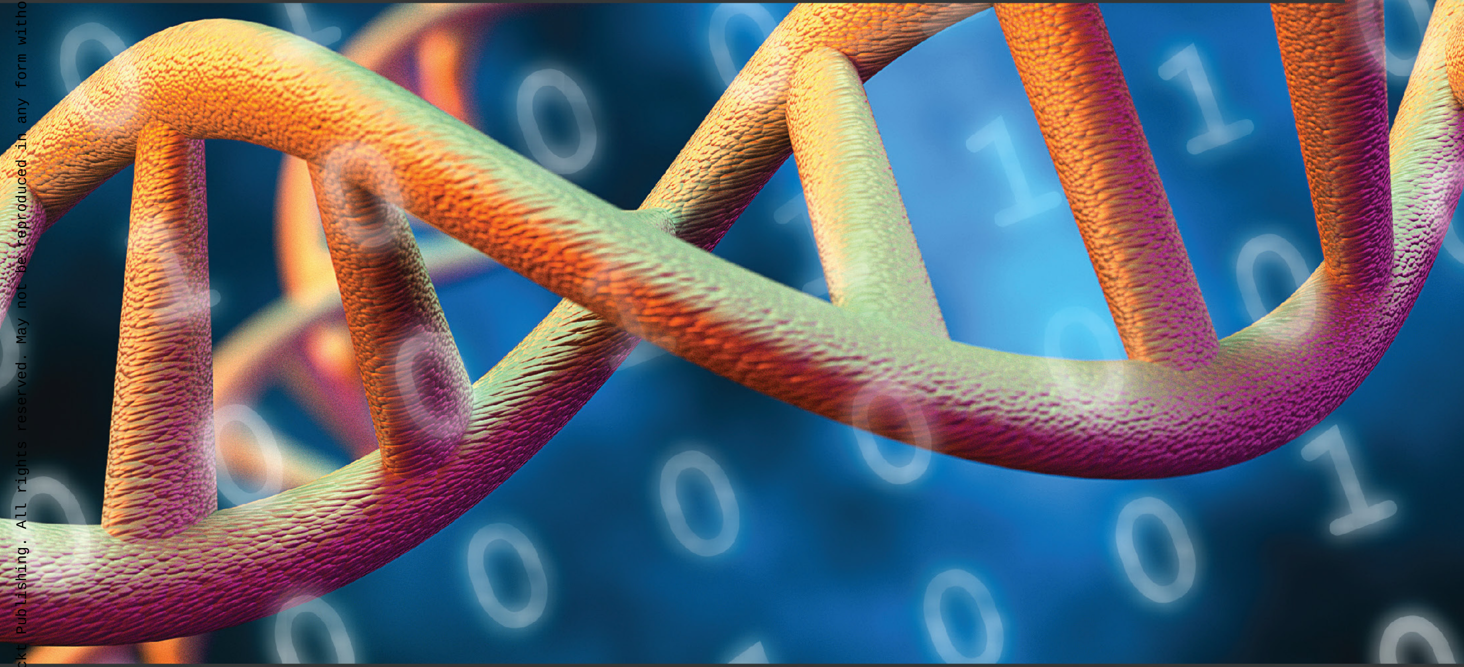
Bioinformatics with Python Cookbook

Third Edition

Use modern Python libraries and applications to solve real-world computational biology problems



Tiago Antao

Bioinformatics with Python Cookbook

Third Edition

Use modern Python libraries and applications to solve real-world computational biology problems

Tiago Antao

packt

BIRMINGHAM—MUMBAI

Bioinformatics with Python Cookbook

Third Edition

Publishing Product Manager: Devika Battike

Senior Editor: David Sugarman

Content Development Editor: Joseph Sunil

Technical Editor: Rahul Limbachiya

Copy Editor: Safis Editing

Project Coordinator: Farheen Fathima

Proofreader: Safis Editing

Indexer: Pratik Shirodkar

Production Designer: Shankar Kalbhor

Marketing Coordinator: Priyanka Mhatre

First published: June 2015

Second edition: November 2018

Third edition: September 2022

Production reference: 1090922

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-80323-642-1

www.packt.com

Contributors

About the author

Tiago Antao is a bioinformatician who is currently working in the field of genomics. A former computer scientist, Tiago moved into 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, UK. Post his doctoral, Tiago worked with human datasets at the University of Cambridge, UK and with mosquito whole-genome sequencing data at the University of Oxford, UK, before helping to set up the bioinformatics infrastructure at the University of Montana, USA. He currently works as a data engineer in the biotechnology field in Boston, MA. He is one of the co-authors of Biopython, a major bioinformatics package written in Python.

About the reviewers

Urminder Singh is a bioinformatician, computer scientist, and developer of multiple open source bioinformatics tools. His educational background encompasses physics, computer science, and computational biology degrees, including a Ph.D. in bioinformatics from Iowa State University, USA.

His diverse research interests include novel gene evolution, precision medicine, sociogenomics, machine learning in medicine, and developing tools and algorithms for big heterogeneous data. You can visit him online at urmi-21.github.io.

Tiffany Ho works as a bioinformatics associate at Embark Veterinary. She holds a BSc from the University of California, Davis in genetics and genomics, and an MPS from Cornell University in plant breeding and genetics.

Table of Contents

Preface xiii

1

Python and the Surrounding Software Ecology 1

Installing the required basic software with Anaconda 2
- Getting ready 2
- How to do it... 4
- There's more... 5

Installing the required software with Docker 7
- Getting ready 7
- How to do it... 8
- See also 8

Interfacing with R via rpy2 9
- Getting ready 9
- How to do it... 10
- There's more... 15
- See also 16

Performing R magic with Jupyter 16
- Getting ready 16
- How to do it... 17
- There's more... 18
- See also 18

2

Getting to Know NumPy, pandas, Arrow, and Matplotlib 19

Using pandas to process vaccine-adverse events 20
- Getting ready 20
- How to do it... 20
- There's more... 25
- See also 26

Dealing with the pitfalls of joining pandas DataFrames 26
- Getting ready 26
- How to do it... 27
- There's more... 29

Reducing the memory usage of pandas DataFrames 29
- Getting ready 29
- How to do it… 29
- See also 32

Accelerating pandas processing with Apache Arrow 32
Getting ready 33
How to do it... 33
There's more... 35

Understanding NumPy as the engine behind Python data science and bioinformatics 36
Getting ready 36
How to do it… 36
See also 39

Introducing Matplotlib for chart generation 39
Getting ready 40
How to do it... 40
There's more... 47
See also 47

3

Next-Generation Sequencing 49

Accessing GenBank and moving around NCBI databases 50
Getting ready 50
How to do it... 51
There's more... 53
See also 54

Performing basic sequence analysis 55
Getting ready 55
How to do it... 55
There's more... 56
See also 57

Working with modern sequence formats 57
Getting ready 57
How to do it... 58
There's more... 64
See also 65

Working with alignment data 66
Getting ready 66
How to do it... 66
There's more... 72
See also 72

Extracting data from VCF files 73
Getting ready 73
How to do it... 74
There's more... 75
See also 76

Studying genome accessibility and filtering SNP data 76
Getting ready 76
How to do it... 78
There's more... 88
See also 88

Processing NGS data with HTSeq 88
Getting ready 89
How to do it... 90
There's more... 92

4

Advanced NGS Data Processing 93

Preparing a dataset for analysis 93
Getting ready 94
How to do it… 94
Using Mendelian error information for quality control 101
How to do it… 101
There's more… 105
Exploring the data with standard statistics 106
How to do it… 106
There's more… 111
Finding genomic features from sequencing annotations 111
How to do it… 111
There's more… 114
Doing metagenomics with QIIME 2 Python API 114
Getting ready 114
How to do it... 116
There's more... 119

5

Working with Genomes 121

Technical requirements 121
Working with high-quality reference genomes 122
Getting ready 122
How to do it... 123
There's more... 127
See also 128
Dealing with low-quality genome references 128
Getting ready 128
How to do it... 129
There's more... 133
See also 134
Traversing genome annotations 134
Getting ready 134
How to do it... 134
There's more... 136
See also 137
Extracting genes from a reference using annotations 137
Getting ready 137
How to do it... 138
There's more... 140
See also 140
Finding orthologues with the Ensembl REST API 141
Getting ready 141
How to do it... 141
There's more... 144
Retrieving gene ontology information from Ensembl 144
Getting ready 144
How to do it... 145
There's more... 149
See also 149

6

Population Genetics 151

Managing datasets with PLINK 152
Getting ready 152
How to do it... 154
There's more... 158
See also 158

Using sgkit for population genetics analysis with xarray 158
Getting ready 159
How to do it... 159
There's more... 163

Exploring a dataset with sgkit 163
Getting ready 163
How to do it... 163
There's more... 167
See also 167

Analyzing population structure 167
Getting ready 168
How to do it... 168
See also 174

Performing a PCA 174
Getting ready 174
How to do it... 175
There's more... 177
See also 177

Investigating population structure with admixture 177
Getting ready 177
How to do it... 178
There's more... 183

7

Phylogenetics 185

Preparing a dataset for phylogenetic analysis 185
Getting ready 186
How to do it... 186
There's more... 192
See also 192

Aligning genetic and genomic data 192
Getting ready 192
How to do it... 193

Comparing sequences 195
Getting ready 195
How to do it... 195
There's more... 200

Reconstructing phylogenetic trees 200
Getting ready 200
How to do it... 201
There's more... 204

Playing recursively with trees 205
Getting ready 205
How to do it... 205
There's more... 209

Visualizing phylogenetic data 210
Getting ready 210
How to do it... 210
There's more... 215

8

Using the Protein Data Bank 217

Finding a protein in multiple databases 218
Getting ready 218
How to do it... 218
There's more 222

Introducing Bio.PDB 222
Getting ready 223
How to do it... 223
There's more 228

Extracting more information from a PDB file 228
Getting ready 228
How to do it... 228

Computing molecular distances on a PDB file 232
Getting ready 233
How to do it... 233

Performing geometric operations 237
Getting ready 237
How to do it... 237
There's more 240

Animating with PyMOL 241
Getting ready 241
How to do it... 241
There's more 247

Parsing mmCIF files using Biopython 247
Getting ready 247
How to do it... 247
There's more 248

9

Bioinformatics Pipelines 249

Introducing Galaxy servers 250
Getting ready 250
How to do it… 250
There's more 252

Accessing Galaxy using the API 252
Getting ready 252
How to do it… 254

Deploying a variant analysis pipeline with Snakemake 260
Getting ready 260
How to do it… 261
There's more 266

Deploying a variant analysis pipeline with Nextflow 267
Getting ready 267
How to do it… 268
There's more 272

10

Machine Learning for Bioinformatics 273

Introducing scikit-learn with a PCA example 273
Getting ready 274
How to do it... 274
There's more... 276

Using clustering over PCA to classify samples 276
Getting ready 277
How to do it... 277
There's more... 282

Exploring breast cancer traits using Decision Trees 282
Getting ready 283
How to do it... 283

Predicting breast cancer outcomes using Random Forests 286
Getting ready 286
How to do it… 286
There's more... 289

11

Parallel Processing with Dask and Zarr 291

Reading genomics data with Zarr 292
Getting ready 292
How to do it... 292
There's more... 297
See also 297

Parallel processing of data using Python multiprocessing 297
Getting ready 297
How to do it... 298
There's more... 299
See also 300

Using Dask to process genomic data based on NumPy arrays 300
Getting ready 300
How to do it... 301
There's more... 305
See also 305

Scheduling tasks with dask.distributed 305
Getting ready 305
How to do it... 307
There's more... 311
See also 311

12

Functional Programming for Bioinformatics 313

Understanding pure functions 314
Getting ready 314
How to do it... 314
There's more... 316

Understanding immutability 316
Getting ready 317
How to do it... 317
There's more... 318

Avoiding mutability as a robust development pattern 318
Getting ready 319
How to do it... 319
There's more... 320

Using lazy programming for pipelining 321
Getting ready 321
How to do it... 321
There's more... 323

The limits of recursion with Python 323
Getting ready 324
How to do it... 324
There's more... 326

A showcase of Python's functools module 326
Getting ready 326
How to do it... 326
There's more... 328
See also... 328

Index 329

Other Books You May Enjoy 338

Preface

Bioinformatics is an active research field that uses a range of simple-to-advanced computations to extract valuable information from biological data, and this book will show you how to manage these tasks using Python.

This updated edition of the Bioinformatics with Python Cookbook begins with a quick overview of the various tools and libraries in the Python ecosystem that will help you convert, analyze, and visualize biological datasets. As you advance through the chapters, you'll cover key techniques for next-generation sequencing, single-cell analysis, genomics, metagenomics, population genetics, phylogenetics, and proteomics with the help of real-world examples. You'll learn how to work with important pipeline systems, such as Galaxy servers and Snakemake, and understand the various modules in Python for functional and asynchronous programming. This book will also help you explore topics such as SNP discovery using statistical approaches under high-performance computing frameworks, including Dask and Spark, and the application of machine learning algorithms to bioinformatics.

By the end of this bioinformatics Python book, you'll be equipped with the knowledge to implement the latest programming techniques and frameworks, empowering you to deal with bioinformatics data on every kind of scale.

Who this book is for

This book is for bioinformatics analysts, data scientists, computational biologists, researchers, and Python developers who want to address intermediate-to-advanced biological and bioinformatics problems. Working knowledge of the Python programming language is expected. Basic knowledge of biology would be helpful.

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 Jupyter Notebooks.

Chapter 2, Getting to Know NumPy, pandas, Arrow, and Matplotlib, introduces the fundamental Python libraries for data science: NumPy for array and matrix processing; Pandas for table-based data manipulation; Arrow to optimize Pandas processing and Matplotlib for charting.

Chapter 3, 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 4, Advanced NGS Processing, covers advanced programming techniques to filter NGS data. This includes the use of mendelian datasets that are then analyzed by standard statistics. We also introduce metagenomic analysis

Chapter 5, 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 6, Population Genetics, describes how to perform population genetics analysis of empirical datasets. For example, in Python, we could perform Principal Components Analysis, computer FST, or structure/admixture plots.

Chapter 7, 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 8, 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 9, 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 will be based on snakemake and nextflow, a type of pipeline that targets programmers.

Chapter 10, Machine Learning for Bioinformatics, introduces machine learning using an intuitive approach to deal with computational biology problems. The chapter covers Principal Components Analysis, Clustering, Decision Trees, and Random Forests.

Chapter 11, Parallel Processing with Dask and Zarr, introduces techniques to deal with very large datasets and computationally intensive algorithms. The chapter will explain how to use parallel computation across many computers (cluster or cloud). We will also discuss the efficient storage of biological data.

Chapter 12, Functional Programming for Bioinformatics, introduces functional programming which permits the development of more sophisticated Python programs that, through lazy programming and immutability are easier to deploy in parallel environments with complex algorithms

To get the most out of this book

Software/Hardware covered in the book	OS Requirements
Python 3.9	Windows, Mac OS X, and Linux (Preferred)
Numpy, Pandas, Matplolib	
Biopython	
Dask, zarr, scikit-learn	

If you are using the digital version of this book, we advise you to type the code yourself or access the code via the GitHub repository (link available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from GitHub at `https://github.com/PacktPublishing/Bioinformatics-with-Python-Cookbook-third-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://packt.link/3KQQO`.

Conventions used

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

`Code in text`: 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: "`call_genotype` has a shape of 56,241x1,1198,2, that is it is dimensioned variants, samples, `ploidy`."

A block of code is set as follows:

```
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)
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
AgamP4_2L | organism=Anopheles_gambiae_PEST | version=AgamP4 |
length=49364325 | SO=chromosome
AgamP4_2R | organism=Anopheles_gambiae_PEST | version=AgamP4 |
length=61545105 | SO=chromosome
```

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: "For the **Chunk** column, see *Chapter 11* – but you can safely ignore it for now."

Tips or important notes

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`.

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.packtpub.com/support/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` 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`.

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`.

Share Your Thoughts

Once you've read *Bioinformatics with Python Cookbook*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

1
Python and the Surrounding Software Ecology

We will start by installing the basic software that is required for most of this book. This will include the **Python** distribution, some fundamental Python libraries, and external bioinformatics software. Here, we will also look at the world outside of 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. Additionally, we will explore the advantages that the **IPython** framework (via Jupyter Lab) can give us in order to efficiently interface with R. Given that source management with Git and GitHub is pervasive, we will make sure that our setup plays well with them. This chapter will set the stage for all of the computational biologies that we will perform in the remainder 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.continuum.io/anaconda/`) distribution and another approach for installing the software is via Docker (which is a server virtualization method based on containers sharing the same operating system kernel; please refer to `https://www.docker.com/`). This will still install Anaconda for you but inside a container. If you are using a Windows-based operating system, you are strongly encouraged to consider changing your operating system or using 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. Learning using a local distribution (Anaconda or something else) is easier than Docker, but given that package management can be complex in Python, Docker images provide a level of stability.

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

Installing the required basic software with Anaconda

Before we get started, we need to install some basic prerequisite software. The following sections will take you through the software and the steps that are needed to install them. Each chapter and section might have extra requirements on top of these – we will make those clear as the book progresses. An alternative way to start is to use the Docker recipe, after which everything will be taken care of for you via a Docker container.

If you are already using a different Python distribution, you are strongly encouraged to consider Anaconda, as it has become the *de-facto* standard for data science and bioinformatics. 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 via **IronPython**). However, here, we are not only concerned 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.

For our code, we will be using Python 3.10. 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 of the heavy-duty analysis will be done on Linux (probably on a Linux **high-performance computing** or **HPC** cluster). **Next-generation sequencing** (**NGS**) data analysis and complex machine learning are 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. Note that 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 will probably be safer if you install the 32-bit version for everything (including Python itself).

Note

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 perhaps not even work with a newer version. Hopefully, any code that you use will indicate the correct dependencies – though this is not guaranteed. In this book, we will fix the precise versions of all software packages, and we will make sure that the code will work with them. It is quite natural that the code might need tweaking with other package versions.

The software developed for this book is available at `https://github.com/PacktPublishing/Bioinformatics-with-Python-Cookbook-third-edition`. To access it, you will need to install Git. Getting used to Git might 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 of 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. Fortunately, since the previous editions of this book, most bioinformatics software has become available via the Bioconda project; therefore, installation is usually easy.

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 install build-essential`), and on macOS, consider **Xcode** (`https://developer.apple.com/xcode/`).

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

Name	**Application**	**URL**	**Purpose**
Project Jupyter	All chapters	`https://jupyter.org/`	Interactive computing
pandas	All chapters	`https://pandas.pydata.org/`	Data processing
NumPy	All chapters	`http://www.numpy.org/`	Array/matrix processing
SciPy	All chapters	`https://www.scipy.org/`	Scientific computing
Biopython	All chapters	`https://biopython.org/`	Bioinformatics library
seaborn	All chapters	`http://seaborn.pydata.org/`	Statistical chart library
R	Bioinformatics and Statistics	`https://www.r-project.org/`	Language for statistical computing
rpy2	R connectivity	`https://rpy2.readthedocs.io`	R interface
PyVCF	NGS	`https://pyvcf.readthedocs.io`	VCF processing

Pysam	NGS	`https://github.com/pysam-developers/pysam`	SAM/BAM processing
HTSeq	NGS/Genomes	`https://htseq.readthedocs.io`	NGS processing
DendroPY	Phylogenetics	`https://dendropy.org/`	Phylogenetics
PyMol	Proteomics	`https://pymol.org`	Molecular visualization
scikit-learn	Machine learning	`http://scikit-learn.org`	Machine learning library
Cython	Big data	`http://cython.org/`	High performance
Numba	Big data	`https://numba.pydata.org/`	High performance
Dask	Big data	`http://dask.pydata.org`	Parallel processing

Figure 1.1 – A table showing the various software packages that are useful in bioinformatics

We will use `pandas` to process most table data. An alternative would be to use just standard Python. `pandas` has become so pervasive in data science that it will probably make sense to just process all tabular data with it (if it fits in memory).

All of our work will be developed inside project Jupyter, namely Jupyter Lab. Jupyter has become the *de facto* standard to write interactive data analysis scripts. Unfortunately, the default format for Jupyter Notebooks is based on JSON. This format is difficult to read, difficult to compare, and needs exporting to be fed into a normal Python interpreter. To obviate that problem, we will extend Jupyter with `jupytext` (`https://jupytext.readthedocs.io/`), which allows us to save Jupyter notebooks as normal Python programs.

How to do it...

To get started, take a look at the following steps:

1. Start by downloading the Anaconda distribution from `https://www.anaconda.com/products/individual`. We will be using version 21.05, although you will probably be fine with the most recent one. You can accept all the installation's default settings, but you might want to make sure that the `conda` binaries are in your path (do not forget to open a new window so that the path can be 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.
2. Let's go ahead with the libraries. We will now create a new `conda` environment called `bioinformatics_base` with `biopython=1.70`, as shown in the following command:

```
conda create -n bioinformatics_base python=3.10
```

3. Let's activate the environment, as follows:

   ```
   conda activate bioinformatics_base
   ```

4. Let's add the `bioconda` and `conda-forge` channels to our source list:

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

5. Also, install the basic packages:

   ```
   conda install \
   biopython==1.79 \
   jupyterlab==3.2.1 \
   jupytext==1.13 \
   matplotlib==3.4.3 \
   numpy==1.21.3 \
   pandas==1.3.4 \
   scipy==1.7.1
   ```

6. Now, let's save our environment so that we can reuse it later to create new environments in other machines or if you need to clean up the base environment:

   ```
   conda list –explicit > bioinformatics_base.txt
   ```

7. We can even install R from `conda`:

   ```
   conda install rpy2 r-essentials r-gridextra
   ```

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

There's more...

If you prefer not to use Anaconda, you will be able to install many of the Python libraries via `pip` using whatever distribution you choose. You will probably need quite a few compilers and build tools – not only C compilers but also C++ and Fortran.

We will not be using the environment we created in the preceding steps. Instead, we will use it as a base to clone working environments from it. This is because environment management with Python – even with the help of the `conda` package system – can still be quite painful. So, we will create a clean environment that we never spoil and can derive from if our development environments become unmanageable.

For example, imagine you want to create an environment for machine learning with `scikit-learn`. You can do the following:

1. Create a clone of the original environment with the following:

```
conda create -n scikit-learn --clone bioinformatics_base
```

2. Add `scikit-learn`:

```
conda activate scikit-learn
conda install scikit-learn
```

When inside JupyterLab, we should open our jupytext files with the notebook, not the text editor. As the jupytext files have the same extension as Python files – this is a feature, not a bug – by default, JupyterLab would use a normal text editor. When we open a jupytext file, we need to override the default. When opening it, right-click and choose **Notebook**, as shown in the following screenshot:

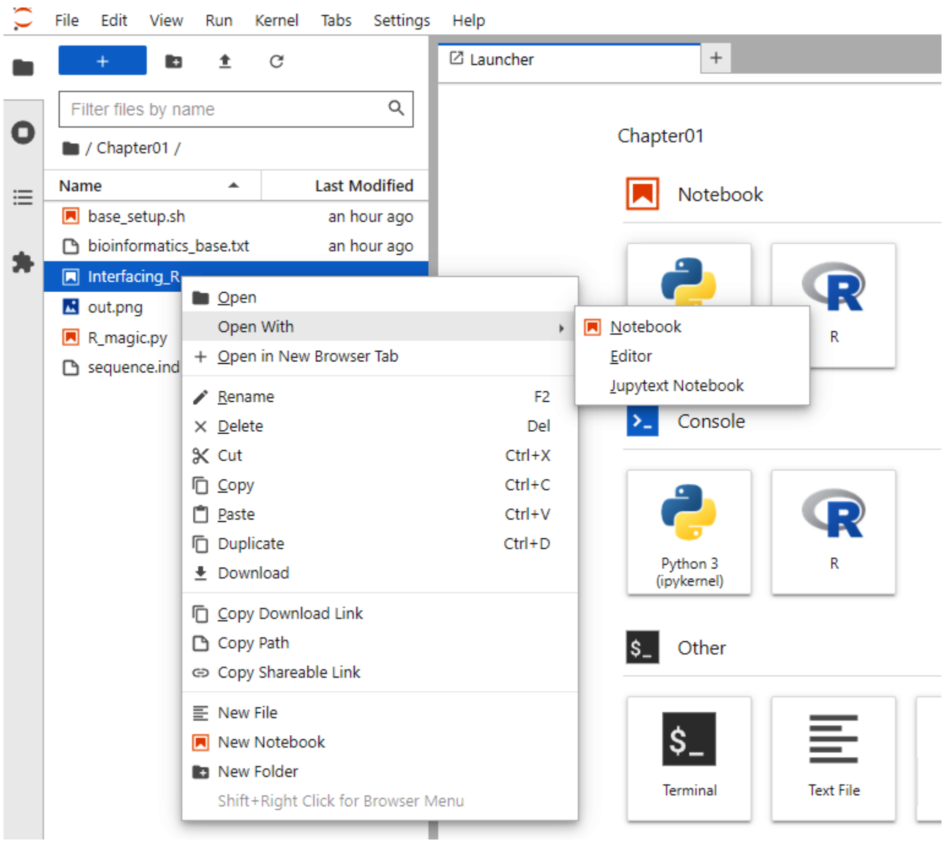


Figure 1.2 – Opening a jupytext file in Notebook

Our jupytext files will not be saving graphical outputs and that will suffice for this book. If you want to have a version with images, this is possible using paired notebooks. For more details, check the Jupytext page (`https://github.com/mwouts/jupytext`).

Warning

As our code is meant to be run inside Jupyter, many times throughout this book, I will not use `print` to output content, as the last line of a cell will be automatically rendered. If you are not using notebooks, remember to do a `print`.

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 could 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 might have a Docker package, it might be too old and buggy.

If you are on Windows or macOS, do not despair; take a look at the Docker site. There are various options available 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 the BIOS, VT-X, or AMD-V is enabled. At the very least, you will need 6 GB of memory, preferably more.

Note

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...

To get started, follow these steps:

1. Use the following command on your Docker shell:

   ```
   docker build -t bio https://raw.githubusercontent.com/
   PacktPublishing/Bioinformatics-with-Python-Cookbook-
   third-edition/main/docker/main/Dockerfile
   ```

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

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

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

3. 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 in `/data` and vice versa.

   `-p 9875:9875` will expose the container's 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 official Docker documentation on how to expose directories.

4. Now you are 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 official Docker documentation (`https://docs.docker.com/`) on how to expose ports.

See also

The following is also worth knowing:

- 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 the necessary 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 been implemented in R. For statistical methods, R is still the most complete framework; moreover, some bioinformatics functionalities are *only* available in R and are 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 to 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-third-edition/blob/main/Datasets.py`, and download the `sequence.index` file. If you are using Jupyter Notebook, open the `Chapter01/Interfacing_R.py` file and simply 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, the population of origin, and important statistical information per lane, such as the number of reads and the number of DNA bases read.

To set up Anaconda, you can run the following:

```
conda create -n bioinformatics_r --clone bioinformatics_base
conda activate bioinformatics_r
conda install r-ggplot2=3.3.5 r-lazyeval r-gridextra rpy2
```

With Docker, you can run the following:

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

Now we can begin.

How to do it...

To get started, follow these steps:

1. Let's start by doing some imports:

```
import os
from IPython.display import Image
import rpy2.robjects as robjects
import rpy2.robjects.lib.ggplot2 as ggplot2
from rpy2.robjects.functions import
SignatureTranslatedFunction
import pandas as pd
import rpy2.robjects as ro
from rpy2.robjects import pandas2ri
from rpy2.robjects import local_converter
```

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

2. 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 to 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 into `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 (however, objects are actually not available in the R namespace; we will discuss this further 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, that is, a sequence of rows where each column has the same type.

3. 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.

4. 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 `26` element. 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.

5. Now, we need to perform some data cleanup. For example, some columns should be interpreted as numbers, but instead, 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 extract the `0` element. 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 (that is values surrounded by `"`), whereas the second print will show numbers.

6. We will need to massage this table a bit more; details on this can be found in the notebook. 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):

```
robjects.r.assign('seq.data', seq_data)
```

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 in the other).

Note

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).

7. Regarding our concrete example based on the Human 1,000 Genomes Project, first, we will plot a histogram with the distribution of center names, where all sequencing lanes were generated. For this, we will use `ggplot`:

```
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).

Then, we 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`. Following this, we annotate 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.

8. Now, we can print the image in the Jupyter Notebook:

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

The following chart is produced:

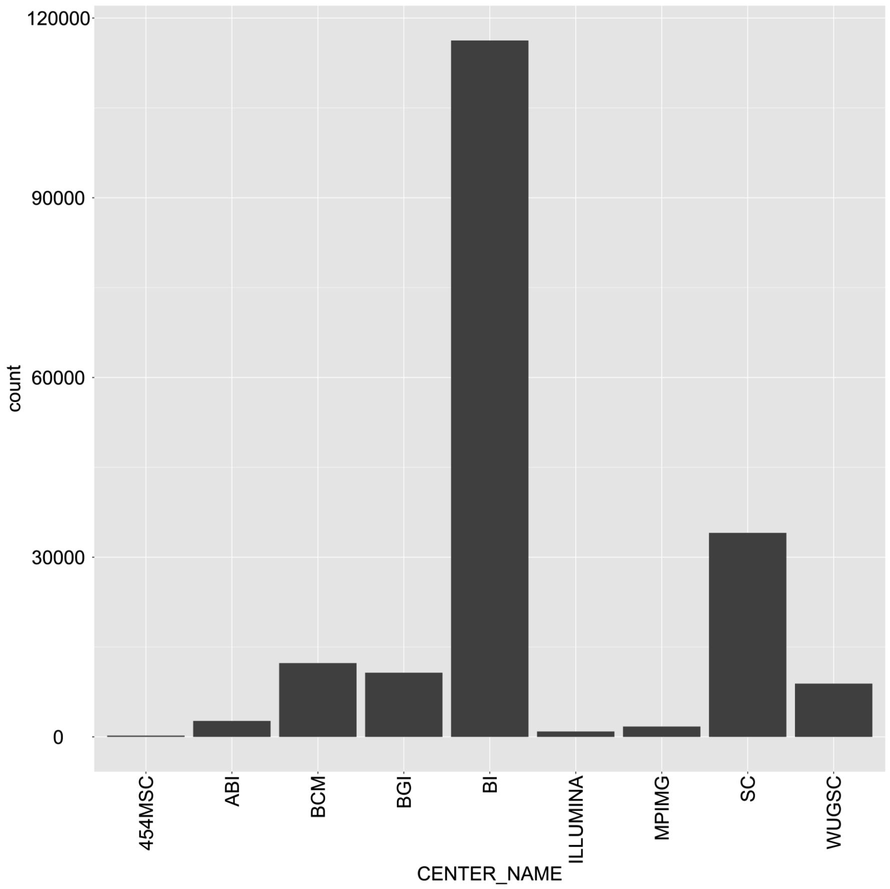


Figure 1.3 – The ggplot2-generated histogram of center names, which is responsible for sequencing the lanes of the human genomic data from the 1,000 Genomes Project

9. As a final example, we will now do a scatter plot of read and base counts for all of 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 of *Chapter 3*, *Next-Generation Sequencing*). Additionally, we are interested in the differences between the different types of sequencing (for instance, exome coverage, high

coverage, and low coverage). First, we generate a DataFrame with just the `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')
```

10. Now we are 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 (please 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 (that is, 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 that the color depends on the `ANALYSIS_GROUP` variable:

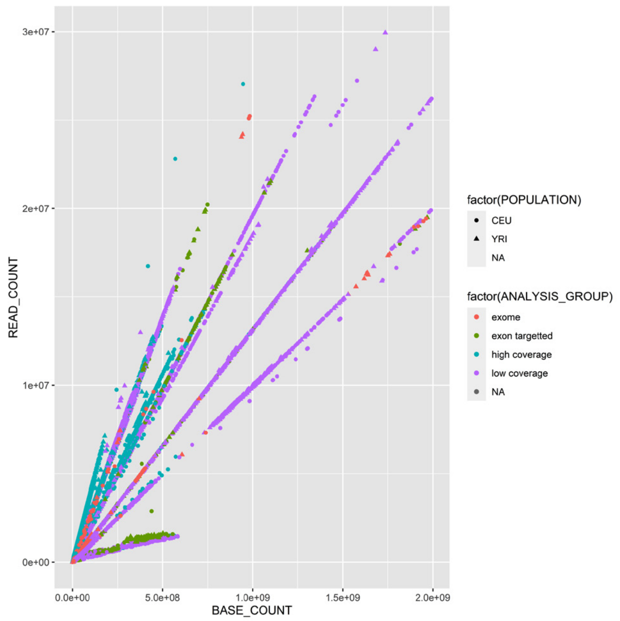


Figure 1.4 – The ggplot2-generated scatter plot with base and read counts for all sequencing lanes read; the color and shape of each dot reflects categorical data (population and the type of data sequenced)

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

```
import rpy2.robjects as ro
from rpy2.robjects import pandas2ri
from rpy2.robjects.conversion import localconverter
with localconverter(ro.default_converter + pandas2ri.
converter):
  pd_yri_ceu = ro.conversion.rpy2py(yri_ceu)
del pd_yri_ceu['PAIRED_FASTQ']
with localconverter(ro.default_converter + pandas2ri.
converter):
  no_paired = ro.conversion.py2rpy(pd_yri_ceu)
robjects.r.assign('no.paired', no_paired)
robjects.r("print(colnames(no.paired))")
```

We start by importing the necessary conversion module – `rpy2` provides many strategies to convert data from R into Python. Here, we are concerned with data frame conversion. 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 many R Bioinformatics packages 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.rproject.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 – surprise, surprise – this is called `ggplot`! (`http://yhat.github.io/ggpy/`).

See also

To learn more about these topics, please refer to the following resources:

- 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`.
- If you work with NGS, you might also want to take look at 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* approach is described in a book aptly named *The Grammar of Graphics*, by 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*, by Wes McKinney, O'Reilly Media, is also an alternative to consider. In the next chapter, we will discuss pandas and use it throughout the book.

Performing R magic with Jupyter

Jupyter provides quite a few extra features compared to standard Python. Among those features, it provides a framework of extensible commands called **magics** (actually, this only works with the IPython kernel of Jupyter since it is actually an IPython feature, but that is the one we are concerned with). Magics allow you to extend the language in many useful ways. There are magic functions that you can use to deal with R. As you will see in our example, it makes R interfacing much easier and more declarative. 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.py`. The notebook is more complete than the recipe presented here and includes more chart examples. For brevity, we will only concentrate on the fundamental constructs to interact with R using magics. If you are using Docker, you can use the following:

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

How to do it...

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

1. 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 `%` starts an IPython-specific directive. Just as a simple example, you can write `%R print(c(1, 2))` onto 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.

2. 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)
   ```

   Then, you can specify that the entire cell should be interpreted as R code by using `%%R` (note the double `%%`).

3. 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.

4. As we have a `pandas` DataFrame, we can operate on it quite easily using the `pandas` interface:

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

5. Let's put this DataFrame back into 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 into 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`.

6. Let's do some final cleanup (for further 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)
```

Additionally, the R magic system 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 their interface. For example, I have updated the R code for the first edition of this book a few times. The current version of the DataFrame assignment returns `pandas` objects, which is a major change.

See also

For more information, check out these links:

- For basic instructions on IPython magics, see `https://ipython.readthedocs.io/en/stable/interactive/magics.html`.
- A list of third-party extensions for IPython, including magic ones can be found at `https://github.com/ipython/ipython/wiki/Extensions-Index`.

2
Getting to Know NumPy, pandas, Arrow, and Matplotlib

One of Python's biggest strengths is its profusion of high-quality science and data processing libraries. At the core of all of them is **NumPy**, which provides efficient array and matrix support. On top of NumPy, we can find almost all of the scientific libraries. For example, in our field, there's **Biopython**. But other generic data analysis libraries can also be used in our field. For example, **pandas** is the *de facto* standard for processing tabled data. More recently, **Apache Arrow** provides efficient implementations of some of pandas' functionality, along with language interoperability. Finally, **Matplotlib** is the most common plotting library in the Python space and is appropriate for scientific computing. While these are general libraries with wide applicability, they are fundamental for bioinformatics processing, so we will study them in this chapter.

We will start by looking at pandas as it provides a high-level library with very broad practical applicability. Then, we'll introduce Arrow, which we will use only in the scope of supporting pandas. After that, we'll discuss NumPy, the workhorse behind almost everything we do. Finally, we'll introduce Matplotlib.

Our recipes are very introductory – each of these libraries could easily occupy a full book, but the recipes should be enough to help you through this book. If you are using Docker, and because all these libraries are fundamental for data analysis, they can be found in the `tiagoantao/bioinformatics_base` Docker image from *Chapter 1*.

In this chapter, we will cover the following recipes:

- Using pandas to process vaccine-adverse events
- Dealing with the pitfalls of joining pandas DataFrames
- Reducing the memory usage of pandas DataFrames
- Accelerating pandas processing with Apache Arrow
- Understanding NumPy as the engine behind Python data science and bioinformatics
- Introducing Matplotlib for chart generation

Using pandas to process vaccine-adverse events

We will be introducing pandas with a concrete bioinformatics data analysis example: we will be studying data from the **Vaccine Adverse Event Reporting System** (**VAERS**, `https://vaers.hhs.gov/`). VAERS, which is maintained by the US Department of Health and Human Services, includes a database of vaccine-adverse events going back to 1990.

VAERS makes data available in **comma-separated values** (**CSV**) format. The CSV format is quite simple and can even be opened with a simple text editor (be careful with very large file sizes as they may crash your editor) or a spreadsheet such as Excel. pandas can work very easily with this format.

Getting ready

First, we need to download the data. It is available at `https://vaers.hhs.gov/data/datasets.html`. Please download the ZIP file: we will be using the 2021 file; do not download a single CSV file only. After downloading the file, unzip it, and then recompress all the files individually with `gzip -9 *csv` to save disk space.

Feel free to have a look at the files with a text editor, or preferably with a tool such as `less` (`zless` for compressed files). You can find documentation for the content of the files at `https://vaers.hhs.gov/docs/VAERSDataUseGuide_en_September2021.pdf`.

If you are using the Notebooks, code is provided at the beginning of them so that you can take care of the necessary processing. If you are using Docker, the base image is enough.

The code can be found in `Chapter02/Pandas_Basic.py`.

How to do it...

Follow these steps:

1. Let's start by loading the main data file and gathering the basic statistics:

```
vdata = pd.read_csv(
    "2021VAERSDATA.csv.gz", encoding="iso-8859-1")
vdata.columns
vdata.dtypes
vdata.shape
```

   We start by loading the data. In most cases, there is no need to worry about the text encoding as the default, UTF-8, will work, but in this case, the text encoding is `legacy iso-8859-1`. Then, we print the column names, which start with `VAERS_ID`, `RECVDATE`, `STATE`, AGE_YRS, and so on. They include 35 entries corresponding to each of the columns. Then, we print the types of each column. Here are the first few entries:

```
VAERS_ID          int64
RECVDATE         object
STATE            object
AGE_YRS         float64
CAGE_YR         float64
CAGE_MO         float64
SEX              object
```

By doing this, we get the shape of the data: `(654986, 35)`. This means 654,986 rows and 35 columns. You can use any of the preceding strategies to get the information you need regarding the metadata of the table.

2. Now, let's explore the data:

```
vdata.iloc[0]
vdata = vdata.set_index("VAERS_ID")
vdata.loc[916600]
vdata.head(3)
vdata.iloc[:3]
vdata.iloc[:5, 2:4]
```

There are many ways we can look at the data. We will start by inspecting the first row, based on location. Here is an abridged version:

```
VAERS_ID                                              916600
RECVDATE                                              01/01/2021
STATE                                                 TX
AGE_YRS                                               33.0
CAGE_YR                                               33.0
CAGE_MO                                               NaN
SEX                                                   F
…
TODAYS_
DATE                                                 01/01/2021
BIRTH_DEFECT                                         NaN
OFC_VISIT                                            Y
```

```
ER_ED_VISIT                                              NaN
ALLERGIES                                            Pcn and
bee venom
```

After we index by VAERS_ID, we can use one ID to get a row. We can use 916600 (which is the ID from the preceding record) and get the same result.

Then, we retrieve the first three rows. Notice the two different ways we can do so:

- Using the head method
- Using the more general array specification; that is, iloc[:3]

Finally, we retrieve the first five rows, but only the second and third columns –iloc[:5, 2:4]. Here is the output:

```
          AGE_YRS  CAGE_YR
VAERS_ID
916600       33.0     33.0
916601       73.0     73.0
916602       23.0     23.0
916603       58.0     58.0
916604       47.0     47.0
```

3. Let's do some basic computations now, namely computing the maximum age in the dataset:

```
vdata["AGE_YRS"].max()
vdata.AGE_YRS.max()
```

The maximum value is 119 years. More importantly than the result, notice the two dialects for accessing AGE_YRS (as a dictionary key and as an object field) for the access columns.

4. Now, let's plot the ages involved:

```
vdata["AGE_YRS"].sort_values().plot(use_index=False)
vdata["AGE_YRS"].plot.hist(bins=20)
```

This generates two plots (a condensed version is shown in the following step). We use pandas plotting machinery here, which uses Matplotib underneath.

5. While we have a full recipe for charting with Matplotlib (*Introducing Matplotlib for chart generation*), let's have a sneak peek here by using it directly:

```
import matplotlib.pylot as plt
fig, ax = plt.subplots(1, 2, sharey=True)
fig.suptitle("Age of adverse events")
vdata["AGE_YRS"].sort_values().plot(
    use_index=False, ax=ax[0],
    xlabel="Obervation", ylabel="Age")
vdata["AGE_YRS"].plot.hist(bins=20,
orientation="horizontal")
```

This includes both figures from the previous steps. Here is the output:

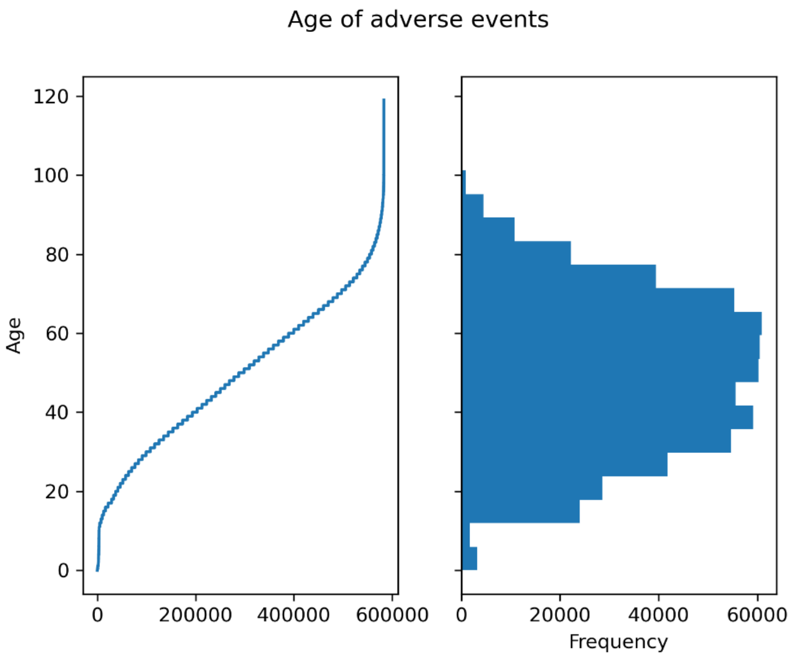


Figure 2.1 – Left – the age for each observation of adverse effect; right – a histogram showing the distribution of ages

6. We can also take a non-graphical, more analytical approach, such as counting the events per year:

```
vdata["AGE_YRS"].dropna().apply(lambda x: int(x)).value_
counts()
```

The output will be as follows:

```
50     11006
65     10948
60     10616
51     10513
58     10362
      ...
```

7. Now, let's see how many people died:

```
vdata.DIED.value_counts(dropna=False)
vdata["is_dead"] = (vdata.DIED == "Y")
```

The output of the count is as follows:

```
NaN    646450
Y        8536
Name: DIED, dtype: int64
```

Note that the type of `DIED` is *not* a Boolean. It's more declarative to have a Boolean representation of a Boolean characteristic, so we create `is_dead` for it.

Tip

Here, we are assuming that NaN is to be interpreted as `False`. In general, we must be careful with the interpretation of NaN. It may mean `False` or it may simply mean – as in most cases – a lack of data. If that were the case, it should not be converted into `False`.

8. Now, let's associate the individual data about deaths with the type of vaccine involved:

```
dead = vdata[vdata.is_dead]
vax = pd.read_csv("2021VAERSVAX.csv.gz",
encoding="iso-8859-1").set_index("VAERS_ID")
vax.groupby("VAX_TYPE").size().sort_values()
vax19 = vax[vax.VAX_TYPE == "COVID19"]
vax19_dead = dead.join(vax19)
```

After we get a DataFrame containing just deaths, we must read the data that contains vaccine information. First, we must do some exploratory analysis of the types of vaccines and their adverse events. Here is the abridged output:

```
            …
HPV9          1506
```

```
FLU4          3342
UNK           7941
VARZOS       11034
COVID19     648723
```

After that, we must choose just the COVID-related vaccines and join them with individual data.

9. Finally, let's see the top 10 COVID vaccine lots that are overrepresented in terms of deaths and how many US states were affected by each lot:

```
baddies = vax19_dead.groupby("VAX_LOT").size().sort_
values(ascending=False)
for I, (lot, cnt) in enumerate(baddies.items()):
    print(lot, cnt, len(vax19_dead[vax19_dead.VAX_LOT ==
lot].groupby""STAT"")))
    if i == 10:
        break
```

The output is as follows:

```
Unknown 254 34
EN6201 120 30
EN5318 102 26
EN6200 101 22
EN6198 90 23
039K20A 89 13
EL3248 87 17
EL9261 86 21
EM9810 84 21
EL9269 76 18
EN6202 75 18
```

That concludes this recipe!

There's more...

The preceding data about vaccines and lots is not completely correct; we will cover some data analysis pitfalls in the next recipe.

In the *Introducing Matplotlib for chart generation* recipe, we will introduce Matplotlib, a chart library that provides the backend for pandas plotting. It is a fundamental component of Python's data analysis ecosystem.

See also

The following is some extra information that may be useful:

- While the first three recipes of this chapter are enough to support you throughout this book, there is plenty of content available on the web to help you understand pandas. You can start with the main user guide, which is available at `https://pandas.pydata.org/docs/user_guide/index.html`.
- If you need to plot data, do not forget to check the visualization part of the guide since it is especially helpful: `https://pandas.pydata.org/docs/user_guide/visualization.html`.

Dealing with the pitfalls of joining pandas DataFrames

The previous recipe was a whirlwind tour that introduced pandas and exposed most of the features that we will use in this book. While an exhaustive discussion about pandas would require a complete book, in this recipe – and in the next one – we are going to discuss topics that impact data analysis and are seldom discussed in the literature but are very important.

In this recipe, we are going to discuss some pitfalls that deal with relating DataFrames through joins: it turns out that many data analysis errors are introduced by carelessly joining data. We will introduce techniques to reduce such problems here.

Getting ready

We will be using the same data as in the previous recipe, but we will jumble it a bit so that we can discuss typical data analysis pitfalls. Once again, we will be joining the main adverse events table with the vaccination table, but we will randomly sample 90% of the data from each. This mimics, for example, the scenario where you only have incomplete information. This is one of the many examples where joins between tables do not have intuitively obvious results.

Use the following code to prepare our files by randomly sampling 90% of the data:

```
vdata = pd.read_csv("2021VAERSDATA.csv.gz",
encoding="iso-8859-1")
vdata.sample(frac=0.9).to_csv("vdata_sample.csv.gz",
index=False)
vax = pd.read_csv("2021VAERSVAX.csv.gz", encoding="iso-8859-1")
vax.sample(frac=0.9).to_csv("vax_sample.csv.gz", index=False)
```

Because this code involves random sampling, the results that you will get will be different from the ones reported here. If you want to get the same results, I have provided the files that I used in the `Chapter02` directory. The code for this recipe can be found in `Chapter02/Pandas_Join.py`.

How to do it...

Follow these steps:

1. Let's start by doing an inner join of the individual and vaccine tables:

```
vdata = pd.read_csv("vdata_sample.csv.gz")
vax = pd.read_csv("vax_sample.csv.gz")
vdata_with_vax = vdata.join(
    vax.set_index("VAERS_ID"),
    on="VAERS_ID",
    how="inner")
len(vdata), len(vax), len(vdata_with_vax)
```

   The `len` output for this code is 589,487 for the individual data, 620,361 for the vaccination data, and 558,220 for the join. This suggests that some individual and vaccine data was not captured.

2. Let's find the data that was not captured with the following join:

```
lost_vdata = vdata.loc[~vdata.index.isin(vdata_with_vax.
index)]
lost_vdata
lost_vax = vax[~vax["VAERS_ID"].isin(vdata.index)]
lost_vax
```

   You will see that 56,524 rows of individual data aren't joined and that there are 62,141 rows of vaccinated data.

3. There are other ways to join data. The default way is by performing a left outer join:

```
vdata_with_vax_left = vdata.join(
    vax.set_index("VAERS_ID"),
    on="VAERS_ID")
vdata_with_vax_left.groupby("VAERS_ID").size().sort_
values()
```

   A left outer join assures that all the rows on the left table are always represented. If there are no rows on the right, then all the right columns will be filled with `None` values.

Warning

There is a caveat that you should be careful with. Remember that the left table – `vdata` – had one entry per `VAERS_ID`. When you left join, you may end up with a case where the left-hand side is repeated several times. For example, the `groupby` operation that we did previously shows that `VAERS_ID` of 962303 has 11 entries. This is correct, but it's not uncommon to have the incorrect expectation that you will still have a single row on the output per row on the left-hand side. This is because the left join returns 1 or more left entries, whereas the inner join above returns 0 or 1 entries, where sometimes, we would like to have precisely 1 entry. Be sure to always test the output for what you want in terms of the number of entries.

4. There is a right join as well. Let's right join COVID vaccines – the left table – with death events – the right table:

```
dead = vdata[vdata.DIED == "Y"]
vax19 = vax[vax.VAX_TYPE == "COVID19"]
vax19_dead = vax19.join(dead.set_index("VAERS_ID"),
on="VAERS_ID", how="right")
len(vax19), len(dead), len(vax19_dead)
len(vax19_dead[vax19_dead.VAERS_ID.duplicated()])
len(vax19_dead) - len(dead)
```

   As you may expect, a right join will ensure that all the rows on the right table are represented. So, we end up with 583,817 COVID entries, 7,670 dead entries, and a right join of 8,624 entries.

   We also check the number of duplicated entries on the joined table and we get 954. If we subtract the length of the dead table from the joined table, we also get, as expected, 954. Make sure you have checks like this when you're making joins.

5. Finally, we are going to revisit the problematic COVID lot calculations since we now understand that we might be overcounting lots:

```
vax19_dead["STATE"] = vax19_dead["STATE"].str.upper()
dead_lot = vax19_dead[["VAERS_ID", "VAX_LOT", "STATE"]].
set_index(["VAERS_ID", "VAX_LOT"])
dead_lot_clean = dead_lot[~dead_lot.index.duplicated()]
dead_lot_clean = dead_lot_clean.reset_index()
dead_lot_clean[dead_lot_clean.VAERS_ID.isna()]
baddies = dead_lot_clean.groupby("VAX_LOT").size().sort_
values(ascending=False)
for i, (lot, cnt) in enumerate(baddies.items()):
    print(lot, cnt, len(dead_lot_clean[dead_lot_clean.
VAX_LOT == lot].groupby("STATE")))
    if i == 10:
        break
```

Note that the strategies that we've used here ensure that we don't get repeats: first, we limit the number of columns to the ones we will be using, then we remove repeated indexes and empty `VAERS_ID`. This ensures no repetition of the `VAERS_ID`, `VAX_LOT` pair, and that no lots are associated with no IDs.

There's more...

There are other types of joins other than left, inner, and right. Most notably, there is the outer join, which assures all entries from both tables have representation.

Make sure you have tests and assertions for your joins: a very common bug is having the wrong expectations for how joins behave. You should also make sure that there are no empty values on the columns where you are joining, as they can produce a lot of excess tuples.

Reducing the memory usage of pandas DataFrames

When you are dealing with lots of information – for example, when analyzing whole genome sequencing data – memory usage may become a limitation for your analysis. It turns out that naïve pandas is not very efficient from a memory perspective, and we can substantially reduce its consumption.

In this recipe, we are going to revisit our VAERS data and look at several ways to reduce pandas memory usage. The impact of these changes can be massive: in many cases, reducing memory consumption may mean the difference between being able to use pandas or requiring a more alternative and complex approach, such as Dask or Spark.

Getting ready

We will be using the data from the first recipe. If you have run it, you are all set; if not, please follow the steps discussed there. You can find this code in `Chapter02/Pandas_Memory.py`.

How to do it...

Follow these steps:

1. First, let's load the data and inspect the size of the DataFrame:

```
import numpy as np
import pandas as pd
vdata = pd.read_csv("2021VAERSDATA.csv.gz",
encoding="iso-8859-1")
vdata.info(memory_usage="deep")
```

Here is an abridged version of the output:

```
RangeIndex: 654986 entries, 0 to 654985
Data columns (total 35 columns):
#   Column        Non-Null Count   Dtype
---  ------        --------------   -----
0   VAERS_ID      654986 non-null  int64
2   STATE         572236 non-null  object
3   AGE_YRS       583424 non-null  float64
6   SEX           654986 non-null  object
8   SYMPTOM_TEXT  654828 non-null  object
9   DIED          8536 non-null    object
31  BIRTH_DEFECT  383 non-null     object
34  ALLERGIES     330630 non-null  object
dtypes: float64(5), int64(2), object(28)
memory usage: 1.3 GB
```

Here, we have information about the number of rows and the type and non-null values of each row. Finally, we can see that the DataFrame requires a whopping 1.3 GB.

2. We can also inspect the size of each column:

```
for name in vdata.columns:
    col_bytes = vdata[name].memory_usage(index=False,
deep=True)
    col_type = vdata[name].dtype
    print(
        name,
        col_type, col_bytes // (1024 ** 2))
```

Here is an abridged version of the output:

```
VAERS_ID int64 4
STATE object 34
AGE_YRS float64 4
SEX object 36
RPT_DATE object 20
SYMPTOM_TEXT object 442
```

```
DIED object 20
ALLERGIES object 34
```

`SYMPTOM_TEXT` occupies 442 MB, so 1/3 of our entire table.

3. Now, let's look at the `DIED` column. Can we find a more efficient representation?

```
vdata.DIED.memory_usage(index=False, deep=True)
vdata.DIED.fillna(False).astype(bool).memory_
usage(index=False, deep=True)
```

The original column takes 21,181,488 bytes, whereas our compact representation takes 656,986 bytes. That's 32 times less!

4. What about the `STATE` column? Can we do better?

```
vdata["STATE"] = vdata.STATE.str.upper()
states = list(vdata["STATE"].unique())
vdata["encoded_state"] = vdata.STATE.apply(lambda state:
states.index(state))
vdata["encoded_state"] = vdata["encoded_state"].
astype(np.uint8)
vdata["STATE"].memory_usage(index=False, deep=True)
vdata["encoded_state"].memory_usage(index=False,
deep=True)
```

Here, we convert the `STATE` column, which is text, into `encoded_state`, which is a number. This number is the position of the state's name in the list state. We use this number to look up the list of states. The original column takes around 36 MB, whereas the encoded column takes 0.6 MB.

As an alternative to this approach, you can look at categorical variables in pandas. I prefer to use them as they have wider applications.

5. We can apply most of these optimizations when we *load* the data, so let's prepare for that. But now, we have a chicken-and-egg problem: to be able to know the content of the state table, we have to do a first pass to get the list of states, like so:

```
states = list(pd.read_csv(
    "vdata_sample.csv.gz",
    converters={
       "STATE": lambda state: state.upper()
    },
    usecols=["STATE"]
)["STATE"].unique())
```

We have a converter that simply returns the uppercase version of the state. We only return the `STATE` column to save memory and processing time. Finally, we get the `STATE` column from the DataFrame (which has only a single column).

6. The ultimate optimization is *not* to load the data. Imagine that we don't need `SYMPTOM_TEXT` – that is around 1/3 of the data. In that case, we can just skip it. Here is the final version:

```
vdata = pd.read_csv(
    "vdata_sample.csv.gz",
    index_col="VAERS_ID",
    converters={
       "DIED": lambda died: died == "Y",
       "STATE": lambda state: states.index(state.upper())
    },
    usecols=lambda name: name != "SYMPTOM_TEXT"
)
vdata["STATE"] = vdata["STATE"].astype(np.uint8)
vdata.info(memory_usage="deep")
```

We are now at 714 MB, which is a bit over half of the original. This could be still substantially reduced by applying the methods we used for `STATE` and `DIED` to all other columns.

See also

The following is some extra information that may be useful:

- If you are willing to use a support library to help with Python processing, check the next recipe on Apache Arrow, which will allow you to have extra memory savings for more memory efficiency.
- If you end up with DataFrames that take more memory than you have available on a single machine, then you must step up your game and use chunking - which we will not cover in the Pandas context - or something that can deal with large data automatically. Dask, which we'll cover in *Chapter 11*, *Parallel Processing with Dask and Zarr*, allows you to work with larger-than-memory datasets with, among others, a pandas-like interface.

Accelerating pandas processing with Apache Arrow

When dealing with large amounts of data, such as in whole genome sequencing, pandas is both slow and memory-consuming. Apache Arrow provides faster and more memory-efficient implementations of several pandas operations and can interoperate with it.

Apache Arrow is a project co-founded by Wes McKinney, the founder of pandas, and it has several objectives, including working with tabular data in a language-agnostic way, which allows for language interoperability while providing a memory- and computation-efficient implementation. Here, we will only be concerned with the second part: getting more efficiency for large-data processing. We will do this in an integrated way with pandas.

Here, we will once again use VAERS data and show how Apache Arrow can be used to accelerate pandas data loading and reduce memory consumption.

Getting ready

Again, we will be using data from the first recipe. Be sure you download and prepare it, as explained in the *Getting ready* section of the *Using pandas to process vaccine-adverse events* recipe. The code is available in `Chapter02/Arrow.py`.

How to do it...

Follow these steps:

1. Let's start by loading the data using both pandas and Arrow:

```
import gzip
import pandas as pd
from pyarrow import csv
import pyarrow.compute as pc
vdata_pd = pd.read_csv("2021VAERSDATA.csv.gz",
encoding="iso-8859-1")
columns = list(vdata_pd.columns)
vdata_pd.info(memory_usage="deep")
vdata_arrow = csv.read_csv("2021VAERSDATA.csv.gz")
tot_bytes = sum([
    vdata_arrow[name].nbytes
    for name in vdata_arrow.column_names])
print(f"Total {tot_bytes // (1024 ** 2)} MB")
```

pandas requires 1.3 GB, whereas Arrow requires 614 MB: less than half the memory. For large files like this, this may mean the difference between being able to process data in memory or needing to find another solution, such as Dask. While some functions in Arrow have similar names to pandas (for example, `read_csv`), that is not the most common occurrence. For example, note the way we compute the total size of the DataFrame: by getting the size of each column and performing a sum, which is a different approach from pandas.

2. Let's do a side-by-side comparison of the inferred types:

```
for name in vdata_arrow.column_names:
    arr_bytes = vdata_arrow[name].nbytes
    arr_type = vdata_arrow[name].type
    pd_bytes = vdata_pd[name].memory_usage(index=False,
deep=True)
    pd_type = vdata_pd[name].dtype
    print(
        name,
        arr_type, arr_bytes // (1024 ** 2),
        pd_type, pd_bytes // (1024 ** 2),)
```

Here is an abridged version of the output:

```
VAERS_ID int64 4 int64 4
RECVDATE string 8 object 41
STATE string 3 object 34
CAGE_YR int64 5 float64 4
SEX string 3 object 36
RPT_DATE string 2 object 20
DIED string 2 object 20
L_THREAT string 2 object 20
ER_VISIT string 2 object 19
HOSPITAL string 2 object 20
HOSPDAYS int64 5 float64 4
```

As you can see, Arrow is generally more specific with type inference and is one of the main reasons why memory usage is substantially lower.

3. Now, let's do a time performance comparison:

```
%timeit pd.read_csv("2021VAERSDATA.csv.gz",
encoding="iso-8859-1")
%timeit csv.read_csv("2021VAERSDATA.csv.gz")
```

On my computer, the results are as follows:

```
7.36 s ± 201 ms per loop (mean ± std. dev. of 7 runs, 1
loop each)
2.28 s ± 70.7 ms per loop (mean ± std. dev. of 7 runs, 1
loop each)
```

Arrow's implementation is three times faster. The results on your computer will vary as this is dependent on the hardware.

4. Let's repeat the memory occupation comparison while not loading the `SYMPTOM_TEXT` column. This is a fairer comparison as most numerical datasets do not tend to have a very large text column:

```
vdata_pd = pd.read_csv("2021VAERSDATA.csv.gz",
encoding="iso-8859-1", usecols=lambda x: x != "SYMPTOM_
TEXT")
vdata_pd.info(memory_usage="deep")
columns.remove("SYMPTOM_TEXT")
vdata_arrow = csv.read_csv(
    "2021VAERSDATA.csv.gz",
     convert_options=csv.ConvertOptions(include_
columns=columns))
vdata_arrow.nbytes
```

pandas requires 847 MB, whereas Arrow requires 205 MB: four times less.

5. Our objective is to use Arrow to load data into pandas. For that, we need to convert the data structure:

```
vdata = vdata_arrow.to_pandas()
vdata.info(memory_usage="deep")
```

There are two very important points to be made here: the pandas representation created by Arrow uses only 1 GB, whereas the pandas representation, from its native `read_csv`, is 1.3 GB. This means that even if you use pandas to process data, Arrow can create a more compact representation to start with.

The preceding code has one problem regarding memory consumption: when the converter is running, it will require memory to hold *both* the pandas and the Arrow representations, hence defeating the purpose of using less memory. Arrow can self-destruct its representation while creating the pandas version, hence resolving the problem. The line for this is `vdata = vdata_arrow.to_pandas(self_destruct=True)`.

There's more...

If you have a very large DataFrame that cannot be processed by pandas, even after it's been loaded by Arrow, then maybe Arrow can do all the processing as it has a computing engine as well. That being said, Arrow's engine is, at the time of writing, substantially less complete in terms of functionality than pandas. Remember that Arrow has many other features, such as language interoperability, but we will not be making use of those in this book.

Understanding NumPy as the engine behind Python data science and bioinformatics

Most of your analysis will make use of NumPy, even if you don't use it explicitly. NumPy is an array manipulation library that is behind libraries such as pandas, Matplotlib, Biopython, and scikit-learn, among many others. While much of your bioinformatics work may not require explicit direct use of NumPy, you should be aware of its existence as it underpins almost everything you do, even if only indirectly via the other libraries.

In this recipe, we will use VAERS data to demonstrate how NumPy is behind many of the core libraries that we use. This is a very light introduction to the library so that you are aware that it exists and that it is behind almost everything. Our example will extract the number of cases from the five US states with more adverse effects, splitting them into age bins: 0 to 19 years, 20 to 39, up to 100 to 119.

Getting ready

Once again, we will be using the data from the first recipe, so make sure it's available. The code for it can be found in `Chapter02/NumPy.py`.

How to do it...

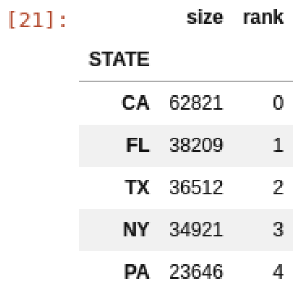
Follow these steps:

1. Let's start by loading the data with pandas and reducing the data so that it's related to the top five US states only:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
vdata = pd.read_csv(
    "2021VAERSDATA.csv.gz", encoding="iso-8859-1")
vdata["STATE"] = vdata["STATE"].str.upper()
top_states = pd.DataFrame({
    "size": vdata.groupby("STATE").size().sort_
values(ascending=False).head(5)}).reset_index()
top_states["rank"] = top_states.index
top_states = top_states.set_index("STATE")
top_vdata = vdata[vdata["STATE"].isin(top_states.index)]
top_vdata["state_code"] = top_vdata["STATE"].apply(
    lambda state: top_states["rank"].at[state]
).astype(np.uint8)
```

```
top_vdata = top_vdata[top_vdata["AGE_YRS"].notna()]
top_vdata.loc[:,"AGE_YRS"] = top_vdata["AGE_YRS"].
astype(int)
top_states
```

The top states are as follows. This rank will be used later to construct a NumPy matrix:



[21]:

STATE	size	rank
CA	62821	0
FL	38209	1
TX	36512	2
NY	34921	3
PA	23646	4

Figure 2.2 – US states with largest numbers of adverse effects

2. Now, let's extract the two NumPy arrays that contain age and state data:

```
age_state = top_vdata[["state_code", "AGE_YRS"]]
age_state["state_code"]
state_code_arr = age_state["state_code"].values
type(state_code_arr), state_code_arr.shape, state_code_
arr.dtype
age_arr = age_state["AGE_YRS"].values
type(age_arr), age_arr.shape, age_arr.dtype
```

Note that the data that underlies pandas is NumPy data (the `values` call for both Series returns NumPy types). Also, you may recall that pandas has properties such as `.shape` or `.dtype`: these were inspired by NumPy and behave the same.

3. Now, let's create a NumPy matrix from scratch (a 2D array), where each row is a state and each column represents an age group:

```
age_state_mat = np.zeros((5,6), dtype=np.uint64)
for row in age_state.itertuples():
    age_state_mat[row.state_code, row.AGE_YRS//20] += 1
age_state_mat
```

The array has five rows – one for each state – and six columns – one for each age group. All the cells in the array must have the same type.

We initialize the array with zeros. There are many ways to initialize arrays, but if you have a very large array, initializing it may take a lot of time. Sometimes, depending on your task, it might be OK that the array is empty at the beginning (meaning it was initialized with random trash). In that case, using `np.empty` will be much faster. We use pandas iteration here: this is not the best way to do things from a pandas perspective, but we want to make the NumPy part very explicit.

4. We can extract a single row – in our case, the data for a state – very easily. The same applies to a column. Let's take California data and then the 0-19 age group:

```
cal = age_state_mat[0,:]
kids = age_state_mat[:,0]
```

Note the syntax to extract a row or a column. It should be familiar to you, given that pandas copied the syntax from NumPy and we encountered it in previous recipes.

5. Now, let's compute a new matrix where we have the fraction of cases per age group:

```
def compute_frac(arr_1d):
    return arr_1d / arr_1d.sum()
frac_age_stat_mat = np.apply_along_axis(compute_frac, 1,
age_state_mat)
```

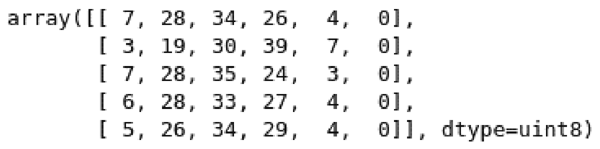
The last line applies the `compute_frac` function to all rows. `compute_frac` takes a single row and returns a new row where all the elements are divided by the total sum.

6. Now, let's create a new matrix that acts as a percentage instead of a fraction – simply because it reads better:

```
perc_age_stat_mat = frac_age_stat_mat * 100
perc_age_stat_mat = perc_age_stat_mat.astype(np.uint8)
perc_age_stat_mat
```

The first line simply multiplies all the elements of the 2D array by 100. Matplotlib is smart enough to traverse different array structures. That line will work if it's presented with an array with any dimensions and would do exactly what is expected.

Here is the result:



```
array([[ 7, 28, 34, 26,  4,  0],
       [ 3, 19, 30, 39,  7,  0],
       [ 7, 28, 35, 24,  3,  0],
       [ 6, 28, 33, 27,  4,  0],
       [ 5, 26, 34, 29,  4,  0]], dtype=uint8)
```

Figure 2.3 – A matrix representing the distribution of vaccine-adverse effects in the five US states with the most cases

7. Finally, let's create a graphical representation of the matrix using Matplotlib:

```
fig = plt.figure()
ax = fig.add_subplot()
ax.matshow(perc_age_stat_mat, cmap=plt.get_cmap("Greys"))
ax.set_yticks(range(5))
ax.set_yticklabels(top_states.index)
ax.set_xticks(range(6))
ax.set_xticklabels(["0-19", "20-39", "40-59", "60-79",
"80-99", "100-119"])
fig.savefig("matrix.png")
```

Do not dwell too much on the Matplotlib code – we are going to discuss it in the next recipe. The fundamental point here is that you can pass NumPy data structures to Matplotlib. Matplotlib, like pandas, is based on NumPy.

See also

The following is some extra information that may be useful:

- NumPy has many more features than the ones we've discussed here. There are plenty of books and tutorials on them. The official documentation is a good place to start: `https://numpy.org/doc/stable/`.
- There are many important issues to discover with NumPy, but probably one of the most important is broadcasting: NumPy's ability to take arrays of different structures and get the operations right. For details, go to `https://numpy.org/doc/stable/user/theory.broadcasting.html`.

Introducing Matplotlib for chart generation

Matplotlib is the most common Python library for generating charts. There are more modern alternatives, such as **Bokeh**, which is web-centered, but the advantage of Matplotlib is not only that it is the most widely available and widely documented chart library but also, in the computational biology world, we want a chart library that is both web- and paper-centric. This is because many of our charts will be submitted to scientific journals, which are equally concerned with both formats. Matplotlib can handle this for us.

Many of the examples in this recipe could also be done directly with pandas (hence indirectly with Matplotlib), but the point here is to exercise Matplotlib.

Once again, we are going to use VAERS data to plot some information about the DataFrame's metadata and summarize the epidemiological data.

Getting ready

Again, we will be using the data from the first recipe. The code can be found in `Chapter02/Matplotlib.py`.

How to do it...

Follow these steps:

1. The first thing that we will do is plot the fraction of nulls per column:

```
import numpy as np
import pandas as pd
import matplotlib as mpl
import matplotlib.pyplot as plt
vdata = pd.read_csv(
    "2021VAERSDATA.csv.gz", encoding="iso-8859-1",
    usecols=lambda name: name != "SYMPTOM_TEXT")
num_rows = len(vdata)
perc_nan = {}
for col_name in vdata.columns:
    num_nans = len(vdata[col_name][vdata[col_name].
isna()])
    perc_nan[col_name] = 100 * num_nans / num_rows
labels = perc_nan.keys()
bar_values = list(perc_nan.values())
x_positions = np.arange(len(labels))
```

`labels` is the column names that we are analyzing, `bar_values` is the fraction of null values, and `x_positions` is the location of the bars on the bar chart that we are going to plot next.

2. Here is the code for the first version of the bar plot:

```
fig = plt.figure()
fig.suptitle("Fraction of empty values per column")
ax = fig.add_subplot()
ax.bar(x_positions, bar_values)
ax.set_ylabel("Percent of empty values")
ax.set_ylabel("Column")
ax.set_xticks(x_positions)
ax.set_xticklabels(labels)
ax.legend()
fig.savefig("naive_chart.png")
```

We start by creating a figure object with a title. The figure will have a subplot that will contain the bar chart. We also set several labels and only used defaults. Here is the sad result:

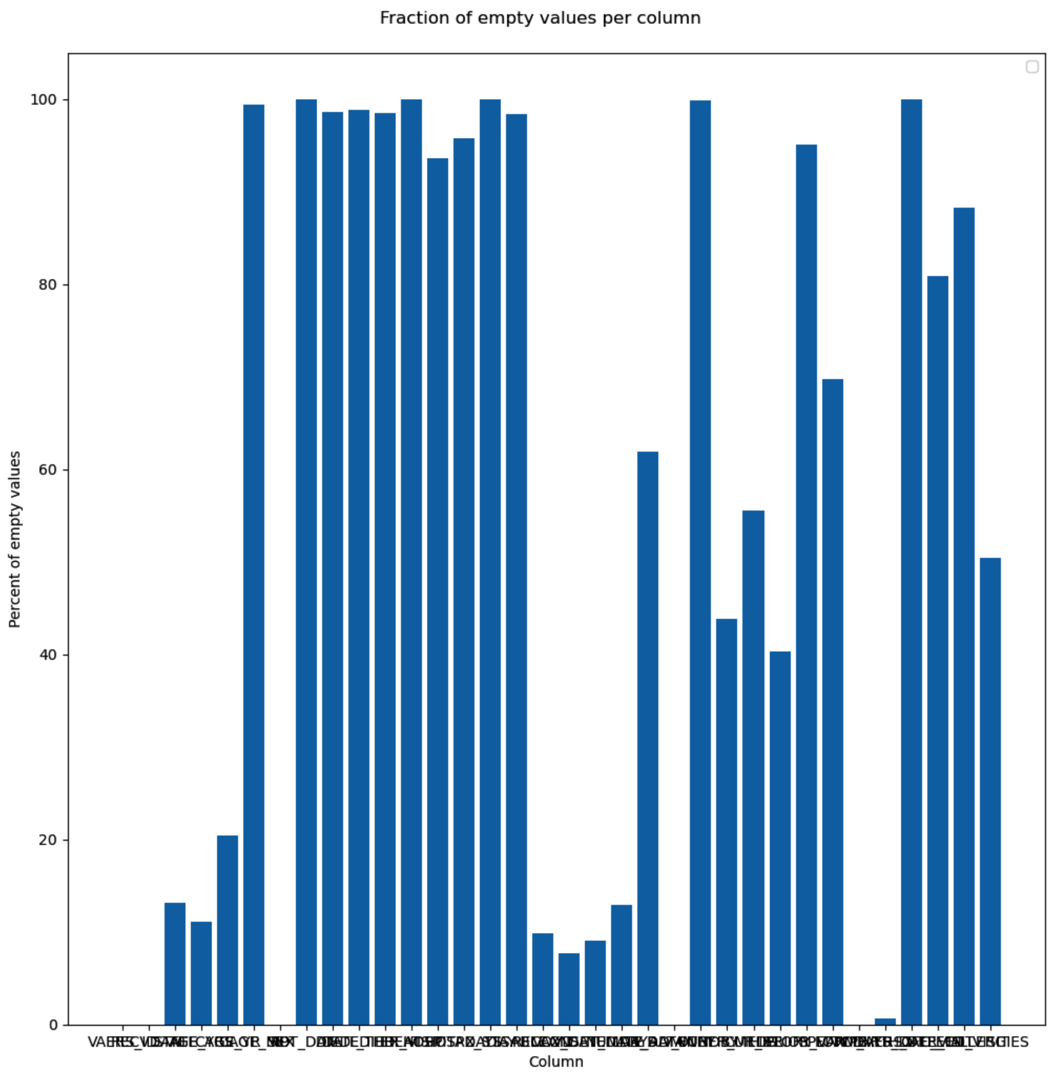


Figure 2.4 – Our first chart attempt, just using the defaults

3. Surely, we can do better. Let's format the chart substantially more:

```
fig = plt.figure(figsize=(16, 9), tight_layout=True,
dpi=600)
fig.suptitle("Fraction of empty values per column",
fontsize="48")
ax = fig.add_subplot()
b1 = ax.bar(x_positions, bar_values)
ax.set_ylabel("Percent of empty values", fontsize="xx-
large")
ax.set_xticks(x_positions)
ax.set_xticklabels(labels, rotation=45, ha="right")
ax.set_ylim(0, 100)
ax.set_xlim(-0.5, len(labels))
for i, x in enumerate(x_positions):
    ax.text(
        x, 2, "%.1f" % bar_values[i], rotation=90,
        va="bottom", ha="center",
        backgroundcolor="white")
fig.text(0.2, 0.01, "Column", fontsize="xx-large")
fig.savefig("cleaner_chart.png")
```

The first thing that we do is set up a bigger figure for Matplotlib to provide a tighter layout. We rotate the *x*-axis tick labels 45 degrees so that they fit better. We also put the values on the bars. Finally, we do not have a standard *x*-axis label as it would be on top of the tick labels. Instead, we write the text explicitly. Note that the coordinate system of the figure can be completely different from the coordinate system of the subplot – for example, compare the coordinates of `ax.text` and `fig.text`. Here is the result:

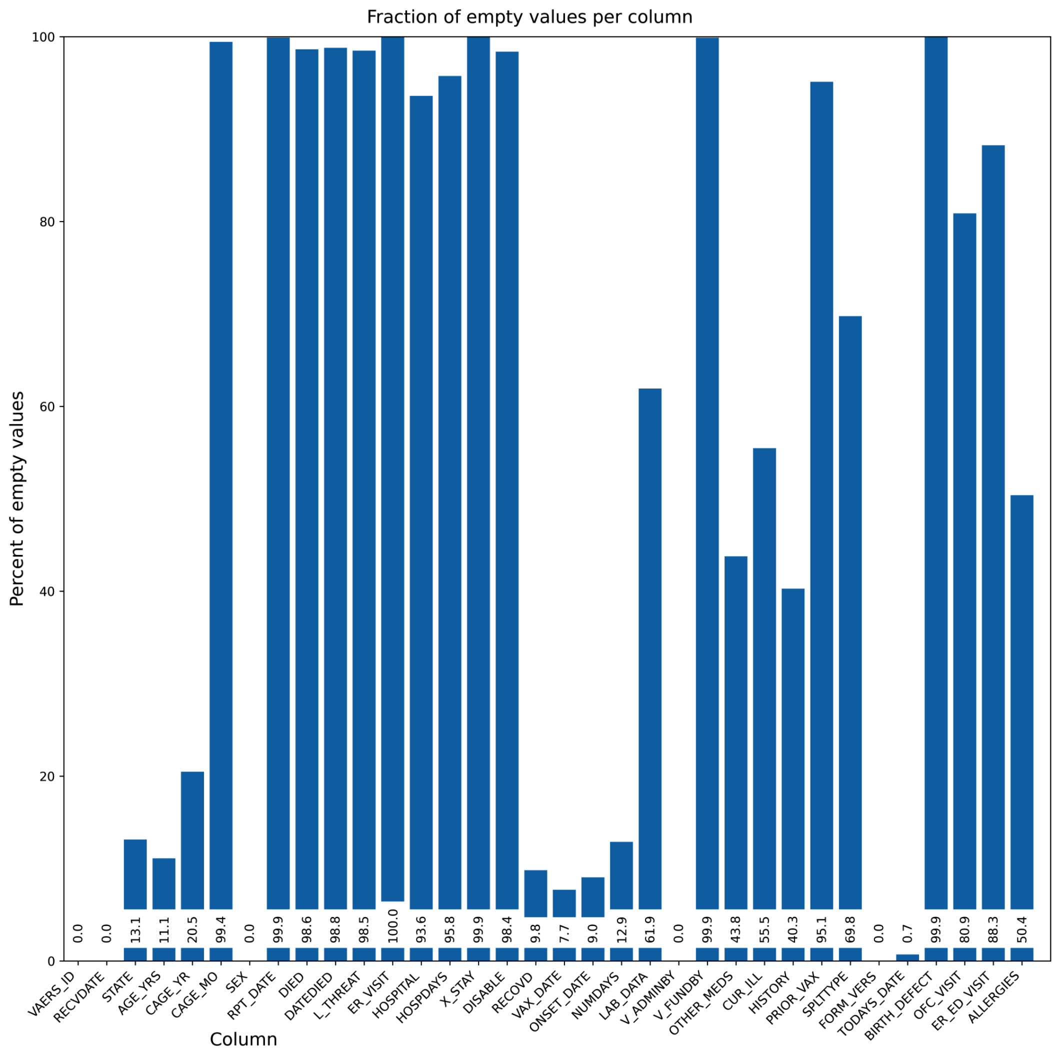


Figure 2.5 – Our second chart attempt, while taking care of the layout

4. Now, we are going to do some summary analysis of our data based on four plots on a single figure. We will chart the vaccines involved in deaths, the days between administration and death, the deaths over time, and the sex of people who have died for the top 10 states in terms of their quantity:

```
dead = vdata[vdata.DIED == "Y"]
vax = pd.read_csv("2021VAERSVAX.csv.gz",
encoding="iso-8859-1").set_index("VAERS_ID")
```

```
vax_dead = dead.join(vax, on="VAERS_ID", how="inner")

dead_counts = vax_dead["VAX_TYPE"].value_counts()
large_values = dead_counts[dead_counts >= 10]
other_sum = dead_counts[dead_counts < 10].sum()
large_values = large_values.append(pd.Series({"OTHER":
other_sum}))

distance_df = vax_dead[vax_dead.DATEDIED.notna() & vax_
dead.VAX_DATE.notna()]
distance_df["DATEDIED"] = pd.to_datetime(distance_
df["DATEDIED"])
distance_df["VAX_DATE"] = pd.to_datetime(distance_
df["VAX_DATE"])
distance_df = distance_df[distance_df.DATEDIED >= "2021"]
distance_df = distance_df[distance_df.VAX_DATE >= "2021"]
distance_df = distance_df[distance_df.DATEDIED >=
distance_df.VAX_DATE]
time_distances = distance_df["DATEDIED"] - distance_
df["VAX_DATE"]
time_distances_d = time_distances.astype(int) / (10**9 *
60 * 60 * 24)

date_died = pd.to_datetime(vax_dead[vax_dead.DATEDIED.
notna()]["DATEDIED"])
date_died = date_died[date_died >= "2021"]
date_died_counts = date_died.value_counts().sort_index()
cum_deaths = date_died_counts.cumsum()

state_dead = vax_dead[vax_dead["STATE"].notna()]
[["STATE", "SEX"]]
top_states = sorted(state_dead["STATE"].value_counts().
head(10).index)
top_state_dead = state_dead[state_dead["STATE"].isin(top_
states)].groupby(["STATE", "SEX"]).size()#.reset_index()
top_state_dead.loc["MN", "U"] = 0  # XXXX
top_state_dead = top_state_dead.sort_index().reset_
index()
```

```
top_state_females = top_state_dead[top_state_dead.SEX ==
"F"][0]
top_state_males = top_state_dead[top_state_dead.SEX ==
"M"][0]
top_state_unk = top_state_dead[top_state_dead.SEX == "U"]
[0]
```

The preceding code is strictly pandas-based and was made in preparation for the plotting activity.

5. The following code plots all the information simultaneously. We are going to have four subplots organized in 2 by 2 format:

```
fig, ((vax_cnt, time_dist), (death_time, state_reps)) =
plt.subplots(
    2, 2,
    figsize=(16, 9), tight_layout=True)
vax_cnt.set_title("Vaccines involved in deaths")
wedges, texts = vax_cnt.pie(large_values)
vax_cnt.legend(wedges, large_values.index, loc="lower
left")

time_dist.hist(time_distances_d, bins=50)
time_dist.set_title("Days between vaccine administration
and death")
time_dist.set_xlabel("Days")
time_dist.set_ylabel("Observations")

death_time.plot(date_died_counts.index, date_died_counts,
".")
death_time.set_title("Deaths over time")
death_time.set_ylabel("Daily deaths")
death_time.set_xlabel("Date")
tw = death_time.twinx()
tw.plot(cum_deaths.index, cum_deaths)
tw.set_ylabel("Cummulative deaths")

state_reps.set_title("Deaths per state stratified by
sex") state_reps.bar(top_states, top_state_females,
label="Females")
state_reps.bar(top_states, top_state_males,
label="Males", bottom=top_state_females)
state_reps.bar(top_states, top_state_unk,
label="Unknown",
                    bottom=top_state_females.values + top_
state_males.values)
state_reps.legend()
state_reps.set_xlabel("State")
state_reps.set_ylabel("Deaths")

fig.savefig("summary.png")
```

We start by creating a figure with 2x2 subplots. The `subplots` function returns, along with the figure object, four axes objects that we can use to create our charts. Note that the legend is positioned in the pie chart, we have used a twin axis on the time distance plot, and we have a way to compute stacked bars on the death per state chart. Here is the result:

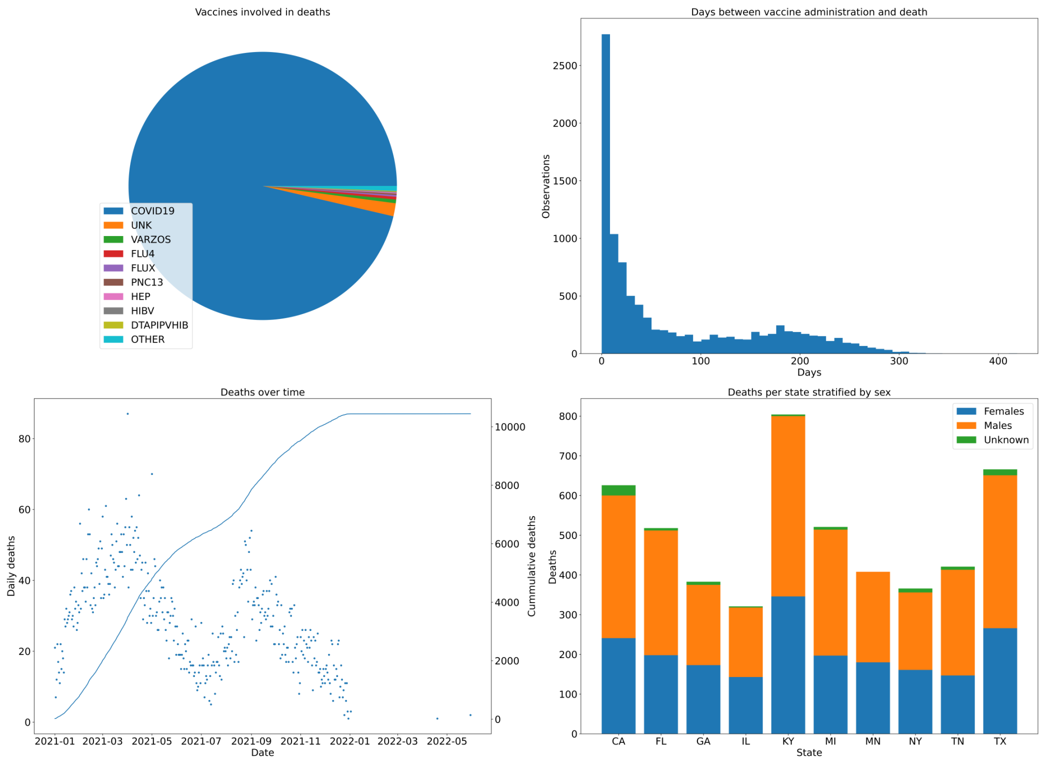


Figure 2.6 – Four combined charts summarizing the vaccine data

There's more...

Matplotlib has two interfaces you can use – an older interface, designed to be similar to MATLAB, and a more powerful **object-oriented** (**OO**) interface. Try as much as possible to avoid mixing the two. Using the OO interface is probably more future-proof. The MATLAB-like interface is below the `matplotlib.pyplot` module. To make things confusing, the entry points for the OO interface are in that module – that is, `matplotlib.pyplot.figure` and `matplotlib.pyplot.subplots`.

See also

The following is some extra information that may be useful:

- The documentation for Matplolib is really, really good. For example, there's a gallery of visual samples with links to the code for generating each sample. This can be found at `https://matplotlib.org/stable/gallery/index.html`. The API documentation is generally very complete.
- Another way to improve the looks of Matplotlib charts is to use the Seaborn library. Seaborn's main purpose is to add statistical visualization artifacts, but as a side effect, when imported, it changes the defaults of Matplotlib to something more palatable. We will be using Seaborn throughout this book; check out the plots provided in the next chapter.

3
Next-Generation Sequencing

Next-generation sequencing (**NGS**) is one of the fundamental technological developments of the century in life sciences. **Whole-genome sequencing** (**WGS**), **restriction site-associated DNA sequencing** (**RAD-Seq**), **ribonucleic acid sequencing** (**RNA-Seq**), **chromatin immunoprecipitation sequencing** (**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 of its own). We will use an existing WGS dataset—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 reference 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 *Chapter 5*, *Working with Genomes*.

As this is not an introductory book, you are expected to know at least what **FASTA** (**FASTA**), FASTQ, **Binary Alignment Map** (**BAM**), and **Variant Call Format** (**VCF**) files are. I will also make use of 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 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.

If you are running the content via Docker, you can use the `tiagoantao/bioinformatics_ngs` image. If you are using Anaconda Python, the required software for the chapter will be introduced in each recipe.

In this chapter, we will cover the following recipes:

- Accessing GenBank and moving around **National Center for Biotechnology Information** (**NCBI**) databases
- Performing basic sequence analysis
- Working with modern sequence formats
- Working with alignment data
- Extracting data from VCF files
- Studying genome accessibility and filtering **single-nucleotide polymorphism** (**SNP**) data
- Processing NGS data with HTSeq

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 from NCBI. We will not only discuss GenBank but also other databases from 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 `Chapter03/Accessing_Databases.py` file.

Tip

You will be accessing a live **application programming interface** (**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 `https://www.ncbi.nlm.nih.gov/books/NBK25497/#chapter2.Usage_Guidelines_and_Requiremen`. Notably, you are required to specify an email address with your query. You should try to avoid a 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 overuse 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:

1. 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.

2. 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 int(rec_list['RetMax']) < int(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 a large number of results (for more information, check the Biopython or NCBI Entrez documentation). Although you now have the **identifiers** (**IDs**) of all of the records, you still need to retrieve the records properly.

3. Now, let's try to retrieve all of these records. The following query will download all matching nucleotide sequences from GenBank, which is 1,374 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 number 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 the 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).

4. 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 *Chapter 5*, *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.

5. 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.

6. Now, let's extract some sequence features that 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` value 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 easily retrieve the sequence from a Python array with ranges, 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.

7. 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).

8. Last but not least, you can access a 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 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 ID, 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 **University of California, Santa Cruz** (**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 **Basic Local Alignment Search Tool** (**BLAST**). 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 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

This additional information will also be useful:

- 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 **question and answer** (**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 of 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 chapter and *Chapter 5*, *Working with Genomes*.

Getting ready

The code for this recipe is available in `Chapter03/Basic_Sequence_Processing.py`. 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, SeqRecord
Entrez.email = "your@email.here"
hdl = Entrez.efetch(db='nucleotide', id=['NM_002299'],
rettype='gb') # Lactase gene
gb_rec = SeqIO.read(hdl, 'gb')
```

Now that we have the GenBank record, let's extract the gene sequence. The record has a bit more than that, but let's get the precise location of the gene first:

```
for feature in gb_rec.features:
    if feature.type == 'CDS':
        location = feature.location  # Note translation
existing
cds = SeqRecord.SeqRecord(gb_rec.seq[location.start:location.
end], 'NM_002299', description='LCT CDS only')
```

Our example sequence is available on the Biopython sequence record.

How to do it...

Let's take a look at the following steps:

1. 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')
```

```
SeqIO.write([cds], w_hdl, 'fasta')
w_hdl.close()
```

The `SeqIO.write` function takes a list of sequences to write (in our case, it's 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 snippet) because this will allocate massive amounts of memory. Use either an iterator or the `SeqIO.write` function several times with a subset of the sequence on each write.

2. 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])
```

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 snippet. For our example, the preceding code will print the following output:

```
NM_002299 LCT CDS only
 ATGGAGCTGT
```

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

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

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

4. Finally, we can translate our gene into a protein:

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

Now, we have the amino acid sequence for our gene.

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 those of you 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 an 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 such as `reverse_complement`.
- The GenBank record from which we started includes a lot of metadata information about the sequence, so be sure to explore it.

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 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 hundreds of 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 that 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 6*, *Population Genetics*.

Tip

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 **Message Digest 5** (**MD5**) checksums. You can compare these checksums with the ones on the files you downloaded by using tools such as md5sum.

The instructions to download the data are at the top of the notebook, as specified in the first cell of `Chapter03/Working_with_FASTQ.py`. This is a fairly small file (27 **megabytes** (**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 snippet:

```
@SRR003258.1 30443AAXX:1:1:1053:1999 length=51
 ACCCCCCCCCACCCCCCCCCCCCCCCCCCCCCCCCACACACACCAACAC
 +
 =IIIIIIIII5IIIIIII>IIII+GIIIIIIIIIIIIII(IIIII01&III
```

Line 1 starts with `@`, followed by a sequence ID 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 sequenced DNA, which is just like a FASTA file. The third line is a + sign, sometimes followed by the description line on the first line.

The fourth line contains quality values for each base that's read on *line 2*. Each letter encodes a 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:

1. 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 **GNU ZIP** (**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.

Tip

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 such as 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 to perform a single iteration or all operations at once. The second option to is open a file several times and repeat the iteration.

2. 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 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

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 comes out of the sequencer unfiltered. In fact, you can even expect something more with regard to the spatial distribution of `N` calls.

3. Let's plot the distribution of `N`s according to their read position:

```
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])
fig.suptitle('Number of N calls as a function of the
distance from the start of the sequencer read')
ax.set_xlim(1, seq_len)
ax.set_xlabel('Read distance')
ax.set_ylabel('Number of N Calls')
```

We 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:

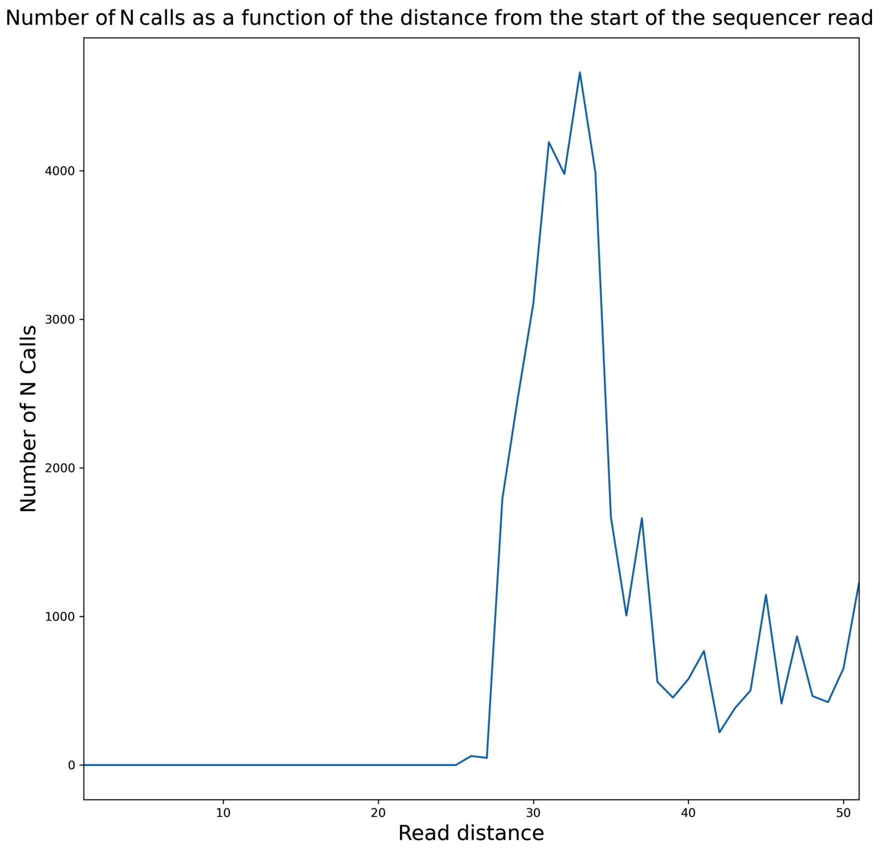


Figure 3.1 – The number of N calls as a function of the distance from the start of the sequencer read

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 the reads?

4. 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** (**bp**) 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.

Note

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 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).

5. 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)])
ax.set_xlabel('Read distance')
ax.set_ylabel('PHRED score')
fig.suptitle('Distribution of PHRED scores as a function
of read distance')
```

In this case, we will ignore both positions sequenced as `25` bp 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 the `boxplot` function of `seaborn`; we are only using this because the output looks slightly better than the standard `boxplot` function of `matplotlib`. 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:

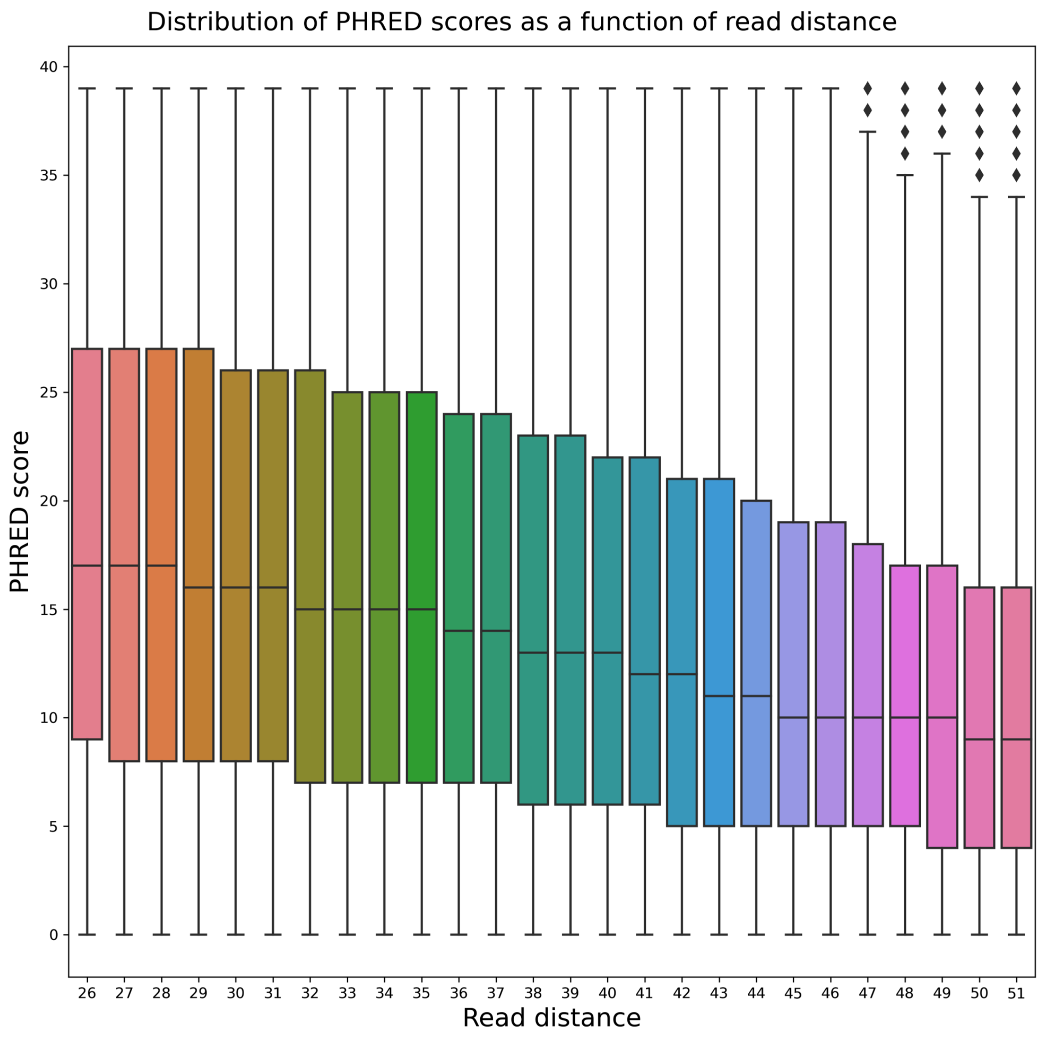


Figure 3.2 – The distribution of Phred scores as a function of the distance from the start of the sequencer read

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 the 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 regard to the programming technique, if you want to keep the pairing information, you might perform something such as this:

```
f1 = gzip.open('X_1.filt.fastq.gz', 'rt, enconding='utf-8')
f2 = gzip.open('X_2.filt.fastq.gz', 'rt, enconding='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` with your `FASTQ` prefix.

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

Here are some links with more information:

- The Wikipedia page on the FASTQ format is quite informative (`http://en.wikipedia.org/wiki/FASTQ_format`).
- 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`.
- 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 *Computational methods for discovering structural variation with next-generation sequencing* paper 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 *Chapter 5*, *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 **gigabytes** (**GB**), and the whole genome alignment (at 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.

The cell at the top of `Chapter03/Working_with_BAM.py` will download the data for you. 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. You can install it with the following command:

```
conda install -c bioconda pysam
```

OK—let's get started.

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.

Tip

One of the most complex operations in NGS is to generate good alignment files from raw sequence data. This 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, 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 the 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:

1. Let's inspect the header files:

```
import pysam
bam = pysam.AlignmentFile('NA18489.chrom20.ILLUMINA.bwa.
YRI.exome.20121211.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 `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).

2. 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 can be iterated over its records. We will transverse it until we find a record whose **Concise Idiosyncratic Gapped Alignment Report** (**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 `bam.getrname(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 are 59,996 and 60,048, which means an alignment of 52 bases. Why are 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 snippet), 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, *Working with modern sequence formats*, 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).

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

```
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)
ax.set_xlabel('Read distance')
ax.set_ylabel('PHRED score')
fig.suptitle('Percentage of mapped calls as a function of
the position from the start of the sequencer read')
```

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 **megabase pairs** (**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 screenshot:

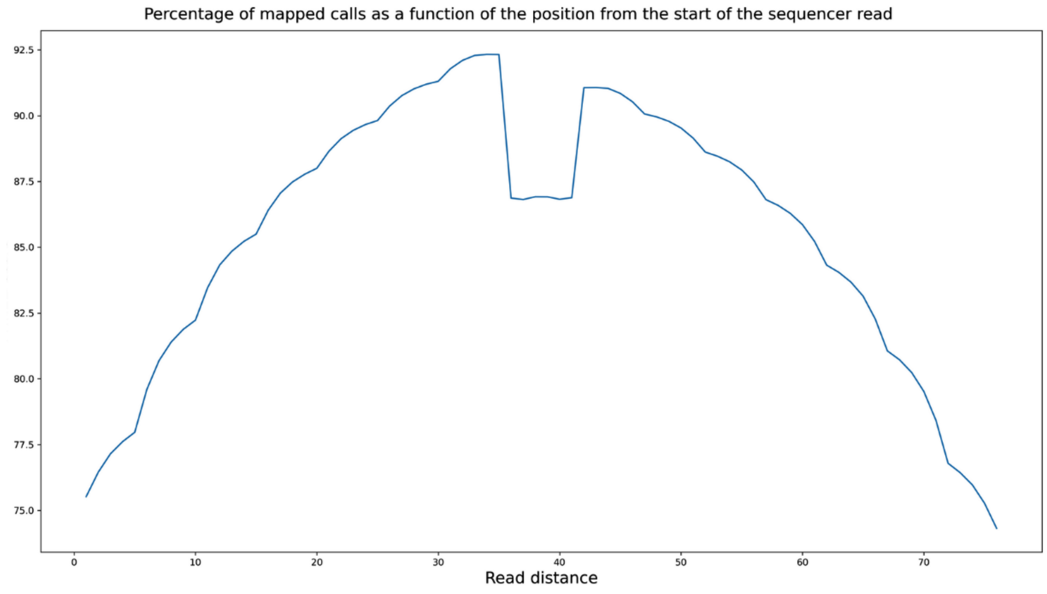


Figure 3.3 – The percentage of mapped calls as a function of the position from the start of the sequencer read

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.

4. 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-')
ax.set_xlabel('Read distance')
ax.set_ylabel('PHRED score')
fig.suptitle('Distribution of PHRED scores as a function
of the position in the read')
```

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 analyses, 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:

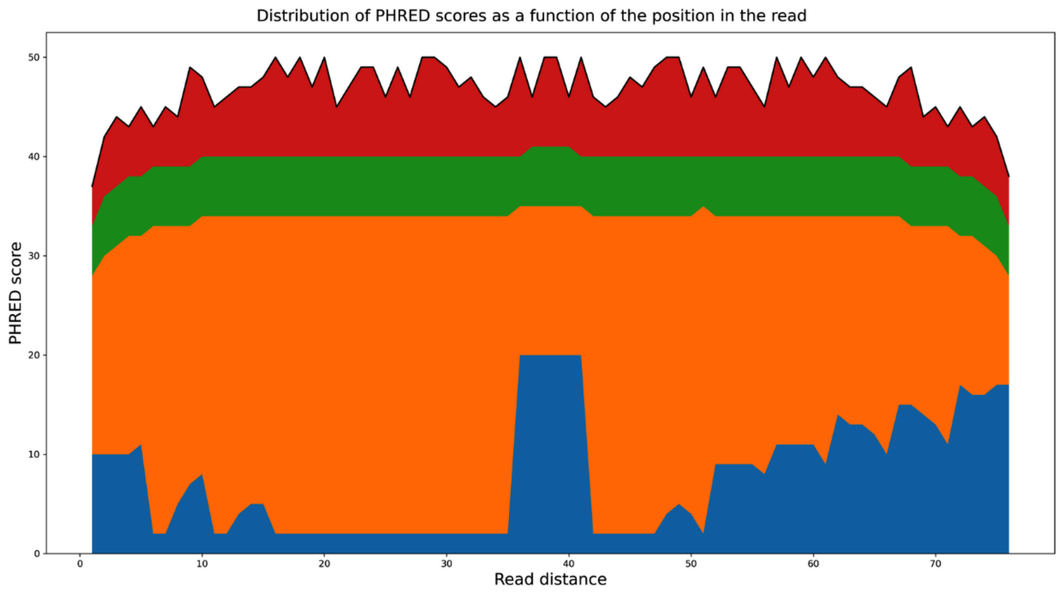


Figure 3.4 – The distribution of Phred scores as a function of the position in the read; the bottom blue color spans from 0 to the 5th percentile; the green color up to the median, red to the 95th percentile, and purple to the maximum

There's more...

Although we will discuss data filtering in the *Studying genome accessibility and filtering SNP data* recipe of this chapter, it's not our objective to explain the SAM format in detail or give a detailed course in data filtering. This task would 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 reducing 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 its own data.

See also

Additional information can be found at the following links:

- The SAM/BAM format is described at `http://samtools.github.io/hts-specs/SAMv1.pdf`.
- You can find an introductory explanation of the SAM format on the Abecasis 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 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 called *SEQanswers* (`http://seqanswers.com/`).
- I would like to repeat the recommendation on Biostars here (which is referred to in the previous recipe, *Working with modern sequence formats*); 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 the **frequently asked questions** (**FAQs**) at `http://www.completegenomics.com/customer-support/faqs/`.

Extracting data from VCF files

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 `cyvcf2` 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 OOM, 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 operation:

```
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
```

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 (the `Chapter03/Working_with_VCF.py` file).

You will need to install `cyvcf2`:

```
conda install –c bioconda cyvcf2
```

Tip

If you have conflict resolution problems, you can try using `pip` instead. This is a last-resort solution that you will find yourself doing with `conda`, as it is incapable of resolving package dependencies, something that happens more often than not. You can execute `pip install cyvcf2`.

How to do it...

Take a look at the following steps:

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

```
from cyvcf2 import VCF
v = VCF('genotypes.vcf.gz')
rec = next(v)
print('Variant Level information')
info = rec.INFO
for info in rec.INFO:
    print(info)
print('Sample Level information')
for fmt in rec.FORMAT:
    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. 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.

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

```
v = VCF('genotypes.vcf.gz')
samples = v.samples
print(len(samples))
variant = next(v)
print(variant.CHROM, variant.POS, variant.ID, variant.
REF, variant.ALT, variant.QUAL, variant.FILTER)
print(variant.INFO)
print(variant.FORMAT)
print(variant.is_snp)
str_alleles = variant.gt_bases[0]
alleles = variant.genotypes[0][0:2]
```

```
is_phased = variant.genotypes[0][2]
print(str_alleles, alleles, is_phased)
print(variant.format('DP')[0])
```

We will start by retrieving the standard information: the chromosome, position, ID, reference base (typically just one) and 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, *Studying genome accessibility and filtering SNP data*).

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.

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

```
from collections import defaultdict
f = VCF('genotypes.vcf.gz')
my_type = defaultdict(int)
num_alts = defaultdict(int)
for variant in f:
    my_type[variant.var_type, variant.var_subtype] += 1
    if variant.var_type == 'snp':
        num_alts[len(variant.ALT)] += 1
print(my_type)
```

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

There's more...

The purpose of this recipe is to get you up to speed with the `cyvcf2` 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.

While `cyvcf2` is quite fast, it can still take a lot of time to process text-based VCF files. 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*, *Bioinformatics Pipelines*. The second strategy is to convert to a more efficient format; we will provide an example of this in *Chapter 6*, *Population Genetics*. Note that VCF developers are working on a **Binary Variant Call Format** (**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

Some useful links are as follows:

- 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 `htslib` are used for variant calling and SAM/BAM management; check out `http://htslib.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.

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 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, as 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 analyses 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 a mosquito vector involved in the transmission of a 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-third-edition/blob/main/Datasets.py` for updates. As usual, the code for downloading this data is available in 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 write, the easier it will be for you to reuse it 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:

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

```
from collections import defaultdict
import functools
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from cyvcf2 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(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, into windows of size 2,000 (100 windows). Every time we find a biallelic SNP, we will add a 1 to the list related to this window in the `window` function.

The `window` function will take a VCF record (a `rec.is_snp` 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 1).

So, now we have a list of 100 elements (each representing 2,000 bp). Each element will be another list that will have a 1 for each biallelic SNP found.

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

2. Let's continue, as follows:

```
def apply_win_funs(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_funs(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 ones; 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 screenshot:

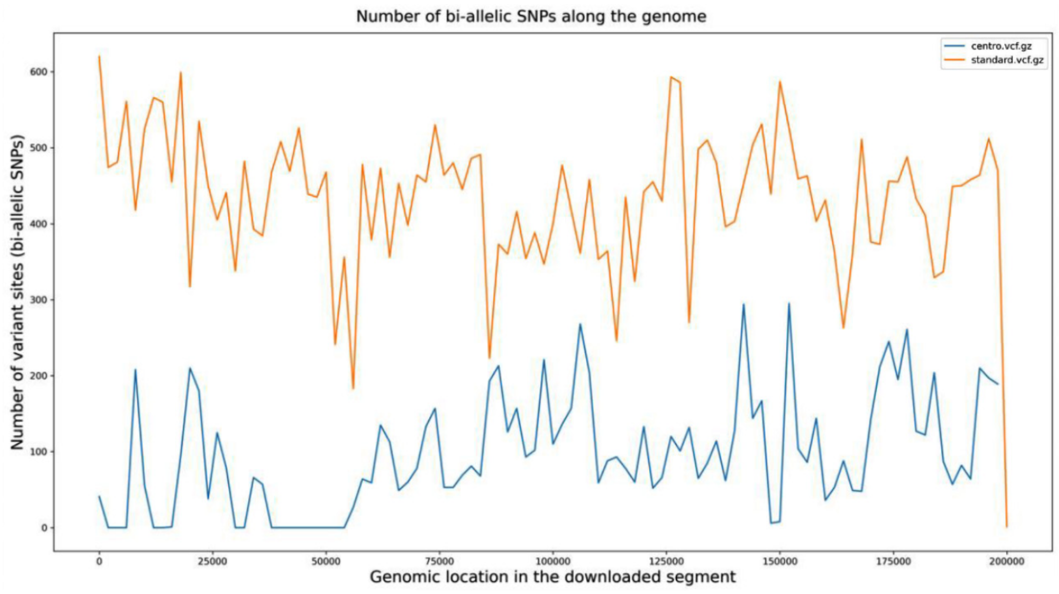


Figure 3.5 – The number of biallelic SNP distributed windows of 2,000 bp in size for an area of 200 kilobase pairs (kbp) near the centromere (orange), and in the middle of the chromosome (blue); both areas come from chromosome 3L for circa 100 Ugandan mosquitoes from the Anopheles 1,000 Genomes Project

Tip

Note that the amount of SNPs in the centromere is smaller than in the middle of the chromosome. This is expected because calling variants in chromosomes is 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!

3. 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` 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 an `MQ0` annotation at the variant level:

```
mq0_wins = {}
size = 5000

def get_sample(rec, annot, my_type):
    return [v for v in rec.format(annot) if v >
np.iinfo(my_type).min]

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=np.int32))
```

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`) that has been cast with a certain type (`my_type=np.int32`). 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.

4. Now, let's print the median and the top 75 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 looks like this:

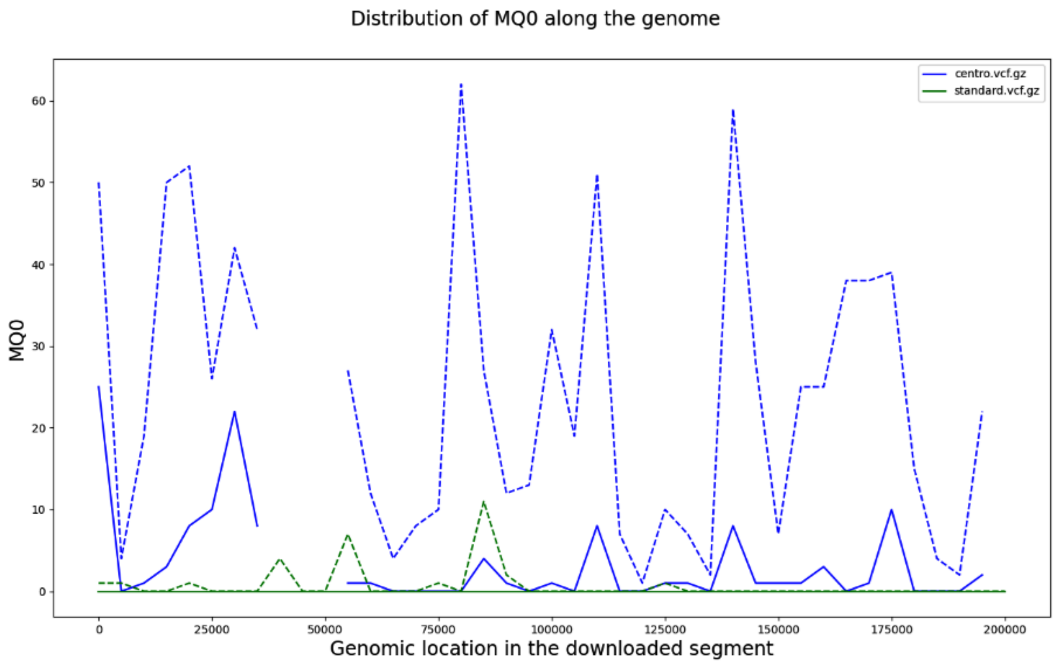


Figure 3.6 – Median (continuous line) and 75th percentile (dashed) of MQ0 of sample SNPs distributed on windows of 5,000 bp in size for an area of 200 kbp near the centromere (blue) and in the middle of chromosome (green); both areas come from chromosome 3L for circa 100 Ugandan mosquitoes from the Anopheles 1,000 Genomes Project

For the `standard.vcf.gz` file, the median `MQ0` is `0` (it's 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).

5. 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 screenshot shows the fraction of calls that are heterozygous at a certain depth:

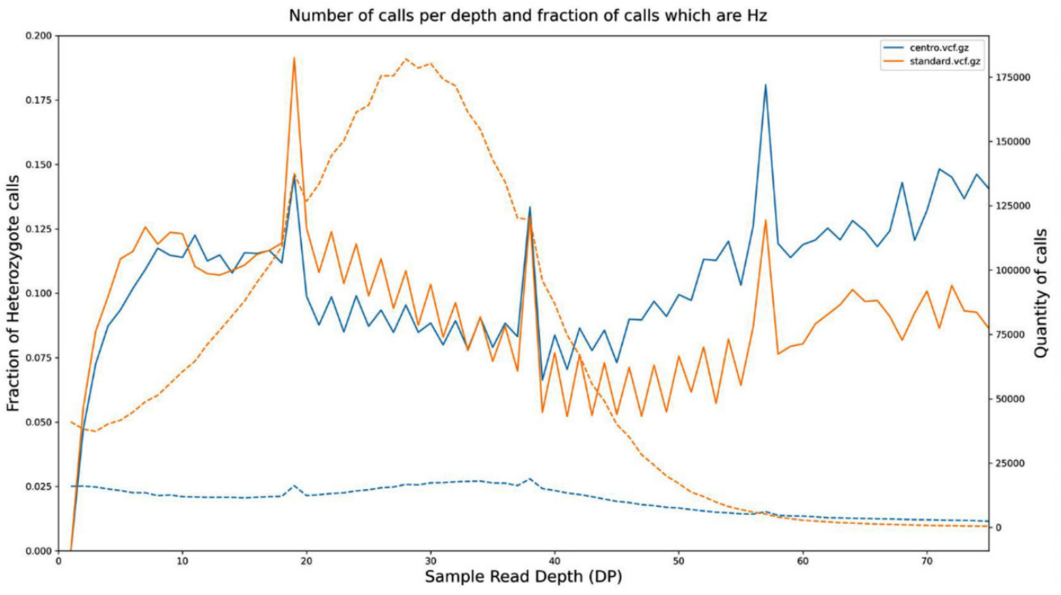


Figure 3.7 – The continuous line represents the fraction of heterozygosite calls computed at a certain depth; in orange is the centromeric area; in blue is the "standard" area; the dashed lines represent the number of sample calls per depth; both areas come from chromosome 3L for circa 100 Ugandan mosquitoes from the Anopheles 1,000 Genomes Project

In the preceding screenshot, there are two considerations to take into account. At a very low depth, the fraction of heterozygote calls is biased—in this case, lower. 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 presented here:

```
def get_sample_relation(recs, f1, f2):
    rel = defaultdict(int)
    for rec in recs:
        if not rec.is_snp:
             continue
        for pos in range(len(rec.genotypes)):
            v1 = f1(rec, pos)
            v2 = f2(rec, pos)
            if v1 is None or v2 == np.iinfo(type(v2)).
min:
                continue  # We ignore Nones
            rel[(v1, v2)] += 1
            # careful with the size, floats: round?
        #break
    return rel get_sample_relation(recs, f1, f2):

rels = {}
for vcf_name in vcf_names:
    recs = VCF(filename=vcf_name)
    rels[vcf_name] = get_sample_relation(
        recs,
        lambda rec, pos: 1 if rec.genotypes[pos][0] !=
rec.genotypes[pos][1] else 0,
        lambda rec, pos: rec.format('DP')[pos][0])
```

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 are 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).

6. Now, let's take a look at the plotting code. 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.

7. 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.

8. 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 effect that we'll work with and convert its effect into an integer (as this 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:
        annot = rec.INFO['EFF']
        master_type = annot.split('(')[0]
        return accepted_eff.index(master_type)
    except ValueError:
        return len(accepted_eff)
```

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

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

10. 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 non-centromeric 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:

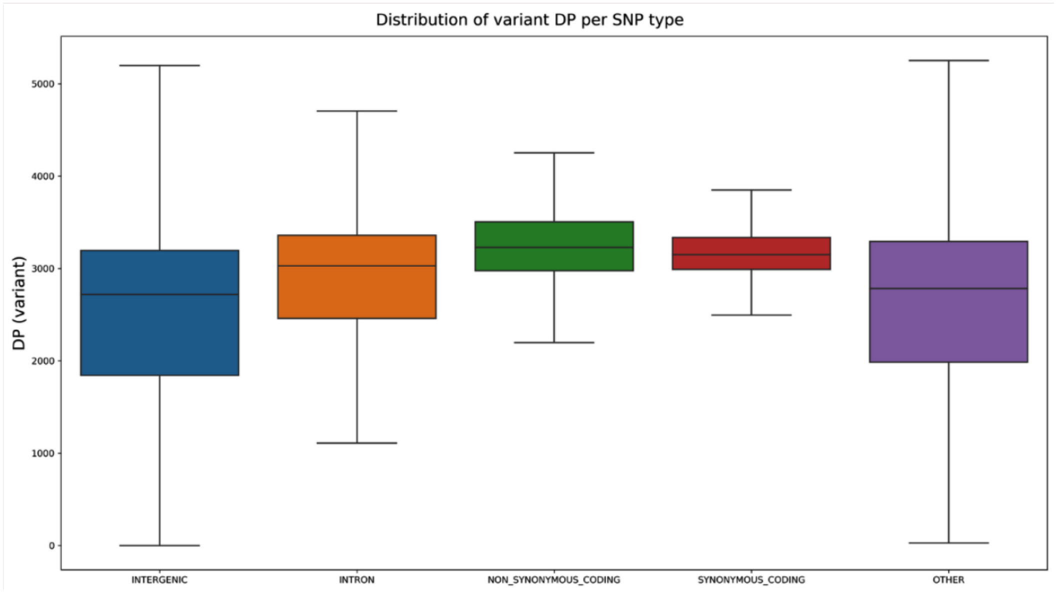


Figure 3.8 – Boxplot for the distribution of variant read depth across different SNP effects

There's more...

The whole issue of filtering SNPs and other genome features will need a book of 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 4*, *Advanced NGS Processing*.

See also

More information can be found via these links:

- There are many filtering rules, but I would like to draw your attention to the need for reasonably good coverage (clearly above 10x). Refer to *Meynert 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 and 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, **General Feature Format** (**GFF**), and **Browser Extensible Data** (**BED**) file formats. It also includes a set of abstractions for processing (mapped) genomic data, encompassing concepts such as 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` 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 called `LCT.bed` is available in the `Chapter03` directory.

The notebook for this code is called `Chapter03/Processing_BED_with_HTSeq.py`.

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

```
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 **Consensus Coding Sequence** (**CCDS**) database (`https://www.ncbi.nlm.nih.gov/CCDS/CcdsBrowse.cgi`).

You will need to install HTSeq:

```
conda install –c bioconda htseq
```

Now, we can begin.

How to do it...

Take a look at the following steps:

1. 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')
```

2. We are now going to extract all the types of features via their 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.

Tip

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

The output for our case looks like this:

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

3. 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 looks like this:

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

4. 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 looks like this:

```
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 regard 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 better than the 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. Even at its worst, the BED format is not very complicated.

HTSeq has much more functionality than this, but 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.

4

Advanced NGS Data Processing

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 the first part of 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 number of errors that our filtering rules might produce. We will also take the opportunity to use the same dataset to find genomic features based on existing annotations.

The last recipe of this chapter will delve into another advanced topic using NGS data: metagenomics. We will QIIME2, a Python package for metagenomics, to analyze data.

If you are using Docker, please use the tiagoantao/bioinformatics_base image. The QIIME2 content has a special setup process that will be discussed in the relevant recipe.

In this chapter, there are the following recipes:

- Preparing a dataset for analysis
- Using Mendelian error information for quality control
- Exploring data with standard statistics
- Finding genomic features from sequencing annotations
- Doing metagenomics with QIIME2

Preparing a 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 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 so, 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 decide whether a certain call is acceptable or not. This was the strategy used by both the Human Genome Project and the Anopheles gambiae 1000 Genomes project.
- 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 gambiae 1000 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 later 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/ag1000g.phase1.ar3.pass.3L.h5
wget -c ftp://ngs.sanger.ac.uk/production/ag1000g/phase1/AR3/
variation/main/hdf5/ag1000g.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 `Chapter04/Preparation.py`. 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:

1. First, start with a few imports:

```
import pickle
import gzip
import random
import numpy as np
import h5py
import pandas as pd
```

2. 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.

3. 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']
```

4. 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 homogenous 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*.

5. 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 acumulate(fun):
    acumulator = {}
    for res in fun():
        if res is not None:
            acumulator[res[0]] = res[1]
    return acumulator
```

6. 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)
```

7. 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 = acumulate(get_mendelian_errors)
```

```
pickle.dump(mendelian_errors, gzip.open('mendelian_
errors.pickle.gz', 'wb'))
```

8. 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 we store a 0 if there is none. An alternative would be to count the number of errors – as we have up to 20 offspring, that would require some sophisticated statistical analysis that we will not be doing here.

9. 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']
```

10. 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)
```

11. We 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)
```

Now, we have prepared the dataset for analysis in this chapter.

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 the offspring can be either AA or AT. They always get an A from the mother, but they can get either an A or a T from the father.
- If 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 `Chapter04/Mendel.py`.

How to do it...

1. We will need a few imports:

```
import random
import matplotlib.pyplot as plt
```

2. 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:

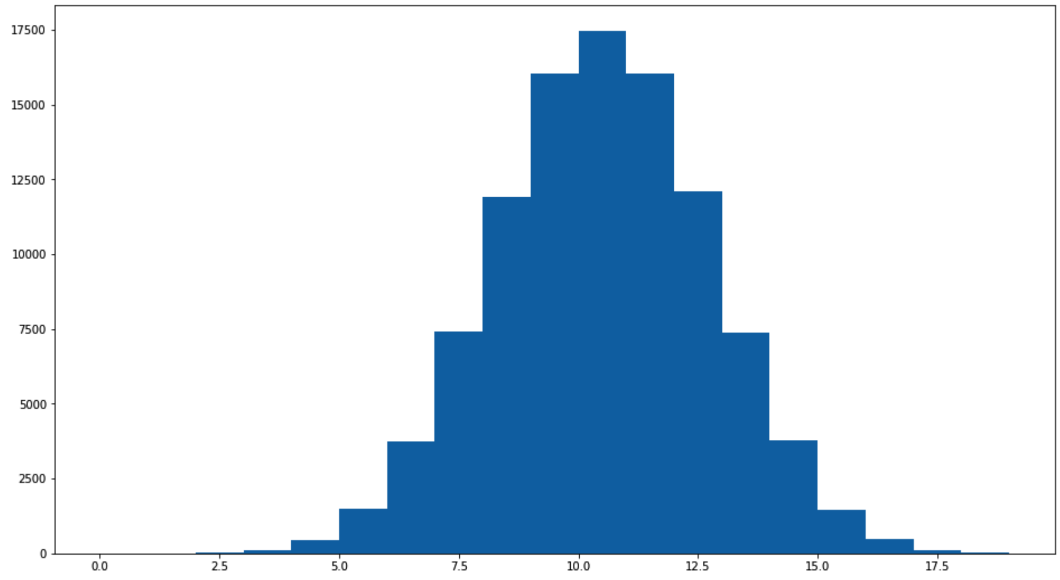


Figure 4.1 - Results from 100,000 simulations: the number of offspring that are heterozygous for certain loci where the mother is AA and the father is heterozygous

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 20, it is very unlikely that all of them are of the same type. This is the information we can use for a less naive interpretation of Mendelian errors.

3. 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:

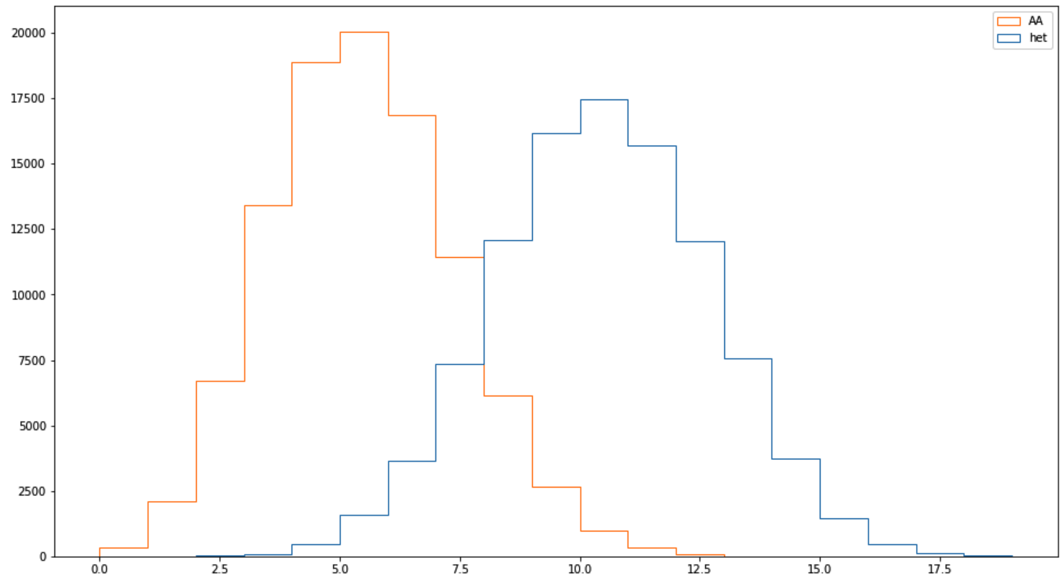


Figure 4.2 - Results from 100,000 simulations: the number of offspring that are AA or heterozygous for a certain locus where both parents are also heterozygous

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).

4. 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)
```

5. Let's see how many errors we have:

```
print(len(mendelian_errors), len(list(filter(lambda x:
x[0] > 0,mendelian_errors.values()))))
```

The output is as follows:

```
(10905732, 541688)
```

Not many of the calls have Mendelian errors – only around 5%, great.

6. 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
```

7. 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])
```

8. 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.

Exploring the data with standard statistics

Now that we have the insights for our Mendelian error analysis, 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 `Chapter04/Exploration.py`.

How to do it...

1. 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
```

2. 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
```

3. Let's ask pandas to show a histogram of all annotations:

```
fig,ax = plt.subplots(figsize=(16,9))
fit_df.hist(column=ordered_features, ax=ax)
```

The following histogram is generated:

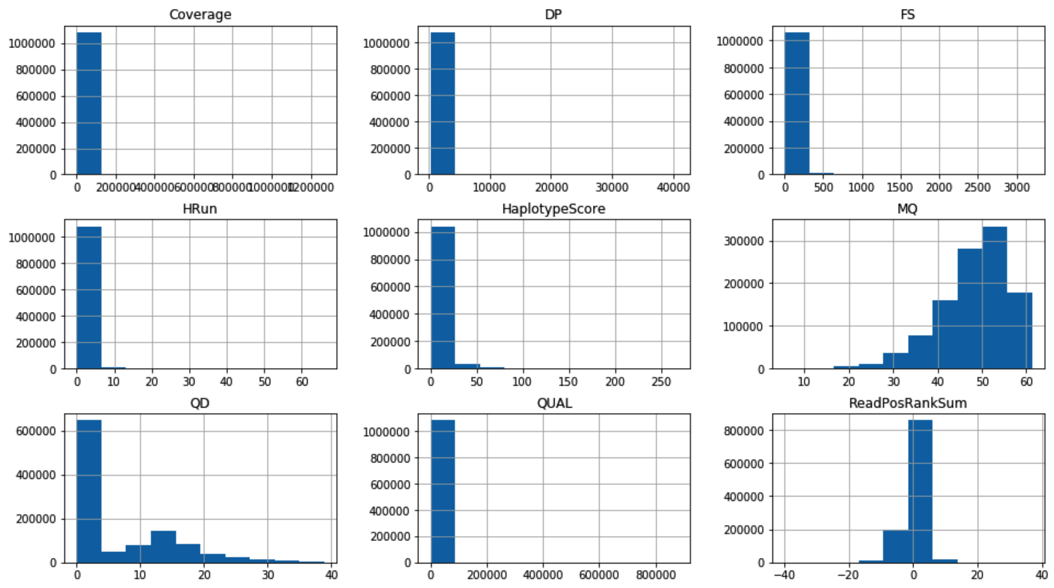


Figure 4.3 - Histogram of all annotations for a dataset with roughly 50% of errors

4. 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)
```

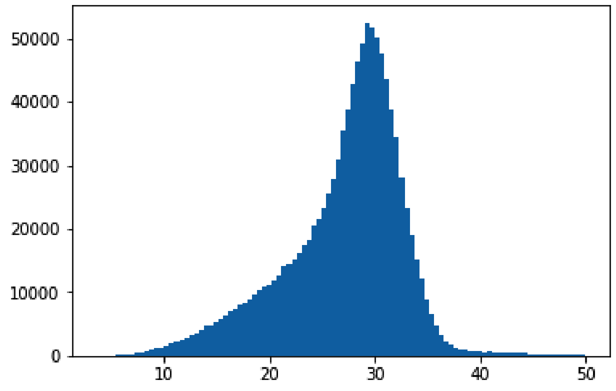


Figure 4.4 - Histogram zooming in on an area of interest for DP

We are actually dividing DP by the number of samples in order to get a more meaningful number.

5. 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]
```

6. Let's have a look at `QUAL` and split it on 0.005, 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)
```

The result is as follows:

```
6507972 6500256 484932 6114096
0.07451353509203788 0.9405931089483245
```

Clearly, `['QUAL']>0.005` gets lots of errors, while not getting lots of OK positions. This is positive, as we have some hope for filtering it.

7. 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:

```
6507972 6500256 460296 5760288
0.07072802402960554 0.8861632526472804
```

8. Let's take an area where there are fewer 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:

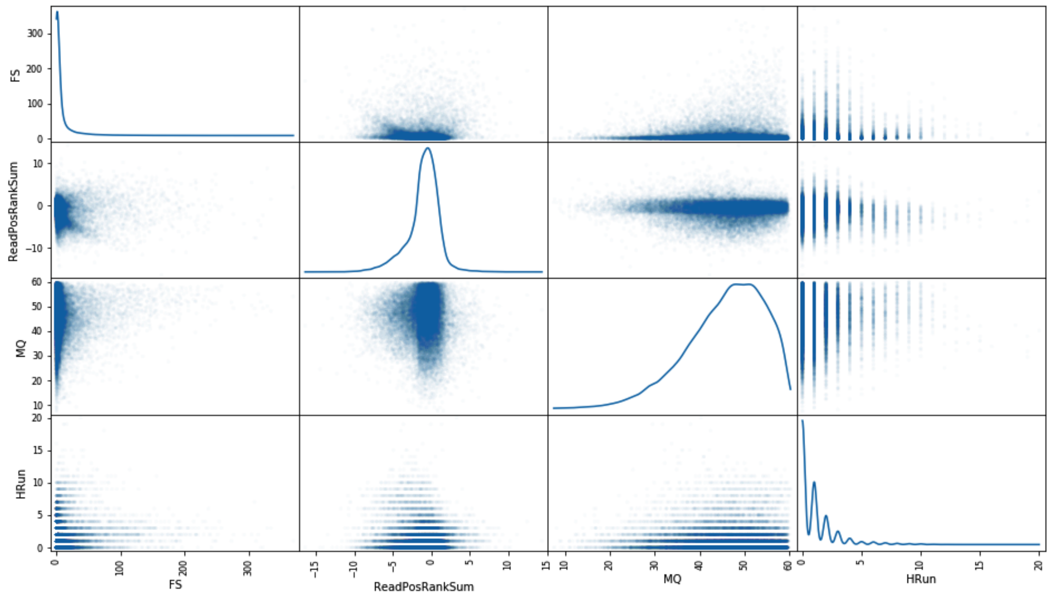


Figure 4.5 - Scatter matrix of annotations of errors for an area of the search space

9. 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:

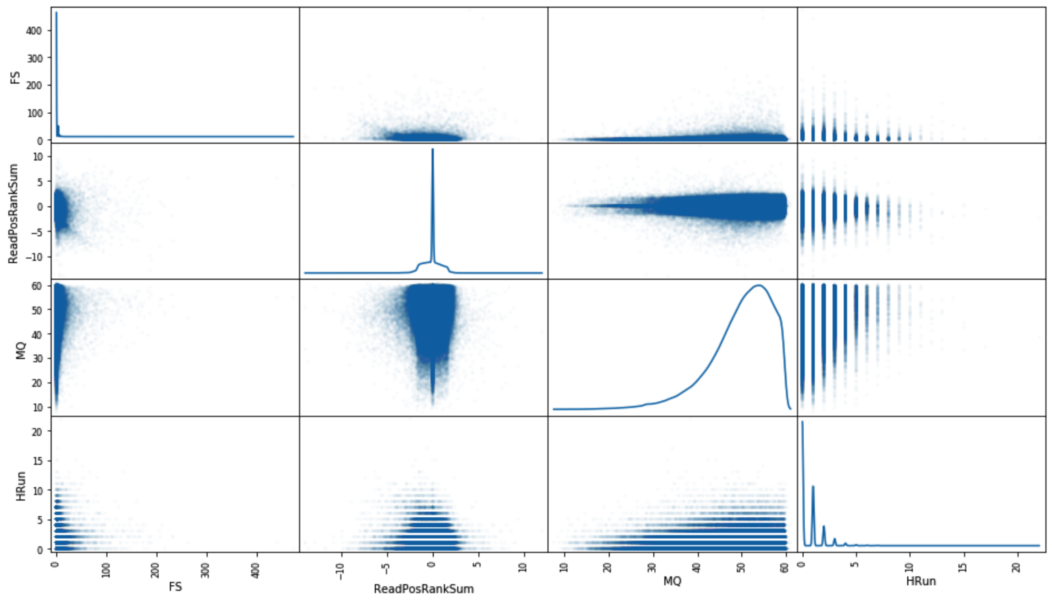


Figure 4.6 - Scatter matrix of annotations of good calls for an area of the search space

10. 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 get the following:

```
10905732 541688 0.04967002673456491
```

Let's remember that there are roughly 10.9 million markers in our full dataset, with around 5% errors.

11. 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))
```

The output is as follows:

```
9625754 32180 0.0033431147315836243
0.8826325458942141
```

So, we reduced the error rate to 0.33% (from 5%), while having only reduced to 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 too 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 conclude 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 in `Chapter04/2L.py`.

How to do it...

We'll get started with the following steps:

1. Let's start with the usual imports:

```
from collections import defaultdict
import gzip

import numpy as np
import matplotlib.pylab as plt
```

2. Let's load the data that we saved in the first recipe:

```
num_parents = 8
dp_2L = np.load(gzip.open('DP_2L.npy.gz', 'rb'))
print(dp_2L.shape)
```

3. And let's print the 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]))
```

The output is as follows:

```
17.0 14.0
23.0 22.0
31.0 29.0
28.0 24.0
32.0 27.0
31.0 31.0
25.0 24.0
24.0 20.0
```

Interestingly, the median for the whole chromosome sometimes does not hold for that big region in the middle, so let's dig further.

4. 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)
```

5. 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_ylim(10, 40)
    ax.plot(*zip(*[(win*window_size, np.mean(lst)) for
win, lst in parent_data.items()]), '.')
```

6. The following plot shows the output:

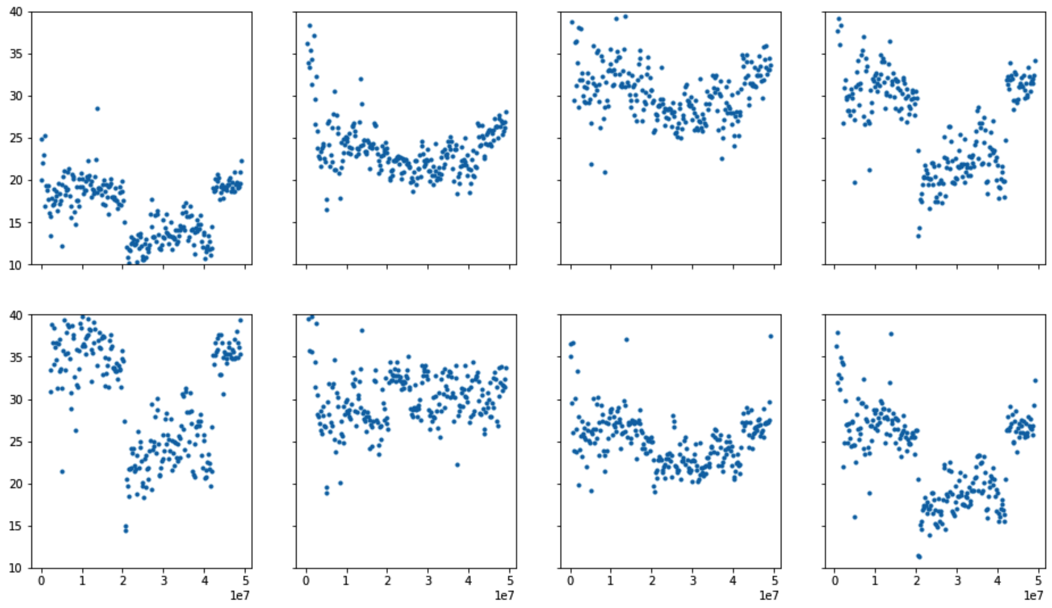


Figure 4.7 - Median DP per window for all parents of the dataset on chromosome arm 2L

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, such as in the third column, there is a bit of drop – not so pronounced. And for 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 for sequencing: Anopheles mosquitoes might have a big chromosomal inversion in the middle of arm 2L. Karyotypes that are not the same as those 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 in 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 is 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).

Doing metagenomics with 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 in 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/tkosciol/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/2022.2/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 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 wget https://data.qiime2.org/distro/core/qiime2-2022.2-
py38-osx-conda.yml
conda env create -n qiime2-2022.2 --file qiime2-2022.2-py38-
osx-conda.yml
```

On Linux, use the following code to create the environment:

```
wget wget https://data.qiime2.org/distro/core/qiime2-2022.2-
py38-linux-conda.yml
conda env create -n qiime2-2022.2 --file qiime2-2022.2-py38-
linux-conda.yml
```

If these instructions do not work, check the QIIME 2 website for an updated version (`https://docs.qiime2.org/2022.2/install/native`). QIIME 2 is updated regularly.

At this stage, you need to enter the QIIME 2 conda environment by using `source activate qiime2-2022.2`. If you want to get to the standard conda environment, use `source deactivate` instead. We will want to install `jupyter lab` and `jupytext`:

```
conda install jupyterlab jupytext
```

You might want to install other packages you want 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
```

TIP

The extension is highly interactive and allows you to look at data from different viewpoints 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). 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 `Chapter4/QIIME2_Metagenomics.py` file.

WARNING

Due to the fluidity of package installation with QIIME, we don't provide a Docker environment for it. This means that if you are working from our Docker installation 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:

1. 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.

2. 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))])
```

3. 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
```

Here is a part of the output from the QIIME extension for Juypter:

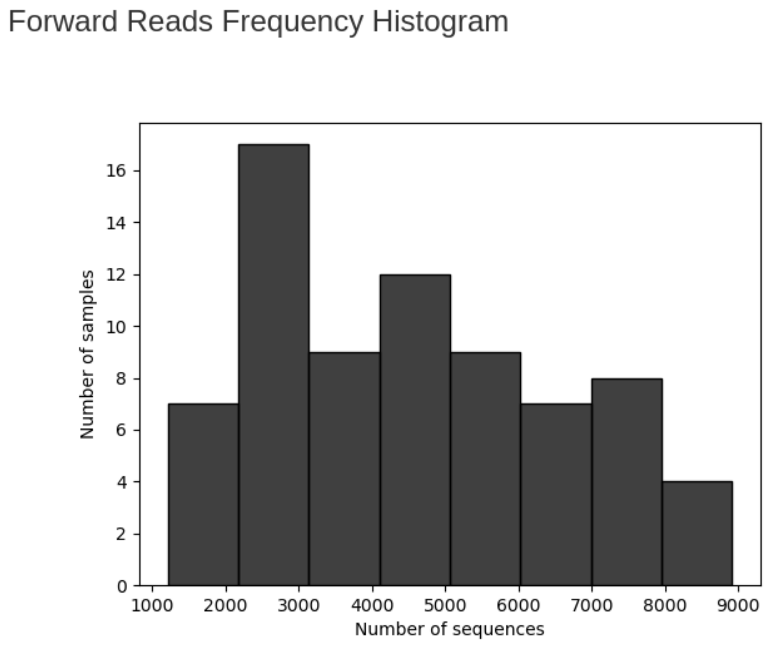


Figure 4.8 - A part of the output of the QIIME2 extension for Jupyter

Remember that the extension is iterative and provides substantially more information than only this chart.

TIP

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.

4. 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
```

5. Let's use the DADA2 (`https://github.com/benjjneb/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)
```

6. 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.

7. Now, let's do the same for the second set:

```
stats_meta2 = metadata_tabulate(input=qual_control2.
denoising_stats.view(Metadata))
stats_meta2.visualization
```

8. 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])
```

9. Then, gather some quality statistics from the merge:

```
ft_sum = ft_summarize(table=table_merge.merged_table)
ft_sum.visualization
```

10. 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). 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.

5
Working with Genomes

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 varying quality, which can range from high quality (by high quality, we only refer to the state of the genome's assembly, which is the focus of this chapter), as with the human genome, to problematic with non-model species. We will also learn how to deal with genome annotations (working with 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 orthologs across species. Finally, we will access a **Gene Ontology** (**GO**) database.

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 with the Ensembl REST API
- Retrieving gene ontology information from Ensembl

Technical requirements

If you are running this chapter's content via Docker, you can use the `tiagoantao/bioinformatics_genomes` image. If you are using Anaconda, the required software for this chapter will be introduced in each relevant section.

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 such as *Plasmodium falciparum*, which has 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 the genome. Along the way, you have different ploidy and sex chromosome organizations.

Tip

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 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 *Plasmodium 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 that's the size of *Plasmodium 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 in this book's Jupyter notebook as `Chapter05/Reference_Genome.py`, in the code bundle for this book. We will need to download the reference genome – you can find the up-to-date location in the aforementioned notebook. To generate the chart at the end of this recipe, we will need `reportlab`:

```
conda install -c bioconda reportlab
```

Now, we're ready to begin.

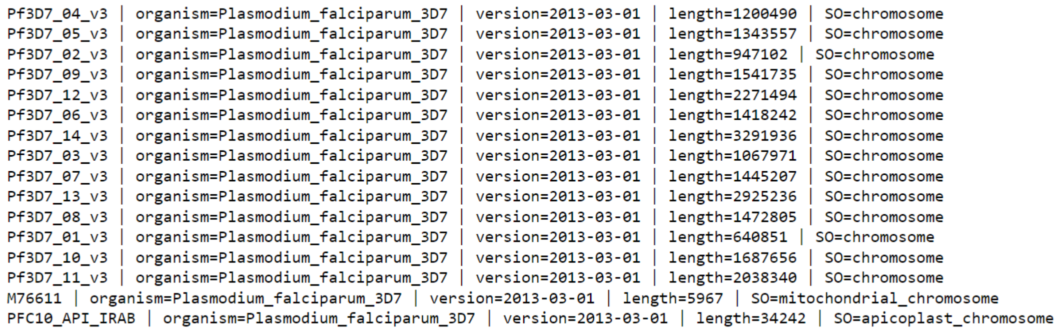
How to do it...

Follow these steps:

1. We will start by inspecting the description of all of the sequences in the reference genome's 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 3*, *Next-Generation Sequencing*. Let's take a look at part of the output:



```
Pf3D7_04_v3 | organism=Plasmodium_falciparum_3D7 | version=2013-03-01 | length=1200490 | SO=chromosome
Pf3D7_05_v3 | organism=Plasmodium_falciparum_3D7 | version=2013-03-01 | length=1343557 | SO=chromosome
Pf3D7_02_v3 | organism=Plasmodium_falciparum_3D7 | version=2013-03-01 | length=947102 | SO=chromosome
Pf3D7_09_v3 | organism=Plasmodium_falciparum_3D7 | version=2013-03-01 | length=1541735 | SO=chromosome
Pf3D7_12_v3 | organism=Plasmodium_falciparum_3D7 | version=2013-03-01 | length=2271494 | SO=chromosome
Pf3D7_06_v3 | organism=Plasmodium_falciparum_3D7 | version=2013-03-01 | length=1418242 | SO=chromosome
Pf3D7_14_v3 | organism=Plasmodium_falciparum_3D7 | version=2013-03-01 | length=3291936 | SO=chromosome
Pf3D7_03_v3 | organism=Plasmodium_falciparum_3D7 | version=2013-03-01 | length=1067971 | SO=chromosome
Pf3D7_07_v3 | organism=Plasmodium_falciparum_3D7 | version=2013-03-01 | length=1445207 | SO=chromosome
Pf3D7_13_v3 | organism=Plasmodium_falciparum_3D7 | version=2013-03-01 | length=2925236 | SO=chromosome
Pf3D7_08_v3 | organism=Plasmodium_falciparum_3D7 | version=2013-03-01 | length=1472805 | SO=chromosome
Pf3D7_01_v3 | organism=Plasmodium_falciparum_3D7 | version=2013-03-01 | length=640851 | SO=chromosome
Pf3D7_10_v3 | organism=Plasmodium_falciparum_3D7 | version=2013-03-01 | length=1687656 | SO=chromosome
Pf3D7_11_v3 | organism=Plasmodium_falciparum_3D7 | version=2013-03-01 | length=2038340 | SO=chromosome
M76611 | organism=Plasmodium_falciparum_3D7 | version=2013-03-01 | length=5967 | SO=mitochondrial_chromosome
PFC10_API_IRAB | organism=Plasmodium_falciparum_3D7 | version=2013-03-01 | length=34242 | SO=apicoplast_chromosome
```

Figure 5.1 – The output showing the FASTA descriptions for the reference genome of Plasmodium falciparum

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

2. 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)
```

Here, we have performed a windowed analysis of all chromosomes, similar to what we did in *Chapter 3*, *Next-Generation Sequencing*. We started by defining a window size of 50 kbp. This is appropriate for *Plasmodium 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 1*) 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, our code is designed to be reused with larger genomes.

3. 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 contain 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's size is 14 blocks of 50 kbp.

Be aware of two unusual features of the *Plasmodium 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).

4. Now, let's create 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 into 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 draw the chromosomes in order (hence the sort). We need the size of the biggest chromosome (14, in *Plasmodium falciparum*) to make sure that the size of chromosomes is printed with the correct scale (the `biggest_chrom` variable).

Then, we create an A4-sized representation of an organism with a PNG output. Note that we draw very small telomeres of 10 bp. This will produce a rectangular-like chromosome. You can make the telomeres bigger, giving them 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% or above 22.0% will be considered an outlier. Remember that for most other species, this will be much higher.

Then, we 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:

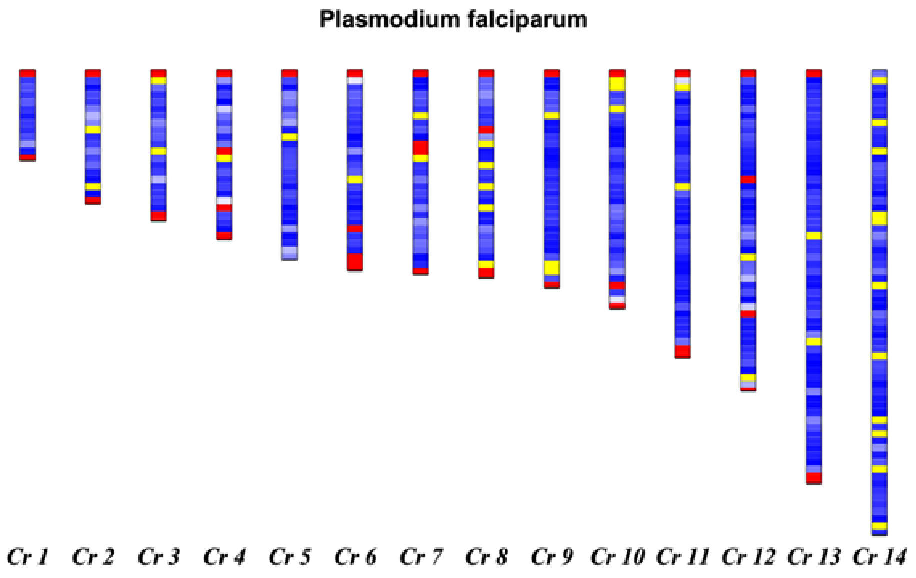


Figure 5.2 – The 14 chromosomes of Plasmodium falciparum, color-coded with the GC content (red is more than 22%, yellow less than 17%, and the blue shades represent a linear gradient between both numbers)

Tip

Biopython code evolved 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, then the standard bearer is Matplotlib, as we learned in *Chapter 2, Getting to Know NumPy, pandas, Arrow, and Matplotlib*. Alternatives include Bokeh, HoloViews, or Python's version of ggplot (or even more sophisticated visualization alternatives, such as Mayavi, **Visualization Toolkit** (**VTK**), and even the Blender API).

5. Finally, you can print the image inline in the notebook:

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

And that completes this recipe!

There's more...

Plasmodium 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.

Plasmodium 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

Here are some sources you can learn more from:

- 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://plasmodb.org/plasmo/`), which you downloaded the *Plasmodium falciparum* genome from, you will find VectorBase (`https://www.vectorbase.org/`) in the next recipe for disease vectors. FlyBase (`http://flybase.org/`) for *Drosophila melanogaster* 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 *Plasmodium 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 learn 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 *Anopheles 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 *N*s instead of ACTGs). The *Anopheles 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 *Plasmodium falciparum* genome (but still 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 in this book's Jupyter notebook at `Chapter05/Low_Quality.py`, in the code bundle for this book. At the start of the notebook, you can find the most up-to-date location of both genomes, along with the code to download them.

How to do it...

Follow these steps:

1. Let's start by listing the chromosomes of the *Anopheles 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 an output that will include the organism chromosomes (along with a few unmapped supercontigs not depicted):

```
AgamP4_2L | organism=Anopheles_gambiae_PEST |
version=AgamP4 | length=49364325 | SO=chromosome
AgamP4_2R | organism=Anopheles_gambiae_PEST |
version=AgamP4 | length=61545105 | SO=chromosome
AgamP4_3L | organism=Anopheles_gambiae_PEST |
version=AgamP4 | length=41963435 | SO=chromosome
AgamP4_3R | organism=Anopheles_gambiae_PEST |
version=AgamP4 | length=53200684 | SO=chromosome
AgamP4_X | organism=Anopheles_gambiae_PEST |
version=AgamP4 | length=24393108 | SO=chromosome
AgamP4_Y_unplaced | organism=Anopheles_gambiae_PEST |
version=AgamP4 | length=237045 | SO=chromosome
AgamP4_Mt | organism=Anopheles_gambiae_PEST |
version=AgamP4 | length=15363 | SO=mitochondrial_
chromosome
```

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 mitochondria (`Mt`), the `X` chromosome, and the `Y` chromosome, which is quite small and has a name that all but indicates that it may not be in the best state. Also, the unknown (`UNKN`) chromosome is a large proportion of the reference genome, to the tune of a chromosome arm.

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

2. Now, let's check the uncalled positions (`Ns`) and their distribution for the *Anopheles gambiae* genome:

```
recs = SeqIO.parse(gzip.open(gambiae_name, 'rt',
encoding='utf-8'), 'fasta')
chrom_Ns = {}
chrom_sizes = {}
for rec in recs:
    if rec.description.find('supercontig') > -1:
        continue
    print(rec.description, rec.id, rec)
    chrom = rec.id.split('_')[1]
    if chrom in ['UNKN']:
        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]
    if len(Ns) > 0:
        max_Ns = max(Ns)
    else:
        max_Ns = 'NA'
    print(f'{chrom} ({size}): %Ns ({round(100 * sum(Ns) / 
size, 1)}), num Ns: {len(Ns)}, max N: {max_Ns}')
```

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

We have two dictionaries: one dictionary that contains chromosome sizes and another that contains the distribution of the sizes of runs of `Ns`. To calculate the runs of `Ns`, we must traverse all autosomes (noting when an `N` position starts and ends). Then, we must 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
Y (237045): %Ns (43.0), num Ns: 63, max N: 7957
Mt (15363): %Ns (0.0), num Ns: 0, max N: NA
```

So, for the `2L` chromosome arm (with a size of 49 Mbp), 1.7% 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`.

3. Now, let's turn our attention to the *Anopheles 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 what we looked at previously, but we print slightly more detailed statistics using NumPy, so we get the following:

```
1320 7811.5 170678.2 58369459 1004 1537.1 39644.7
```

Thus, we have `1371` scaffolds (against seven entries on the *Anopheles gambiae* genome) with a median size of `7811.5` (a mean of `17,0678.2`). The biggest scaffold is 5.8 Mbp, while the smallest scaffold is 1,004 bp. The tenth percentile for size is `1537.1`, while the ninetieth is `39644.7`.

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

```
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 a large variance (sizes between 0% and above 90%), and a tighter variance (between 0% and 25%) for the largest scaffolds:

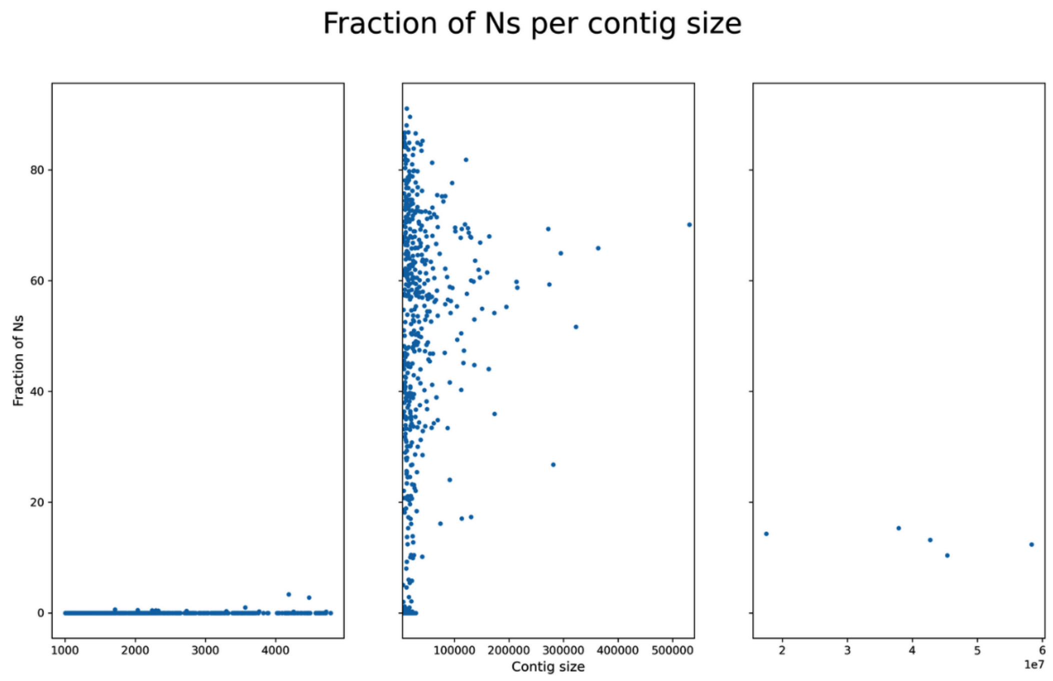


Figure 5.3 – The fraction of scaffolds that are N as a function of their size

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 can 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 on the performance of your aligner. In any case, the general idea is that you should be careful because the scaffold size and number will rear their 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

Here are some resources you can learn more from:

- Tools such as RepeatMasker can be 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 learn how to parse and analyze GFF files while using the annotation of the *Anopheles gambiae* genome as an example.

Getting ready

Use the `Chapter05/Annotations.py` notebook file, which is provided in the code bundle for this book. The up-to-date location of the GFF file that we will be using can be found at the top of the notebook.

You will need to install `gffutils`:

```
conda install -c bioconda gffutils
```

Now, we're ready to start.

How to do it...

Follow these steps:

1. 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.

2. Now, let's 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 into a list (it's safe to do so here).

3. Let's list all seqids:

```
seqids = set()
for e in db.all_features():
    seqids.add(e.seqid)
for seqid in seqids:
    print(seqid)
```

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

4. 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 seqid in seqids:
    cnt = 0
    for gene in db.region(seqid=seqid,
featuretype='protein_coding_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(f'seqid {seqid}, number of genes {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 seqids while extracting all protein-coding genes (using `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 that spans the largest region.

We follow a similar procedure to find the gene and the largest number of exons. Finally, we print a dictionary that contains the distribution of the number of alternative transcripts per gene (`num_mRNAs`) and the distribution of the 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 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 for this library is concerned with helping you process all kinds of awkward variations (refer to `https://pythonhosted.org/gffutils/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

Here are some resources you can learn more from:

- The GFF spec can be found at `https://www.sanger.ac.uk/resources/software/gff/spec.html`.
- Probably the best explanation of the GFF format, along with the most common versions and GTF, can be found at `http://gmod.org/wiki/GFF3`.

Extracting genes from a reference using annotations

In this recipe, we will learn 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). First, we will 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 `Chapter05/Getting_Gene.py` notebook file.

How to do it...

Follow these steps:

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

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

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

```
AgamP4_2L       VEuPathDB       protein_coding_
gene     2358158 2431617 .       +       .       ID=AGAP0
04707;Name=para;description=voltage-gated sodium channel
AgamP4_2L +
```

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

2. 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.id == gene.seqid:
        my_seq = rec.seq
        break
```

The output will be as follows:

```
AgamP4_2L | organism=Anopheles_gambiae_PEST |
version=AgamP4 | length=49364325 | SO=chromosome
```

3. 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('')
    for CDS in CDSs:
        my_cds = Seq.Seq(str(my_seq[CDS.start - 1:CDS.
end]))
        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.

We 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.

4. Although we have the `gene_id` at hand, we only want 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
```

5. Now, let's 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)
```

6. 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 avoided 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 chapter and *Chapter 3*, *Next Generation Sequencing*. While it's conceptually trivial, it's unfortunately full of booby traps.

> **Tip**
> 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 contain 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

Here are some resources you can learn more from:

- 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 of CDSs and exons can be found at `https://www.biostars.org/p/65162/`.

Finding orthologues with the Ensembl REST API

In this recipe, we will learn 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 to 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 since we are using web APIs, internet access will be needed. The amount of data that can be 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 `Chapter05/Orthology.py` notebook file.

How to do it...

Follow these steps:

1. 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.

2. Then, we will 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 a URL starting with the `http://rest.ensembl.org/info/species` prefix for the REST request. The preceding link will not work on your browser, by the way; it should only be used via a REST API.

3. Now, let's 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 maintaining our LCT identifier.

4. 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']
```

Tip

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.

5. Just for your information, we can now get the sequence of the area containing the gene. Note that this is probably the whole interval, so 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)
```

6. 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 8*, *Using the Protein Data Bank*), and WikiGene.

7. 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 acquired the orthologues directly for the `horse` genome by specifying a `target_species` parameter on `do_request`. However, this code allows you to inspect all the 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 3*, *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.

8. 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. Accessing NCBI Entrez databases was covered in *Chapter 3*, *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, you will learn how to use gene ontology information again by querying the Ensembl REST API. Gene ontologies are controlled vocabularies for annotating 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: the 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 since we are using web APIs, internet access will be needed. The amount of data that will be transferred will be limited.

As usual, you can find this content in the `Chapter05/Gene_Ontology.py` notebook file. We will make use of the `do_request` function, which was defined in *Step 1* of the previous recipe (*Finding orthologues with the Ensembl REST API*). To draw GO trees, we will use `pygraphviz`, a graph-drawing library:

```
conda install pygraphviz
```

OK – we're all set.

How to do it...

Follow these steps:

1. Let's start by retrieving all GO terms associated with the LCT gene (you learned 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 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]
```

2. 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.

3. 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('\t is 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).

4. Let's define a function that will 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
```

5. Finally, we will print a tree of relationships for the `lactase activity` term. For this, we will use the `pygraphivz` 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:

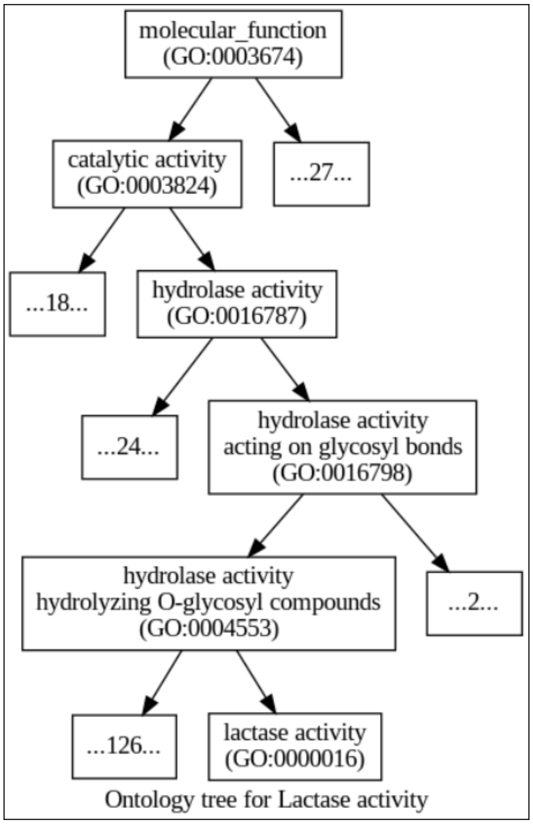

molecular_function (GO:0003674)
catalytic activity (GO:0003824)
...27...
...18...
hydrolase activity (GO:0016787)
...24...
hydrolase activity acting on glycosyl bonds (GO:0016798)
hydrolase activity hydrolyzing O-glycosyl compounds (GO:0004553)
...2...
...126...
lactase activity (GO:0000016)
Ontology tree for Lactase activity

Figure 5.4 – An ontology tree for the "lactase activity" term (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://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://amigo.geneontology.org/amigo/term/GO:0005761#display-lineage-tab` and click on **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 later in this book.

See also

Here are some resources you can learn more from:

- As mentioned previously, 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, such as AmiGO (`http://amigo.geneontology.org`) and QuickGO (`http://www.ebi.ac.uk/QuickGO/`).
- One of the most common analyses performed with GO is 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/`).

6
Population Genetics

Population genetics is the study of the changes in the 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 elsewhere. Also, we will not cover genomic structural variations 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 using the **Fixation Index** (**FST**) with computing F-statistics, **Principal Components Analysis** (**PCA**), and studying 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 as we will see in *Chapter 11*.

There is no such thing as a default file format for population genetics data. The bleak reality of this field is that there is a plenitude of formats, most of them developed with a specific application in mind; therefore, none are generically applicable. Some of the efforts to create a more general format (or even just a file converter to support many formats) had 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 PLINK (`https://www.cog-genomics.org/plink/2.0/`), which was originally developed to perform **Genome-Wide Association Studies** (**GWAS**) with human data but has many more applications. If you have **Next-Generation Sequencing** (**NGS**) data, you may question, why not use the **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. If you use Docker, use the image tiagoantao/bioinformatics_popgen.

In this chapter, we will cover the following recipes:

- Managing datasets with PLINK
- Using sgkit for population genetics analysis with xarray
- Exploring a dataset with sgkit
- Analyzing population structure
- Performing a PCA
- Investigating population structure with admixture

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.

Warning

Note that neither PLINK nor any similar programs were developed for their file formats. There was probably no objective to create 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 a 'file conversion mentality'. Beyond 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 3*, *Next-Generation Sequencing*, and that the HapMap project is in many ways the precursor to 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 3*, *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 3*, *Next-Generation Sequencing*, and, of course, the HapMap site (`https://www.genome.gov/10001688/international-hapmap-project`) 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

Table 6.1 - The populations in the Genome Project

Note

We will be using data from the HapMap project that 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. Make sure that you have approximately 20 GB of disk space for this chapter. The files can be found at `https://ftp.ncbi.nlm.nih.gov/hapmap/genotypes/hapmap3_r3/plink_format/`.

Decompress the PLINK file using the following commands:

```
bunzip2 hapmap3_r3_b36_fwd.consensus.qc.poly.map.gz
bunzip2 hapmap3_r3_b36_fwd.consensus.qc.poly.ped.gz
```

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 `Chapter06/Data_Formats.py` Notebook file, where everything has been taken care of.

Finally, most of this recipe will make heavy usage of PLINK (`https://www.cog-genomics.org/plink/2.0/`). Python will mostly be used as the glue language to call PLINK.

How to do it...

Take a look at the following steps:

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

```
from collections import defaultdict
f = open('relationships_w_pops_041510.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 the 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 (information that can be found in 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 in a separate, ad hoc file is that the PLINK format makes no provision for the population structure (this format makes provision only for the case and 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 data formats in population genetics: 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.

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

```
import os
os.system('plink2 --pedmap hapmap3_r3_b36_fwd.consensus.
qc.poly --out hapmap10 --thin 0.1 --geno 0.1 --export
ped')
os.system('plink2 --pedmap hapmap3_r3_b36_fwd.consensus.
qc.poly --out hapmap1 --thin 0.01 --geno 0.1 --export
ped')
```

With Jupyter Notebook, you can just do this instead:

```
!plink2 --pedmap hapmap3_r3_b36_fwd.consensus.qc.poly
--out hapmap10 --thin 0.1 --geno 0.1 --export ped
!plink2 --pedmap hapmap3_r3_b36_fwd.consensus.qc.poly
--out hapmap1 --thin 0.01 --geno 0.1 --export ped
```

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 lower than 90 percent (with the `--geno 0.1` parameter).

Note

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 an 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.

3. 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')
        try:
            chrom = int(toks[0])
        except ValueError:
            rs = toks[1]
            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('plink2 –-pedmap hapmap1 --out hapmap1_auto
--exclude exclude1.txt --export ped')
os.system('plink2 --pedmap hapmap10 --out hapmap10_auto
--exclude exclude10.txt --export ped')
```

4. Let's create a function that generates a list with all the SNPs not belonging to autosomes. With human data, that means all non-numeric chromosomes. If you use another species, be careful with your chromosome coding because PLINK is geared toward human data. If your species are diploid, 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` for some alternatives (such as the `--allow-extra-chr` flag).
5. We then create autosome-only PLINK files for subsample datasets of 10 and 1 percent (prefixed as `hapmap10_auto` and `hapmap1_auto`).
6. 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('plink2 --pedmap hapmap10_auto --filter-
founders --out hapmap10_auto_noofs --export ped')
```

Note

This step is representative of the fact that most population genetic analyses 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.

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

```
os.system('plink2 --pedmap hapmap10_auto_noofs --indep-
pairwise 50 10 0.1 --out keep --export ped')
os.system('plink2 --pedmap hapmap10_auto_noofs --extract
keep.prune.in --recode --out hapmap10_auto_noofs_ld
--export ped')
```

   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.

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

```
os.system('plink2 --file hapmap10_auto_noofs_ld
--recode12 tab --out hapmap10_auto_noofs_ld_12 --export
ped 12')
os.system('plink2 --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 a PCA* 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, a smaller file size). We will use this in the admixture recipe.

9. 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('plink2 --pedmap hapmap10_auto_noofs --chr 2
--out hapmap10_auto_noofs_2 --export ped')
```

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 between 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 to be 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 the 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 the constraints required by the software.

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 5*, *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 sets 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` 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 lacking in much of genetics software.

Using sgkit for population genetics analysis with xarray

Sgkit is the most advanced Python library for doing population genetics analysis. It's a modern implementation, leveraging almost all of the fundamental data science libraries in Python. When I say almost all, I am not exaggerating; it uses NumPy, pandas, xarray, Zarr, and Dask. NumPy and pandas were introduced in *Chapter 2*. Here, we will introduce xarray as the main data container for sgkit. Because I feel that I cannot ask you to get to know data engineering libraries to an extreme level, I will gloss over the Dask part (mostly by treating Dask structures as equivalent NumPy structures). You can find more advanced details about out-of-memory Dask data structures in *Chapter 11*.

Getting ready

You will need to run the previous recipe because its output is required for this one: we will be using one of the PLINK datasets. You will need to install sgkit.

As usual, this is available in the `Chapter06/Sgkit.py` 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:

1. Let's load the `hapmap10_auto_noofs_ld` dataset generated in the previous recipe:

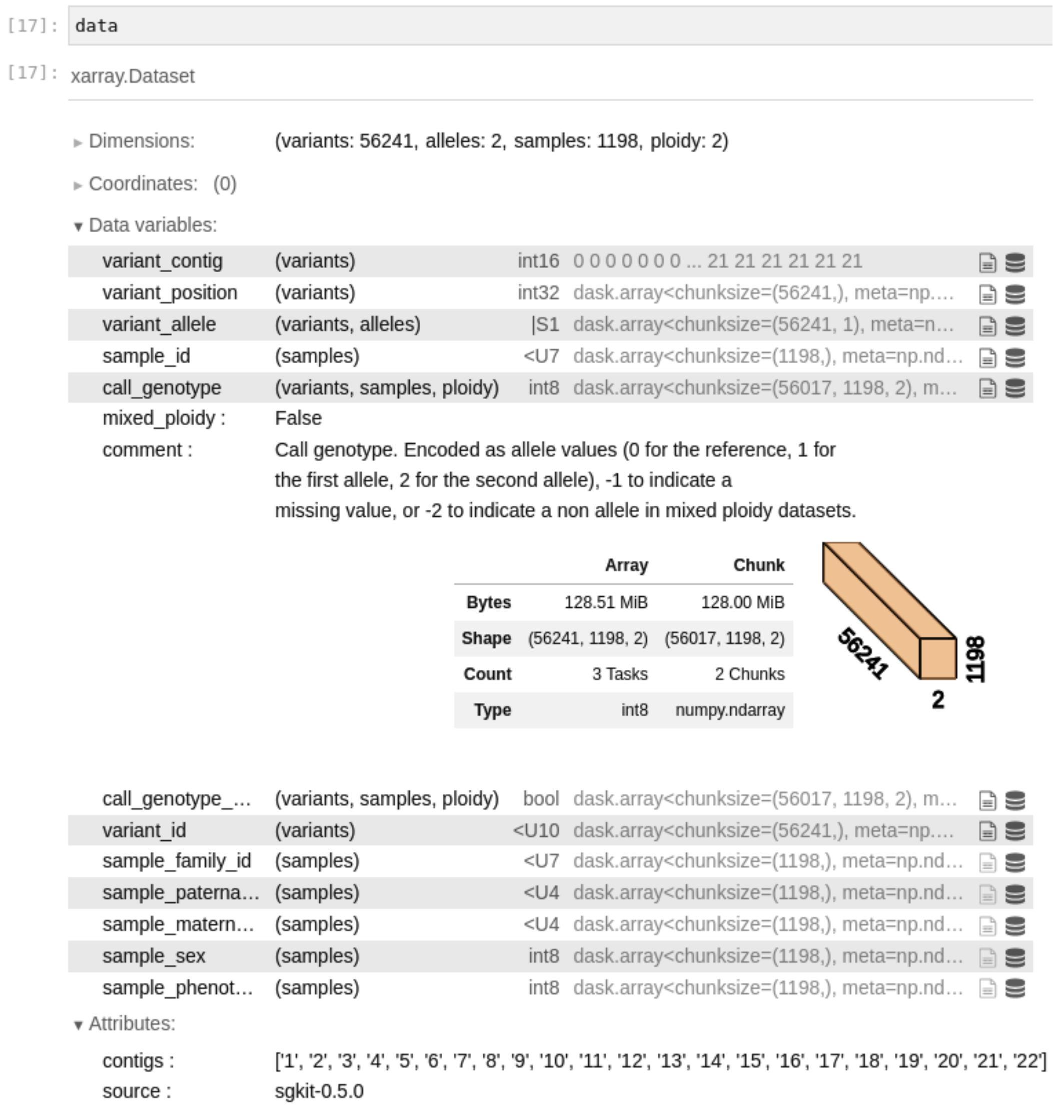
   ```
   import numpy as np
   from sgkit.io import plink

   data = plink.read_plink(path='hapmap10_auto_noofs_ld',
   fam_sep='\t')
   ```

   Remember that we are loading a set of PLINK files. It turns out that sgkit creates a very rich and structured representation for that data. That representation is based on an xarray dataset.

2. Let's check the structure of our data – if you are in a notebook, just enter the following:

   ```
   data
   ```

   `sgkit` – if in a notebook – will generate the following representation:



```
[17]: data
```

```
[17]: xarray.Dataset

▸ Dimensions:      (variants: 56241, alleles: 2, samples: 1198, ploidy: 2)
▸ Coordinates:  (0)
▾ Data variables:
    variant_contig     (variants)                   int16   0 0 0 0 0 0 0 ... 21 21 21 21 21 21
    variant_position   (variants)                   int32   dask.array<chunksize=(56241,), meta=np....
    variant_allele     (variants, alleles)          |S1     dask.array<chunksize=(56241, 1), meta=n...
    sample_id          (samples)                    <U7     dask.array<chunksize=(1198,), meta=np.nd...
    call_genotype      (variants, samples, ploidy)  int8    dask.array<chunksize=(56017, 1198, 2), m...
    mixed_ploidy :     False
    comment :          Call genotype. Encoded as allele values (0 for the reference, 1 for
                       the first allele, 2 for the second allele), -1 to indicate a
                       missing value, or -2 to indicate a non allele in mixed ploidy datasets.
```

	Array	Chunk
Bytes	128.51 MiB	128.00 MiB
Shape	(56241, 1198, 2)	(56017, 1198, 2)
Count	3 Tasks	2 Chunks
Type	int8	numpy.ndarray

```
    call_genotype_...  (variants, samples, ploidy)  bool    dask.array<chunksize=(56017, 1198, 2), m...
    variant_id         (variants)                   <U10    dask.array<chunksize=(56241,), meta=np....
    sample_family_id   (samples)                    <U7     dask.array<chunksize=(1198,), meta=np.nd...
    sample_paterna...  (samples)                    <U4     dask.array<chunksize=(1198,), meta=np.nd...
    sample_matern...   (samples)                    <U4     dask.array<chunksize=(1198,), meta=np.nd...
    sample_sex         (samples)                    int8    dask.array<chunksize=(1198,), meta=np.nd...
    sample_phenot...   (samples)                    int8    dask.array<chunksize=(1198,), meta=np.nd...
▾ Attributes:
    contigs :          ['1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11', '12', '13', '14', '15', '16', '17', '18', '19', '20', '21', '22']
    source :           sgkit-0.5.0
```

Figure 6.1 - An overview of the xarray data loaded by sgkit for our PLINK file

`data` is an xarray DataSet. An xarray DataSet is essentially a dictionary in which each value is a Dask array. For our purposes, you can assume it is a NumPy array. In this case, we can see that we have **56241** variants for **1198** samples. We have **2** alleles per variant and a ploidy of **2**.

In the notebook, we can expand each entry. In our case, we expanded `call_genotype`. This is a three-dimensional array, with `variants`, `samples`, and `ploidy` dimensions.

The type of the array is `int8`. After this, we can find some metadata relevant to the entry, `mixed_ploidy`, and comment. Finally, you have a summary of the Dask implementation. The **Array** column presents details about the size and shape of the array. For the **Chunk** column, see *Chapter 11* – but you can safely ignore it for now.

3. Another way to get summary information, which is especially useful if you are not using notebooks, is by inspecting the `dims` field:

```
print(data.dims)
```

The output should be self-explanatory:

```
Frozen({'variants': 56241, 'alleles': 2, 'samples': 1198,
'ploidy': 2})
```

4. Let's extract some information about the samples:

```
print(len(data.sample_id.values))
print(data.sample_id.values)
print(data.sample_family_id.values)
print(data.sample_sex.values)
```

The output is as follows:

```
1198
['NA19916' 'NA19835' 'NA20282' ... 'NA18915' 'NA19250'
'NA19124']
['2431' '2424' '2469' ... 'Y029' 'Y113' 'Y076']
[1 2 2 ... 1 2 1]
```

We have `1198` samples. The first one has a sample ID of `NA19916`, a family ID of `2431`, and a sex of `1` (Male). Remember that, given PLINK as the data source, a sample ID is not enough to be a primary key (you can have different samples with the same sample ID). The primary key is a composite of the sample ID and sample family ID.

TIP

You might have noticed that we add `.values` to all the data fields: this is actually rendering a lazy Dask array into a materialized NumPy one. For now, I suggest that you ignore it, but if you revisit this chapter after reading *Chapter 11*, `.values` is akin to the `compute` method in Dask.

The `.values` call is no nuisance – the reason our code works is that our dataset is small enough to fit into memory, which is great for our teaching example. But if you have a very large dataset, the preceding code is too naive. Again, *Chapter 11* will help you with this. For now, the simplicity is pedagogical.

5. Before we look at the variant data, we have to be aware of how sgkit stores `contigs`:

```
print(data.contigs)
```

The output is as follows:

```
['1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11',
'12', '13', '14', '15', '16', '17', '18', '19', '20',
'21', '22']
```

The `contigs` here are the human autosomes (you will not be so lucky if your data is based on most other species – you will probably have some ugly identifier here).

6. Now, let's look at the variants:

```
print(len(data.variant_contig.values))
print(data.variant_contig.values)
print(data.variant_position.values)
print(data.variant_allele.values)
print(data.variant_id.values)
```

Here is an abridged version of the output:

```
56241
[ 0  0  0 ... 21 21 21]
[  557616   782343   908247 ... 49528105 49531259
49559741]
[[b'G' b'A']
 ...
 [b'C' b'A']]
['rs11510103' 'rs2905036' 'rs13303118' ... 'rs11705587'
'rs7284680'
 'rs2238837']
```

We have `56241` variants. The `contig` index is `0`, which if you look at the step from the previous recipe, is chromosome `1`. The variant is in position `557616` (against build 36 of the human genome) and has possible alleles `G` and `A`. It has an SNP ID of `rs11510103`.

7. Finally, let's look at the `genotype` data:

```
call_genotype = data.call_genotype.values
print(call_genotype.shape)
first_individual = call_genotype[:,0,:]
first_variant = call_genotype[0,:,:]
first_variant_of_first_individual = call_genotype[0,0,:]
```

```
print(first_variant_of_first_individual)
print(data.sample_family_id.values[0], data.sample_
id.values[0])
print(data.variant_allele.values[0])
```

`call_genotype` has a shape of 56,241 x 1,1198,2, which is its dimensioned variants, samples, and ploidy.

To get all variants for the first individual, you fixate the second dimension. To get all the samples for the first variant, you fixate the first dimension.

If you print the first individual's details (sample and family ID), you get `2431` and `NA19916` – as expected, exactly as in the first case in the previous sample exploration.

There's more...

This recipe is mostly an introduction to xarray, disguised as a sgkit tutorial. There is much more to be said about xarray – be sure to check `https://docs.xarray.dev/`. It is worth reiterating that xarray depends on a plethora of Python data science libraries and that we are glossing over Dask for now.

Exploring a dataset with sgkit

In this recipe, we will perform an initial exploratory analysis of one of our generated datasets. Now that we have some basic knowledge of xarray, we can actually try to do some data analysis. In this recipe, we will ignore population structure, an issue we will return to in the following one.

Getting ready

You will need to have run the first recipe and should have the `hapmap10_auto_noofs_ld` files available. There is a Notebook file with this recipe called `Chapter06/Exploratory_Analysis.py`. You will need the software that you installed for the previous recipe.

How to do it...

Take a look at the following steps:

1. We start by loading the PLINK data with sgkit, exactly as in the previous recipe:

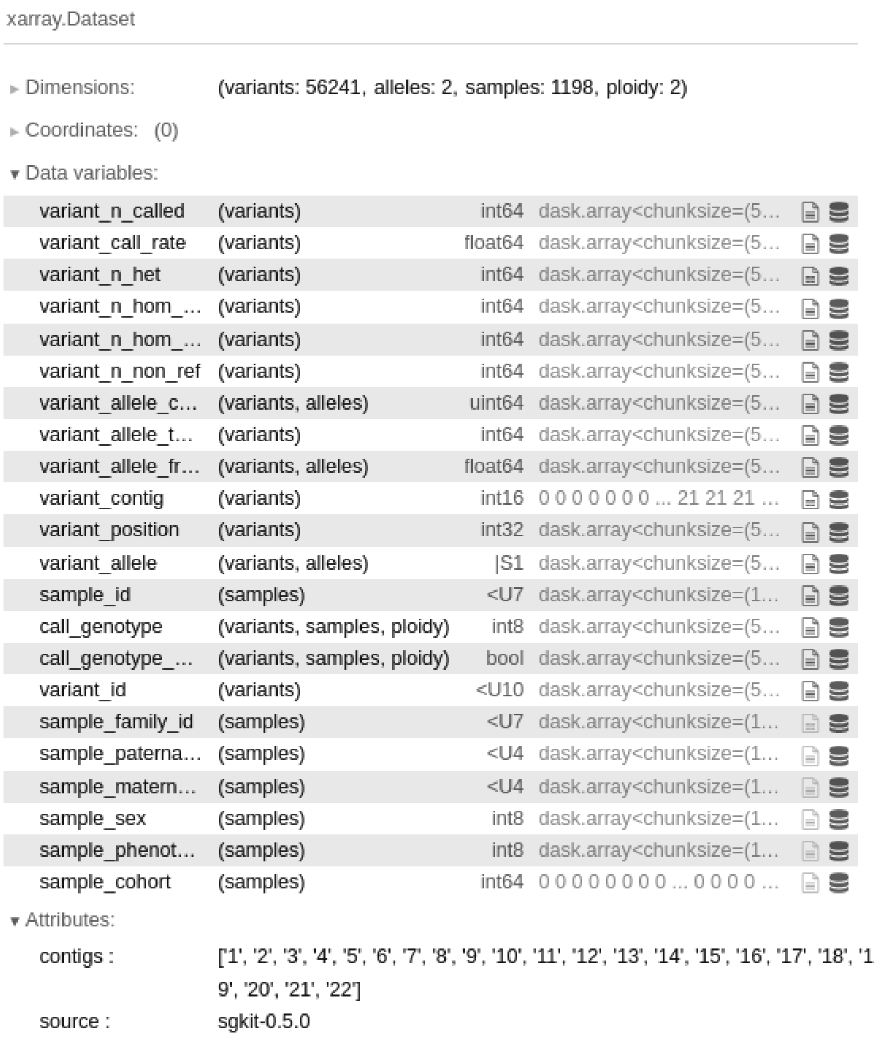
```
import numpy as np
import xarray as xr
import sgkit as sg
from sgkit.io import plink
```

```
data = plink.read_plink(path='hapmap10_auto_noofs_ld',
fam_sep='\t')
```

2. Let's ask sgkit for `variant_stats`:

```
variant_stats = sg.variant_stats(data)
variant_stats
```

The output is the following:



```
xarray.Dataset

▸ Dimensions:      (variants: 56241, alleles: 2, samples: 1198, ploidy: 2)
▸ Coordinates:  (0)
▾ Data variables:
  variant_n_called      (variants)                   int64  dask.array<chunksize=(5...
  variant_call_rate     (variants)                 float64  dask.array<chunksize=(5...
  variant_n_het         (variants)                   int64  dask.array<chunksize=(5...
  variant_n_hom_...     (variants)                   int64  dask.array<chunksize=(5...
  variant_n_hom_...     (variants)                   int64  dask.array<chunksize=(5...
  variant_n_non_ref     (variants)                   int64  dask.array<chunksize=(5...
  variant_allele_c...   (variants, alleles)         uint64  dask.array<chunksize=(5...
  variant_allele_t...   (variants)                   int64  dask.array<chunksize=(5...
  variant_allele_fr...  (variants, alleles)        float64  dask.array<chunksize=(5...
  variant_contig        (variants)                   int16  0 0 0 0 0 0 0 ... 21 21 21 ...
  variant_position      (variants)                   int32  dask.array<chunksize=(5...
  variant_allele        (variants, alleles)            |S1  dask.array<chunksize=(5...
  sample_id             (samples)                      <U7  dask.array<chunksize=(1...
  call_genotype         (variants, samples, ploidy)   int8  dask.array<chunksize=(5...
  call_genotype_...     (variants, samples, ploidy)   bool  dask.array<chunksize=(5...
  variant_id            (variants)                    <U10  dask.array<chunksize=(5...
  sample_family_id      (samples)                      <U7  dask.array<chunksize=(1...
  sample_paterna...     (samples)                      <U4  dask.array<chunksize=(1...
  sample_matern...      (samples)                      <U4  dask.array<chunksize=(1...
  sample_sex            (samples)                     int8  dask.array<chunksize=(1...
  sample_phenot...      (samples)                     int8  dask.array<chunksize=(1...
  sample_cohort         (samples)                    int64  0 0 0 0 0 0 0 0 ... 0 0 0 0 ...
▾ Attributes:
  contigs :        ['1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11', '12', '13', '14', '15', '16', '17', '18', '1
                   9', '20', '21', '22']
  source :         sgkit-0.5.0
```

Figure 6.2 - The variant statistics provided by sgkit's variant_stats

3. Let's now look at the statistic, `variant_call_rate`:

```
variant_stats.variant_call_rate.to_series().describe()
```

There is more to unpack here than it may seem. The fundamental part is the `to_series()` call. Sgkit is returning a Pandas series to you – remember that sgkit is highly integrated with Python data science libraries. After you get the Series object, you can call the Pandas `describe` function and get the following:

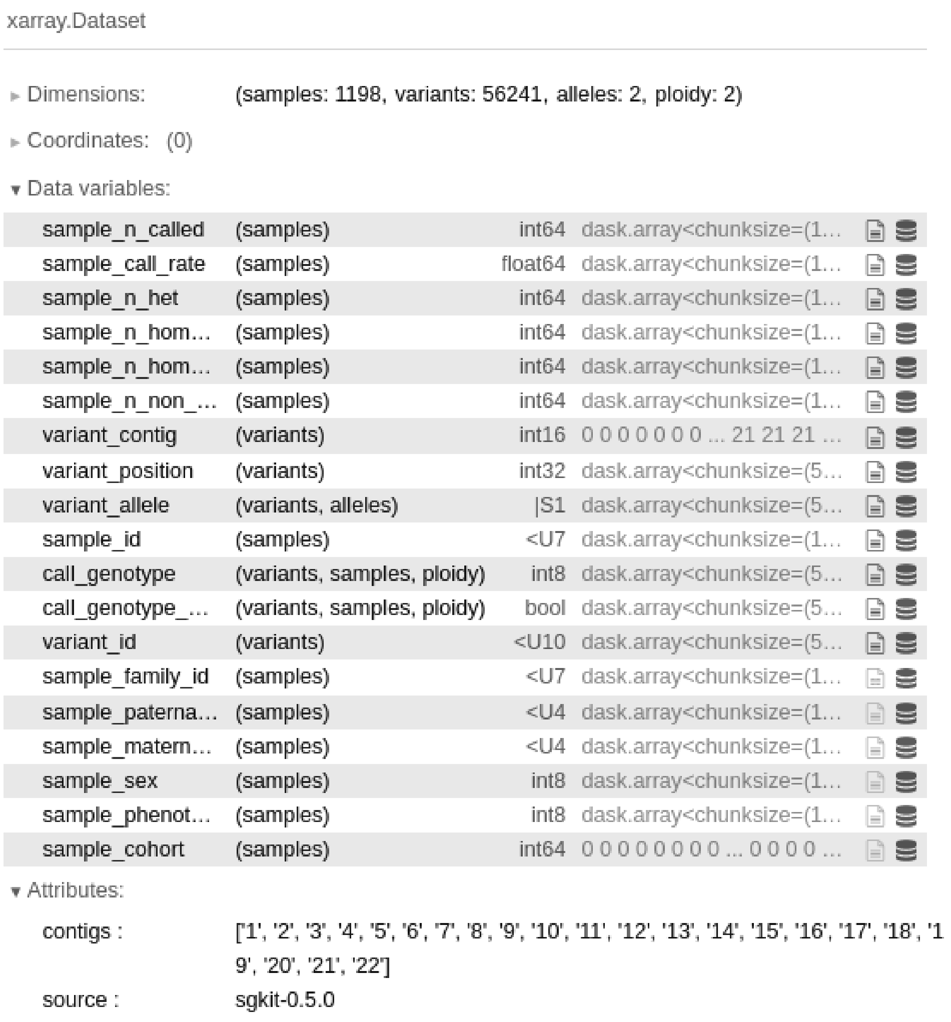
```
count    56241.000000
mean         0.997198
std          0.003922
min          0.964107
25%          0.996661
50%          0.998331
75%          1.000000
max          1.000000
Name: variant_call_rate, dtype: float64
```

Our variant call rate is quite good, which is not shocking because we are looking at array data – you would have worse numbers if you had a dataset based on NGS.

4. Let's now look at sample statistics:

```
sample_stats = sg.sample_stats(data)
sample_stats
```

Again, sgkit provides a lot of sample statistics out of the box:



```
xarray.Dataset

▸ Dimensions:      (samples: 1198, variants: 56241, alleles: 2, ploidy: 2)
▸ Coordinates:  (0)
▾ Data variables:
   sample_n_called     (samples)                        int64   dask.array<chunksize=(1…
   sample_call_rate    (samples)                      float64   dask.array<chunksize=(1…
   sample_n_het        (samples)                        int64   dask.array<chunksize=(1…
   sample_n_hom…       (samples)                        int64   dask.array<chunksize=(1…
   sample_n_hom…       (samples)                        int64   dask.array<chunksize=(1…
   sample_n_non_…      (samples)                        int64   dask.array<chunksize=(1…
   variant_contig      (variants)                       int16   0 0 0 0 0 0 0 ... 21 21 21 …
   variant_position    (variants)                       int32   dask.array<chunksize=(5…
   variant_allele      (variants, alleles)                |S1   dask.array<chunksize=(5…
   sample_id           (samples)                          <U7   dask.array<chunksize=(1…
   call_genotype       (variants, samples, ploidy)       int8   dask.array<chunksize=(5…
   call_genotype_…     (variants, samples, ploidy)       bool   dask.array<chunksize=(5…
   variant_id          (variants)                        <U10   dask.array<chunksize=(5…
   sample_family_id    (samples)                          <U7   dask.array<chunksize=(1…
   sample_paterna…     (samples)                          <U4   dask.array<chunksize=(1…
   sample_matern…      (samples)                          <U4   dask.array<chunksize=(1…
   sample_sex          (samples)                         int8   dask.array<chunksize=(1…
   sample_phenot…      (samples)                         int8   dask.array<chunksize=(1…
   sample_cohort       (samples)                        int64   0 0 0 0 0 0 0 0 ... 0 0 0 0 …
▾ Attributes:
   contigs :           ['1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11', '12', '13', '14', '15', '16', '17', '18', '1
                       9', '20', '21', '22']
   source :            sgkit-0.5.0
```

Figure 6.3 - The sample statistics obtained by calling sample_stats

5. We will now have a look at sample call rates:

```
sample_stats.sample_call_rate.to_series().hist()
```

This time, we plot a histogram of sample call rates. Again, sgkit gets this for free by leveraging Pandas:

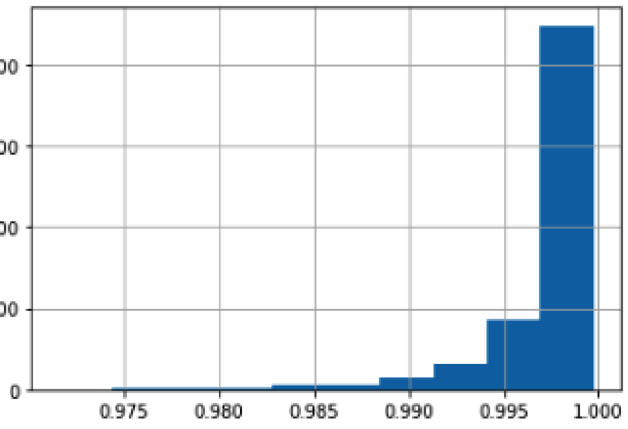


Figure 6.4 - The histogram of sample call rates

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. Indeed, sgkit is prepared to deal with that because it leverages Dask. It would be too complex to introduce Dask at this stage, but for large datasets, *Chapter 11* will discuss ways to address those.

See also

- A list of R packages for statistical genetics is available at `http://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.*

Analyzing population structure

Previously, we introduced data analysis with sgkit ignoring the population structure. Most datasets, including the one we are using, actually do have a population structure. Sgkit provides functionality to analyze genomic datasets with population structure and that is what we are going to investigate here.

Getting ready

You will need to have run the first recipe, and should have the `hapmap10_auto_noofs_ld` data we produced and also the original population meta data `relationships_w_pops_041510.txt` file downloaded. There is a Notebook file with the `06_PopGen/Pop_Stats.py` recipe in it.

How to do it...

Take a look at the following steps:

1. First, let's load the PLINK data with sgkit:

```
from collections import defaultdict
from pprint import pprint
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import xarray as xr
import sgkit as sg
from sgkit.io import plink

data = plink.read_plink(path='hapmap10_auto_noofs_ld',
fam_sep='\t')
```

2. Now, let's load the data assigning individuals to populations:

```
f = open('relationships_w_pops_041510.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))

pops = list(pop_ind.keys())
```

We end up with a dictionary, `pop_ind`, where the key is the population code, and the value is a list of samples. Remember that a sample primary key is the family ID and the sample ID.

We also have a list of populations in the `pops` variable.

3. We now need to inform sgkit about to which population or cohort each sample belongs:

```
def assign_cohort(pops, pop_ind, sample_family_id,
sample_id):
    cohort = []
    for fid, sid in zip(sample_family_id, sample_id):
        processed = False
        for i, pop in enumerate(pops):
            if (fid, sid) in pop_ind[pop]:
                processed = True
                cohort.append(i)
                break
        if not processed:
            raise Exception(f'Not processed {fid},
{sid}')
    return cohort
cohort = assign_cohort(pops, pop_ind, data.sample_family_
id.values, data.sample_id.values)
data['sample_cohort'] = xr.DataArray(
    cohort, dims='samples')
```

Remember that each sample in sgkit has a position in an array. So, we have to create an array where each element refers to a specific population or cohort within a sample. The `assign_cohort` function does exactly that: it takes the metadata that we loaded from the `relationships` file and the list of samples from the sgkit file, and gets the population index for each sample.

4. Now that we have loaded population information structure into the sgkit dataset, we can start computing statistics at the population or cohort level. Let's start by getting the number of monomorphic loci per population:

```
cohort_allele_frequency = sg.cohort_allele_
frequencies(data)['cohort_allele_frequency'].values
monom = {}
for i, pop in enumerate(pops):
    monom[pop] = len(list(filter(lambda x: x,
np.isin(cohort_allele_frequency[:, i, 0], [0, 1]))))
pprint(monom)
```

We start by asking sgkit to calculate the allele frequencies per cohort or population. After that, we filter all loci per population where the allele frequency of the first allele is either `0` or `1` (that is, there is the fixation of one of the alleles). Finally, we print it. Incidentally, we use the `pprint.pprint` function to make it look a bit better (the function is quite useful for more complex structures if you want to render the output in a readable way):

```
{'ASW': 3332,
 'CEU': 8910,
 'CHB': 11130,
 'CHD': 12321,
 'GIH': 8960,
 'JPT': 13043,
 'LWK': 3979,
 'MEX': 6502,
 'MKK': 3490,
 'TSI': 8601,
 'YRI': 5172}
```

5. Let's get the minimum allele frequency for all loci per population. This is still based in `cohort_allele_frequency` – so no need to call sgkit again:

```
mafs = {}
for i, pop in enumerate(pops):
    min_freqs = map(
        lambda x: x if x < 0.5 else 1 - x,
        filter(
            lambda x: x not in [0, 1],
            cohort_allele_frequency[:, i, 0]))
    mafs[pop] = pd.Series(min_freqs)
```

We create Pandas `Series` objects for each population, as this permits lots of helpful functions, such as plotting.

6. We will now print the MAF histograms for the `YRI` and `JPT` populations. We will leverage Pandas and Matplotlib for this:

```
maf_plot, maf_ax = plt.subplots(nrows=2, sharey=True)
mafs['YRI'].hist(ax=maf_ax[0], bins=50)
maf_ax[0].set_title('*YRI*')
mafs['JPT'].hist(ax=maf_ax[1], bins=50)
maf_ax[1].set_title('*JPT*')
maf_ax[1].set_xlabel('MAF')
```

We get Pandas to generate the histograms and put the results in a Matplotlib plot. The result is the following:

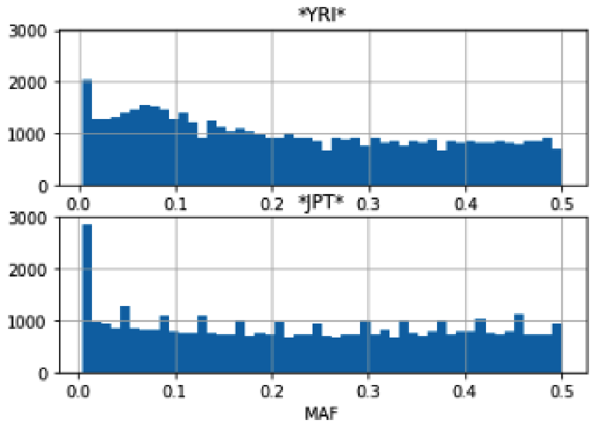


Figure 6.5 - A MAF histogram for the YRI and JPT populations

7. We are now going to concentrate on computing the FST. The FST is a widely used statistic that tries to represent the genetic variation created by population structure. Let's compute it with sgkit:

```
fst = sg.Fst(data)
fst = fst.assign_coords({"cohorts_0": pops, "cohorts_1":
pops})
```

The first line computes `fst`, which, in this case, will be pairwise `fst` across cohorts or populations. The second line assigns names to each cohorts by using the xarray coordinates feature. This makes it easier and more declarative.

8. Let's compare `fst` between the `CEU` and `CHB` populations with `CHB` and `CHD`:

```
remove_nan = lambda data: filter(lambda x: not
np.isnan(x), data)
ceu_chb = pd.Series(remove_nan(fst.stat_Fst.
sel(cohorts_0='CEU', cohorts_1='CHB').values))
chb_chd = pd.Series(remove_nan(fst.stat_Fst.
sel(cohorts_0='CHB', cohorts_1='CHD').values))
ceu_chb.describe()
chb_chd.describe()
```

We take the pairwise results returned by the `sel` function from `stat_FST` to both compare and create a Pandas Series with it. Note that we can refer to populations by name, as we have prepared the coordinates in the previous step.

9. Let's plot the distance matrix across populations based on the multi-locus pairwise FST. Before we do it, we will prepare the computation:

```
mean_fst = {}
for i, pop_i in enumerate(pops):
    for j, pop_j in enumerate(pops):
        if j <= i:
            continue
        pair_fst = pd.Series(remove_nan(fst.stat_Fst.
sel(cohorts_0=pop_i, cohorts_1=pop_j).values))
        mean = pair_fst.mean()
        mean_fst[(pop_i, pop_j)] = mean
min_pair = min(mean_fst.values())
max_pair = max(mean_fst.values())
```

We compute all the FST values for the population pairs. The execution of this code will be demanding in terms of time and memory, as we are actually requiring Dask to perform a lot of computations to render our NumPy arrays.

10. We can now do a pairwise plot of all mean FSTs across populations:

```
sns.set_style("white")
num_pops = len(pops)
arr = np.ones((num_pops - 1, num_pops - 1, 3),
dtype=float)
fig = plt.figure(figsize=(16, 9))
ax = fig.add_subplot(111)
for row in range(num_pops - 1):
    pop_i = pops[row]
    for col in range(row + 1, num_pops):
        pop_j = pops[col]
        val = mean_fst[(pop_i, pop_j)]
        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_title('Multilocus Pairwise FST')
ax.set_xticks(range(num_pops - 1))
ax.set_xticklabels(pops[1:])
ax.set_yticks(range(num_pops - 1))
ax.set_yticklabels(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 (a lower FST) and blue means more different (a higher FST). 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 pairwise FST between these two populations:

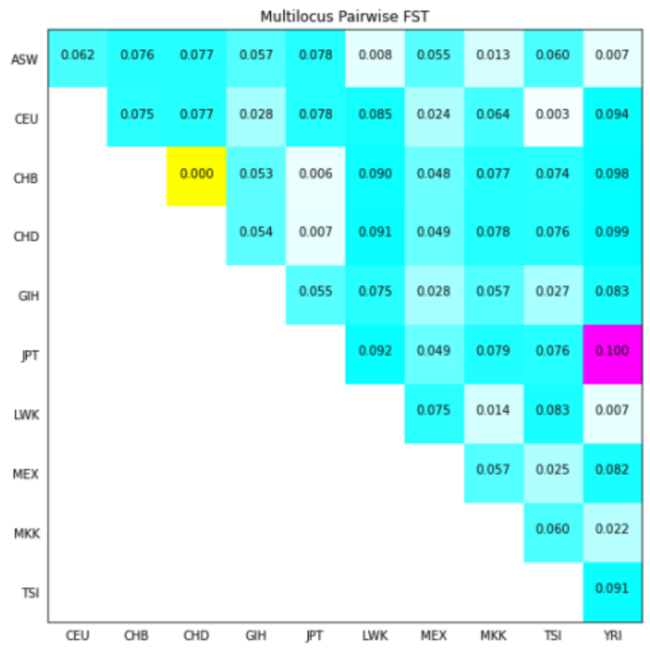


Figure 6.6 - The average pairwise FST across the 11 populations in the HapMap project for all autosomes

See also

- F-statistics is an immensely complex topic, so I will direct you firstly to the Wikipedia page at `http://en.wikipedia.org/wiki/F-statistics`.
- A very good explanation can be found in Holsinger and Weir's paper (*Genetics in geographically structured populations: defining, estimating, and interpreting FST*) in *Nature Reviews Genetics*, at `http://www.nature.com/nrg/journal/v10/n9/abs/nrg2611.html`.

Performing a PCA

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. In this recipe, we will use SmartPCA – for a native machine learning experience with scikit-learn, see *Chapter 10*.

TIP

You actually have a third option: using sgkit. However, I want to show you alternatives on how to perform computations. There are two good reasons for this. Firstly, you might prefer not to use sgkit – while I recommend it, I don't want to force it – and secondly, you might be required to run an alternative method that is not implemented in sgkit. PCA is actually a good example of this: a reviewer on a paper might require you to run a published and widely used method such as EIGENSOFT SmartPCA.

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 with SmartPCA and scikit-learn easier.

I have a simple library to help with some genomics processing. 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 `Chapter06/PCA.py` recipe, but you will still need to run the first recipe.

How to do it...

Take a look at the following steps:

1. Let's load the metadata, as follows:

```
f = open('relationships_w_pops_041510.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.

2. 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.

3. 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 the PCA weights (which we will not use, but you should inspect), normalized weights, and then the principal components (usually up to PC `10`) per individual.

4. Then, we plot PC `1` and PC `2`, 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 at the 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:

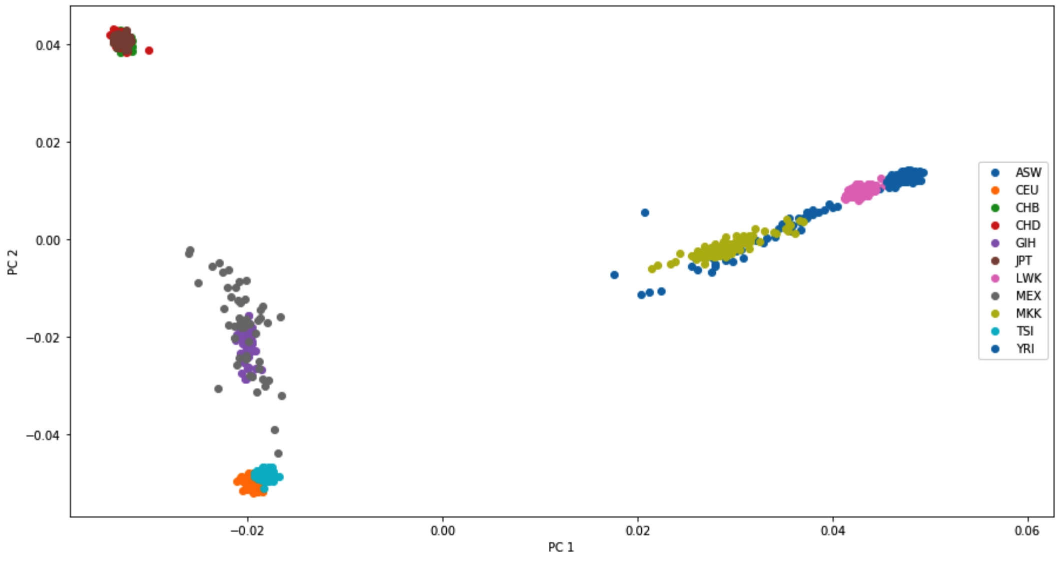


Figure 6.7 - PC 1 and PC 2 of the HapMap data, as produced by SmartPCA

Note

Note that PCA plots can be symmetrical in any axis across runs, as the 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.

There's more...

An interesting question here is which method you should use – SmartPCA or scikit-learn, which we will use in *Chapter 10*. 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.'s *Genes mirror geography within Europe* on *Nature*, where a PCA of Europeans mapped almost perfectly to a map of Europe. This can be found at `http://www.nature.com/nature/journal/v456/n7218/abs/nature07331.html`. Note that there is nothing about PCA that assures it will map to geographical features (just check our PCA earlier).
- The SmartPCA is described in Patterson et al.'s *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. The 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 (`https://dalexander.github.io/admixture/download.html`).

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 previous recipe, you will use the pygenomics library to help; 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 Chapter06/Admixture.py recipe, but you will still need to run the recipe first.

How to do it...

Take a look at the following steps:

1. First, let's define our k (a number of ancestral populations) range of interest, as follows:

```
k_range = range(2, 10)  # 2..9
```

2. Let's run admixture for all our k (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))
```

Note

This is the worst possible way of running admixture and will probably take more than 3 hours if you do it this way. This is because it will run all k from 2 to 9 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 *Chapter 11*.

3. 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()
```

4. 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`, the lower, the better. It should be clear from this graph that we should maybe run some more `K` (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 would be a very technical debate on whether there is such 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 the `K` results:

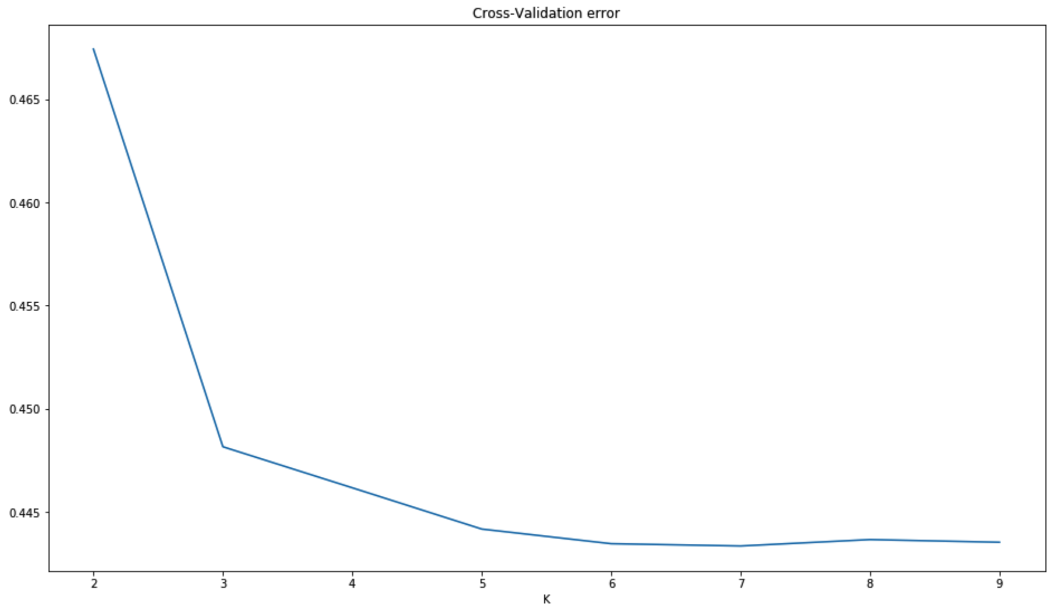


Figure 6.8 - The error by K

5. We will need the metadata for the population information:

```
f = open('relationships_w_pops_041510.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.

6. 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 `k` that you decided to study. Here, we will load the `Q` file for all `k` that we studied and store them in a dictionary where the individual ID is the key.

7. 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 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://github.com/tiagoantao/pygenomics/blob/master/genomics/popgen/admix/__init__.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 take a look (by the way, it's not very difficult to improve upon it).

8. 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 the components per individual and their 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 `K` (we have chosen a `K` of `7` here).

Note that all `K` 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**):

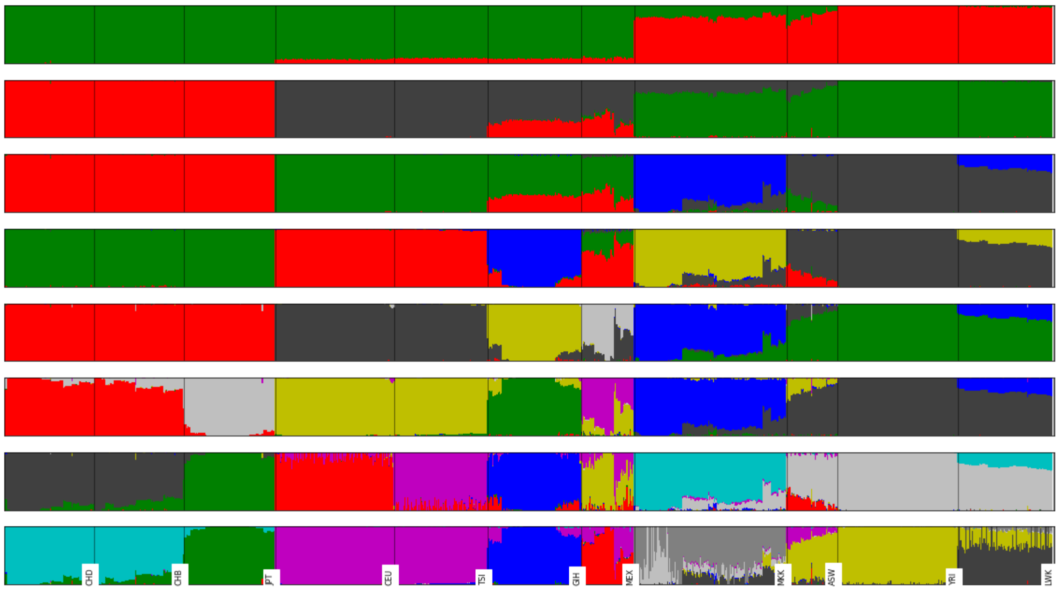


Figure 6.9 - A stacked admixture plot (between K of 2 and 9) for the HapMap example

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 `K` 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.

7
Phylogenetics

Phylogenetics is the application of molecular sequencing that is used 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.

In this book, we will take the practical approach that is mentioned 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 in *Science*. It is available at `https://pubmed.ncbi.nlm.nih.gov/25214632/`. Here, we will try to follow a similar methodology to arrive at similar results to the paper.

In this chapter, we will use DendroPy (a phylogenetics library) and Biopython. The `bioinformatics_phylo` Docker image includes all the necessary software.

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

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, which was 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 a sequenced form. You can read more at `https://en.wikipedia.org/wiki/Ebolavirus`.

If you have already gone through the previous chapters, you might panic looking at the potential data sizes involved here; this is not a problem at all because these are genomes of viruses that are each around 19 kbp in size. So, our approximately 100 genomes are actually quite light.

To do this analysis, we will need to install `dendropy`. If you are using Anaconda, please perform the following:

```
conda install –c bioconda dendropy
```

As usual, this information is available in the corresponding Jupyter Notebook file, which is available at `Chapter07/Exploration.py`.

How to do it...

Take a look at the following steps:

1. First, let's start by 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'])
```

Here, we have three functions: one to retrieve data from the most recent EBOV outbreak, another to retrieve data from the previous EBOV outbreaks, and one to retrieve data 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 the code to download more genomes. For our purpose, the subset that we will download is enough.

2. 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 the `dendropy` functions to create FASTA files that are retrieved from GenBank records through conversion. The ID of each sequence in the FASTA file is produced by a lambda function that uses the species and the year, alongside the GenBank accession number.

3. 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 (please refer to *Chapter 3*, *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 the FASTA files in order to extract the genes. We put each gene into a different file and only take two virus species. We also get translated proteins, which are available in the records for each gene.

4. 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. Then, we extract all the 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 additional details regarding the API.

5. 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 size of 18,925.2, and a maximum size of 18,959. This is a small genome when compared to eukaryotes.

Note that at the very end, the memory structure has been 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.

6. 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 the species and the EBOV breakdown is detailed next (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).

7. Let's extract the basic statistics of a gene in 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 (that is, the name and the 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 discover, it's more robust with certain tasks (such as file parsing). More 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 *Phylip*, which can be found at `http://evolution.genetics.washington.edu/phylip.html`.
- We will use the Nexus and Newick formats in future recipes (`http://evolution.genetics.washington.edu/phylip/newicktree.html`), but do 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. Additionally, 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:

```
conda install –c bioconda mafft trimal muscle=3.8
```

As usual, this information is available in the corresponding Jupyter Notebook file at `Chapter07/Alignment.py`. You will need to run the previous notebook beforehand, 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:

1. Now, we will run MAFFT to align the genomes, as shown in the following code. This task is CPU-intensive and memory-intensive, and it 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 preceding parameters are the same as the ones specified in the supplementary material of the paper. We will use the Biopython interface to call MAFFT.

2. 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.

3. Additionally, we can run `MUSCLE` to align the 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 the proteins. Therefore, we have to convert the alignment into the gene sequence form.

4. 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

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 the sequences we aligned in the previous recipe. We will perform gene-wide 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 `Chapter07/Comparison.py`.

How to do it...

Take a look at the following steps:

1. Let's start analyzing the gene data. For simplicity, we will only use data from two other species of the genus Ebola virus that are available in the extended dataset, that is, 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
```

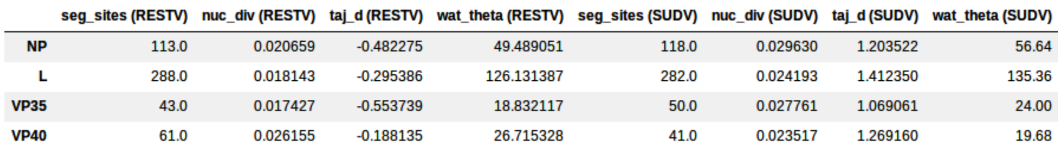
We get four genes that we stored in the first recipe and aligned in the second.

We load all the files (which are FASTA formatted) and create a dictionary with all of 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.

2. 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 out the *There's more...* 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)
```

3. First, let's look at the output, 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

Figure 7.1 – A DataFrame for the virus dataset

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.

The 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 in Jupyter, just putting `df` at the end will render the DataFrame and the cell output, too. If you are not in a notebook, use `print(df)` (you can also perform this in a notebook, but it will not look as pretty).

4. 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.

5. 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 will 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
```

6. 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
```

Now the number of segregating sites is 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 phylogenetic and 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 also includes it, but it is named `raxml`. Note that the binary is called `raxmlHPC`. You can perform the following command to install it:

```
conda install –c bioconda raxml
```

The preceding code 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 might 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 `Chapter07/Reconstruction.py`. You will need the output of the previous recipe to complete this one.

How to do it...

Take a look at the following steps:

1. 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 (at least 10 GB).

Be prepared to wait for some time. Depending on your computer, this could take more than one hour. If it takes longer, consider restarting the process, as sometimes a RAxML bug might occur.

We will run RAxML with the GTRΓ 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 of the process, we will 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 might want to ignore all of these files, but we will discuss this a little in the alternative Biopython step.

2. We will save trees for future analysis; in our case, it will be a 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.

3. 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 we come to the proper recipe later, but if you look at the following diagram and compare it with the results from the paper, you will clearly see 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, by adding their sizes. This is easy to solve; we will deal with this in the *Visualizing phylogenetic data* recipe:

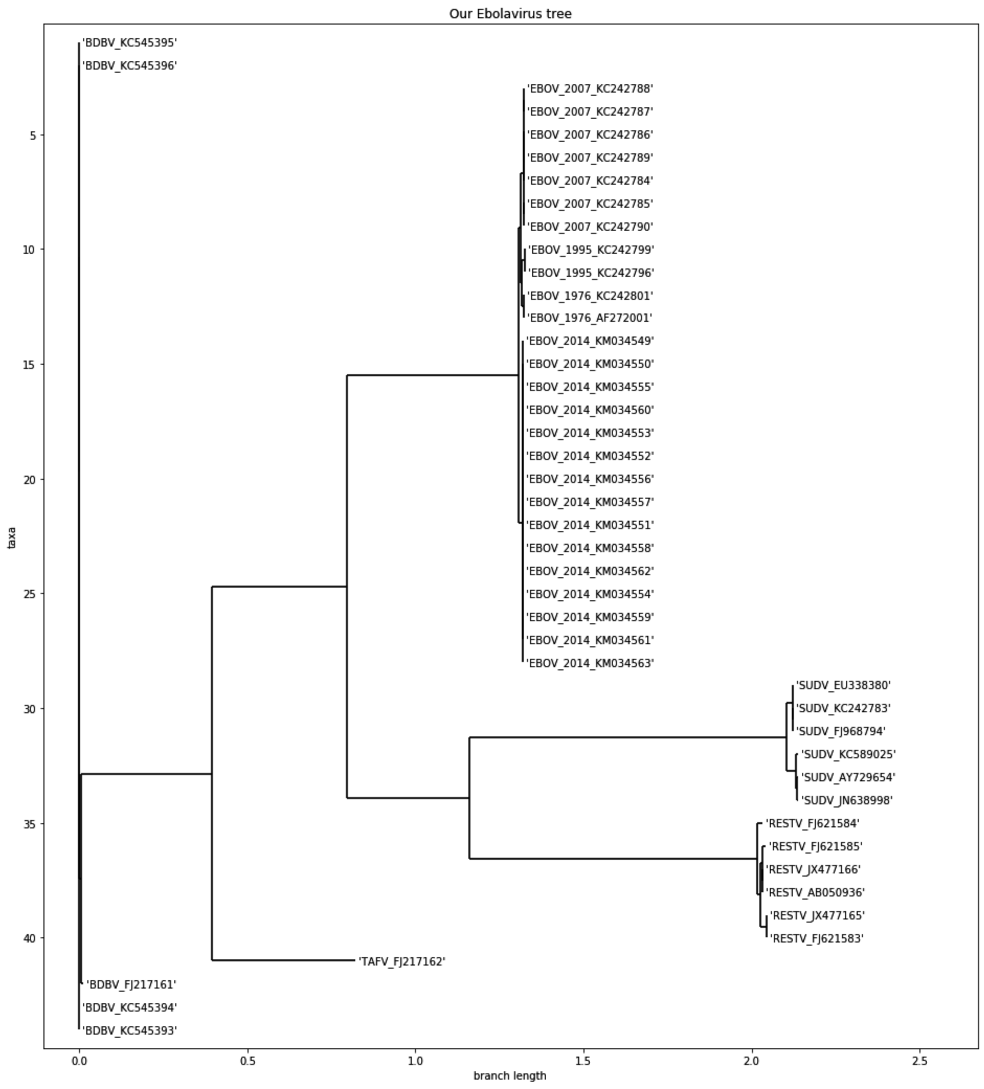


Figure 7.2 – The phylogenetic tree that we generated with RAxML for all Ebola viruses

4. 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 10,000 if you do not do so) and the working directory. With RAxML, the working directory specification requires the absolute path.

5. 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; however, 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 10 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 in 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 in Python, as the topic is vast. Having said that, it's not common for introductory Python books to discuss recursive programming at length. Usually, recursive programming techniques are 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. Additionally, phylogenetic trees can have weights on their edges. Therefore, be mindful of this when you read the documentation; if the 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 mostly devised 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 `Chapter07/Trees.py` notebook file. Here, we will use DendroPy's tree representations. 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:

1. 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')
```

2. 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 `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 `0`. 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 (that is, it is 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.

3. 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 is `0`; if not, then it's `1` plus the maximum height of its entire offspring.

Note that we use a map over a lambda function to get the heights of all the children of the current node. Then, we choose the maximum (the `max` function performs a `reduce` operation here because it summarizes all of the values that are reported). If you are relating this to MapReduce frameworks, you are correct; they are inspired by functional programming dialects like these.

4. Now, let's compute the number of offspring for each node. By now, this should be quite easy to understand:

```
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)
```

5. Now we will print all of 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 so far 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.

6. 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**). That's because it implements an efficient data structure when you operate at both extremes.

The algorithm starts by putting the root node onto 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 of its children into 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 shown in the following diagram. The code is small, but not trivial:

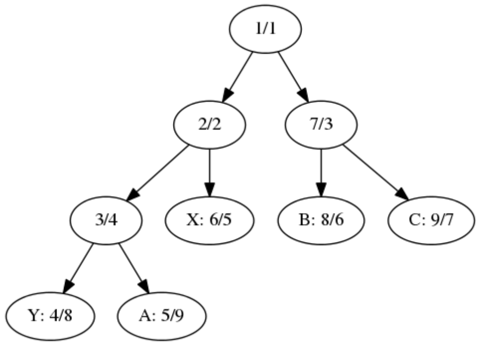


Figure 7.3 – Visiting a tree; the first number indicates the order in which that node is visited traversing depth-first, while the second assumes breadth-first

7. 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 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 deep copy of the tree structure. As our function and the ladderization are destructive (they 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 outbreak year as in the case of EBOV, we remove all of 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_%28data_structure%29`.

Note that the use of `lambda` functions and `map` is not encouraged as a Python dialect; you can read some (older) 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 within functional and recursive programming. The more common dialect will be based on a list of comprehensions.

In any case, the functional dialect based on using the map and reduce operations 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 only has 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 of the 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 Chapter07/Visualization.py notebook file.

How to do it...

Take a look at the following steps:

1. Let's load all of the phylogenetic data:

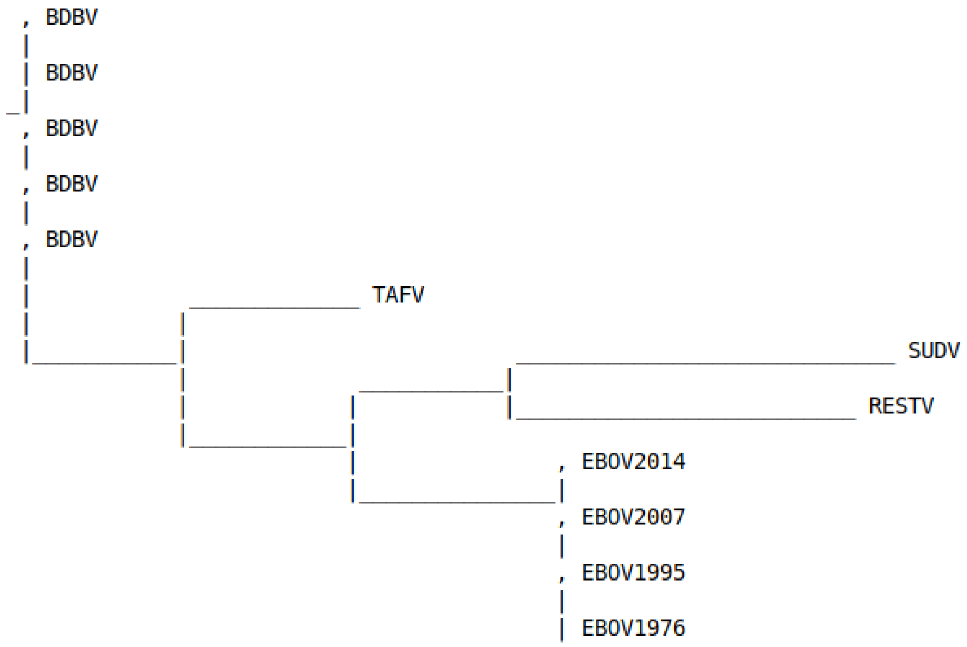
```
from copy import deepcopy
from Bio import Phylo
ebola_tree = Phylo.read('my_ebola.nex', 'nexus')
ebola_tree.name = 'Ebolavirus tree'
ebola_simple_tree = Phylo.read('ebola_simple.nex',
'nexus')
ebola_simple_tree.name = 'Ebolavirus simplified tree'
```

   For all of the trees that we read, we will change the name of the tree, as the name will be printed later.

2. 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:



```
 , BDBV
 |
 | BDBV
_|
 , BDBV
 |
 , BDBV
 |
 , BDBV
 |
 |            ______________ TAFV
 |           |
 |___________|                           ________________________________ SUDV
             |              ____________|
             |             |            |_____________________________ RESTV
             |_____________|
                           |                , EBOV2014
                           |________________|
                                            , EBOV2007
                                            |
                                            , EBOV1995
                                            |
                                            | EBOV1976
```

Figure 7.4 – The ASCII representation of a simplified Ebola virus dataset

3. `Bio.Phylo` 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 at 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:

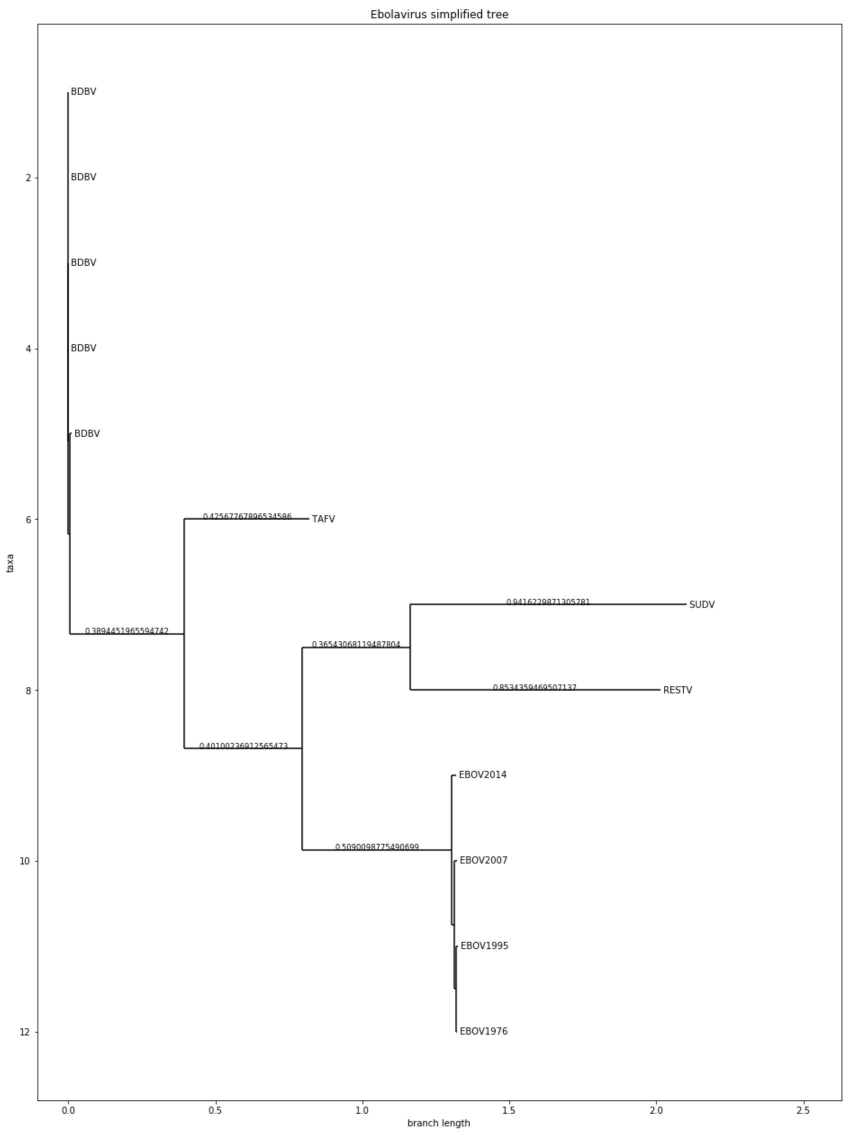


Figure 7.5 – A matplotlib-based version of the simplified dataset with branch lengths added

4. Now we will plot the complete dataset, but we will color each bit of the tree differently. If a subtree only has a single virus species, it will get its own color. EBOV will have two colors, that is, 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 in the following way. 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 gray. 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 deep copy 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:

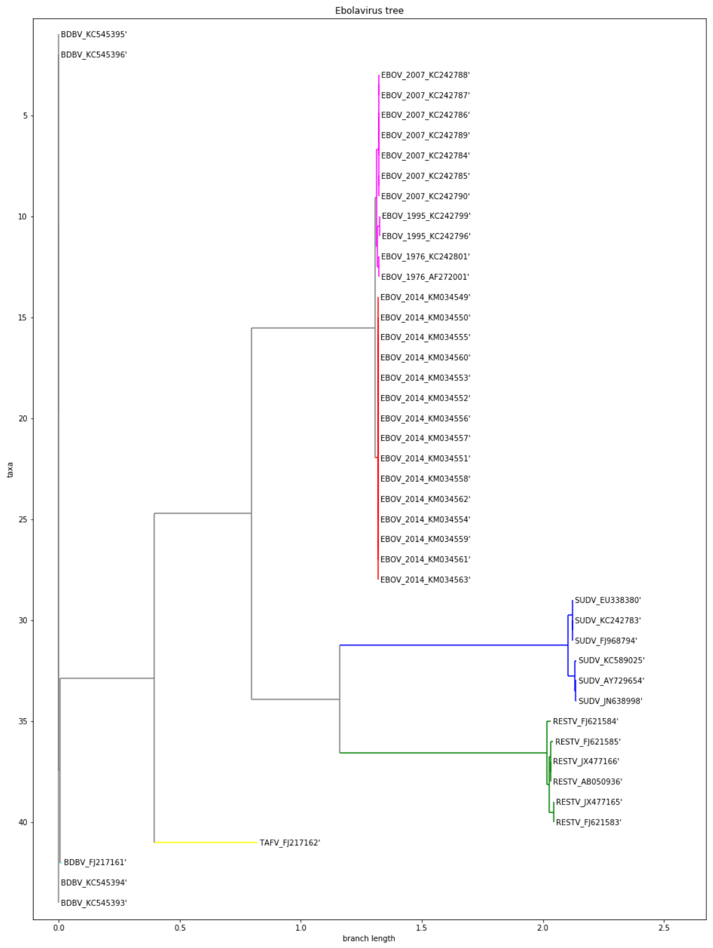


Figure 7.6 – A ladderized and colored phylogenetic tree with the complete Ebola virus dataset

There's more...

Tree and graph visualization is a complex topic; arguably, here, the tree's visualization is rigorous but far from pretty. One alternative to DendroPy, which has more visualization features, is ETE (`http://etetoolkit.org/`). General alternatives for drawing trees and graphs include Cytoscape (`https://cytoscape.org/`) and Gephi (`http://gephi.github.io/`). If you want to know more about the algorithms for rendering trees and graphs, check out the Wikipedia page at `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.

8
Using the Protein Data Bank

Proteomics is the study of proteins, including their function and structure. One of the main objectives of this field is to characterize the three-dimensional structure of proteins. One of the most widely known computational resources in the proteomics field is the **Protein Data Bank** (**PDB**), a repository with the structural data of large biomolecules. Of course, many databases focus on protein primary structure instead; these are somewhat similar to the genomic databases that we saw in *Chapter 2*, *Getting to Know NumPy, pandas, Arrow, and Matplotlib*.

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 within 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 that is probably more suited to a book about chemoinformatics.

Throughout this chapter, we will use a classic example of a protein: the 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 by now: accessing databases, especially for a protein's primary structure (as in, sequences of amino acids).

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

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 the data, we will use Biopython and the REST API (we used a similar approach in *Chapter 5*, *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 `Chapter08/Intro.py` Notebook file.

How to do it...

Take a look at the following steps:

1. 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
```

2. We can now query all the `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,datab
ase(PDB),database(HGNC)',
 limit='50')
```

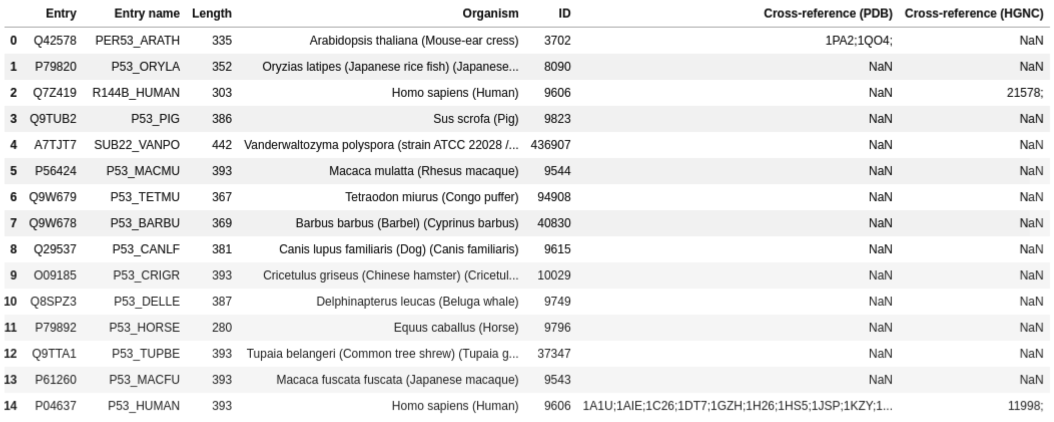
We will query the `p53` gene and request to see all entries that are reviewed (as in, manually curated). The output will be in a tabular format. We will request a maximum of 50 results, specifying the desired columns.

We could have restricted the output to just human data, but for this example, let's include all available species.

3. 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 use `pandas` for easy processing of the tab-delimited list and pretty printing. The abridged output of the Notebook is as follows:



	Entry	Entry name	Length	Organism	ID	Cross-reference (PDB)	Cross-reference (HGNC)
0	Q42578	PER53_ARATH	335	Arabidopsis thaliana (Mouse-ear cress)	3702	1PA2;1QO4;	NaN
1	P79820	P53_ORYLA	352	Oryzias latipes (Japanese rice fish) (Japanese...	8090	NaN	NaN
2	Q7Z419	R144B_HUMAN	303	Homo sapiens (Human)	9606	NaN	21578;
3	Q9TUB2	P53_PIG	386	Sus scrofa (Pig)	9823	NaN	NaN
4	A7TJT7	SUB22_VANPO	442	Vanderwaltozyma polyspora (strain ATCC 22028 /...	436907	NaN	NaN
5	P56424	P53_MACMU	393	Macaca mulatta (Rhesus macaque)	9544	NaN	NaN
6	Q9W679	P53_TETMU	367	Tetraodon miurus (Congo puffer)	94908	NaN	NaN
7	Q9W678	P53_BARBU	369	Barbus barbus (Barbel) (Cyprinus barbus)	40830	NaN	NaN
8	Q29537	P53_CANLF	381	Canis lupus familiaris (Dog) (Canis familiaris)	9615	NaN	NaN
9	O09185	P53_CRIGR	393	Cricetulus griseus (Chinese hamster) (Cricetul...	10029	NaN	NaN
10	Q8SPZ3	P53_DELLE	387	Delphinapterus leucas (Beluga whale)	9749	NaN	NaN
11	P79892	P53_HORSE	280	Equus caballus (Horse)	9796	NaN	NaN
12	Q9TTA1	P53_TUPBE	393	Tupaia belangeri (Common tree shrew) (Tupaia g...	37347	NaN	NaN
13	P61260	P53_MACFU	393	Macaca fuscata fuscata (Japanese macaque)	9543	NaN	NaN
14	P04637	P53_HUMAN	393	Homo sapiens (Human)	9606	1A1U;1AIE;1C26;1DT7;1GZH;1H26;1HS5;1JSP;1KZY;1...	11998;

Figure 8.1 - An abridged list of species for which there is a TP53 protein

4. 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) &
    (uniprot_list['Entry name'].str.contains('P53'))]
```

```
['Entry'].iloc[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 there is an error with your network services, it may be a network or server problem. If this is the case, just retry at a later date.

5. 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')
 MEEPQSDPSVEPPLSQETFSDLWKLLPENNVLSPLPSQAMDDLMLSPDDIEQWFTED
PGPDEAPRMPEAAPPVAPAPAAPTPAAPAPAPSWPLSSSVPSQKTYQGSYGFRLGF
LHSGTAKSVTCTYSPALNKMFCQLAKTCPVQLWVDSTPPPGTRVRAMAIYKQSQHM
TEVVRRCPHHERCSDSDGLAPPQHLIRVEGNLRVEYLDDRNTFRHSVVVPYEPPEVG
SDCTTIHYNYMCNSSCMGGMNRRPILTIITLEDSSGNLLGRNSFEVRVCACPGRDRR
TEEENLRKKGEPHHELPPGSTKRALPNNTSSSPQPKKKPLDGEYFTLQIRGRERFEM
FRELNEALELKDAQAGKEPGGSRAHSSHLKSKKGQSTSRHKKLMFKTEGPDSD
```

6. 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 all of the information here, just a summary of it. We print a number of features of the sequence with one example per type, a number of external database references, plus databases that are referred to, and a number of GO entries, along with three examples. Currently, there are 1,509 features, 923 external references, and 173 GO terms just for this protein. Here is a highly abridged version of the output:

```
Total features: 1509
type: CHAIN
location: [0:393]
id: PRO_0000185703
qualifiers:
    Key: note, Value: Cellular tumor antigen p53

type: DNA_BIND
location: [101:292]
qualifiers:

type: REGION
location: [0:320]
qualifiers:
    Key: evidence, Value: ECO:0000269|PubMed:25732823
    Key: note, Value: Interaction with CCAR2
[...]
Cross references:  923
dict_keys(['EMBL', 'CCDS', 'PIR', 'RefSeq', 'PDB',
'PDBsum', 'BMRB', 'SMR', 'BioGRID', 'ComplexPortal',
'CORUM', 'DIP', 'ELM', 'IntAct', 'MINT', 'STRING',
'BindingDB', 'ChEMBL', 'DrugBank', 'MoonDB', 'TCDB',
'GlyGen', 'iPTMnet', 'MetOSite', 'PhosphoSitePlus',
'BioMuta', 'DMDM', 'SWISS-2DPAGE', 'CPTAC', 'EPD',
'jPOST', 'MassIVE', 'MaxQB', 'PaxDb', 'PeptideAtlas',
'PRIDE', 'ProteomicsDB', 'ABCD', 'Antibodypedia',
'CPTC', 'DNASU', 'Ensembl', 'GeneID', 'KEGG', 'MANE-
Select', 'UCSC', 'CTD', 'DisGeNET', 'GeneCards',
'GeneReviews', 'HGNC', 'HPA', 'MalaCards', 'MIM',
'neXtProt', 'OpenTargets', 'Orphanet', 'PharmGKB',
'VEuPathDB', 'eggNOG', 'GeneTree', 'InParanoid', 'OMA',
'OrthoDB', 'PhylomeDB', 'TreeFam', 'PathwayCommons',
'Reactome', 'SABIO-RK', 'SignaLink', 'SIGNOR', 'BioGRID-
ORCS', 'ChiTaRS', 'EvolutionaryTrace', 'GeneWiki',
'GenomeRNAi', 'Pharos', 'PRO', 'Proteomes', 'RNAct',
'Bgee', 'ExpressionAtlas', 'Genevisible', 'GO', 'CDD',
'DisProt', 'Gene3D', 'IDEAL', 'InterPro', 'PANTHER',
'Pfam', 'PRINTS', 'SUPFAM', 'PROSITE'])
Annotation SOURCES: 173
('GO:0005813', 'C:centrosome', 'IDA:UniProtKB')
('GO:0036310', 'F:ATP-dependent DNA/DNA annealing
activity', 'IDA:UniProtKB')
('GO:0006914', 'P:autophagy', 'IMP:CAFA')
```

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`.

Introducing Bio.PDB

Here, we will introduce Biopython's `PDB` module for working with the PDB. 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 already be aware of the basic PDB data model of model, chain, residue, and atom objects. A good explanation of *Biopython's Structural Bioinformatics FAQ* can be found at http://biopython.org/wiki/The_Biopython_Structural_Bioinformatics_FAQ.

You can find this content in the Chapter08/PDB.py Notebook file.

Of the three models that we will download, the 1TUP model is the one that will be used in the remainder of the 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:

1. 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.

2. 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.

3. 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 the indentation for ease of reading and annotate lists of elements with the `->` prefix. For an example of recursive functions, refer to the previous chapter, *Chapter 7*, *Phylogenetics*. For an advanced discussion on recursion in Python, go to the last chapter, *Chapter 12*, *Functional Programming for Bioinformatics*. The abridged output is as follows:

```
                name: tumor suppressor p53 complexed with
dna
                head: antitumor protein/dna
              idcode: 1TUP
     deposition_date: 1995-07-11
        release_date: 1995-07-11
    structure_method: x-ray diffraction
          resolution: 2.2
 structure_reference:
                   -> n.p.pavletich,k.a.chambers,c.o.pabo
the dna-binding domain of p53 contains the four conserved
regions and the major mutation hot spots genes dev. v. 7
2556 1993 issn 0890-9369
              author: Y.Cho,S.
Gorina,P.D.Jeffrey,N.P.Pavletich
            compound:
                       2:
                        misc:
```

```
                     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')
                        chain: f
                   engineered: yes
has_missing_residues: True
    missing_residues:
                    -> {'model': None, 'res_name': 'ARG',
'chain': 'A', 'ssseq': 290, 'insertion': None}
keywords: antigen p53, antitumor protein/dna complex
              journal: AUTH   Y.CHO,S.
GORINA,P.D.JEFFREY,N.P.PAVLETICHTITL   CRYSTAL
STRUCTURE OF A P53 TUMOR SUPPRESSOR-DNATITL 2 COMPLEX:
UNDERSTANDING TUMORIGENIC MUTATIONS.REF    SCIENCE57
```

4. 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 the 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 such as `grep` will easily extract this information.

Note that for the `1TUP` model, chains `A`, `B`, and `C` are from the protein, while chains `E` and `F` are from the DNA. This information will be useful in the future.

5. 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.
```

6. 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 in each of the protein chains.

7. 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 the atoms in 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 the model, chain, residue, and finally, the atom.

8. 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 `Chapter08/Stats.py` Notebook file.

How to do it...

We'll get started, using the following steps:

1. 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') p53_1tup = parser.get_structure('P 53',
'pdb1tup.ent')
```

2. 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, the number of atoms per chain, and the 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 to (it adds up to the number of atoms, not residues).

Notice the water (`W`), nucleotide (`DA`, `DC`, `DG`, and `DT`), and zinc (`ZN`) residues, which add to the amino acid ones.

3. 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}
```

4. 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 that you are interested in. In this case, we will print boundaries for all the chains and the whole model. Numbers don't convey it so intuitively, so we will get a little bit more graphical.

5. To get 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 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 three-dimensional 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 of the following graph:

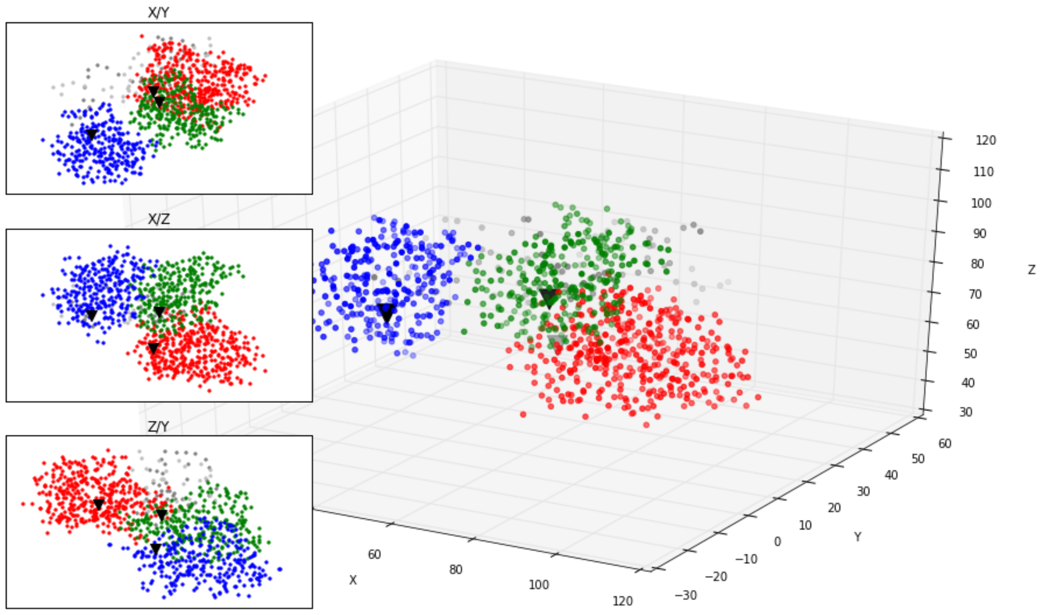


Figure 8.2 - The spatial distribution of the protein chains – the main figure is a 3D plot and the left subplots are planar views (X/Y, X/Z, and Z/Y)

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 `Chapter08/Distance.py` Notebook file.

How to do it...

Take a look at the following steps:

1. 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')
```

2. We will now get our zincs, against which we will perform comparisons later:

```
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.

3. 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.

4. We now compute the atoms near our zincs, the distance of which can be 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 significantly reduced. However, imagine that we had more, or that we were doing a pairwise comparison among all the 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, while more comparisons take a lot of time. We will get back to this soon.

5. 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]
```

6. As we have seen previously, this specific case is not very expensive, but let's time it anyway:

```
import timeit
nexecs = 10
print(timeit.timeit('get_closest_atoms(p53_1tup, zns[0], 
4.0)',
      'from __main__ import get_closest_atoms, p53_1tup, 
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.

7. 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 the 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.

8. Now, let's time against it:

```
print(timeit.timeit('get_closest_alternative(p53_1tup,
zns[0], 4.0)',
      'from __main__ import get_closest_alternative,
p53_1tup, zns',
      number=nexecs) / nexecs * 1000)
```

On the same machine that we used in the preceding example, it takes 16 milliseconds, which means that it is roughly three times faster.

9. However, is this 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_1tup,
zns[0], distance)',
          'from __main__ import get_closest_atoms,
p53_1tup, zns, distance',
          number=nexecs) / nexecs * 1000)
print('Optimized')
for distance in [1, 4, 16, 64, 128]:
    print(timeit.timeit('get_closest_
alternative(p53_1tup, zns[0], distance)',
          'from __main__ import get_closest_alternative,
p53_1tup, 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 depending on the distance of the closest atoms; the larger the distance, the more cases that 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 whole models.

Getting ready

You can find this content in the `Chapter08/Mass.py` Notebook file.

How to do it...

Let's take a look at the following steps:

1. 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')
```

2. 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)
```

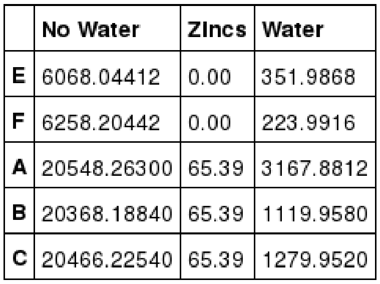
So, we have `H_ ZN` (zinc) and `W` (water), which are `HETATM` types; the vast majority are standard PDB atoms.

3. 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)
```

The `get_mass` function returns the mass of all atoms in the list that pass an acceptance criterion function. Here, the default acceptance criterion involves not being a water residue.

We then compute the mass for all chains. We have three versions: just amino acids, zincs, and water. 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

Figure 8.3 - The mass for all protein chains

4. Let's compute the geometric center and the 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.

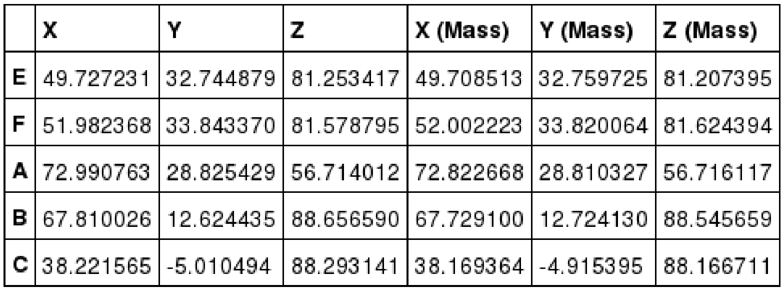
We then compute the geometric center and the center of mass by redefining the `weight` function with a value of each atom equal to its mass. The geometric center is computed, irrespective of its molecular weights.

For example, you may want to compute the center of mass of the protein without DNA chains.

5. 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)
```

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

Figure 8.4 - The center of mass and the geometric center of each protein chain

There's more

Although this is not a book based on the protein structure determination technique, it's important to 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/protexpl/help_hyd.htm` for more information.

Animating with PyMOL

Here, we will create a video of the p53 1TUP model. For that, we will use the PyMOL visualization library. We will start our animation by moving around the p53 1TUP model and then zooming in; as we zoom in, we change the rendering strategy so that you can see deeper into the model. You can find a version of the video that you will generate at https://odysee.com/@Python:8/protein_video:8.

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, or Linux, you can use the `apt-get install pymol` command. If you are on Conda, I suggest not using it, as the dependencies will be easy to resolve – furthermore, you will be installing a 30-day-trial-only version requiring a license, whereas the version above is fully open source. If you are not on Debian or Linux, I suggest that you install the open source version available for your operating system.

PyMOL is more of an interactive program than a Python library, so I strongly encourage you to 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 `Chapter08`. We will use the `PyMol_Movie.py` file in this recipe.

How to do it...

Take a look at the following steps:

1. 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 for understanding what is going on. Here, we will enable all of the models for viewing purposes, hiding all (because the default view is of lines and this is not good enough), then making the zincs visible as spheres.

At this stage, bar zinc, everything else is invisible.

2. 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 corresponds to us moving around the protein (surface-based, thus opaque) and the other corresponds to us diving 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 be rendered as sticks, and we add a mesh to chain A so that the relationship with the DNA becomes clearer.

3. 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 `0` 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.

4. In the first 150 frames, we move around using the initial scene. We move around the model a bit and then move nearer to 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,
-20.000000000))
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,
-20.000000000))
cmd.mview()
```

We define three points; the first two align with the DNA and the last point 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:

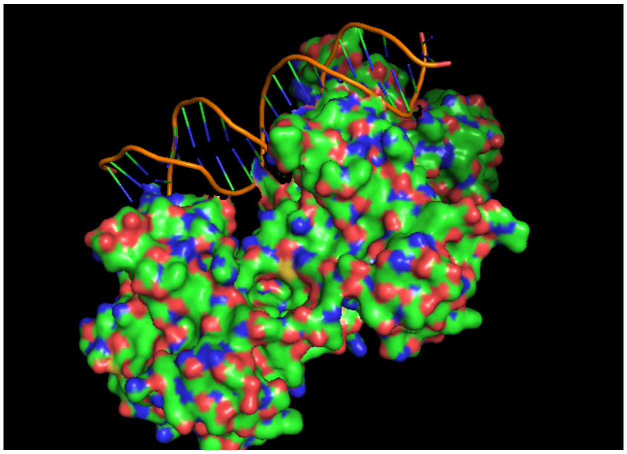


Figure 8.5 - Frame 0 and scene DS0

5. We now change the scene, in preparation for going inside the protein:

```
cmd.frame(200)
cmd.scene('S1')
cmd.mview()
```

The following screenshot shows the current position:

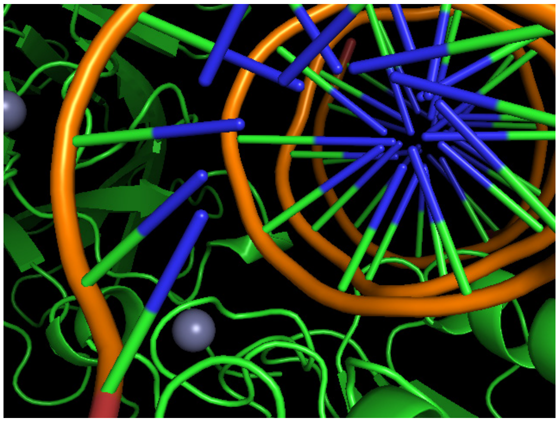


Figure 8.6 - Frame 200 near the DNA molecule and scene S1

6. 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:

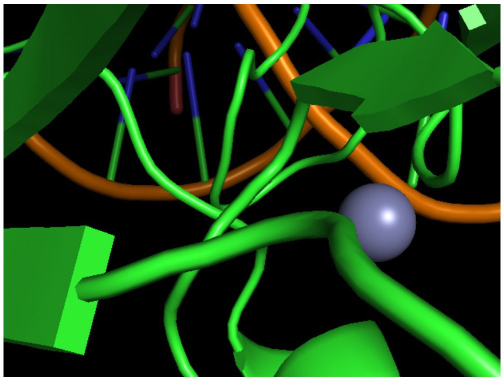


Figure 8.7 - Frame 350 – scene S1 on the verge of changing to S2

7. Finally, 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):

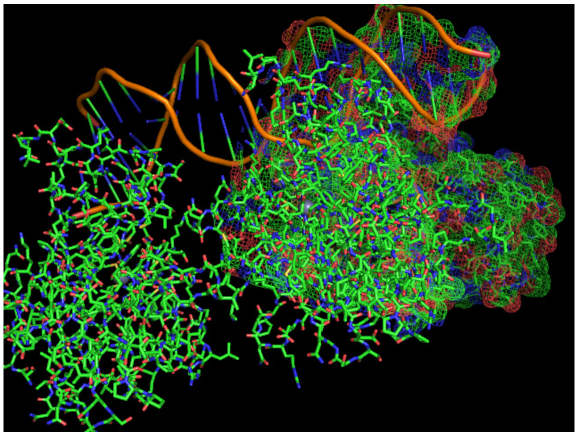


Figure 8.8 - Frame 450 and scene S2

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 that you can use to generate videos from images. PyMOL can generate a MPEG, but it requires the installation of 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` in the GitHub repository; you will see that the second PNG call will start before the first one has finished. Try the script code, 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 currently exists.

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-third-Edition/blob/master/Datasets.py` under `1TUP.cif`.

You can find this content in the `Chapter08/mmCIF.py` Notebook file.

How to do it...

Take a look at the following steps:

1. 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')
```

2. 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.
```

3. Many of the fields are not available in 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 sense of it, but it 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. We 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`.

9
Bioinformatics Pipelines

Pipelines are fundamental within 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 several fundamental viewpoints, chiefly reproducibility, maintainability, and extensibility.

In bioinformatics, you can find three main types of pipeline system:

- Frameworks such as Galaxy (`https://usegalaxy.org`), which are geared toward users, that is, they expose easy-to-use user interfaces and hide most of the underlying machinery.
- Programmatic workflows – geared toward code interfaces that, while generic, originate from the bioinformatics space. Two examples are Snakemake (`https://snakemake.readthedocs.io/`) and Nextflow (`https://www.nextflow.io/`).
- Totally generic workflow systems such as Apache Airflow (`https://airflow.incubator.apache.org/`), which take a less data-centric approach to workflow management.

In this chapter, we will discuss Galaxy, which is especially important for bioinformaticians supporting users that are less inclined to code their own solutions. While you may not be a typical user of these pipeline systems, you might still have to support them. Fortunately, Galaxy provides APIs, which will be our main focus.

We will also be discussing Snakemake and Nextflow as generic workflow tools with programmatic interfaces that originated in the bioinformatics space. We will cover both tools, as they are the most common in the field. We will solve a similar bioinformatics problem using both Snakemake and Nextflow. We will get a taste of both frameworks and hopefully be able to decide on a favorite.

The code for these recipes is presented not as notebooks, but as Python scripts available in the `Chapter09` directory of the book's repository.

In this chapter, you will find recipes for the following:

- Introducing Galaxy servers
- Accessing Galaxy using the API

- Developing a variant analysis pipeline with Snakemake
- Developing a variant analysis pipeline with Nextflow

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 you start, you are strongly advised to approach Galaxy as a user. You can do this by creating a free account at `http://usegalaxy.org`, and playing around with it a bit. Reaching a level of understanding that includes knowledge of the 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. The level of complexity will vary 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 may also be able to use existing public servers. Note that the interfaces of public servers can change over time, so what works today may 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:

1. First, we pull the Galaxy Docker image with the following command:

   ```
   docker pull bgruening/galaxy-stable:20.09
   ```

   This will pull Björn Grüning's amazing Docker Galaxy image. Use the `20.09` label, as shown in the preceding command; anything more recent could break this recipe and the next recipe.
2. Create a directory on your system. This directory will hold the persistent output of the Docker container across runs.

Note

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.

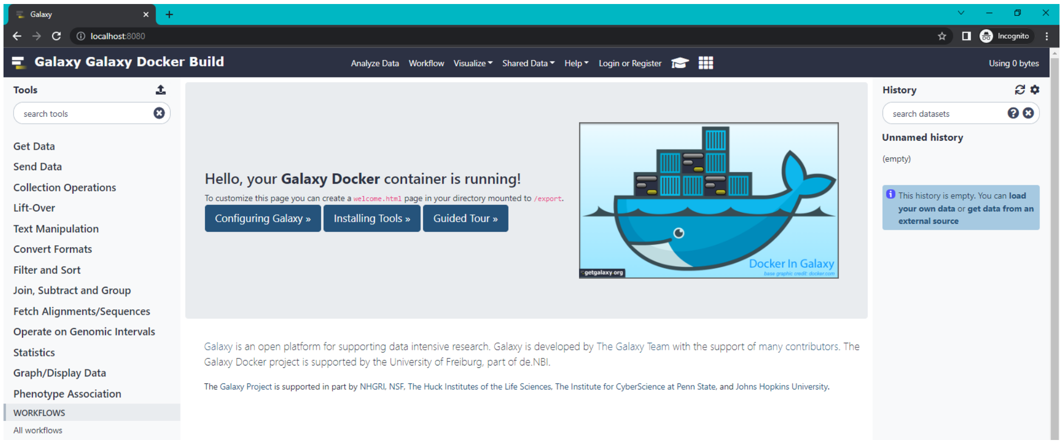
3. We can now run the image with the following command:

```
docker run -d -v YOUR_DIRECTORY:/export -p 8080:80 -p
8021:21 bgruening/galaxy-stable:20.09
```

   Replace `YOUR_DIRECTORY` with the full path to the directory that you created in *step 2*. If the preceding command fails, make sure you have permission to run Docker. This will vary across operating systems.

4. 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, and workflows.

   Point your browser to `http://localhost:8080`. If you get any errors, wait a few seconds. You should see the following screen:



Hello, your **Galaxy Docker** container is running!

To customize this page you can create a `welcome.html` page in your directory mounted to `/export`.

Galaxy is an open platform for supporting data intensive research. Galaxy is developed by The Galaxy Team with the support of many contributors. The Galaxy Docker project is supported by the University of Freiburg, part of de.NBI.

The Galaxy Project is supported in part by NHGRI, NSF, The Huck Institutes of the Life Sciences, The Institute for CyberScience at Penn State, and Johns Hopkins University.

This history is empty. You can **load your own data** or **get data from an external source**

Figure 9.1 - The Galaxy Docker home page

5. Now log in (see the top bar) with the default username and password combination: `admin` and `password`.
6. From the top menu, choose **User**, and inside, choose **Preferences**.
7. 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: `fakekey`. 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: the user as `admin`, password as `password`, and API key as `fakekey`. 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 book on system administration or DevOps, but a programming one. If you visit `https://github.com/bgruening/docker-galaxy-stable`, you will see that there is an infinite number 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 previous instructions for accessing 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 using Galaxy's FTP server.

Getting ready

If you did not go through the previous recipe, please read the corresponding *There's more...* section. The code was tested in a local server, as prepared in the preceding recipe, so 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 regard to it. Our script will be configured via a YAML file, for example:

```
rest_protocol: http
server: localhost
rest_port: 8080
sftp_port: 8022
user: admin
```

```
password: password
api_key: fakekey
```

Our script will not accept this file as plain text, 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), 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.

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 at `Chapter09/pipelines/galaxy/encrypt.py` in the GitHub repository.

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 `ruaml.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 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 in `Chapter09/pipelines/galaxy/api.py`:

1. We start by decrypting our configuration file. We need to supply a password:

```
import base64
from collections import defaultdict
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
Import paramiko

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.

2. 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'
```

3. Finally, we are able to connect to the Galaxy server:

```
gi = GalaxyInstance(url=rest_url, key=api_key)
gi.verify = False
```

4. We 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.

5. Afterward, we create the `bioinf_example` history :

```
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.

6. We are going to upload the file now; this requires an SFTP connection. The file is supplied with this code:

```
print('Uploading file')
transport = paramiko.Transport((server, sftp_port))
transport.connect(None, user, password)
sftp = paramiko.SFTPClient.from_transport(transport)
sftp.put('LCT.bed', 'LCT.bed')
sftp.close()
transport.close()
```

7. 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'])
```

8. Let's summarize the contents of our history:

```
def summarize_contents(contents):
 summary = defaultdict(list)
 for item in contents:
 summary['íd'].append(item['id'])
 summary['híd'].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:

```
                 íd  híd     name  type extension
0  f2db41e1fa331b3e    1  LCT.bed  file      auto
```

9. 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 consists of the following:

```
{'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',
 'visible': True}
```

10. 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()
```

This will print a long list of tools.

11. 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()
```

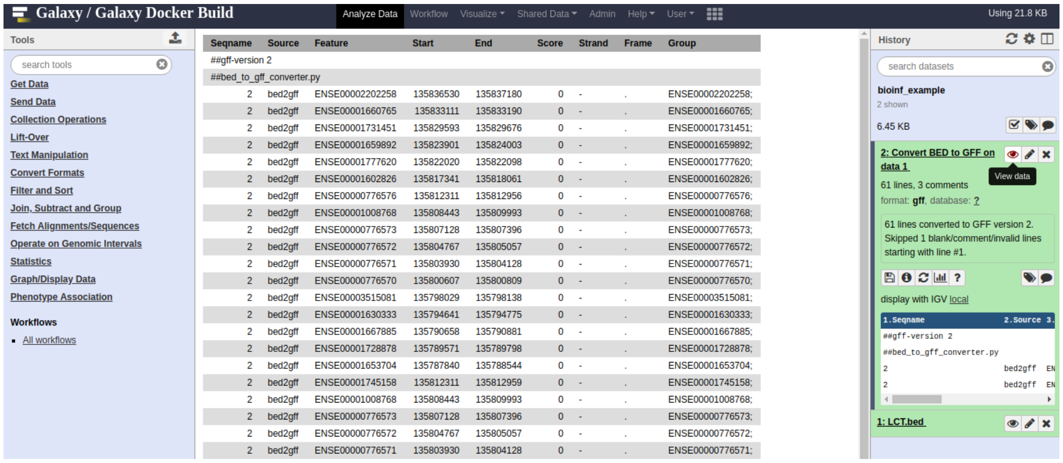
The tool's name was available in the preceding step. Note that we get the first element of a list as, in theory, there could be more than one version of the tool installed. The abridged output is as follows:

```
{'config_file': '/galaxy-central/lib/galaxy/datatypes/
converters/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'}
```

12. 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 similar to the following:



Seqname	Source	Feature	Start	End	Score	Strand	Frame	Group
##gff-version 2								
##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	ENSE00001659892	135823901	135824003	0	-	.	ENSE00001659892;
2	bed2gff	ENSE00001777620	135822020	135822098	0	-	.	ENSE00001777620;
2	bed2gff	ENSE00001602826	135817341	135818061	0	-	.	ENSE00001602826;
2	bed2gff	ENSE00000776576	135812311	135812956	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	135800607	135800809	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	135812959	0	-	.	ENSE00001745158;
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;

bioinf_example
2 shown
6.45 KB

2: Convert BED to GFF on data 1
61 lines, 3 comments
format: gff, database: ?

61 lines converted to GFF version 2. Skipped 1 blank/comment/invalid lines starting with line #1.

display with IGV local

```
1.Seqname                 2.Source 3.
##gff-version 2
##bed_to_gff_converter.py
2                         bed2gff  EN
2                         bed2gff  EN
```

1: LCT.bed

Figure 9.2 - Checking the results of our script via Galaxy's web interface

Thus, we have accessed Galaxy using its REST API.

Deploying a variant analysis pipeline with Snakemake

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. In this chapter, we explore two widely used programmer-friendly pipelines: `snakemake` and Nextflow. In this recipe, we consider `snakemake`.

Snakemake is implemented in Python and shares many traits with it. That being said, its fundamental inspiration is a Makefile, the framework used by the venerable `make`-building system.

Here, we will develop a mini variant analysis pipeline with `snakemake`. The objective here is not to get the scientific part right – we cover that in other chapters – but to see how to create pipelines with `snakemake`. Our mini pipeline will download HapMap data, subsample it at 1%, do a simple PCA, and draw it.

Getting ready

You will need Plink 2 installed alongside `snakemake`. To display execution strategies, you will also need Graphviz to draw the execution. We will define the following tasks:

1. Downloading data
2. Uncompressing it

3. Sub-sampling it at 1%
4. Computing the PCA on the 1% sub-sample
5. Charting the PCA

Our pipeline recipe will have two parts: the actual coding of the pipeline in `snakemake` and a support script in Python.

The `snakemake` code for this can be found in `Chapter09/snakemake/Snakefile`, whereas the Python support script is in `Chapter09/snakemake/plot_pca.py`.

How to do it...

1. The first task is downloading the data:

```
from snakemake.remote.HTTP import RemoteProvider as
HTTPRemoteProvider
HTTP = HTTPRemoteProvider()
download_root = "https://ftp.ncbi.nlm.nih.gov/hapmap/
genotypes/hapmap3_r3"

remote_hapmap_map = f"{download_root}/plink_format/
hapmap3_r3_b36_fwd.consensus.qc.poly.map.gz"
remote_hapmap_ped = f"{download_root}/plink_format/
hapmap3_r3_b36_fwd.consensus.qc.poly.ped.gz"
remote_hapmap_rel = f"{download_root}/relationships_w_
pops_041510.txt"

rule plink_download:
    input:
        map=HTTP.remote(remote_hapmap_map, keep_
local=True),
        ped=HTTP.remote(remote_hapmap_ped, keep_
local=True),
        rel=HTTP.remote(remote_hapmap_rel, keep_
local=True)

    output:
        map="scratch/hapmap.map.gz",
        ped="scratch/hapmap.ped.gz",
        rel="data/relationships.txt"
    shell:
        "mv {input.map} {output.map};"
        "mv {input.ped} {output.ped};"
        "mv {input.rel} {output.rel}"
```

Snakemake's language is Python-dependent, as you can see from the very first lines, which should be easy to understand from a Python perspective. The fundamental part is the rule. It has a set of input streams, which are rendered via `HTTP.remote` in our case, as we are dealing with remote files, followed by the output. We put two files in a `scratch` directory (the ones that are still uncompressed) and one in the `data` directory. Finally, our pipeline code is a simple shell script that moves the downloaded HTTP files to their final location. Note how the shell script refers to inputs and outputs.

2. With this script, downloading the files is easy. Run the following on the command line:

```
snakemake -c1 data/relationships.txt
```

This tells `snakemake` that you want to materialize `data/relationships.txt`. We will be using a single core, `-c1`. As this is an output of the `plink_download` rule, the rule will then be run (unless the file is already available – in that case, `snakemake` will do nothing). Here is an abridged version of the output:

```
Building DAG of jobs...
Using shell: /usr/bin/bash
Provided cores: 1 (use --cores to define parallelism)
Rules claiming more threads will be scaled down.
Job stats:
job                count    min threads    max threads
--------------  -------  -------------  -------------
plink_download        1              1              1
total                 1              1              1

Select jobs to execute...

[Mon Jun 13 18:54:26 2022]
rule plink_download:
    input: ftp.ncbi.nlm.nih.gov/hapmap/ge [...]
    output: [..], data/relationships.txt
    jobid: 0
    reason: Missing output files: data/relationships.txt
    resources: tmpdir=/tmp

Downloading from remote: [...]relationships_w_
pops_041510.txt
Finished download.
[...]
Finished job 0.
1 of 1 steps (100%) done
```

Snakemake gives you some information about which jobs will be executed and starts running those.

3. Now that we have the data, let's see the rule for uncompressing it:

```
PLINKEXTS = ['ped', 'map']

rule uncompress_plink:
    input:
        "scratch/hapmap.{plinkext}.gz"

    output:
        "data/hapmap.{plinkext}"

    shell:
        "gzip -dc {input} > {output}"
```

The most interesting feature here is the way that we can specify multiple files to download. Note how the `PLINKEXTS` list is converted into discrete `plinkext` elements in the code. You can execute by requesting an output from the rule.

4. Now, let's subsample our data to 1%:

```
rule subsample_1p:
    input:
        "data/hapmap.ped",
        "data/hapmap.map"

    output:
        "data/hapmap1.ped",
        "data/hapmap1.map"
    run:
        shell(f"plink2 --pedmap {input[0][:-4]} --out
{output[0][:-4]} --thin 0.01 --geno 0.1 --export ped")
```

The new content is in the last two lines: we are not using `script`, but `run`. This tells `snakemake` that the execution is Python-based with a few extra functions available. Here we see the shell function, which executes a shell script. The string is a Python `f`-string – note the reference to the `snakemake` `input` and `output` variables in the string. You could put more complex Python code here – for example, you could iterate over the inputs.

TIP

Here, we are assuming that Plink is available, as we pre-installed it, but `snakemake` does provide some functionality to deal with dependencies. More specifically, `snakemake` rules can be annotated with a `YAML` file pointing to `conda` dependencies.

5. Now that we have our data sub-sampled, let's compute the PCA. In this case, we will use Plink's internal PCA framework to do the computation:

```
rule plink_pca:
    input:
        "data/hapmap1.ped",
        "data/hapmap1.map"

    output:
        "data/hapmap1.eigenvec",
        "data/hapmap1.eigenval"

    shell:
        "plink2 --pca --file data/hapmap1 -out data/
hapmap1"
```

6. As with most pipeline systems, `snakemake` constructs a **Directed Acyclic Graph** (**DAG**) of operations to execute. At any moment, you can ask `snakemake` to present you a DAG of what you will execute to generate your request. For example, to generate the PCA, use the following:

```
snakemake --dag data/hapmap1.eigenvec | dot -Tsvg > bio.
svg
```

This would generate the following figure:

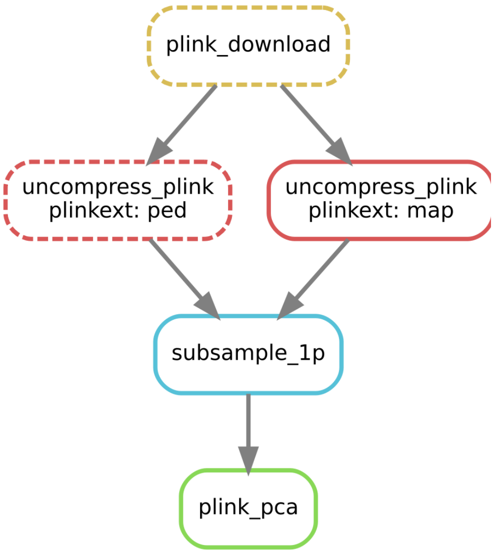


Figure 9.3 - The DAG to compute the PCA

7. Finally, let's generate the `plot` rule for the PCA:

```
rule plot_pca:
    input:
        "data/hapmap1.eigenvec",
        "data/hapmap1.eigenval"

    output:
        "pca.png"

    script:
        "./plot_pca.py"
```

   The `plot` rule introduces a new type of execution, `script`. In this case, an external Python script is called to process the rule.

8. Our Python script to generate the chart is the following:

```
import pandas as pd

eigen_fname = snakemake.input[0] if snakemake.input[0].
endswith('eigenvec') else snakemake.input[1]
```

```
pca_df = pd.read_csv(eigen_fname, sep='\t')
ax = pca_df.plot.scatter(x=2, y=3, figsize=(16, 9))
ax.figure.savefig(snakemake.output[0])
```

The Python script has access to the `snakemake` object. This object exposes the content of the rule: note how we make use of `input` to get the PCA data and `output` to generate the image.

9. Finally, the code to produce a rough chart is as follows:

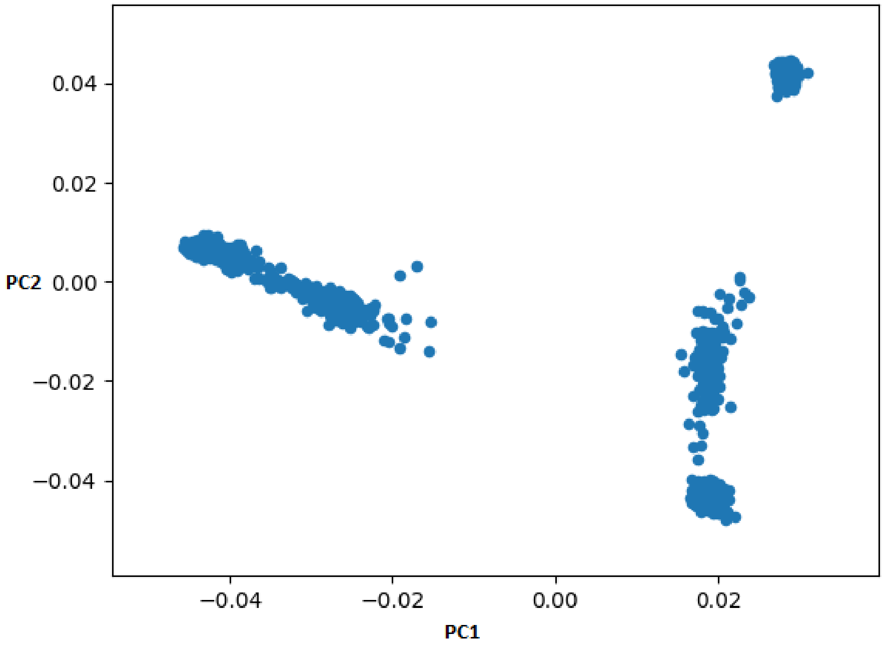


Figure 9.4 - A very rough PCA produced by the Snakemake pipeline

There's more

The preceding recipe was made to run on a simple configuration of `snakemake`. There are many more ways to construct rules in `snakemake`.

The most important issue that we didn't discuss is the fact that `snakemake` can execute code in many different environments, from the local computer (as in our case), on-premises clusters, to the cloud. It would be unreasonable to ask for anything more than using a local computer to try `snakemake`, but don't forget that `snakemake` can manage complex computing environments.

Remember that `snakemake`, while implemented in Python, is conceptually based on `make`. It's a subjective analysis to decide whether you like the (snake)make design. For an alternative design approach, check the next recipe, which uses Nextflow.

Deploying a variant analysis pipeline with Nextflow

There are two main players in the pipeline framework space in bioinformatics: `snakemake` and Nextflow. They provide pipeline functionality whilst having different design approaches. Snakemake is based on Python, but its language and philosophy come from the `make` tool used to compile complex programs with dependencies. Nextflow is Java-based (more precisely, it's implemented in Groovy – a language that works on top of the Java Virtual Machine) and has its own **Domain Specific Language** (**DSL**) for implementing pipelines. The main purpose of this recipe (and the previous recipe) is to give you a flavor of Nextflow so that you can compare it with `snakemake` and choose the one that better suits your needs.

TIP

There are many perspectives on how to evaluate a pipeline system. Here, we present a perspective based on the language used to specify the pipeline. However, there are others that you should consider when choosing a pipeline system. For example, how well does it support your execution environment (such as an on-premises cluster or a cloud), does it support your tools (or allow easy development of extensions to deal with new tools), and does it provide good recovery and monitoring functionalities?

Here, we will develop a pipeline with Nextflow that provides the same functionality as we implemented with `snakemake`, thus allowing for a fair comparison from the pipeline design point of view. The objective here is not to get the scientific part right – we cover that in other chapters – but to see how to create pipelines with `snakemake`. Our mini pipeline will download HapMap data, sub-sample it at 1%, do a simple PCA, and draw it.

Getting ready

You will need Plink 2 installed along with Nextflow. Nextflow itself requires some software from the Java space: notably the Java Runtime Environment and Groovy.

We will define the following tasks:

1. Downloading data
2. Uncompressing it
3. Sub-sampling it at 1%
4. Computing the PCA on the 1% sub-sample
5. Charting the PCA

The Nextflow code for this can be found in `Chapter09/nextflow/pipeline.nf`.

How to do it...

1. The first task is downloading the data:

```
nextflow.enable.dsl=2

download_root = "https://ftp.ncbi.nlm.nih.gov/hapmap/
genotypes/hapmap3_r3"

 process plink_download {
  output:
  path 'hapmap.map.gz'
  path 'hapmap.ped.gz'

  script:
  """
  wget $download_root/plink_format/hapmap3_r3_b36_fwd.
consensus.qc.poly.map.gz -O hapmap.map.gz
  wget $download_root/plink_format/hapmap3_r3_b36_fwd.
consensus.qc.poly.ped.gz -O hapmap.ped.gz
   """
}
```

Remember that the underlying language for the pipeline is not Python but Groovy, so the syntax will be a bit different, such as using braces for blocks or ignoring indentation.

We create a process (a pipeline building block in Nextflow) called `plink_download`, which downloads the Plink files. It only specifies outputs. The first output will be the `hapmap.map.gz` file and the second output will be `hapmap.ped.gz`. This process will have two output channels (another Nextflow concept, akin to a stream), which can be consumed by another process.

The code for the process is, by default, a bash script. It's important to note how the script outputs files with names that are synchronized with the output section. Also, see how we refer to the variables defined in the pipeline (`download_root`, in our case).

2. Let's now define a process to consume the channels with the HapMap files and decompress them:

```
process uncompress_plink {
  publishDir 'data', glob: '*', mode: 'copy'

  input:
  path mapgz
```

```
    path pedgz

    output:
    path 'hapmap.map'
    path 'hapmap.ped'

    script:
    """
    gzip -dc $mapgz > hapmap.map
    gzip -dc $pedgz > hapmap.ped
    """
}
```

There are three issues of note in this process: we now have a couple of inputs (remember that we have a couple of outputs from the previous process). Our script also now refers to input variables (`$mapgz` and `$pedgz`). Finally, we publish the output by using `publishDir`. Therefore, any files that are not published will only be stored temporarily.

3. Let's specify a first version of the workflow that downloads and uncompresses the files:

```
workflow {
    plink_download | uncompress_plink
}
```

4. We can execute the workflow by running the following on the shell:

```
nextflow run pipeline.nf -resume
```

The `resume` flag at the end will make sure that the pipeline will continue from whatever step was already completed. The steps are, on local execution, stored in the `work` directory.

5. If we remove the `work` directory, we don't want to download the HapMap files if they were already published. As this is outside the `work` directory, hence not directly tracked, we need to change the workflow to track the data in the published directory:

```
workflow {
    ped_file = file('data/hapmap.ped')
    map_file = file('data/hapmap.map')
    if (!ped_file.exists() | !map_file.exists()) {
        plink_download | uncompress_plink
    }
}
```

There are alternative ways of doing this, but I wanted to introduce a bit of Groovy code, as you might sometimes have to write code in Groovy. There are ways to use Python code, as you will see soon.

6. Now, we need to subsample the data:

```
process subsample_1p {
  input:
  path 'hapmap.map'
  path 'hapmap.ped'

  output:
  path 'hapmap1.map'
  path 'hapmap1.ped'

  script:
  """
  plink2 --pedmap hapmap --out hapmap1 --thin 0.01 --geno
0.1 --export ped
  """
}
```

7. Let's now compute the PCA using Plink:

```
process plink_pca {
  input:
  path 'hapmap.map'
  path 'hapmap.ped'
  output:
  path 'hapmap.eigenvec'
  path 'hapmap.eigenval'
   script:
  """
  plink2 --pca --pedmap hapmap -out hapmap
  """
}
```

8. Finally, let's plot the PCA:

```
process plot_pca {
  publishDir '.', glob: '*', mode: 'copy'

  input:
  path 'hapmap.eigenvec'
  path 'hapmap.eigenval'

  output:
  path 'pca.png'

  script:
  """
  #!/usr/bin/env python
  import pandas as pd

  pca_df = pd.read_csv('hapmap.eigenvec', sep='\t')
  ax = pca_df.plot.scatter(x=2, y=3, figsize=(16, 9))
  ax.figure.savefig('pca.png')
  """
}
```

The new feature of this code is that we specify the bash script using the shebang (`#!`) operator, which allows us to call an external scripting language to process the data.

Here is our final workflow:

```
workflow {
    ped_file = file('data/hapmap.ped')
    map_file = file('data/hapmap.map')
    if (!ped_file.exists() | !map_file.exists()) {
        plink_download | uncompress_plink | subsample_1p
| plink_pca | plot_pca
    }
    else {
        subsample_1p(
            Channel.fromPath('data/hapmap.map'),
            Channel.fromPath('data/hapmap.ped')) | plink_
pca | plot_pca
    }
}
```

We either download the data or use the already downloaded data.

While there are other dialects for designing the complete workflow, I would like you to note how we use `subsample_1p` when the files are available; we can explicitly pass two channels to a process.

9. We can run the pipeline and request an HTML report on execution:

```
nextflow run pipeline.nf -with-report report/report.html
```

The report is quite exhaustive and will allow you to figure out the parts of the pipeline that are expensive from different perspectives, whether related to time, memory, CPU consumption, or I/O.

There's more

This was a simple introductory example of Nextflow, which hopefully will allow you to get a flavor of the framework, particularly so that you can compare it with `snakemake`. Nextflow has many more functionalities and you are encouraged to check out its website.

As with `snakemake`, the most important issue that we didn't discuss is the fact that Nextflow can execute code in many different environments, from the local computer, on-premises clusters, to the cloud. Check Nextflow's documentation to see which computing environments are currently supported.

As important as the underlying language is, Groovy with Nextflow and Python with `snakemake`, make sure to compare other factors. This includes not only where both pipelines can execute (locally, in a cluster, or in a cloud) but also the design of the framework, as they use quite different paradigms.

10
Machine Learning for Bioinformatics

Machine learning is used in a wide variety of contexts and computational biology is not an exception. Machine learning has countless applications in the field, probably the oldest and most known being the use of **Principal Component Analysis** (**PCA**) to study population structure using genomics. There are many other potential applications as this is a burgeoning field. In this chapter, we are going to introduce machine learning concepts from a bioinformatics perspective.

Given that machine learning is a very complex topic that could easily fill a book, here we intend to take an intuitive approach that will allow you to broadly understand how some machine learning techniques can be useful to tackle biological problems. If you find these techniques useful, you will understand the fundamental concepts and can proceed to more detailed literature.

If you are using Docker, and because all the libraries in this chapter are fundamental for data analysis, they all can be found on the Docker image `tiagoantao/bioinformatics_ml`.

In this chapter, we will cover the following recipes:

- Introducing scikit-learn with a PCA example
- Using clustering over PCA to classify samples
- Exploring breast cancer traits using Decision Trees
- Predicting breast cancer outcomes using Random Forests

Introducing scikit-learn with a PCA example

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. In *Chapter 6*, we saw a PCA implementation based on using an external application. In this recipe, we will implement the same PCA for population genetics but will use the `scikit-learn` library. Scikit-learn is one of the fundamental Python libraries for machine learning and this recipe is an introduction to the library. PCA is a form of unsupervised machine learning – we don't provide information about the class of the sample. We will discuss supervised techniques in the other recipes of this chapter.

As a reminder, we will compute PCA for 11 human populations from the HapMap project.

Getting ready

You will need to run the first recipe from *Chapter 6* in order to generate the `hapmap10_auto_noofs_ld_12` PLINK file (with alleles recorded as 1 and 2). From a population genetics perspective, we require LD-pruned markers to produce a reliable PCA. We will not risk using the offspring here because it would probably bias the result. Our recipe will require the `pygenomics` library, which can be installed using the following command:

```
pip install pygenomics
```

The code is in the `Chapter10/PCA.py` notebook.

How to do it...

Take a look at the following steps:

1. We start by loading the metadata for our samples. In our case, we will be loading the human population that each sample belongs to:

```
import os
from sklearn.decomposition import PCA
import numpy as np
from genomics.popgen.pca import plot

f = open('../Chapter06/relationships_w_pops_041510.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()
```

2. We now get the order of individuals along with the number of SNPs that we will be processing:

```
f = open('../Chapter06/hapmap10_auto_noofs_ld_12.ped')
ninds = 0
ind_order = []
for line in f:
    ninds += 1
    toks = line[:100].replace(' ', '\t').split('\t')
    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()
```

3. We create the array that will be fed to the PCA:

```
pca_array = np.empty((ninds, nsnps), dtype=int)
print(pca_array.shape)
f = open('../Chapter06/hapmap10_auto_noofs_ld_12.ped')
for ind, line in enumerate(f):
    snps = line.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()
```

4. Finally, we compute the PCA with up to eight components. We then get the 8-D coordinates for all samples using a `transform` method.

```
my_pca = PCA(n_components=8)
my_pca.fit(pca_array)
trans = my_pca.transform(pca_array)
```

5. Finally, we plot the PCA:

```
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)
```

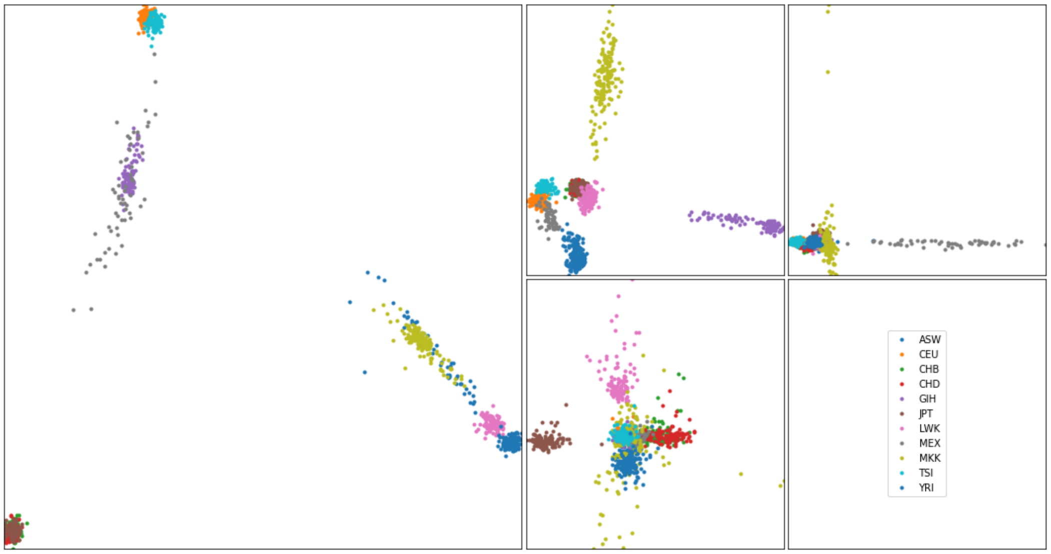


Figure 10.1 - PC1 to PC8 for our dataset as produced by scikit-learn

There's more...

For publication in scientific journals, I would recommend using the recipe in *Chapter 6*, simply because it's based on a published and highly regarded method. That being said, the results from this code are qualitatively similar and cluster data in a very similar fashion (the inversion of direction on the vertical axis, if you compare it with the figure in *Chapter 6*, is irrelevant when interpreting a PCA chart).

Using clustering over PCA to classify samples

PCA in genomics allows us to see how samples cluster. In many cases, individuals from the same population will be in the same area of the chart. But we would like to go further and predict where new individuals fall in terms of populations. To do that, we will start with PCA data, as it does dimensionality reduction – making working with the data easier – and then apply a K-Means clustering algorithm to predict where new samples fall. We will use the same dataset as in the recipe above. We will use all our samples save one to train the algorithm, and then we will predict where the remaining sample falls.

K-Means clustering can be an example of a supervised algorithm. In these types of algorithms, we need a training dataset so that the algorithm is able to learn. After training the algorithm, it will be able to predict a certain outcome for new samples. In our case, we are hoping that we can predict the population.

WARNING

The current recipe intends to serve as a gentle introduction to supervised algorithms and the concepts behind them. The way we are training the algorithm is far from optimal. The issue of properly training a supervised algorithm will be alluded to in the last recipe of this chapter.

Getting ready

We will be using the same data as in the previous recipe. The code for this recipe can be found in `Chapter10/Clustering.py`.

How to do it...

Let's have a look:

1. We start by loading the population information – this is similar to what we did in the previous recipe:

```
import os
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
import numpy as np
from genomics.popgen.pca import plot

f = open('../Chapter06/relationships_w_pops_041510.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()
```

```
f = open('../Chapter06/hapmap10_auto_noofs_ld_12.ped')
ninds = 0
ind_order = []
for line in f:
    ninds += 1
    toks = line[: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
print (nsnps)
f.close()
```

2. We now load all sample data – SNPs – into a NumPy array:

```
all_array = np.empty((ninds, nsnps), dtype=int)
f = open('../Chapter06/hapmap10_auto_noofs_ld_12.ped')
for ind, line in enumerate(f):
    snps = line.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
        all_array[ind, pos] = my_code
f.close()
```

3. We separate the array into two datasets, namely, a training case with all individuals except one, and a case to test with a single individual:

```
predict_case = all_array[-1, :]
pca_array = all_array[:-1,:]

last_ind = ind_order[-1]
last_ind, ind_pop[last_ind]
```

Our test case is individual Y076/NA19124, who we know belongs to the Yoruban population.

4. We now compute the PCA for the training set that we will use for K-Means clustering:

```
my_pca = PCA(n_components=2)
my_pca.fit(pca_array)
trans = my_pca.transform(pca_array)

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

Here is the output, which will be useful to check clustering results:

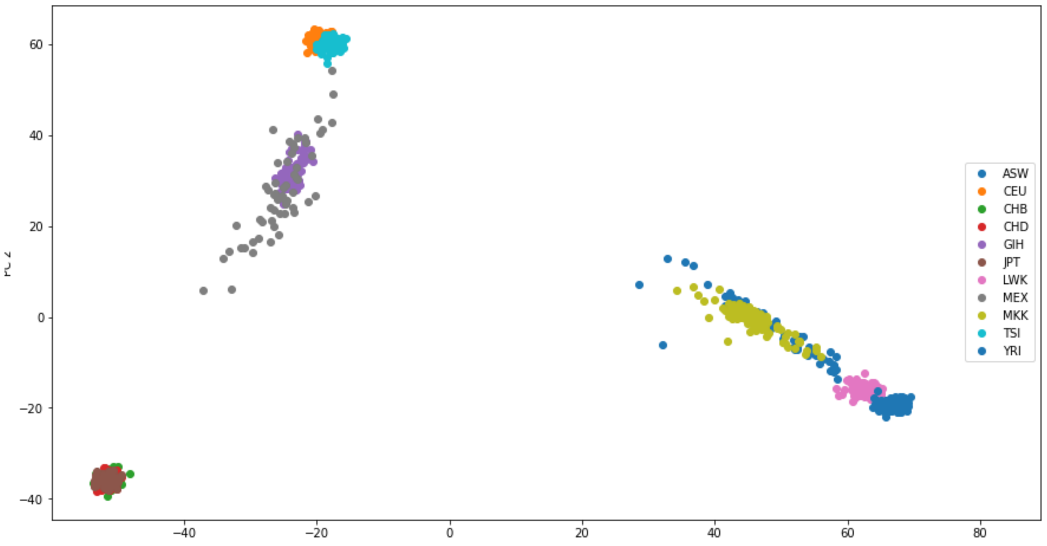


Figure 10.2 - PC1 and PC2 with populations color-coded

5. Before we start computing K-means clustering, let's write a function to plot the clustering surface from running the algorithm:

```
def plot_kmeans_pca(trans, kmeans):
    x_min, x_max = trans[:, 0].min() - 1, trans[:,
0].max() + 1
    y_min, y_max = trans[:, 1].min() - 1, trans[:,
1].max() + 1
    mesh_x, mesh_y = np.meshgrid(np.arange(x_min, x_max,
0.5), np.arange(y_min, y_max, 0.5))
```

```
    k_surface = kmeans.predict(np.c_[mesh_x.ravel(),
mesh_y.ravel()]).reshape(mesh_x.shape)
    fig, ax = plt.subplots(1,1, dpi=300)
    ax.imshow(
        k_surface, origin="lower", cmap=plt.cm.Pastel1,
        extent=(mesh_x.min(), mesh_x.max(), mesh_y.min(),
mesh_y.max()),
    )
    ax.plot(trans[:, 0], trans[:, 1], "k.", markersize=2)
    ax.set_title("KMeans clustering of PCA data")
    ax.set_xlim(x_min, x_max)
    ax.set_ylim(y_min, y_max)
    ax.set_xticks(())
    ax.set_yticks(())
    return ax
```

6. Let's now fit the algorithm with our samples. Because we have 11 populations, we will train for 11 clusters:

```
kmeans11 = KMeans(n_clusters=11).fit(trans)
plot_kmeans_pca(trans, kmeans11)
```

Here is the output:

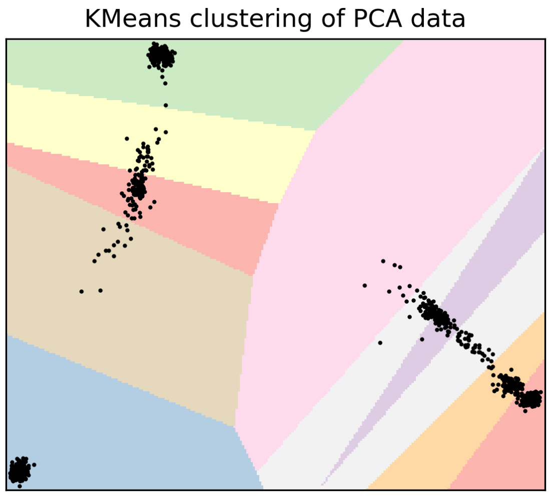


Figure 10.3 - The cluster surface for 11 clusters

If you compare with the figure here, you can intuitively see that the clustering makes little sense: it doesn't map to the known populations very well. One could argue that this clustering algorithm with 11 clusters is not very useful.

TIP

There are many other clustering algorithms implemented in scikit-learn, and in several scenarios, they might perform better than K-means. You can find them at `https://scikit-learn.org/stable/modules/clustering.html`. It is doubtful that, in this specific case, any alternative would perform much better for 11 clusters.

7. While it seems K-means clustering cannot resolve the 11 populations, maybe it can still provide some predictions if we use a different number of clusters. Simply by looking at the chart, we see four separate blocks. What would be the result if we used four clusters?

```
kmeans4 = KMeans(n_clusters=4).fit(trans)
plot_kmeans_pca(trans, kmeans4)
```

Here is the output:

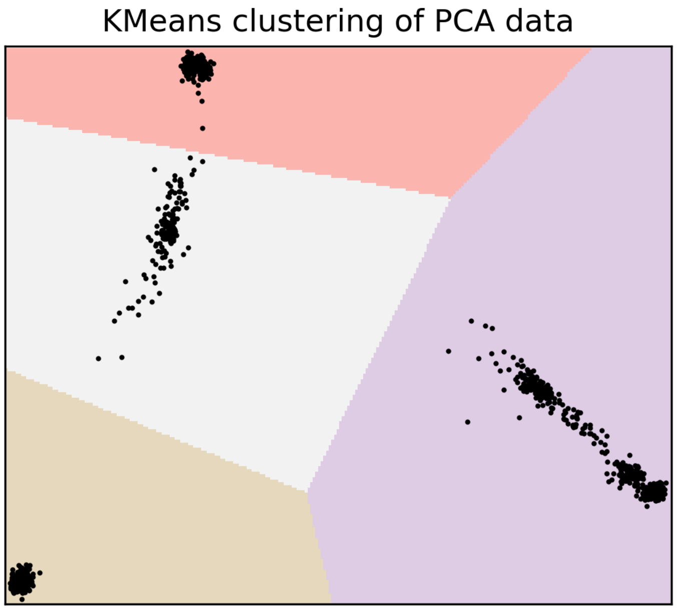


Figure 10.4 - The cluster surface for four clusters

The four groups are now mostly clear. But do they make intuitive sense? If they do, we can make use of this clustering approach. And in fact, they do. The cluster on the left is composed of African populations, the top cluster European ones, and the bottom one East Asians. The middle one is the most cryptic as it contains both Gujarati and Mexican descendants, but that mix comes originally from PCA and it is not caused by clustering itself.

8. Let's see how prediction does for the single case we left out:

```
pca_predict = my_pca.transform([predict_case])
kmeans4.predict(pca_predict)
```

   Our sample is predicted to be in cluster 1. We need to dig a little deeper now.

9. Let's find out what cluster 1 means. We take the last individual from the training set, who is also a Yoruba, and see to which cluster he is allocated:

```
last_train = ind_order[-2]
last_train, ind_pop[last_train]
kmeans4.predict(trans)[0]
```

It is indeed cluster 1, so the prediction is correct.

There's more...

It is worth reiterating that we are trying to achieve an intuitive understanding of machine learning. At this stage, you should have a grasp of what you can gain from supervised learning, and also example usage of a clustering algorithm. There is much more to be said about the procedure to train a machine learning algorithm, something that we will partially unveil in the last recipe.

Exploring breast cancer traits using Decision Trees

One of the first problems that we have when we receive a dataset is deciding what to start analyzing. At the very beginning, there is quite often a feeling of loss about what to do first. Here, we will present an exploratory approach based on Decision Trees. The big advantage of Decision Trees is that they will give us the rules that constructed the decision tree, allowing us a first tentative understanding of what is going on with our data.

In this example, we will be using a dataset with trait observations from patients with breast cancer. The dataset with 699 data entries includes information such as clump thickness, uniformity of cell size, or type of chromatin. The outcome is either a benign or malignant tumor. The features are encoded with values from 0 to 10. More information about the project can be found at `http://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+%28diagnostic%29`.

Getting ready

We are going to download the data along with the documentation:

```
wget http://archive.ics.uci.edu/ml/machine-learning-databases/
breast-cancer-wisconsin/breast-cancer-wisconsin.data
wget http://archive.ics.uci.edu/ml/machine-learning-databases/
breast-cancer-wisconsin/breast-cancer-wisconsin.names
```

The data file is formatted as a CSV file. Information about the content can be found in the second downloaded file.

The code for this recipe can be found in `Chapter10/Decision_Tree.py`.

How to do it...

Follow these steps:

1. The first thing we do is to remove a small fraction of individuals that have incomplete data:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn import tree
f = open('breast-cancer-wisconsin.data')
w = open('clean.data', 'w')
for line in f:
    if line.find('?') > -1:
        continue
    w.write(line)
f.close()
w.close()
```

TIP

Removing individuals with incomplete data is adequate in this case because they are a small fraction of the dataset, and we are only doing exploratory analysis. For cases with lots of missingness or when we are trying to do something more rigorous, you will have to use methods to deal with missing data, which we will not explore here.

2. We are now going to read the data, giving names to all columns:

```
column_names = [
    'sample_id', 'clump_thickness', 'uniformity_cell_
size',
    'uniformity_cell shape', 'marginal_adhesion',
    'single_epithelial_cell_size', 'bare_nuclei',
    'bland_chromatin', 'normal_nucleoli', 'mitoses',
    'class'
]
samples = pd.read_csv('clean.data', header=None,
names=column_names, index_col=0)
```

3. We will now separate the features from the outcome and recode the outcome using 0 and 1:

```
training_input = samples.iloc[:,:-1]
target = samples.iloc[:,-1].apply(lambda x: 0 if x == 2
else 1)
```

4. Let's now create a Decision Tree based on this data with a max depth of 3:

```
clf = tree.DecisionTreeClassifier(max_depth=3)
clf.fit(training_input, target)
```

5. Let's start by seeing which features are the most important:

```
importances = pd.Series(
    clf.feature_importances_ * 100,
    index=training_input.columns).sort_
values(ascending=False)
importances
```

The following are the features ranked by importance:

```
uniformity_cell_size          83.972870
uniformity_cell shape          7.592903
bare_nuclei                    4.310045
clump_thickness                4.124183
marginal_adhesion              0.000000
single_epithelial_cell_size    0.000000
bland_chromatin                0.000000
normal_nucleoli                0.000000
mitoses                        0.000000
```

Remember that this is just exploratory analysis. In the next recipe, we will try to produce more reliable rankings. The reason why the bottom features are zero is we asked for a max depth of 3 and in that case, it is possible that not all features are used.

6. We can do some native analysis of the accuracy of our implementation:

```
100 * clf.score(training_input, target)
```

We get a performance of 96%. We shouldn't be testing the algorithm with its own training set as this is quite circular. We will revisit that in the next recipe.

7. Finally, let's plot the decision tree:

```
fig, ax = plt.subplots(1, dpi=300)
tree.plot_tree(clf,ax=ax, feature_names=training_input.
columns, class_names=['Benign', 'Malignant'])
```

This results in the following output:

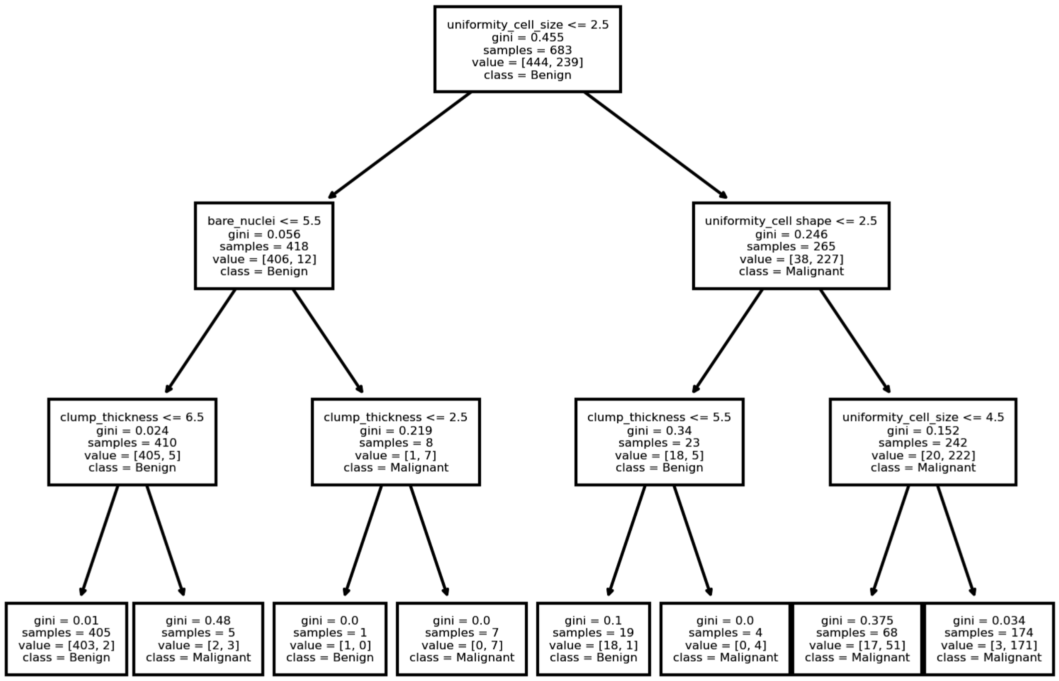


Figure 10.5 - The decision tree for the breast cancer dataset

Let's look at the root node to start: it has a criterion of `uniformity_cell_size < 2.5` and a classification of benign. The main feature of splitting the tree is the uniformity of the cell size. The classification of benign at the top node comes simply from the fact that most samples on the dataset are benign. Now look at the right node from the root: it has 265 samples, most of which are malignant and with criteria of `uniformity_cell_shape < 2.5`.

These rules allow you to have an initial understanding of what might be driving the dataset. Decision Trees are not very precise, so take these as your initial step.

Predicting breast cancer outcomes using Random Forests

We are now going to predict the outcomes for some patients using Random Forests. A random forest is an ensemble method (it will use several instances of other machine learning algorithms) that uses many decision trees to arrive at robust conclusions about the data. We are going to use the same example as in the previous recipe: breast cancer traits and outcomes.

This recipe has two main goals: to introduce you to random forests and issues regarding the training of machine learning algorithms.

Getting ready

The code for this recipe can be found in `Chapter10/Random_Forest.py`.

How to do it...

Take a look at the code:

1. We start, as in the previous recipe, by getting rid of samples with missing information:

```
import pandas as pd
import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.tree import export_graphviz
f = open('breast-cancer-wisconsin.data')
w = open('clean.data', 'w')
for line in f:
    if line.find('?') > -1:
        continue
    w.write(line)
f.close()
w.close()
```

2. We now load the cleaned data:

```
column_names = [
    'sample_id', 'clump_thickness', 'uniformity_cell_
size',
    'uniformity_cell shape', 'marginal_adhesion',
    'single_epithelial_cell_size', 'bare_nuclei',
    'bland_chromatin', 'normal_nucleoli', 'mitoses',
    'class'
]
samples = pd.read_csv('clean.data', header=None,
names=column_names, index_col=0)
samples
```

3. We separate the data read in features and outcomes:

```
training_input = samples.iloc[:, :-1]
target = samples.iloc[:, -1]
```

4. We create a classifier and fit the data to it:

```
clf = RandomForestClassifier(max_depth=3, n_
estimators=200)
clf.fit(training_input, target)
```

The most important parameter here is `n_estimators`: we are requesting the Forest be constructed with 200 trees.

5. We now rank the features in order of importance:

```
importances = pd.Series(
    clf.feature_importances_ * 100,
    index=training_input.columns).sort_
values(ascending=False)
importances
```

Here is the output:

```
uniformity_cell_size           30.422515
uniformity_cell shape          21.522259
bare_nuclei                    18.410346
single_epithelial_cell_size    10.959655
```

```
bland_chromatin                 9.600714
clump_thickness                 3.619585
normal_nucleoli                 3.549669
marginal_adhesion               1.721133
mitoses                         0.194124
```

The result is non-deterministic, meaning that you might have different results. Also, note that the Random Forest has quite different numbers than the Decision Tree of the previous recipe. This is to be expected as the Decision Tree is a single estimator where the Forest weighs 200 trees and is more reliable.

6. We can score this case:

```
clf.score(training_input, target)
```

I get a result of 97.95%. You might get a slightly different value as the algorithm is stochastic. As we said in the previous recipe, getting a score from the training set is quite circular and far from best practice.

7. In order to have a more realistic view of the accuracy of the algorithm, we need to separate our data into two parts – a training set and a test set:

```
for test_size in [0.01, 0.1, 0.2, 0.5, 0.8, 0.9, 0.99]:
    X_train, X_test, y_train, y_test = train_test_split(
        trainning_input, target, test_size=test_size)
    tclf = RandomForestClassifier(max_depth=3)
    tclf.fit(X_train, y_train)
    score = tclf.score(X_test, y_test)
    print(f'{1 - test_size:.1%} {score:.2%}')
```

The output is the following (remember that you will get different values):

```
99.0% 71.43%
90.0% 94.20%
80.0% 97.81%
50.0% 97.66%
20.0% 96.89%
10.0% 94.80%
1.0% 92.02%
```

If you only train with 1% of the data, you only get 71% accuracy, whereas if you train with more, the accuracy goes above 90%. Note that accuracy does not monotonically increase with the size of the training set. Deciding on the size of the training set is a complex affair with various issues causing unexpected side effects.

There's more...

We only scratched the surface of training and testing machine learning algorithms. For example, supervised datasets are normally split into 3, not 2 (training, test, and cross-validation). There are many more issues that you need to consider in order to train your algorithm and many more types of algorithms. In this chapter, we tried to develop basic intuition to understand machine learning, but this is nothing more than your starting point if you intend to follow this route.

11
Parallel Processing with Dask and Zarr

Bioinformatics datasets are growing at an exponential rate. Data analysis strategies based on standard tools such as Pandas assume that datasets are able to fit in memory (though with some provision for out-of-core analysis) or that a single machine is able to efficiently process all the data. This is, unfortunately, not realistic for many modern datasets.

In this chapter, we will introduce two libraries that are able to deal with very large datasets and expensive computations:

- Dask is a library that allows parallel computing that can scale from a single computer to very large cloud and cluster environments. Dask provides interfaces that are similar to Pandas and NumPy while allowing you to deal with large datasets spread over many computers.
- Zarr is a library that stores compressed and chunked multidimensional arrays. As we will see, these arrays are tailored to deal with very large datasets processed in large computer clusters, while still being able to process data on a single computer if need be.

Our recipes will introduce these advanced libraries using data from mosquito genomics. You should look at this code as a starting point to get you on the path to processing large datasets. Parallel processing of large datasets is a complex topic, and this is the beginning—not the end—of your journey.

Because all these libraries are fundamental for data analysis, if you are using Docker, they all can be found on the `tiagoantao/bioinformatics_dask` Docker image.

In this chapter, we will cover the following recipes:

- Reading genomics data with Zarr
- Parallel processing of data using Python multiprocessing
- Using Dask to process genomic data based on NumPy arrays
- Scheduling tasks with `dask.distributed`

Reading genomics data with Zarr

Zarr (`https://zarr.readthedocs.io/en/stable/`) stores array-based data—such as NumPy —in a hierarchical structure on disk and cloud storage. The data structures used by Zarr to represent arrays are not only very compact but also allow for parallel reading and writing, something we will see in the next recipes. In this recipe, we will be reading and processing genomics data from the Anopheles gambiae 1000 Genomes project (`https://malariagen.github.io/vector-data/ag3/download.html`). Here, we will simply do sequential processing to ease the introduction to Zarr; in the following recipe, we will do parallel processing. Our project will be computing the missingness for all genomic positions sequenced for a single chromosome.

Getting ready

The Anopheles 1000 Genomes data is available from **Google Cloud Platform** (**GCP**). To download data from GCP, you will need `gsutil`, available from `https://cloud.google.com/storage/docs/gsutil_install`. After you have `gsutil` installed, download the data (~2 **gigabytes** (**GB**)) with the following lines of code:

```
mkdir -p data/AG1000G-AO/
gsutil -m rsync -r \
         -x '.*/calldata/(AD|GQ|MQ)/.*' \
         gs://vo_agam_release/v3/snp_genotypes/all/AG1000G-AO/
\
         data/AG1000G-AO/ > /dev/null
```

We download a subset of samples from the project. After downloading the data, the code to process it can be found in `Chapter11/Zarr_Intro.py`.

How to do it...

Take a look at the following steps to get started:

1. Let's start by checking the structure made available inside the Zarr file:

```
import numpy as np
import zarr
mosquito = zarr.open('data/AG1000G-AO')
print(mosquito.tree())
```

We start by opening the Zarr file (as we will soon see, this might not actually be a file). After that, we print the tree of data available inside it:

```
/
├── 2L
│   └── calldata
│       └── GT (48525747, 81, 2) int8
├── 2R
│   └── calldata
│       └── GT (60132453, 81, 2) int8
├── 3L
│   └── calldata
│       └── GT (40758473, 81, 2) int8
├── 3R
│   └── calldata
│       └── GT (52226568, 81, 2) int8
├── X
│   └── calldata
│       └── GT (23385349, 81, 2) int8
└── samples (81,) |S24
```

The Zarr file has five arrays: four correspond to chromosomes in the mosquito—`2L`, `2R`, `3L`, `3R`, and `X` (`Y` is not included)—and one has a list of 81 samples included in the file. The last array has the sample names included—we have 81 samples in this file. The chromosome data is made of 8-bit integers (`int8`), and the sample names are strings.

2. Now, let's explore the data for chromosome `2L`. Let's start with some basic information:

```
gt_2l = mosquito['/2L/calldata/GT']
gt_2l
```

Here is the output:

```
<zarr.core.Array '/2L/calldata/GT' (48525747, 81, 2)
int8>
```

We have an array of `4852547` **single-nucleotide polymorphisms** (**SNPs**), for `81` samples. For each SNP and sample, we have `2` alleles.

3. Let's now inspect how the data is stored:

```
gt_2l.info
```

The output looks like this:

```
Name                 : /2L/calldata/GT
Type                 : zarr.core.Array
Data type            : int8
Shape                : (48525747, 81, 2)
Chunk shape          : (300000, 50, 2)
Order                : C
Read-only            : False
Compressor           : Blosc(cname='lz4', clevel=5,
shuffle=SHUFFLE, blocksize=0)
Store type           : zarr.storage.DirectoryStore
No. bytes            : 7861171014 (7.3G)
No. bytes stored     : 446881559 (426.2M)
Storage ratio        : 17.6
Chunks initialized : 324/324
```

There is a lot to unpack here, but for now, we will concentrate on the store type, bytes stored, and storage ratio. The `Store type` value is `zarr.storage.DirectoryStore`, so the data is not in a single file but inside a directory. The raw size of the data is `7.3` GB! But Zarr uses a compressed format that reduces the size to `426.2` **megabytes** (**MB**). This means a compression ratio of `17.6`.

4. Let's peek at how the data is stored inside the directory. If you list the contents of the `AG1000G-AO` directory, you will find the following structure:

```
.
├── 2L
│   └── calldata
│       └── GT
├── 2R
│   └── calldata
│       └── GT
├── 3L
│   └── calldata
│       └── GT
├── 3R
│   └── calldata
│       └── GT
├── samples
```

```
└── X
    └── calldata
        └── GT
```

5. If you list the contents of `2L/calldata/GT`, you will find plenty of files encoding the array:

```
0.0.0
0.1.0
1.0.0
...
160.0.0
160.1.0
```

There are 324 files inside the `2L/calldata/GT` directory. Remember from a previous step that we have a parameter called `Chunk shape` with a value of `(300000, 50, 2)`.

Zarr splits the array into chunks—bits that are easier to process in memory than loading the whole array. Each chunk has 30000x50x2 elements. Given that we have 48525747 SNPs, we need 162 chunks to represent the number of SNPs (48525747/300000 = 161.75) and then multiply it by 2 for the number of samples (81 samples/50 per chunk = 1.62). Hence, we end up with 162*2 chunks/files.

Tip

Chunking is a technique widely used to deal with data that cannot be fully loaded into memory in a single pass. This includes many other libraries such as Pandas or Zarr. We will see an example with Zarr later. The larger point is that you should be aware of the concept of chunking as it is applied in many cases requiring big data.

6. Before we load the Zarr data for processing, let's create a function to compute some basic genomic statistics for a chunk. We will compute missingness, the number of ancestral homozygotes, and the number of heterozygotes:

```
def calc_stats(my_chunk):
    num_miss = np.sum(np.equal(my_chunk[:,:,0], -1),
axis=1)
    num_anc_hom = np.sum(
        np.all([
            np.equal(my_chunk[:,:,0], 0),
            np.equal(my_chunk[:,:,0], my_chunk[:,:,1])],
axis=0), axis=1)
    num_het = np.sum(
        np.not_equal(
            my_chunk[:,:,0],
            my_chunk[:,:,1]), axis=1)
    return num_miss, num_anc_hom, num_het
```

If you look at the previous function, you will notice that there is nothing Zarr-related: it's just NumPy code. Zarr has a very light **application programming interface** (**API**) that exposes most of the data inside NumPy, making it quite easy to use if you know NumPy.

7. Finally, let's traverse our data—that is, traverse our chunks to compute our statistics:

```
complete_data = 0
more_anc_hom = 0
total_pos = 0
for chunk_pos in range(ceil(max_pos / chunk_pos_size)):
    start_pos = chunk_pos * chunk_pos_size
    end_pos = min(max_pos + 1, (chunk_pos + 1) * chunk_
pos_size)
    my_chunk = gt_2l[start_pos:end_pos, :, :]
    num_samples = my_chunk.shape[1]
    num_miss, num_anc_hom, num_het = calc_stats(my_chunk)
    chunk_complete_data = np.sum(np.equal(num_miss, 0))
    chunk_more_anc_hom = np.sum(num_anc_hom > num_het)
    complete_data += chunk_complete_data
    more_anc_hom += chunk_more_anc_hom
    total_pos += (end_pos - start_pos)
print(complete_data, more_anc_hom, total_pos)
```

Most of the code takes care of the management of chunks and involves arithmetic to decide which part of the array to access. The important part in terms of ready Zarr data is the `my_chunk = gt_2l[start_pos:end_pos, :, :]` line. When you slice a Zarr array, it will automatically return a NumPy array.

Tip

Be very careful with the amount of data that you bring into memory. Remember that most Zarr arrays will be substantially bigger than the memory that you have available, so if you try to load it, your application and maybe even your computer will crash. For example, if you do `all_data = gt_2l[:, :, :]`, you will need around 8 GB of free memory to load it—as we have seen earlier, the data is 7.3 GB in size.

There's more...

Zarr has many more features than those presented here, and while we will explore some more in the next recipes, there are some possibilities that you should be aware of. For example, Zarr is one of the only libraries that allow for concurrent writing of data. You can also change the internal format of a Zarr representation.

As we have seen here, Zarr is able to compress data in very efficient ways—this is made possible by using the Blosc library (`https://www.blosc.org/`). You can change the internal compression algorithm of Zarr data owing to the flexibility of Blosc.

See also

There are alternative formats to Zarr—for example, **Hierarchical Data Format 5** (**HDF5**) (`https://en.wikipedia.org/wiki/Hierarchical_Data_Format`) and **Network Common Data Form** (**NetCDF**) (`https://en.wikipedia.org/wiki/NetCDF`). While these are more common outside the bioinformatics space, they have less functionality—for example, a lack of concurrent writes.

Parallel processing of data using Python multiprocessing

When dealing with lots of data, one strategy is to process it in parallel so that we make use of all available **central processing unit** (**CPU**) power, given that modern machines have many cores. In a theoretical best-case scenario, if your machine has eight cores, you can get an eight-fold increase in performance if you do parallel processing.

Unfortunately, typical Python code only makes use of a single core. That being said, Python has built-in functionality to use all available CPU power; in fact, Python provides several avenues for that. In this recipe, we will be using the built-in `multiprocessing` module. The solution presented here works well in a single computer and if the dataset fits into memory, but if you want to scale it in a cluster or the cloud, you should consider Dask, which we will introduce in the next two recipes.

Our objective here will again be to compute some statistics around missingness and heterozygosity.

Getting ready

We will be using the same data as in the previous recipe. The code for this recipe can be found in `Chapter11/MP_Intro.py`.

How to do it...

Follow these steps to get started:

1. We will be using the exact same function as in the previous recipe to calculate statistics—this is heavily NumPy-based:

```
import numpy as np
import zarr

def calc_stats(my_chunk):
    num_miss = np.sum(np.equal(my_chunk[:,:,0], -1), 
axis=1)
    num_anc_hom = np.sum(
        np.all([
            np.equal(my_chunk[:,:,0], 0),
            np.equal(my_chunk[:,:,0], my_chunk[:,:,1])], 
axis=0), axis=1)
    num_het = np.sum(
        np.not_equal(
            my_chunk[:,:,0],
            my_chunk[:,:,1]), axis=1)
    return num_miss, num_anc_hom, num_het
```

2. Let's access our mosquito data:

```
mosquito = zarr.open('data/AG1000G-AO')
gt_2l = mosquito['/2L/calldata/GT']
```

3. While we are using the same function to calculate statistics, our approach will be different for the whole dataset. First, we compute all the intervals for which we will call `calc_stats`. The intervals will be devised to match perfectly with the chunk division for variants:

```
chunk_pos_size = gt_2l.chunks[0]
max_pos = gt_2l.shape[0]

intervals = []
for chunk_pos in range(ceil(max_pos / chunk_pos_size)):
    start_pos = chunk_pos * chunk_pos_size
    end_pos = min(max_pos + 1, (chunk_pos + 1) * chunk_
pos_size)
    intervals.append((start_pos, end_pos))
```

It is important that our interval list is related to the chunking on disk. The computation will be efficient as long as this mapping is as close as possible.

4. We are now going to separate the code to compute each interval in a function. This is important as the `multiprocessing` module will execute this function many times on each process that it creates:

```
def compute_interval(interval):
    start_pos, end_pos = interval
    my_chunk = gt_2l[start_pos:end_pos, :, :]
    num_samples = my_chunk.shape[1]
    num_miss, num_anc_hom, num_het = calc_stats(my_chunk)
    chunk_complete_data = np.sum(np.equal(num_miss, 0))
    chunk_more_anc_hom = np.sum(num_anc_hom > num_het)
    return chunk_complete_data, chunk_more_anc_hom
```

5. We are now finally going to execute our code over several cores:

```
with Pool() as p:
    print(p)
    chunk_returns = p.map(compute_interval, intervals)
    complete_data = sum(map(lambda x: x[0], chunk_
returns))
    more_anc_hom = sum(map(lambda x: x[1], chunk_
returns))
    print(complete_data, more_anc_hom)
```

The first line creates a context manager using the `multiprocessing.Pool` object. The `Pool` object, by default, creates several processes numbered `os.cpu_count()`. The pool provides a `map` function that will call our `compute_interval` function across all processes created. Each call will take one of the intervals.

There's more...

This recipe provides a small introduction to parallel processing with Python without the need to use external libraries. That being said, it presents the most important building block for concurrent parallel processing with Python.

Due to the way thread management is implemented in Python, threading is not a viable alternative for real parallel processing. Pure Python code cannot be run in parallel using multithreading.

Some libraries that you might use—and this is normally the case with NumPy—are able to make use of all underlying processors even when executing a sequential piece of code. Make sure that when making use of external libraries, you are not overcommitting processor resources: this happens when you have multiple processes, and underlying libraries also make use of many cores.

See also

- There is way more to be discussed about the `multiprocessing` module. You can start with the standard documentation at `https://docs.python.org/3/library/multiprocessing.html`.
- To understand why Python-based multithreading doesn't make use of all CPU resources, read about the **Global Interpreter Lock** (**GIL**) at `https://realpython.com/python-gil/`.

Using Dask to process genomic data based on NumPy arrays

Dask is a library that provides advanced parallelism that can scale from a single computer to very large clusters or a cloud operation. It also provides the ability to process datasets that are larger than memory. It is able to provide interfaces that are similar to common Python libraries such as NumPy, Pandas, or scikit-learn.

We are going to repeat a subset of the example from previous recipes—namely, compute missingness for the SNPs in our dataset. We will be using an interface similar to NumPy that is offered by Dask.

Before we start, be aware that the semantics of Dask are quite different from libraries such as NumPy or Pandas: it is a lazy library. For example, when you specify a call equivalent to—say—`np.sum`, you are not actually calculating a sum, but adding a task that in the future will eventually calculate it. Let's get into the recipe to make things clearer.

Getting ready

We are going to rechunk the Zarr data in a completely different way. The reason we do that is so that we can visualize task graphs during the preparation of our algorithm. Task graphs with five operations are easier to visualize than task graphs with hundreds of nodes. For practical purposes, you should not rechunk in so little chunks as we do here. In fact, you will be perfectly fine if you don't rechunk this dataset at all. We are only doing it for visualization purposes:

```
import zarr
mosquito = zarr.open('data/AG1000G-AO/2L/calldata/GT')
zarr.array(
    mosquito,
    chunks=(1 + 48525747 // 4, 81, 2),
    store='data/rechunk')
```

We will end up with very large chunks, and while that is good for our visualization purpose, they might be too big to fit in memory.

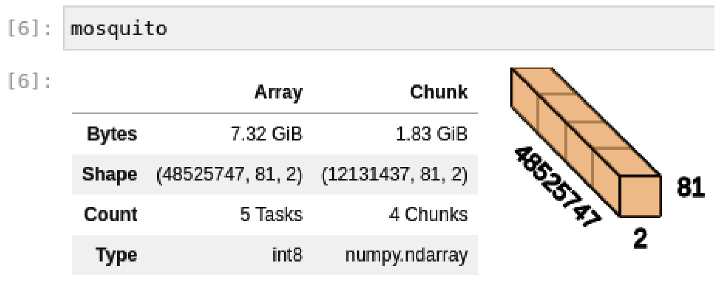
The code for this recipe can be found in `Chapter11/Dask_Intro.py`.

How to do it...

1. Let's first load the data and inspect the size of the DataFrame:

```
import numpy as np
import dask.array as da

mosquito = da.from_zarr('data/rechunk')
mosquito
```

Here is the output if you are executing inside Jupyter:



```
[6]: mosquito
[6]:
```

	Array	Chunk
Bytes	7.32 GiB	1.83 GiB
Shape	(48525747, 81, 2)	(12131437, 81, 2)
Count	5 Tasks	4 Chunks
Type	int8	numpy.ndarray

Figure 11.1 - Jupyter output for a Dask array, summarizing our data

The full array takes up `7.32` GB. The most important number is the chunk size: `1.83` GB. Each worker will need to have enough memory to process a chunk. Remember that we are only using such a smaller number of chunks to be able to plot the tasks here.

Because of the large chunk sizes, we end up with just four chunks.

We did not load anything in memory yet: we just specified that we want to eventually do it. We are creating a task graph to be executed, not executing—at least for now.

2. Let's see which tasks we have to execute to load the data:

```
mosquito.visualize()
```

Here is the output:

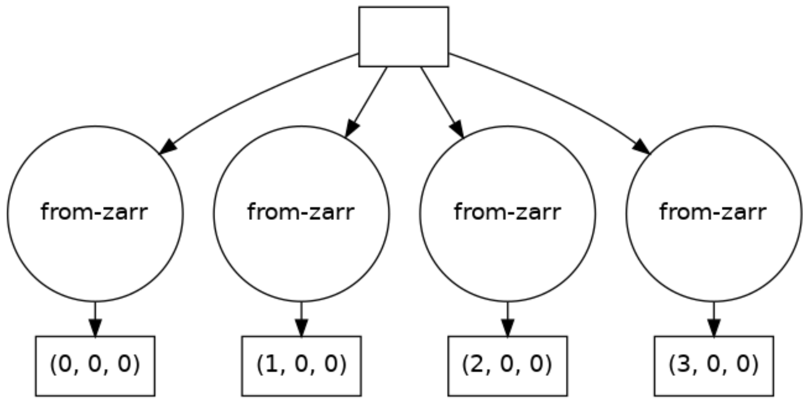


Figure 11.2 - Tasks that need to be executed to load our Zarr array

We thus have four tasks to execute, one for each chunk.

3. Now, let's look at the function to compute missingness per chunk:

```
def calc_stats(variant):
    variant = variant.reshape(variant.shape[0] // 2, 2)
    misses = np.equal(variant, -1)
    return misses
```

The function per chunk will operate on NumPy arrays. Note the difference: the code that we use to work on the main loop works with Dask arrays, but at the chunk level the data is presented as a NumPy array. Hence, the chunks have to fit in memory as NumPy requires that.

4. Later, when we actually use the function, we need to have a **two-dimensional** (**2D**) array. Given that the array is **three-dimensional** (**3D**), we will need to reshape the array:

```
mosquito_2d = mosquito.reshape(
    mosquito.shape[0],
    mosquito.shape[1] * mosquito.shape[2])
mosquito_2d.visualize()
```

Here is the task graph as it currently stands:

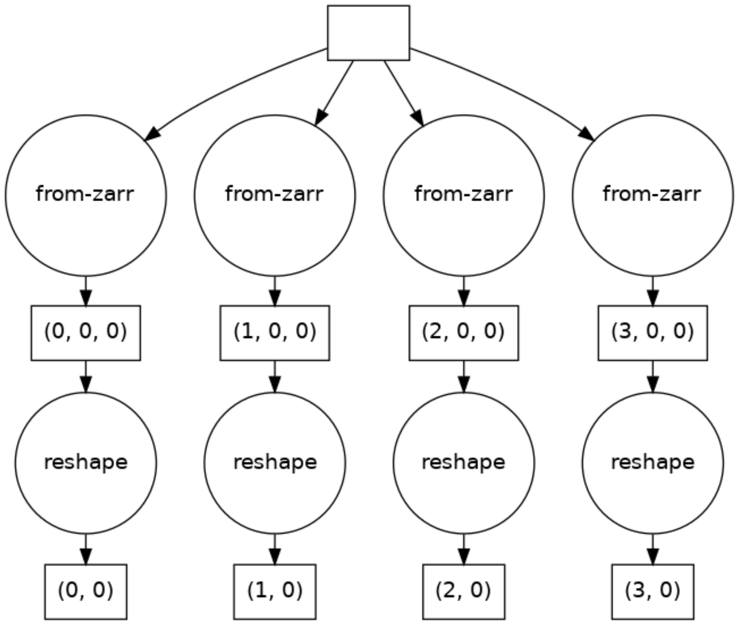


Figure 11.3 - The task graph to load genomic data and reshape it

The `reshape` operation is happening at the `dask.array` level, not at the NumPy level, so it just added nodes to the task graph. There is still no execution.

5. Let's now prepare to execute the function—meaning adding tasks to our task graph—over all our dataset. There are many ways to execute it; here, we are going to use the `apply_along_axis` function that `dask.array` provides and is based on the equally named function from NumPy:

```
max_pos = 10000000
stats = da.apply_along_axis(
    calc_stats, 1, mosquito_2d[:max_pos,:],
    shape=(max_pos,), dtype=np.int64)
stats.visualize()
```

For now, we are only going to study the first million positions. As you can see in the task graph, Dask is smart enough to only add an operation to the chunk involved in the computation:

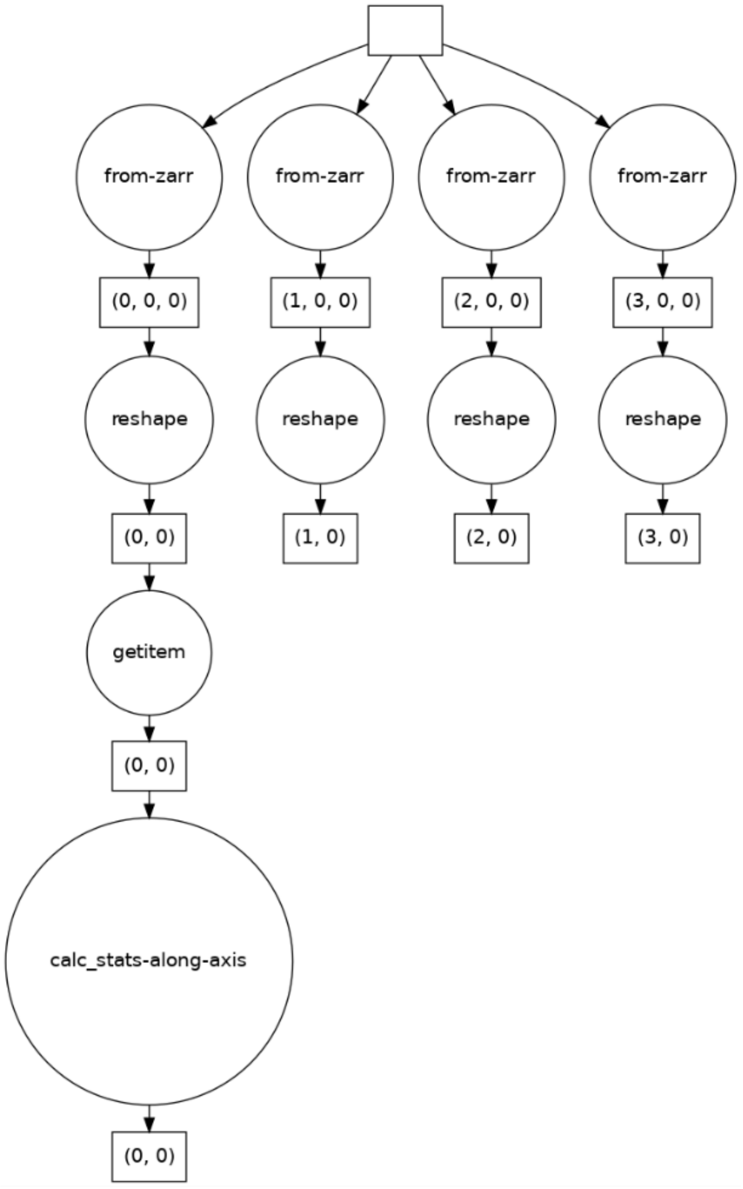


Figure 11.4 - The complete task graph including statistical computing

6. Remember that we haven't computed anything until now. It is now time to actually execute the task graph:

```
stats = stats.compute()
```

This will start the computation. Precisely how the computation is done is something we will discuss in the next recipe.

WARNING

Because of the chunk size, this code might crash your computer. You will be safe with at least 16 GB of memory. Remember that you can use smaller chunk sizes—and you *should use* smaller chunk sizes. We just used chunk sizes like this in order to be able to generate the task graphs shown earlier (if not, they would have possibly hundreds of nodes and would be unprintable).

There's more...

We didn't spend any time here discussing strategies to optimize the code for Dask—that would be a book of its own. For very complex algorithms, you will need to research further into the best practices.

Dask provides interfaces similar to other known Python libraries such as Pandas or scikit-learn that can be used for parallel processing. You can also use it for general algorithms that are not based on existing libraries.

See also

- For best practices with Dask, your best starting point is the Dask documentation itself, especially `https://docs.dask.org/en/latest/best-practices.html`.

Scheduling tasks with dask.distributed

Dask is extremely flexible in terms of execution: we can execute locally, on a scientific cluster, or on the cloud. That flexibility comes at a cost: it needs to be parameterized. There are several alternatives to configure a Dask schedule and execution, but the most generic is `dask.distributed` as it is able to manage different kinds of infrastructure. Because I cannot assume you have access to a cluster or a cloud such as **Amazon Web Services** (**AWS**) or GCP, we will be setting up computation on your local machine, but remember that you can set up `dask.distributed` on very different kinds of platforms.

Here, we will again compute simple statistics over variants of the Anopheles 1000 Genomes project.

Getting ready

Before we start with `dask.distributed`, we should note that Dask has a default scheduler that actually can change depending on the library you are targeting. For example, here is the scheduler for our NumPy example:

```
import dask
from dask.base import get_scheduler
```

```
import dask.array as da

mosquito = da.from_zarr('data/AG1000G-AO/2L/calldata/GT')
print(get_scheduler(collections=[mosquito]).__module__)
```

The output will be as follows:

```
dask.threaded
```

Dask uses a threaded scheduler here. This makes sense for a NumPy array: NumPy implementation is itself multithreaded (real multithreaded with parallelism). We don't want lots of processes running when the underlying library is running in parallel to itself. If you had a Pandas DataFrame, Dask would probably choose a multiprocessor scheduler. As Pandas is not parallel, it makes sense for Dask to run in parallel itself.

OK—now that we have that important detail out of the way, let's get back to preparing our environment.

`dask.distributed` has a centralized scheduler and a set of workers, and we need to start those. Run this code in the command line to start the scheduler:

```
dask-scheduler --port 8786 --dashboard-address 8787
```

We can start workers on the same machine as the scheduler, like this:

```
dask-worker --nprocs 2 –nthreads 1 127.0.0.1:8786
```

I specified two processes with a single thread per process. This is reasonable for NumPy code, but the actual configuration will depend on your workload (and be completely different if you are on a cluster or a cloud).

Tip

You actually do not need to start the whole process manually as I did here. `dask.distributed` will start something for you—not really optimized for your workload—if you don't prepare the system yourself (see the following section for details). But I wanted to give you a flavor of the effort as in many cases, you will have to set up the infrastructure yourself.

Again, we will be using data from the first recipe. Be sure you download and prepare it, as explained in its *Getting ready* section. We won't be using the rechunked part—we will be doing it in our Dask code in the following section. Our code is available in `Chapter11/Dask_distributed.py`.

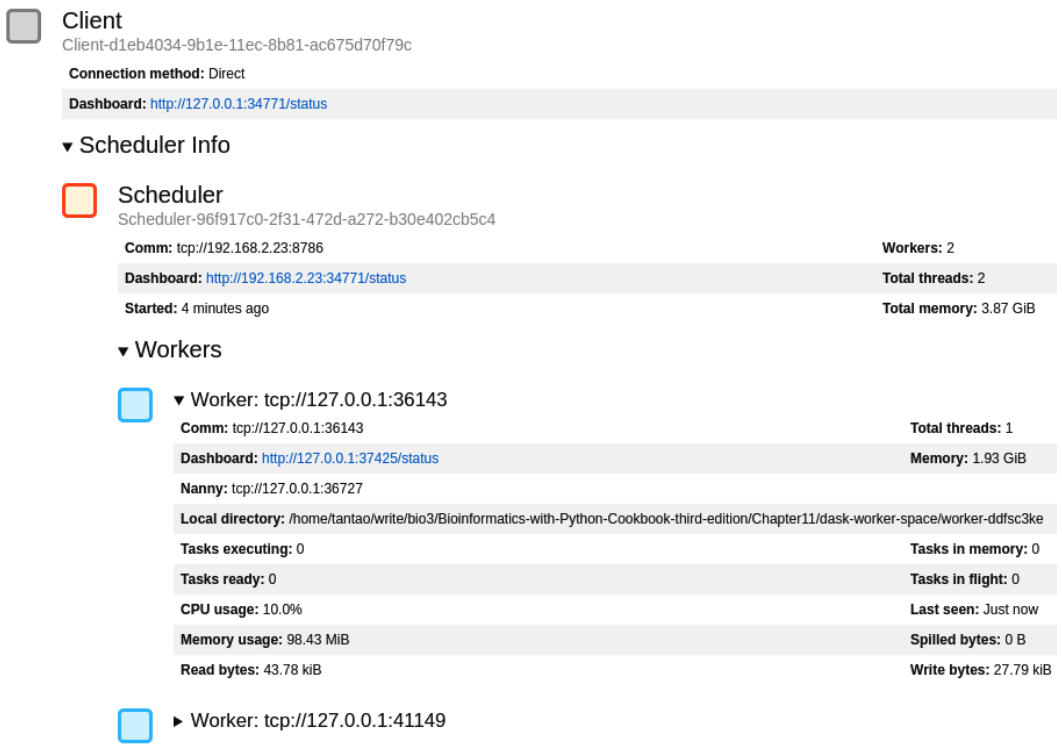
How to do it...

Follow these steps to get started:

1. Let's start by connecting to the scheduler that we created earlier:

```
import numpy as np
import zarr
import dask.array as da
from dask.distributed import Client

client = Client('127.0.0.1:8786')
client
```

If you are on Jupyter, you will get a nice output summarizing the configuration you created in the *Getting ready* part of this recipe:



```
Client
Client-d1eb4034-9b1e-11ec-8b81-ac675d70f79c
Connection method: Direct
Dashboard: http://127.0.0.1:34771/status

▼ Scheduler Info

  Scheduler
  Scheduler-96f917c0-2f31-472d-a272-b30e402cb5c4
  Comm: tcp://192.168.2.23:8786                          Workers: 2
  Dashboard: http://192.168.2.23:34771/status            Total threads: 2
  Started: 4 minutes ago                                 Total memory: 3.87 GiB

  ▼ Workers

    ▼ Worker: tcp://127.0.0.1:36143
      Comm: tcp://127.0.0.1:36143                        Total threads: 1
      Dashboard: http://127.0.0.1:37425/status           Memory: 1.93 GiB
      Nanny: tcp://127.0.0.1:36727
      Local directory: /home/tantao/write/bio3/Bioinformatics-with-Python-Cookbook-third-edition/Chapter11/dask-worker-space/worker-ddfsc3ke
      Tasks executing: 0                                 Tasks in memory: 0
      Tasks ready: 0                                     Tasks in flight: 0
      CPU usage: 10.0%                                   Last seen: Just now
      Memory usage: 98.43 MiB                            Spilled bytes: 0 B
      Read bytes: 43.78 kiB                              Write bytes: 27.79 kiB

    ▶ Worker: tcp://127.0.0.1:41149
```

Figure 11.5 - Summary of your execution environment with dask.distributed

You will notice the reference to a dashboard here. `dask.distributed` provides a real-time dashboard over the web that allows you to track the state of the computation. Point your browser to `http://127.0.0.1:8787/` to find it, or just follow the link provided in *Figure 11.5*.

As we still haven't done any computations, the dashboard is mostly empty. Be sure to explore the many tabs along the top:

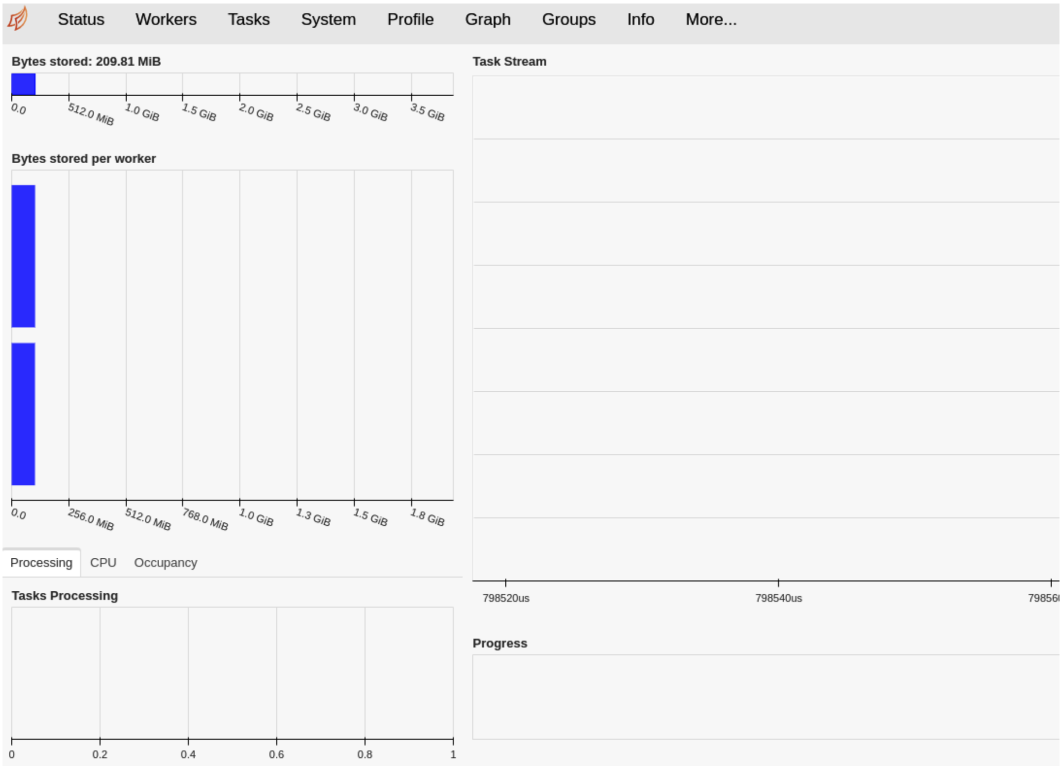
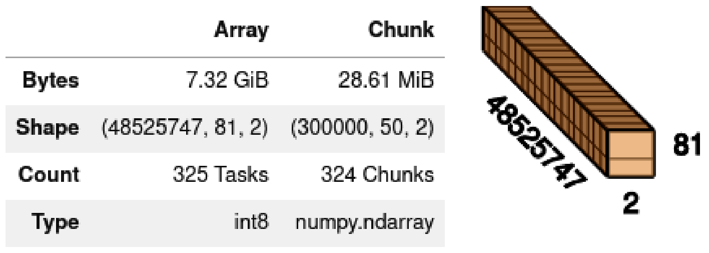


Figure 11.6 - The starting state of the dask.distributed dashboard

2. Let's load the data. More rigorously, let's prepare the task graph to load the data:

```
mosquito = da.from_zarr('data/AG1000G-AO/2L/calldata/GT')
mosquito
```

Here is the output on Jupyter:



	Array	Chunk
Bytes	7.32 GiB	28.61 MiB
Shape	(48525747, 81, 2)	(300000, 50, 2)
Count	325 Tasks	324 Chunks
Type	int8	numpy.ndarray

48525747
81
2

Figure 11.7 - Summary of the original Zarr array for chromosome 2L

3. To facilitate visualization, let's rechunk again. We are also going to have a single chunk for the second dimension, which is the samples. This is because our computation of missingness requires all the samples, and it makes little sense—in our specific case—to have two chunks per sample:

```
mosquito = mosquito.rechunk((mosquito.shape[0]//8, 81,
2))
```

   As a reminder, we have very large chunks, and you might end up with memory problems. If that is the case, then you can run it with the original chunks. It's just that the visualization will be unreadable.

4. Before we continue, let's ask Dask to not only execute the rechunking but also to have the results of it at the ready in the workers:

```
mosquito = mosquito.persist()
```

   The `persist` call makes sure the data is available in the workers. In the following screenshot, you can find the dashboard somewhere in the middle of the computation. You can find which tasks are executing on each node, a summary of tasks done and to be done, and the bytes stored per worker. Of note is the concept of **spilled to disk** (see the top left of the screen). If there is not enough memory for all chunks, they will temporarily be written to disk:

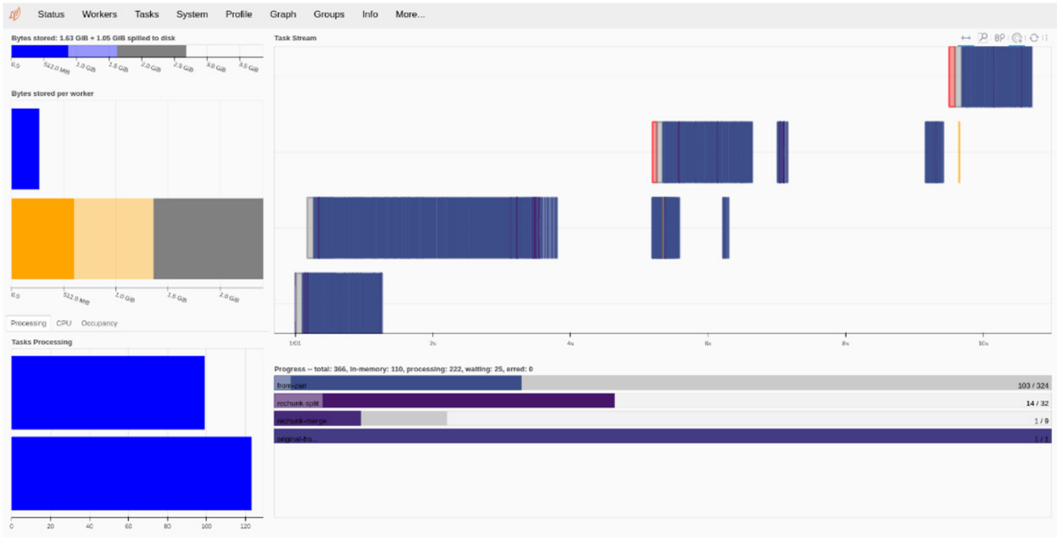


Figure 11.8 - The dashboard state while executing the persist function for rechunking the array

5. Let's now compute the statistics. We will use a different approach for the last recipe—we will ask Dask to apply a function to each chunk:

```
def calc_stats(my_chunk):
    num_miss = np.sum(
        np.equal(my_chunk[0][0][:,:,0], -1),
        axis=1)
    return num_miss
stats = da.blockwise(
    calc_stats, 'i', mosquito, 'ijk',
    dtype=np.uint8)
stats.visualize()
```

Remember that each chunk is not a `dask.array` instance but a NumPy array, so the code works on NumPy arrays. Here is the current task graph. There are no operations to load the data, as the function performed earlier executed all of those:

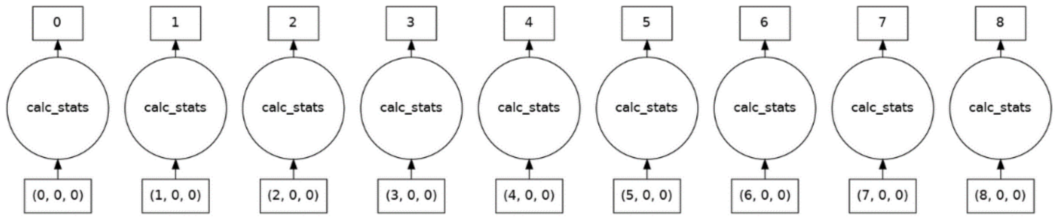


Figure 11.9 - Calls to the calc_stats function over chunks starting with persisted data

6. Finally, we can get our results:

```
stat_results = stats.compute()
```

There's more...

There is substantially more that can be said about the `dask.distributed` interface. Here, we introduced the basic concepts of its architecture and the dashboard.

`dask.distributed` provides an asynchronous interface based on the standard `async` module of Python. Due to the introductory nature of this chapter, we won't address it, but you are recommended to look at it.

See also

- You can start with the documentation of `dask.distributed` at `https://distributed.dask.org/en/stable/`.
- In many cases, you will need to deploy your code in a cluster or the cloud. Check out the deployment documentation for resources on different platforms: `https://docs.dask.org/en/latest/deploying.html`.
- After you have mastered the content here, studying asynchronous computation in Python would be your next step. Check out `https://docs.python.org/3/library/asyncio-task.html`.

12
Functional Programming for Bioinformatics

Python is a multi-paradigm language that allows you to express computations in multiple different ways. It is sometimes called an **object-oriented** (**OO**) language: sure, you can write code in an OO dialect, but you can also use other styles. Most code in Python is written in an imperative style: there is no structuring along a class hierarchy, as typical of the OO paradigm, and most code changes state that, for example, if you write `x = x + 1`, you are changing the state of variable `x`.

If you write complex code, especially code that requires parallel processing, imperative and OO paradigms will hit some limits. For simple scripts that run on a single machine, imperative and OO styles will do fine, but bioinformatics is a big-data enterprise, and you will often need to scale up and scale out, as there is lots of information to process, and many algorithms are computationally heavy.

Functional programming is quite useful for the complex and parallel problems that are becoming more common within bioinformatics. Many modern architectures for high-throughput data analysis are based on functional programming ideas, for example, the MapReduce paradigm originally developed at Google and on platforms such as Apache Spark. These ideas are also directly applicable to Dask and even to making sequential code more robust.

Functional programming is based on functions, pure function evaluation, and avoiding mutability. In this chapter, we are going to take a very practical approach to presenting these concepts, with examples of their use cases within typical bioinformatics applications. By the end, you should have a basic conceptual grasp of functional programming, but above all, you will understand its utility and applicability.

If you are using Docker, and because all these libraries are fundamental for data analysis, they all can be found in the Docker image `tiagoantao/bioinformatics_base`.

In this chapter, we will cover the following recipes:

- Understanding pure functions
- Understanding immutability
- Avoiding mutability as a robust development pattern
- Using lazy programming for pipelining
- The limits of recursion with Python
- A showcase of Python's `functools` module

Understanding pure functions

A pure function has a couple of important properties: for the same inputs, it produces the same outputs, and it has no side effects (it doesn't change global variables and doesn't do I/O). In this recipe, we are going to introduce a few simple examples to make the concept clear. We are mostly – but not exclusively – interested in the second property: the lack of side effects. In later recipes, it will be made clear why pure functions can be quite useful.

We are going to develop a very simple example where we are counting the genes sequenced per sample. We will have a database on a text file with counts for genes. For example, we might have sequenced LCT and TP53 on a sample, and LCT, MRAP2, and POMC on another. The total count would be: TP53: 1, LCT: 2, MRPA2: 1, and POMC: 1. We will be using a CSV file that can be easily read with Pandas or even just the CSV module.

Getting ready

We will be using a simple CSV file as our little database. The code for this recipe can be found in `Chapter12/Pure.py`. The database file is `my_genes.csv` and there is a saved original state at `my_genes.csv.base` in case you need to get back to the original database state.

How to do it...

Take a look at the following steps to get started:

1. Let's start by creating two simple functions to load and save the data. We also need some time to restore the original database:

```
import shutil
import pandas as pd

def restore_db(file_name):
    shutil.copyfile(f'{file_name}.base', file_name)
```

```
def load(file_name):
    df = pd.read_csv(file_name).set_index('gene')
    return dict(df['count'])

 def save(dict_db, file_name):
    pd.Series(dict_db).to_csv(
        file_name, index_label='gene', header=['count'])
```

We use Pandas to take care of persistence. This is a very simple example; you could have also only used the `csv` module.

2. We will consider three alternative functions to report the genes seen in a sample; here is the first:

```
def add_sample_csv(gene_list):
    gene_count = load('my_genes.csv')
    for gene in gene_list:
        gene_count[gene]=gene_count(gene,0)+1

    save(gene_count, 'my_genes.csv')
```

This function automatically persists the data with a new sample. It has the side effect of reading and writing a file, so it's not a pure function.

3. Here is a second alternative to also report the genes in a sample:

```
def add_sample_global_dict(gene_list):
    global gene_count
    for gene in gene_list:
        gene_count[gene] = gene_count.get(0) + 1
```

This function uses a global variable from the module and updates it, a side effect. We will not be using this function, but it is provided as a *bad* example of a function with side effects. We should avoid global variables.

4. Here is a pure function variant:

```
def add_sample_dict(dict_db, gene_list):
    for gene in gene_list:
        dict_db[gene] = dict_db.get(0) + 1
    return dict_db
```

This function mutates one of its parameters – `dict_db` – and as we will see in the next recipe, this is not a best practice from a functional programming perspective. However, it always returns the same result for equal output and doesn't have side effects, so it will be a good first approach.

5. Imagine that we now run the following:

```
add_sample_csv(['MC4R', 'TYR'])
```

   But, if we run it 10 times by mistake, what would be the consequence? We would over-report both genes 9 times, corrupting the content of our database from a logical perspective.

6. As an alternative, consider the following:

```
add_sample_dict(gene_count, ['MC4R', 'TYR'])
```

If you run it 10 times, what would be the consequence? It would still be an error (as we are changing `gene_count`), but at least it is not yet committed to the disk. This dialect would be much more comfortable when doing exploratory data analysis – you could run it knowing that no external data would be damaged. In the next recipe, we are going to see an alternative to make re-running code less problematic.

There's more...

In the next few recipes, we are going to see, with very practical cases, why pure functions can be very useful. But there is one specific case that would be cumbersome to explain with an example and as such, we are going to present the theory here.

Pure functions make parallelizing code much easier. Imagine a framework to execute distributed computing such as Dask or Spark. Such a framework has to deal with the potential case where hardware fails, and code needs to be moved and potentially repeated elsewhere. If your code writes to disk (as an example of a side effect) every time it's run, it's much more complicated for distributed frameworks to recover. If your code has no side effects, the distributed framework could repeat your function without any concern for data consistency. Indeed, many distributed frameworks do not allow you to have side effects in your parallelizable code. They also might frown at data structure mutability, something we are going to discuss now.

Understanding immutability

Another common trait of functional programming is that data structures are typically immutable. This is a tough concept to get around when you are used to imperative programming – the idea of programming without objects that change state over time. Here, we are going to see a simple example of making a function from the previous recipe work in an immutable way: that is, so that no objects are changed, and if we need to pass new information, we create new ones.

This recipe will give a short presentation on immutability from a data structure perspective. It will be, in a sense, the standard presentation that you can find in most books. Our main consideration, though, is to discuss mutability as a code design pattern, the topic of the following recipe. But for this, we need to understand immutability first.

We will be looking at two functions: one that mutates data structures and another that doesn't. This will be done in the context of the example that we followed in the previous recipes of this chapter.

Getting ready

We will be using the same data as in the previous recipe. The code for this recipe can be found in `Chapter12/Mutability.py`.

How to do it...

Follow these steps:

1. From the previous recipe, we have a function that mutates the input dictionary:

```
def add_sample_dict(dict_db, gene_list):
    for gene in gene_list:
        dict_db.get(gene,0) +1
```

   If you call this code with, say, `add_sample_dict(gene_count, ['DEPP'])`, while there are no output parameters, your dictionary's `gene_count` will be *mutated* to add `1` to the gene `DEPP`. If you run this function an excessive number of times – which is typical when doing exploratory data analysis – you would change your dictionary erroneously.

2. Contrast the previous implementation with the following:

```
def add_sample_new_dict(dict_db, gene_list):
    my_dict_db = dict(dict_db)
    for gene in gene_list:
        my_dict_db[gene] = my_dict_db.get(0) + 1
    return my_dict_db
```

   In this case, we are copying `dict_db` to a new dictionary, `my_dict_db`, and adding the extra gene list. There is a memory and time cost of making the copy of the existing dictionary, but the original one, `dict_db`, is not changed.

3. If you use this implementation, you are sure that your input parameters are never changed:

```
new_gene_count = add_sample_new_dict(gene_count,
['DEPP'])
```

`gene_count` is not changed and `new_gene_count` is a completely new dictionary. You can repeat the execution as many times you want without being worried about the impact of each execution.

There's more...

This simple recipe provides an example of a function that doesn't mutate its parameters. We are now in a situation where we can execute this function as many times as we want – by mistake or on purpose – without any consequences for the rest of the code. This is quite useful when we are testing code or running exploratory data analysis, as we don't have to be worried about side effects. This also facilitates the work of distributed executors such as Dask or Spark, as they don't have to worry about checking the state of existing objects: they are fixed.

There are also important lessons here for general software design, even if you don't use distributed executors. That is what we are going to investigate in the following recipe.

Avoiding mutability as a robust development pattern

The previous recipe introduced the concept of immutable data structures. In this recipe, we are going to discuss a design pattern that avoids persistent database mutability in your code until the very end. In terms of pseudocode, most applications in a long script work as follows:

```
Do computation 1
Write computation 1 to disk or database
Do computation 2
Write computation 2 to disk or database
….
Do computation n
Write computation n to disk or database
```

Here, we are going to present an alternative paradigm and discuss why it is generally better from a resilience point of view:

```
Do computation 1
Write computation 1 to temporary storage
Do computation 2
Write computation 2 to temporary storage
...
Do computation n
Write computation n to temporary storage
Take all temporary data and write it to definitive disk and
database
```

First, we will show the code for both approaches, and then discuss why, for complex and sophisticated scripts, the latter is a better approach in most cases.

We will use the same example as in the previous recipes: we will report on genes seen in different samples.

Getting ready

We will be using the same data as in the previous recipe. The code for this recipe can be found in `Chapter12/Persistence1.py` and `Chapter12/Persistence2.py`.

How to do it...

Let's set up both solutions along with some shared code.

1. Both solutions still use `load` and `save` functions:

```
import pandas as pd

def restore_db(file_name):
    shutil.copyfile(f'{file_name}.base', file_name)

def load(file_name):
    df = pd.read_csv(file_name).set_index('gene')
    return dict(df['count'])

def save(dict_db, file_name):
    pd.Series(dict_db).to_csv(
        file_name, index_label='gene', header=['count'])
```

2. The first alternative, where we save as we go, is the following:

```
def add_sample_csv(gene_list):
    gene_count = load('my_genes.csv')
    for gene in gene_list:
        gene_count[gene]=gene_count(gene,0)+1
    save(gene_count, 'my_genes.csv')

add_sample_csv(['MC4R', 'TYR'])
add_sample_csv(['LCT', 'HLA-A'])
add_sample_csv(['HLA-B', 'HLA-C'])
```

Every time we add a sample, we update the main database. While this solution is not very efficient from an I/O perspective, that is not the issue at hand here – actually, most design patterns of this kind tend to be more efficient from an I/O perspective than what we are going to look at next.

3. The second alternative, when we store definitive data at the end, is the following:

```
def add_sample_new_dict(dict_db, gene_list):
    my_dict_db = dict(dict_db)  # next recipe
    for gene in gene_list:
        dict_db.get(gene,0) +1
    return my_dict_db

gene_count = load('my_genes.csv')
gene_count = add_sample_new_dict(gene_count, ['MC4R',
'TYR'])
gene_count = add_sample_new_dict(gene_count, ['LCT',
'HLA-A'])
gene_count = add_sample_new_dict(gene_count, ['HLA-B',
'HLA-C'])
save(gene_count, 'my_genes.csv')
```

In this case, we only update the main database at the very end. We use an in-memory dictionary to maintain the intermediate data.

There's more...

So, why do this? The solution that delays writing the final data is, if anything, more complex, as we have to potentially store partial results along the way and manage that (which we do with the `gene_count` dictionary in this case – a simple example). In fact, this is true but code has bugs, disk storage fills up, and computers break in the real world, so facing this is the more important consideration in pragmatic terms.

Imagine that your first solution stops working in the middle of the execution for some reason. The database is in an unknown state, as you don't know where the code was. You simply cannot restart the code from scratch, as part of it might already have been executed. So, you have to check what the state of the database is, see where the code stopped, and partially run what is missing, or find a way to roll back whatever part was executed. It is both cumbersome and bug-prone.

Now, imagine that there is a problem during the execution of the second approach: you do not have to worry about the intermediate state of the data because it doesn't affect your final database and it will be completely recomputed in a second execution. You can just re-run the code, without worrying about inconsistent states. The only exception is the very last line, but not only will this be less common but you can also concentrate all your effort on this point to make the process as resilient as possible.

Can this always be done? Or does it only work in some cases? Most code can be done so that it creates temporary files. Also, much of bioinformatics analysis is not real-time transactional, so it's normally easy to update a SQL database only at the very end. And you now have a single point of failure that is relevant to your whole code – if there is a problem, you know you only need to concentrate on that very final part.

This approach, where possible, will make the day-to-day maintenance of complex code much easier.

Using lazy programming for pipelining

Lazy programming is a strategy where we defer computation until it's really needed. It has many advantages compared with its counterpart, eager programming, where we compute everything as soon as we invoke a computation.

Python provides many mechanisms for lazy programming – indeed, one of the biggest changes from Python 2 to Python 3 is that the language became lazier.

To understand lazy programming, we are going again to take our gene database and do an exercise with it. We are going to check whether we have at least *n* genes with *y* reads each (for example, three genes with five reads each). This can be, say, a measure of the quality of our database – that is, a measure of whether we have enough genes with a certain number of samples.

We are going to consider two implementations: one lazy and one eager. We will then compare the implications of both approaches.

Getting ready

The code for this recipe can be found in `Chapter12/Lazy.py`.

How to do it...

Take a look at these two different approaches:

1. Let's start with the eager implementation:

```
import pandas as pd

def load(file_name):
    df = pd.read_csv(file_name).set_index('gene')
    return dict(df['count'])
```

```
def get_min_reads(all_data, min_reads):
    return {
        gene: count
        for gene, count in all_data.items()
        if count >= min_reads
    }

def has_min_observations(subset_data, min_observations):
    return len(subset_data) >= min_observations

print(has_min_observations(
    get_min_reads(
        load('my_genes.csv'), 4
    ), 3))
```

Notice that the code loads all the data, checks all entries for a count of `4`, and sees if there are at least `3` observations with that count.

2. This is an alternative, lazy version:

```
def get_rec(file_name):
    with open(file_name) as f:
        f.readline()  # header
        for line in f:
            toks = line.strip().split(',')
            yield toks[0], int(toks[1])

def gene_min_reads(source, min_reads):
    for gene, count in source:
        if count >= min_reads:
            yield gene

def gene_min_observations(subset_source, min_
observations):
    my_observations = 0
    for gene in subset_source:
        my_observations += 1
```

```
            if my_observations == min_observations:
                return True
        return False

print(gene_min_observations(
        gene_min_reads(
            get_rec('my_genes.csv'), 4
        ), 2))
```

This code behaves in a completely different way. First, `get_rec` and `gene_min_reads` are generators (note the `yield`), so they will only return results when required, one by one. `gene_min_observations` will return as soon as it has seen the necessary number of observations. This means that the only data that will be read and processed is the bare minimum to arrive at a result. In a worst-case scenario, this is still the whole file, but in many cases, it can be much less.

There's more...

The advantages of this approach can be seen with very large files. While there are no perceivable advantages with our toy file, if the file is too large to fit in memory, the first approach would simply crash! Also, every step of the pipeline will require enough memory to store all the data that it consumes, and all of the steps will need to be in-memory at the same time. So, even for medium-sized files, memory problems can occur with the eager version that would be avoided with the lazy version.

Many times, lazy code also does substantially less processing: as in the previous example, it may stop computation before seeing all the data. This is not always assured but happens with many use cases.

As a historical note, Python 2 was substantially more eager than Python 3. It is actually one of the main changes from Python 2 to Python 3. As a trivial example, consider the behavior of `range` in Python 2 versus 3.

Part of our lazy implementation can be written in a more succinct way by using some Python built-in modules – we will revisit this in the final recipe.

The limits of recursion with Python

Recursion – a function calling itself – is a fundamental concept in functional programming. Many languages that cater to functional programming are able to recurse ad infinitum. Python is not one of those languages. You can use recursion in Python, but you have to be aware of its limits, namely that recursion incurs quite a lot of memory costs and that it cannot be used to replace iteration – a typical case when writing functional programs.

To see the limits of recursion with Python, we are going to implement the Fibonacci sequence in several ways. As a reminder, Fibonacci can be defined as follows:

```
Fib(0) = 0
Fib(1) = 1
Fib(n) = Fib(n -1) + Fib(n −2)
```

We are also computing the factorial function, but in this case only in a recursive way:

```
Fact(1) = 1
Fact(n) = n * Fact(n −1 )
```

Getting ready

Our code is available in `Chapter12/Recursion.py`.

How to do it...

Follow these steps to get started:

1. Let's start with an iterative version of Fibonacci:

    ```
    def fibo_iter(n):
        if n < 2:
            return n
        last = 1
        second_last = 0
        for _i in range(2, n + 1):
            result = second_last + last
            second_last = last
            last = result
        return result
    ```

    This version is perfectly sufficient and also quite efficient. We are not suggesting that the iterative version is in any way lesser than the recursive version. On the contrary, with Python, it will probably perform better.

2. Here is a recursive version:

    ```
    def fibo_naive(n):
        if n == 0:
            return 0
    ```

```
    if n == 1:
        return 1
    return fibo_naive(n - 1) + fibo_naive(n - 2)
```

If you run it, you will see that the performance suffers quite substantially when compared with the iterative version. Furthermore, if you ask for `fibo_naive(1000)` or a similarly large number, the code will not perform well at all (it can take many hours for 1,000 cases), whereas the iterative version will perform adequately. In the following recipe, we will actually fix part of this. But for now, let's dig deeper into the issues with recursion and Python.

3. To make the case quite obvious, let's implement a factorial function that is as simple as it can get from a recursive implementation perspective (even simpler than Fibonacci):

```
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n - 1)
```

If you run this function with a small number, such as `factorial(5)`, you will get the correct answer, `5`. But if you try a large number, such as `factorial(20000)`, you will get the following:

```
Traceback (most recent call last):
  File "Recursion.py", line 8, in <module>
    factorial(20000)
  File "Recursion.py", line 4, in factorial
    return n * factorial(n - 1)
  File "Recursion.py", line 4, in factorial
    return n * factorial(n - 1)
  File "Recursion.py", line 4, in factorial
    return n * factorial(n - 1)
  [Previous line repeated 995 more times]
  File "Recursion.py", line 2, in factorial
    if n == 1:
RecursionError: maximum recursion depth exceeded in
comparison
```

This error suggests that there is a limit to how much you can recurse in Python: Python can only recurse to a certain limit (see `sys.setrecursionlimit` to change this). While you can change the number of recursions to something bigger, there will always be a limit bound by the memory necessary to maintain the stack. There are also alternative ways to explicitly implement recursion, but at the cost of speed. Many languages implement a feature called **Tail Call Optimization** (**TCO**), which allows for infinite levels of recursion with high performance, but Python doesn't implement it.

There's more...

You can use recursion in Python for simple cases without many recurring calls, but in Python, recursion is not a general substitute for iterative solutions. Recursion is probably the least well-implemented major feature from the functional world in Python.

A showcase of Python's functools module

Python has three built-in modules that greatly help when writing code in a functional dialect: `functools`, `operator`, and `itertools`. In this recipe, we are going to briefly discuss the `functools` module. For example, the fundamental `reduce` function (where part of the name of MapReduce comes from) is only available if you import `functools`.

While a detailed exploration of these modules would be too long for a single recipe, we are going to showcase some functionality by improving some of the code of the previous recipes with the functionality from `functools` and showcase some illustrative examples of the utility of the module.

Getting ready

Our code is available in `Chapter12/Builtin.py`. We will make references to previous recipes.

How to do it...

Let's look at several illustrative examples:

1. Remember that our recursive implementation of a factorial function in the previous recipe was not very efficient? Let's mitigate many of its problems in a very simple way:

```
import functools

@functools.cache
def fibo(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fibo(n - 1) + fibo(n - 2)
```

If you remember the recursive code in the previous recipe, it was very slow, even for small numbers. Doing `fibo_naive(1000)` could take hours. This function, just by adding the `@functools.cache` decorator, can process much higher numbers far more quickly. This is due to the `cache` decorator implementing memoization.

What is memoization?

Memoization is the process through which the computer caches the results of execution and, the next time it is called, instead of computing it again, just returns the stored result. For example, the first time you call `fibo(10)`, to get the result, the function is actually executed, but the second time you call it, the result was cached on the first run and returned without execution. Memoization only works correctly if the function always returns the same output for an equal input and has no side effects. That is, memoization only works with pure functions.

2. For another example of an alternative to the existing code using a functional approach, take the function from the *Using lazy programming for pipelining* recipe:

```
def gene_min_reads(source, min_reads):
    for gene, count in source:
        if count >= min_reads:
            yield gene
```

   There is already some functional flavor here, as we are using a generator.

3. This function can be written in a more functional dialect:

```
def gene_min_reads(source, min_reads):
    return map(
        lambda x: x[0],
        filter(lambda x: x[1] >= min_reads,
        source.items()))
```

   There is a lot to unpack here. First, look at the built-in `filter` function: it will apply the function defined in the first parameter to objects of the iterator on the second parameter. In our case, objects where the second element is larger or equal to `min_reads` will be preserved. Then, the `map` function takes each object (which is of the type `('GENE', count)`) and returns only the first part. `map` and `filter` functions are very common in dialects in functional programming. Also note the quintessential functional concept of the anonymous function, a `lambda`: this is a function definition that will only be used in a single place – it's quite useful for very tiny functions. In this case, there is no direct reason to rewrite in this fashion (especially because the generator type of the previous definition already provides the most important feature of a functional dialect to our problem), but you will find many cases where this type of representation is more succinct and expressive.

4. Another important concept that you will probably need when using distributed systems is partial function application – here is a simple example using the most basic of arithmetic functions:

```
def multiplication(x, y):
    return x * y
double = functools.partial(multiplication, 2)
double(3)
```

`double` is defined as partially resolving one of the arguments of `multiplication` – hence, we use the term partial function application. This is not only a stylistic issue, as sometimes in distributed platforms (or even just with the `map` function of multiprocessing pools), you are limited to only supplying a single parameter – hence, a partial function application becomes a necessity.

There's more...

There are many more interesting applications in the `functools` module to check out. The `itertools` and `operator` modules also have many functions that are idiomatic from a functional perspective.

See also...

There are several functional programming libraries for Python – for example, see the following:

- Toolz (`http://toolz.readthedocs.org/`) with extra iterators, function utilities, and functional dialects for dictionary.
- Pyrsistent implements immutable data structures: `https://github.com/tobgu/pyrsistent`.
- For general resources on functional programming for Python, be sure to check the awesome functional Python list on GitHub: `https://github.com/sfermigier/awesome-functional-python`.

Index

Symbols

1,000 Genomes Project
  URL 65

A

admixture
  reference link 177
  used, for investigating population structure 177-183
African Yorubans (YRI) 57
Ag3.0 data downloads
  reference link 292
Amazon Web Services (AWS) 305
AmiGO
  reference link 149
Anaconda
  used for installing prerequisite software 2-7
annotations
  used, for extracting genes from reference with 137-140
Apache Airflow
  URL 249
Apache Arrow
  used, for accelerating pandas processing 32-35
application programming interface (API) 50, 296
API key
  used, for accessing Galaxy 252-260
Arlequin software suite
  reference link 200

B

BAM format
  reference link 72
Basic Local Alignment Search Tool (BLAST) 54
binary Variant Call Format (BCF)
  reference link 75
BioBlend
  URL 252
Bioconda
  URL 2
bioinformatics
  pipeline system 249
Bio.PDB
  about 222
  working with 223-228
Biopython
  about 50
  mmCIF files, parsing 247, 248
  reference link 54

Biopython SeqIO
  reference link 57
Biostars
  URL 54
Blosc
  URL 297
Bokeh 39
breast cancer outcomes
  predicting, with Random Forests 286-289
breast cancer traits
  exploring, with Decision Trees 282-286
Browser Extensible Data (BED) 88
Burrows-Wheeler Aligner (BWA)
  about 66
  reference link 72

C

Center for Disease Control (CDC) 192
chart generation
  Matplotlib, using 39-46
chloroquine resistance transporter (CRT) 51
chunking 295
chunk sizes
  usage 305
comma-separated values (CSV) 20
Comprehensive R Archive Network (CRAN)
  URL 15
Concise Idiosyncratic Gapped Alignment Report (CIGAR) 68
Consensus Coding Sequence (CCDS)
  reference link 89
copy number variation (CNV) 73, 114, 151
coroutines
  reference link 311
CPython 2
Cytoscape
  URL 215

D

DADA2 plugin
  reference link 118
Dask
  about 300
  used, for processing genomic data based on NumPy arrays 300-304
Dask best practices
  reference link 305
dask.distributed
  reference link 311
  used, for scheduling tasks 305-311
data
  extracting, from VCF files 73-75
  parallel processing, with Python multiprocessing 297-299
data alignment
  working with 66-72
DataFrames through joins
  pitfalls 26-28
dataset
  preparing, for phylogenetic analysis 185-191
dataset for analysis
  preparing 93-100
datasets
  exploring, with sgkit 163-167
  managing, with PLINK 152-158
Decision Trees
  used, for exploring breast cancer traits 282-286
deploy dask clusters
  reference link 311
Directed Acyclic Graph (DAG) 264
DNA sequence analysis
  performing 55-57

Docker
  about 2
  URL 8
  used, for installing prerequisite software 7, 8
Domain Specific Language (DSL) 267

E

EIGENSOFT
  reference link 174
Ensembl
  reference link 54
  used, for retrieving gene ontology information 144-148
Ensembl REST API
  orthologues, finding with 141-144
  reference link 144
ETE library
  URL 192

F

FASTQ format
  reference link 65
fastStructure
  reference link 177
first-in, first-out (FIFO) 208
Fixation Index (FST) 151
FlyBase
  URL 128
functional programming 313
functools module
  exploring 326-328

G

Galaxy
  about 250
  accessing, with API key 252-260
  URL 249-252
Galaxy server
  about 250
  installing 250, 251
GenBank
  accessing 50
Gene Ontology (GO) 220
gene ontology information
  retrieving, from Ensembl 144-148
  URL 149
General Feature Format (GFF) 88, 134
General Transfer Format (GTF) 134
genes
  extracting, from reference with annotations 137-140
Genetic codes
  reference link 57
genetic data
  aligning 192-194
genome accessibility
  studying 76-88
Genome Analysis Toolkit (GATK) 66, 93
genome annotations
  traversing 134-136
Genome Browser
  reference link 144
genomes
  reference link 128
Genome-Wide Association Studies (GWAS) 151
genomic data
  aligning 192-194

genomic data based on NumPy arrays
  processing, with Dask 300-304
genomic features
  finding, from sequencing annotations 111-113
genomics data
  reading, with Zarr 292-296
geometric operations
  performing 237-240
Gephi
  URL 215
GFF format
  reference link 137
GFF spec
  reference link 137
ggplot2 library 12
Google Cloud Platform (GCP) 292
Graphviz
  URL 149
gsutil
  installation link 292

H

Han Chinese in Beijing (CHB) 58
HapMap
  reference link 152
Hierarchical Data Format 5 (HDF5)
  reference link 297
high-performance computing (HPC) 2
high-quality reference genomes
  working with 122-128
HTSeq
  URL 88
  used, for processing NGS data 88-92
htslib package
  URL 73
Hydrogens, in PDB files
  reference link 240

I

immutability
  about 316
  examples 317, 318
insertions/deletions (INDELs) 73
IPython magic
  reference link 18
IPython third-party extensions
  reference link 18
IronPython 2
IUPAC ambiguity 134

J

Japanese in Tokyo (JPT) 58
Java Virtual Machine (JVM) 2
Jupyter
  used, for performing R magic 16-18
jupytext
  reference link 4
Jython 2

K

K-Means clustering
  about 277
  using, to classify samples over PCA 276-282

L

last-in, first-out (LIFO) 208
lazy programming
  about 321
  using, for pipelining 321-323

Linkage Disequilibrium- (LD-) 152
lookup table (LUT) 68
low-quality genome references
  dealing with 128-132

M

MAFFT
  reference link 192
magics command 16
Matplotlib
  reference link 47
  used, for chart generation 39-46
Mendelian error information
    for quality control
  using 101-105
Message Digest 5 (MD5) 58
metagenomics
  with QIIME 2 Python API 114-119
mmCIF files
  parsing, with Biopython 247, 248
mmCIF format
  reference link 228
modern sequence formats
  working with 57-64
molecular distances
  computing, on PDB file 232-237
multiple databases
  protein, finding in 218-221
multiprocessing
  reference link 300
MUSCLE
  reference link 192
mutability
  avoiding, as robust development
    pattern 318-320
MySQL tables for Ensembl
  reference link 140

N

National Center for Biotechnology
    Information (NCBI) 190
NCBI databases
  about 50-54
  data, fetching 51-53
  data, searching 51-53
  reference link 54
Network Common Data Form (NetCDF)
  reference link 297
Nextflow
  URL 249
  variant analysis pipeline, deploying 267-272
next-generation sequencing (NGS) 2, 93, 151
NGS data
  processing, with HTSeq 88-92
NumPy
  about 36
  using, in Python data science and
    bioinformatics 36-39

O

object-oriented (OO) interface 47
object-oriented (OO) language 313
orders of magnitude (OOMs) 58
orthologues
  finding, with Ensembl REST API 141-144
outer join 29

P

p53
  reference link 222
pandas
  using, to process
    vaccine-adverse events 20-25

pandas DataFrames
  memory usage, reducing 29-32
pandas processing
  accelerating, with Apache Arrow 32-35
Panther
  URL 149
parallel processing of data
  with Python multiprocessing 297-299
PDB file
  molecular distances, computing on 228-232
  used, for extracting information 232-237
Phred quality score
  about 62
  reference link 59
phylogenetic analysis
  dataset, preparing 185-191
phylogenetic data
  visualizing 210-214
phylogenetic trees
  reconstructing 200-204
pipelining
  lazy programming, using for 321-323
PlasmoDB
  reference link 128
Plasmodium falciparum 127
PLINK
  reference link 151
  used, for managing datasets 152-158
population structure
  analyzing 167-174
  investigating, with admixture 177-183
prerequisite software
  installing, with Anaconda 2-7
  installing, with Docker 7, 8
Principal Component Analysis (PCA)
  about 151, 273
  performing 174-177
  using, in scikit-learn 273-276
protein
  finding, in multiple databases 218-221
Protein Data Bank (PDB) 217
proteomics 217
pure functions
  about 314, 316
  examples 314-316
PyMOL
  animating with 241-246
  URL 241
pysamstats library
  reference link 72
Python
  about 313
  limits of recursion 324, 325
Python Global Interpreter Lock (GIL)
  reference link 300
Python multiprocessing
  used, for parallel processing of data 297-299

Q

QuickGO
  reference link 149

R

Random Forests
  used, for predicting breast cancer outcomes 286-289
recursion
  about 323
  limits of recursion, with Python 324, 325
recursive algorithms 205-209
regular expression (regex) 90
RepeatMasker
  reference link 134

R magic
  performing, with Jupyter 16-18
R packages for Bioinformatics
  reference link 15
R programming
  interfacing with, via rpy2 9-15
rpy2
  about 9
  used, for interfacing with R programming 9-15
rpy library
  reference link 16

S

SAM format
  reference link 72
sample read depth (DP) 83
scikit-learn
  with Principal Component Analysis (PCA) 273-276
SEQanswers
  reference link 72
seq.data variable 12
Sequence Alignment Map (SAM) 66
Sequence Read Archive (SRA) 53
sequences
  comparing 195-200
sequencing annotations
  used, for finding genomic features 111-113
sgkit
  about 158
  used, for exploring datasets 163-167
  using, for population genetics analysis with xarray 158-163
Single-Nucleotide Polymorphism (SNP) 151, 293
Singularity
  reference link 8
Snakemake
  URL 249
  variant analysis pipeline, deploying 260-266
SNP data
  filtering 76-88
SnpEff
  URL 77
standard statistics data
  exploring 106-110
Structural Bioinformatics FAQ, Biopython
  reference link 223

T

Tail Call Optimization (TCO) 326
tasks
  scheduling, with dask.distributed 305-311
three-dimensional (3D) array 302
Toolz
  URL 328
trees
  playing, recursively with 205-209
trimAl
  reference link 192
two-dimensional (2D) array 302

U

UCSC genome browser
  reference link 140
UniProt REST interface
  reference link 222
University of California, Santa Cruz (UCSC)
  reference link 54
Utah Residents with Northern and Western European Ancestry (CEU) 57

V

Vaccine Adverse Event Reporting System (VAERS)
  reference link 20
vaccine-adverse events
  processing, with pandas 20-25
VAERS data
  used, for demonstrating NumPy usages 36-39
variant analysis pipeline
  deploying, with Nextflow 267-272
  deploying, with Snakemake 260-266
Variant Call Format (VCF) 151
VCF files
  used, for extracting data 73-75
VectorBase
  URL 128
Vertebrate Genome Annotation (Vega) 143
VirtualBox 2
voltage-gated sodium channel (VGSC) 137

X

xarray
  sgkit, using for population genetics analysis 158-163

Z

Zarr
  reference link 292
  used, for reading genomics data 292-296
Zarr arrays
  memory 296

Share Your Thoughts

Now you've finished *Bioinformatics with Python Cookbook*, we'd love to hear your thoughts! If you purchased the book from Amazon, please click here to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.