using 259, 260, 261, 263  
high-quality reference genomes  
  working with 75, 76, 77, 79, 80  
HTSeq  
  NGS data, processing 69, 71, 72  
  reference 69  
htslib package  
  reference 53

**I**

insertions/deletions (INDELs) 53  
IPython magics  
  reference 25  
island model  
  population structure, simulating 156, 158, 160  
IUPAC ambiguity codes  
  reference 87

**J**

Java Virtual Machine (JVM) 9  
Jupyter Notebooks  
  R magic, performing 23, 24

**K**

Kyoto Encyclopedia of Genes and Genomes  
  (KEGG)  
  reference 309

## L

- last-in, first-out (LIFO) 191
- linkage disequilibrium (LD) pruning 105
- linkage disequilibrium
  - reference 112
- low-quality genome references
  - dealing with 81, 82, 83, 84

## M

- MAFFT
  - reference 176
- Mendelian error information
  - using, for quality control 317, 318, 319, 320
- metagenomics
  - performing, with QIIME 2 Python API 280, 282, 283, 284
- mmCIF files
  - parsing, with Biopython 230
- mmCIF format
  - reference 211
- modern sequence formats
  - working with 36, 37, 38, 39, 41, 42
- molecular distances
  - computing, on PDB file 215, 217, 219
- MUSCLE
  - reference 176
- MySQL tables, for Ensembl
  - reference 94

## N

- National Center for Biotechnology Information (NCBI)
  - about 27, 173
  - reference 36
- NCBI databases
  - about 27, 32
  - data, fetching from 28, 31
  - reference 33
- NetworkX
  - reference 303, 309
- next-generation sequencing (NGS) 26, 105
- NGS data
  - processing, with HTSeq 69, 71, 72
- Numba

code, optimizing 274, 275, 277

## O

- OpenStreetMap
  - reference 297, 303
- orthologues
  - finding, with Ensembl REST API 94, 96, 97

## P

- page to paired-end reads, Illumina
  - reference 44
- pandas
  - reference 23
- Panther
  - reference 103
- parallel computing
  - performing, with Dask 264, 265, 266, 268
- parallel processing, HDF5
  - reference 264
- Parquet
  - reference 269
  - using 269, 270
- PDB file
  - information, extracting from 211, 212, 214
  - molecular distances, computing 215, 217, 219
- Phred quality score
  - reference 38, 44
- phylogenetic analysis
  - dataset, preparing for 170, 172, 173, 175
- phylogenetic data
  - visualizing 193, 194, 195, 197, 198
- phylogenetic trees
  - constructing 185
  - reconstructing 183, 187
- phylogenetics
  - about 169
  - reference 176
- PhyloXML
  - reference 176
- Picard
  - reference 52
- Pillow
  - reference 297
- pipeline systems, bioinformatics
  - Apache Airflow 232

Galaxy 232  
 Script of Scripts (SoS) 232  
 pipelines 232  
**Planemo**  
 reference 244  
**PlasmoDB**  
 reference 81  
**PLINK**  
 datasets, managing 105, 106, 107, 108, 110, 111  
 reference 105, 112  
 population genetics 104  
 population structure  
     investigating, with admixture 135, 136, 138, 140  
     simulating, island model used 156, 158, 159  
     simulating, stepping-stone model used 156, 158, 160  
 Principal Components Analysis  
     performing 130, 131, 132, 134  
     references 135  
 Protein Data Bank (PDB) 201  
 protein interactions  
     plotting, with Cytoscape 303, 305, 307  
 protein  
     finding, in multiple databases 202, 203, 204, 206  
 proteomics 201  
 pygraphviz library  
     reference 103  
 PyMOL  
     animating with 223, 224, 226, 227, 228  
     references 223, 230  
 pysamstats library  
     reference 52  
 Python Global Interpreter Lock (GIL)  
     reference 156  
 Python Package Index (PyPI) 9

**Q**  
 QIIME 2 Python API  
     metagenomics, performing 280, 281, 282, 284  
 QIIME 2  
     reference 281  
 quality control  
     Mendelian error information, using for 317, 318

319, 320  
 QuickGO  
     reference 103

**R**  
 R magic  
     performing, with Jupyter Notebooks 23, 24  
 R packages, for statistical genetics  
     reference 123  
 R  
     reference 22  
 RAxML  
     reference 183  
 reference genomes of model organisms, Ensembl  
     reference 81  
 RepeatMasker  
     reference 87  
 resilient distributed datasets (RDDs) 273  
 REST API  
     reference 296  
 REST  
     Global Biodiversity Information Facility,  
         accessing 291, 293, 294  
 rkt  
     reference 15  
 rpy2  
     interfacing, with R 15, 16, 17, 18, 20

**S**  
 SAM/BAM format  
     reference 52  
 Sample Read Depth (DP) 63  
 scikit-allel  
     reference 264  
 Script of Scripts (SoS)  
     reference 232  
 selection  
     simulating 149, 150, 151, 153  
 SEQanswers  
     reference 52  
 SeqIO  
     reference 38  
 sequence alignment map (SAM) format 45  
 Sequence Read Archive (SRA) 32  
 sequences

comparing 178, 180, 182  
sequencing statistics  
computing, Spark used 271, 272  
shared chromosomal segments  
inferring, with Germline 285, 286, 287, 288, 289  
simuPOP programming 143  
simuPOP  
reference 149  
Single-Nucleotide Polymorphism (SNP) 105  
single-nucleotide polymorphisms (SNPs) 32  
Singularity  
reference 15  
SNP data  
filtering 56, 58, 60, 62, 63, 65, 67  
SnpEff  
reference 57  
Spark  
sequencing statistics, computing 271, 272  
standard statistics  
used, for exploring data 324, 326, 328, 329  
stepping-stone model  
population structure, simulating 156, 158, 160

**T**

third-party magic extensions, IPython  
reference 25

Tile Map Services  
reference 297

tiling coordinates  
reference 303

trees  
reference 193  
working with 188, 189, 191

trimAl  
reference 176

**U**

UCSC Genome Bioinformatics  
reference 32

UCSC genome browser  
reference 94

UniProt  
reference 206

**V**

variant analysis pipeline  
deploying, with Airflow 250, 251, 253, 254, 255, 257

Variant Call Format (VCF) 27

VCF  
data, analyzing in 53, 54  
reference 56

VectorBase  
reference 81

Vertebrate Genome Annotation (Vega) project 97

voltage-gated sodium channel (VGSC) gene 90

**W**

well-known text  
reference 302

whole genome sequencing (WGS) 26

**X**

Xcode  
reference 11

**Z**

Zarr  
reference 264