

Phuong Vothihong, Martin Czygan,
Ivan Idris, Magnus Vilhelm Persson,
Luiz Felipe Martins

Python: End-to-end Data Analysis

Learning Path

Leveraging the power of Python to clean, scrape, analyze,
and visualize your data



Packt>

Python: End-to-end Data Analysis

Leverage the power of Python to clean, scrape,
analyze, and visualize your data

A course in three modules



BIRMINGHAM - MUMBAI

Python: End-to-end Data Analysis

Copyright © 2016 Packt Publishing

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

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

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

Published on: May 2017

Production reference: 1050517

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

ISBN 978-1-78839-469-7

www.packtpub.com

Credits

Authors

Phuong Vo.T.H

Martin Czygan

Ivan Idris

Magnus VilhelmPersson

Luiz Felipe Martins

Reviewers

Dong Chao

Hai Minh Nguyen

Kenneth Emeka Odoh

Bill Chambers

Alexey Grigorev

Dr. VahidMirjalili

Michele Usuelli

Hang (Harvey) Yu

Laurie Lugrin

Chris Morgan

Michele Pratusevich

Content Development Editor

Aishwarya Pandere

Graphics

Jason Monteiro

Production Coordinator

Deepika Naik

Preface

The use of Python for data analysis and visualization has only increased in popularity in the last few years.

The aim of this book is to develop skills to effectively approach almost any data analysis problem, and extract all of the available information. This is done by introducing a range of varying techniques and methods such as uni- and multi-variate linear regression, cluster finding, Bayesian analysis, machine learning, and time series analysis. Exploratory data analysis is a key aspect to get a sense of what can be done and to maximize the insights that are gained from the data. Additionally, emphasis is put on presentation-ready figures that are clear and easy to interpret.

What this learning path covers

Module 1, Getting Started with Python Data Analysis, shows how to work with time-oriented data in Pandas. How do you clean, inspect, reshape, merge, or group data – these are the concerns in this chapter. The library of choice in the course will be Pandas again.

Module 2, Python Data Analysis Cookbook, demonstrates how to visualize data and mentions frequently encountered pitfalls. Also, discusses statistical probability distributions and correlation between two variables.

Module 3, Mastering Python Data Analysis, introduces linear, multiple, and logistic regression with in-depth examples of using SciPy and stats models packages to test various hypotheses of relationships between variables.

What you need for this learning path

Module 1:

There are not too many requirements to get started. You will need a Python programming environment installed on your system. Under Linux and Mac OS X, Python is usually installed by default. Installation on Windows is supported by an excellent installer provided and maintained by the community. This book uses a recent Python 2, but many examples will work with Python 3 as well.

The versions of the libraries used in this book are the following: NumPy 1.9.2, Pandas 0.16.2, matplotlib 1.4.3, tables 3.2.2, pymongo 3.0.3, redis 2.10.3, and scikit-learn 0.16.1. As these packages are all hosted on PyPI, the Python package index, they can be easily installed with pip. To install NumPy, you would write:

```
$ pip install numpy
```

If you are not using them already, we suggest you take a look at virtual environments for managing isolating Python environment on your computer. For Python 2, there are two packages of interest there: virtualenv and virtualenvwrapper. Since Python 3.3, there is a tool in the standard library called pyvenv (<https://docs.python.org/3/library/venv.html>), which serves the same purpose.

Most libraries will have an attribute for the version, so if you already have a library installed, you can quickly check its version:

```
>>>import redis
>>>redis.__version__ '2.10.3'
```

This works well for most libraries. A few, such as pymongo, use a different attribute (pymongo uses just version, without the underscores). While all the examples can be run interactively in a Python shell, we recommend using IPython. IPython started as a more versatile Python shell, but has since evolved into a powerful tool for exploration and sharing. We used IPython 4.0.0 with Python 2.7.10. IPython is a great way to work interactively with Python, be it in the terminal or in the browser.

Module 2:

First, you need a Python 3 distribution. I recommend the full Anaconda distribution as it comes with the majority of the software we need. I tested the code with Python 3.4 and the following packages:

- joblib 0.8.4
- IPython 3.2.1

- NetworkX 1.9.1
- NLTK 3.0.2
- Numexpr 2.3.1
- pandas 0.16.2
- SciPy 0.16.0
- seaborn 0.6.0
- sqlalchemy 0.9.9
- statsmodels 0.6.1
- matplotlib 1.5.0
- NumPy 1.10.1
- scikit-learn 0.17
- dautil0.0.1a29

For some recipes, you need to install extra software, but this is explained whenever the software is required.

Module 3:

All you need to follow through the examples in this book is a computer running any recent version of Python. While the examples use Python 3, they can easily be adapted to work with Python 2, with only minor changes. The packages used in the examples are NumPy, SciPy, matplotlib, Pandas, stats models, PyMC, Scikit-learn. Optionally, the packages basemap and cartopy are used to plot coordinate points on maps. The easiest way to obtain and maintain a Python environment that meets all the requirements of this book is to download a prepackaged Python distribution. In this book, we have checked all the code against Continuum Analytics' Anaconda Python distribution and Ubuntu Xenial Xerus (16.04) running Python 3.

To download the example data and code, an Internet connection is needed.

Who this learning path is for

This learning path is for developers, analysts, and data scientists who want to learn data analysis from scratch. This course will provide you with a solid foundation from which to analyze data with varying complexity. A working knowledge of Python (and a strong interest in playing with your data) is recommended.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this course — what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the course's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt course, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this course from your account at <http://www.packtpub.com>. If you purchased this course elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the SUPPORT tab at the top.
3. Click on Code Downloads & Errata.
4. Enter the name of the course in the Search box.
5. Select the course for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this course from.
7. Click on Code Download.

You can also download the code files by clicking on the Code Files button on the course's webpage at the Packt Publishing website. This page can be accessed by entering the course's name in the Search box. Please note that you need to be logged into your Packt account.

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

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac

7-Zip / PeaZip for Linux

The code bundle for the course is also hosted on GitHub at <https://github.com/PacktPublishing/Python-End-to-end-Data-Analysis>. We also have other code bundles from our rich catalog of books, videos, and courses available at <https://github.com/PacktPublishing/>. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our courses – maybe a mistake in the text or the code – we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this course. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your course, clicking on the Errata Submission Form link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the course in the search field. The required information will appear under the Errata section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this course, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

Module 1: Getting Started with Python Data Analysis

Chapters 1: Introducing Data Analysis and Libraries	3
Data analysis and processing	4
An overview of the libraries in data analysis	7
Python libraries in data analysis	9
NumPy	10
Pandas	10
Matplotlib	11
PyMongo	11
The scikit-learn library	11
Summary	11
Chapters 2: NumPy Arrays and Vectorized Computation	13
NumPy arrays	14
Data types	14
Array creation	16
Indexing and slicing	18
Fancy indexing	19
Numerical operations on arrays	20
Array functions	21
Data processing using arrays	23
Loading and saving data	24
Saving an array	24
Loading an array	25
Linear algebra with NumPy	26
NumPy random numbers	27
Summary	30

Chapters 3: Data Analysis with Pandas	33
An overview of the Pandas package	33
The Pandas data structure	34
Series	34
The DataFrame	36
The essential basic functionality	40
Reindexing and altering labels	40
Head and tail	41
Binary operations	42
Functional statistics	43
Function application	45
Sorting	46
Indexing and selecting data	48
Computational tools	49
Working with missing data	51
Advanced uses of Pandas for data analysis	54
Hierarchical indexing	54
The Panel data	56
Summary	58
Chapters 4: Data Visualization	61
The matplotlib API primer	62
Line properties	65
Figures and subplots	67
Exploring plot types	70
Scatter plots	70
Bar plots	71
Contour plots	72
Histogram plots	74
Legends and annotations	75
Plotting functions with Pandas	78
Additional Python data visualization tools	80
Bokeh	81
MayaVi	81
Summary	83
Chapters 5: Time Series	85
Time series primer	85
Working with date and time objects	86
Resampling time series	94

Downsampling time series data	94
Upsampling time series data	97
Time zone handling	99
Timedeltas	100
Time series plotting	101
Summary	105
Chapters 6: Interacting with Databases	107
Interacting with data in text format	107
Reading data from text format	107
Writing data to text format	112
Interacting with data in binary format	113
HDF5	114
Interacting with data in MongoDB	115
Interacting with data in Redis	120
The simple value	120
List	121
Set	122
Ordered set	123
Summary	124
Chapters 7: Data Analysis Application Examples	127
Data munging	128
Cleaning data	130
Filtering	133
Merging data	136
Reshaping data	139
Data aggregation	141
Grouping data	144
Summary	146
Chapters 8: Machine Learning Models with scikit-learn	147
An overview of machine learning models	147
The scikit-learn modules for different models	148
Data representation in scikit-learn	150
Supervised learning – classification and regression	152
Unsupervised learning – clustering and dimensionality reduction	158
Measuring prediction performance	162
Summary	164

Module 2: Python Data Analysis Cookbook

Chapter 1: Laying the Foundation for Reproducible Data Analysis	167
Introduction	168
Setting up Anaconda	168
Getting ready	169
How to do it...	169
There's more...	170
See also	170
Installing the Data Science Toolbox	170
Getting ready	171
How to do it...	171
How it works...	172
See also	172
Creating a virtual environment with virtualenv and virtualenvwrapper	172
Getting ready	173
How to do it...	173
See also	174
Sandboxing Python applications with Docker images	174
Getting ready	174
How to do it...	174
How it works...	176
See also	176
Keeping track of package versions and history in IPython Notebook	176
Getting ready	177
How to do it...	177
How it works...	179
See also	179
Configuring IPython	179
Getting ready	180
How to do it...	180
See also	181
Learning to log for robust error checking	182
Getting ready	182
How to do it...	182
How it works...	185
See also	185

Unit testing your code	185
Getting ready	185
How to do it...	186
How it works...	187
See also	187
Configuring pandas	188
Getting ready	188
How to do it...	188
Configuring matplotlib	190
Getting ready	191
How to do it...	191
How it works...	194
See also	194
Seeding random number generators and NumPy print options	194
Getting ready	194
How to do it...	194
See also	196
Standardizing reports, code style, and data access	196
Getting ready	197
How to do it...	197
See also	199
Chapter 2: Creating Attractive Data Visualizations	201
Introduction	202
Graphing Anscombe's quartet	202
How to do it...	202
See also	205
Choosing seaborn color palettes	205
How to do it...	205
See also	208
Choosing matplotlib color maps	208
How to do it...	208
See also	209
Interacting with IPython Notebook widgets	209
How to do it...	209
See also	213
Viewing a matrix of scatterplots	213
How to do it...	213
Visualizing with d3.js via mpld3	215
Getting ready	215
How to do it...	216

Creating heatmaps	217
Getting ready	217
How to do it...	217
See also	219
Combining box plots and kernel density plots with violin plots	220
How to do it...	220
See also	221
Visualizing network graphs with hive plots	221
Getting ready	222
How to do it...	222
Displaying geographical maps	224
Getting ready	224
How to do it...	224
Using ggplot2-like plots	226
Getting ready	227
How to do it...	227
Highlighting data points with influence plots	228
How to do it...	229
See also	231
Chapter 3: Statistical Data Analysis and Probability	233
Introduction	234
Fitting data to the exponential distribution	234
How to do it...	234
How it works...	236
See also	236
Fitting aggregated data to the gamma distribution	237
How to do it...	237
See also	238
Fitting aggregated counts to the Poisson distribution	238
How to do it...	239
See also	241
Determining bias	241
How to do it...	242
See also	244
Estimating kernel density	244
How to do it...	244
See also	246
Determining confidence intervals for mean, variance, and standard deviation	247
How to do it...	247

See also	249
Sampling with probability weights	249
How to do it...	250
See also	252
Exploring extreme values	253
How to do it...	253
See also	256
Correlating variables with Pearson's correlation	257
How to do it...	257
See also	260
Correlating variables with the Spearman rank correlation	260
How to do it...	260
See also	263
Correlating a binary and a continuous variable with the point biserial correlation	263
How to do it...	263
See also	265
Evaluating relations between variables with ANOVA	265
How to do it...	266
See also	267
Chapter 4: Dealing with Data and Numerical Issues	269
Introduction	269
Clipping and filtering outliers	270
How to do it...	270
See also	272
Winsorizing data	273
How to do it...	273
See also	274
Measuring central tendency of noisy data	275
How to do it...	275
See also	277
Normalizing with the Box-Cox transformation	278
How to do it...	278
How it works	280
See also	280
Transforming data with the power ladder	280
How to do it...	281
Transforming data with logarithms	282
How to do it...	283

Rebinning data	284
How to do it...	285
Applying logit() to transform proportions	286
How to do it...	287
Fitting a robust linear model	288
How to do it...	289
See also	291
Taking variance into account with weighted least squares	291
How to do it...	291
See also	294
Using arbitrary precision for optimization	294
Getting ready	294
How to do it...	294
See also	296
Using arbitrary precision for linear algebra	297
Getting ready	297
How to do it...	297
See also	299
Chapter 5: Web Mining, Databases, and Big Data	301
Introduction	302
Simulating web browsing	302
Getting ready	303
How to do it...	303
See also	305
Scraping the Web	305
Getting ready	306
How to do it...	306
Dealing with non-ASCII text and HTML entities	308
Getting ready	308
How to do it...	308
See also	310
Implementing association tables	310
Getting ready	310
How to do it...	310
Setting up database migration scripts	313
Getting ready	314
How to do it...	314
See also	314

Adding a table column to an existing table	314
Getting ready	314
How to do it...	315
Adding indices after table creation	316
Getting ready	316
How to do it...	316
How it works...	317
See also	317
Setting up a test web server	317
Getting ready	318
How to do it...	318
Implementing a star schema with fact and dimension tables	319
How to do it...	320
See also	324
Using HDFS	325
Getting ready	325
How to do it...	325
See also	326
Setting up Spark	326
Getting ready	327
How to do it...	327
See also	327
Clustering data with Spark	327
Getting ready	328
How to do it...	328
How it works...	331
There's more...	331
See also	331
Chapter 6: Signal Processing and Timeseries	333
Introduction	333
Spectral analysis with periodograms	334
How to do it...	334
See also	336
Estimating power spectral density with the Welch method	336
How to do it...	336
See also	338
Analyzing peaks	338
How to do it...	338
See also	340

Measuring phase synchronization	340
How to do it...	341
See also	342
Exponential smoothing	343
How to do it...	343
See also	345
Evaluating smoothing	346
How to do it...	346
See also	348
Using the Lomb-Scargle periodogram	349
How to do it...	349
See also	351
Analyzing the frequency spectrum of audio	351
How to do it...	352
See also	354
Analyzing signals with the discrete cosine transform	354
How to do it...	355
See also	356
Block bootstrapping time series data	357
How to do it...	357
See also	359
Moving block bootstrapping time series data	359
How to do it...	360
See also	362
Applying the discrete wavelet transform	363
Getting started	364
How to do it...	364
See also	366
Chapter 7: Selecting Stocks with Financial Data Analysis	367
Introduction	368
Computing simple and log returns	368
How to do it...	369
See also	369
Ranking stocks with the Sharpe ratio and liquidity	370
How to do it...	370
See also	372
Ranking stocks with the Calmar and Sortino ratios	372
How to do it...	372
See also	374

Analyzing returns statistics	374
How to do it...	375
Correlating individual stocks with the broader market	377
How to do it...	377
Exploring risk and return	380
How to do it...	380
See also	381
Examining the market with the non-parametric runs test	382
How to do it...	382
See also	384
Testing for random walks	385
How to do it...	385
See also	386
Determining market efficiency with autoregressive models	387
How to do it...	387
See also	389
Creating tables for a stock prices database	389
How to do it...	390
Populating the stock prices database	391
How to do it...	391
Optimizing an equal weights two-asset portfolio	396
How to do it...	397
See also	399
Chapter 8: Text Mining and Social Network Analysis	401
Introduction	401
Creating a categorized corpus	402
Getting ready	402
How to do it...	403
See also	405
Tokenizing news articles in sentences and words	405
Getting ready	405
How to do it...	405
See also	406
Stemming, lemmatizing, filtering, and TF-IDF scores	406
Getting ready	408
How to do it...	408
How it works	409

See also	410
Recognizing named entities	410
Getting ready	410
How to do it...	411
How it works	412
See also	412
Extracting topics with non-negative matrix factorization	412
How to do it...	413
How it works	414
See also	414
Implementing a basic terms database	414
How to do it...	415
How it works	418
See also	418
Computing social network density	418
Getting ready	419
How to do it...	419
See also	420
Calculating social network closeness centrality	420
Getting ready	420
How to do it...	420
See also	421
Determining the betweenness centrality	421
Getting ready	421
How to do it...	422
See also	422
Estimating the average clustering coefficient	423
Getting ready	423
How to do it...	423
See also	424
Calculating the assortativity coefficient of a graph	424
Getting ready	424
How to do it...	425
See also	425
Getting the clique number of a graph	425
Getting ready	426
How to do it...	426
See also	426

Creating a document graph with cosine similarity	427
How to do it...	428
See also	430
Chapter 9: Ensemble Learning and Dimensionality Reduction	431
Introduction	432
Recursively eliminating features	432
How to do it...	433
How it works	434
See also	434
Applying principal component analysis for dimension reduction	435
How to do it...	435
See also	436
Applying linear discriminant analysis for dimension reduction	437
How to do it...	437
See also	438
Stacking and majority voting for multiple models	438
How to do it...	439
See also	441
Learning with random forests	442
How to do it...	442
There's more...	444
See also	445
Fitting noisy data with the RANSAC algorithm	445
How to do it...	446
See also	448
Bagging to improve results	449
How to do it...	449
See also	451
Boosting for better learning	452
How to do it...	452
See also	454
Nesting cross-validation	455
How to do it...	455
See also	458
Reusing models with joblib	458
How to do it...	458
See also	459
Hierarchically clustering data	460
How to do it...	460
See also	461

Taking a Theano tour	462
Getting ready	462
How to do it...	462
See also	464
Chapter 10: Evaluating Classifiers, Regressors, and Clusters	465
Introduction	466
Getting classification straight with the confusion matrix	466
How to do it...	467
How it works	468
See also	469
Computing precision, recall, and F1-score	469
How to do it...	470
See also	472
Examining a receiver operating characteristic and the area under a curve	472
How to do it...	473
See also	474
Visualizing the goodness of fit	475
How to do it...	475
See also	476
Computing MSE and median absolute error	476
How to do it...	477
See also	479
Evaluating clusters with the mean silhouette coefficient	479
How to do it...	479
See also	481
Comparing results with a dummy classifier	482
How to do it...	482
See also	484
Determining MAPE and MPE	485
How to do it...	485
See also	487
Comparing with a dummy regressor	487
How to do it...	487
See also	489
Calculating the mean absolute error and the residual sum of squares	490
How to do it...	490
See also	492

Examining the kappa of classification	492
How to do it...	493
How it works	495
See also	495
Taking a look at the Matthews correlation coefficient	495
How to do it...	495
See also	497
Chapter 11: Analyzing Images	499
Introduction	499
Setting up OpenCV	500
Getting ready	500
How to do it...	501
How it works	502
There's more	503
Applying Scale-Invariant Feature Transform (SIFT)	503
Getting ready	503
How to do it...	503
See also	505
Detecting features with SURF	505
Getting ready	506
How to do it...	506
See also	507
Quantizing colors	507
Getting ready	508
How to do it...	508
See also	509
Denoising images	509
Getting ready	510
How to do it...	510
See also	511
Extracting patches from an image	511
Getting ready	512
How to do it...	512
See also	514
Detecting faces with Haar cascades	514
Getting ready	515
How to do it...	515
See also	517
Searching for bright stars	517
Getting ready	518

How to do it...	518
See also	520
Extracting metadata from images	521
Getting ready	521
How to do it...	521
See also	523
Extracting texture features from images	523
Getting ready	524
How to do it...	524
See also	526
Applying hierarchical clustering on images	526
How to do it...	526
See also	527
Segmenting images with spectral clustering	527
How to do it...	528
See also	529
Chapter 12: Parallelism and Performance	531
Introduction	531
Just-in-time compiling with Numba	533
Getting ready	533
How to do it...	533
How it works	534
See also	535
Speeding up numerical expressions with Numexpr	535
How to do it...	535
How it works	536
See also	536
Running multiple threads with the threading module	536
How to do it...	536
See also	539
Launching multiple tasks with the concurrent.futures module	540
How to do it...	540
See also	542
Accessing resources asynchronously with the asyncio module	543
How to do it...	543
See also	546
Distributed processing with execnet	546
Getting ready	547
How to do it...	547
See also	549

Profiling memory usage	550
Getting ready	550
How to do it...	550
See also	551
Calculating the mean, variance, skewness, and kurtosis on the fly	551
Getting ready	552
How to do it...	552
See also	556
Caching with a least recently used cache	556
Getting ready	556
How to do it...	556
See also	559
Caching HTTP requests	559
Getting ready	559
How to do it...	560
See also	560
Streaming counting with the Count-min sketch	561
How to do it...	562
See also	563
Harnessing the power of the GPU with OpenCL	564
Getting ready	564
How to do it...	564
See also	566
Appendix A: Glossary	567
Appendix B: Function Reference	573
IPython	573
Matplotlib	574
NumPy	575
pandas	576
Scikit-learn	577
SciPy	578
Seaborn	578
Statsmodels	579
Appendix C: Online Resources	581
IPython notebooks and open data	581
Mathematics and statistics	582
Presentations	582

Appendix D: Tips and Tricks for Command-Line and Miscellaneous Tools	585
IPython notebooks	585
Command-line tools	586
The alias command	586
Command-line history	587
Reproducible sessions	587
Docker tips	588

Module 3: Mastering Python Data Analysis

Preface	1
Chapter 1: Tools of the Trade	7
Before you start	7
Using the notebook interface	9
Imports	10
An example using the Pandas library	10
Summary	18
Chapter 2: Exploring Data	19
The General Social Survey	20
Obtaining the data	20
Reading the data	21
Univariate data	23
Histograms	23
Making things pretty	28
Characterization	29
Concept of statistical inference	32
Numeric summaries and boxplots	33
Relationships between variables – scatterplots	37
Summary	40
Chapter 3: Learning About Models	41
Models and experiments	41
The cumulative distribution function	42
Working with distributions	51
The probability density function	61
Where do models come from?	63
Multivariate distributions	68
Summary	70
Chapter 4: Regression	71
Introducing linear regression	72
Getting the dataset	73
Testing with linear regression	81
Multivariate regression	91
Adding economic indicators	91

Taking a step back	98
Logistic regression	100
Some notes	107
Summary	107
Chapter 5: Clustering	108
Introduction to cluster finding	109
Starting out simple – John Snow on cholera	110
K-means clustering	116
Suicide rate versus GDP versus absolute latitude	116
Hierarchical clustering analysis	122
Reading in and reducing the data	122
Hierarchical cluster algorithm	132
Summary	137
Chapter 6: Bayesian Methods	138
The Bayesian method	138
Credible versus confidence intervals	139
Bayes formula	139
Python packages	140
U.S. air travel safety record	141
Getting the NTSB database	141
Binning the data	147
Bayesian analysis of the data	150
Binning by month	158
Plotting coordinates	160
Cartopy	160
Mpl toolkits – basemap	162
Climate change – CO₂ in the atmosphere	163
Getting the data	164
Creating and sampling the model	166
Summary	173
Chapter 7: Supervised and Unsupervised Learning	174
Introduction to machine learning	174
Scikit-learn	175
Linear regression	176
Climate data	176
Checking with Bayesian analysis and OLS	181
Clustering	183
Seeds classification	188

Visualizing the data	189
Feature selection	194
Classifying the data	196
The SVC linear kernel	198
The SVC Radial Basis Function	199
The SVC polynomial	200
K-Nearest Neighbour	200
Random Forest	201
Choosing your classifier	202
Summary	203
Chapter 8: Time Series Analysis	204
Introduction	204
Pandas and time series data	206
Indexing and slicing	209
Resampling, smoothing, and other estimates	212
Stationarity	218
Patterns and components	220
Decomposing components	221
Differencing	227
Time series models	229
Autoregressive – AR	230
Moving average – MA	232
Selecting p and q	233
Automatic function	234
The (Partial) AutoCorrelation Function	234
Autoregressive Integrated Moving Average – ARIMA	235
Summary	236
Appendix: More on Jupyter Notebook and matplotlib Styles	238
Jupyter Notebook	238
Useful keyboard shortcuts	239
Command mode shortcuts	239
Edit mode shortcuts	239
Markdown cells	240
Notebook Python extensions	241
Installing the extensions	241
Codefolding	243
Collapsible headings	245
Help panel	247
Initialization cells	247
NbExtensions menu item	249

Ruler	249
Skip-traceback	250
Table of contents	252
Other Jupyter Notebook tips	254
External connections	255
Export	255
Additional file types	255
Matplotlib styles	256
Useful resources	261
General resources	261
Packages	262
Data repositories	264
Visualization of data	265
Summary	266
Index	267

Module 1

Getting Started with Python Data Analysis

Learn to use powerful Python libraries for effective data processing and analysis

1

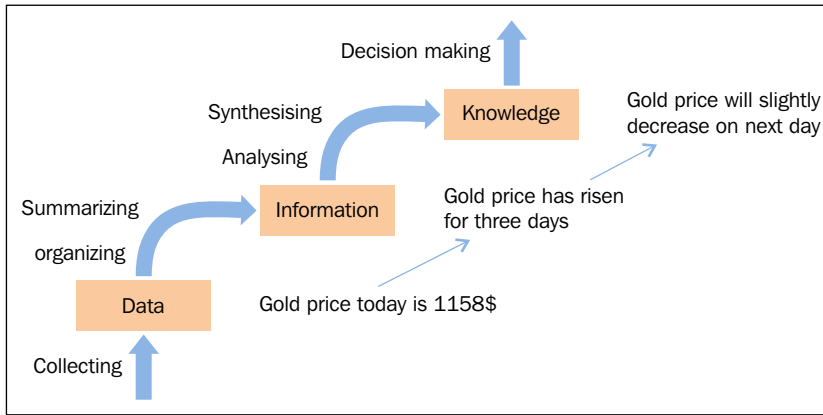
Introducing Data Analysis and Libraries

Data is raw information that can exist in any form, usable or not. We can easily get data everywhere in our lives; for example, the price of gold on the day of writing was \$ 1.158 per ounce. This does not have any meaning, except describing the price of gold. This also shows that data is useful based on context.

With the relational data connection, information appears and allows us to expand our knowledge beyond the range of our senses. When we possess gold price data gathered over time, one piece of information we might have is that the price has continuously risen from \$1.152 to \$1.158 over three days. This could be used by someone who tracks gold prices.

Knowledge helps people to create value in their lives and work. This value is based on information that is organized, synthesized, or summarized to enhance comprehension, awareness, or understanding. It represents a state or potential for action and decisions. When the price of gold continuously increases for three days, it will likely decrease on the next day; this is useful knowledge.

The following figure illustrates the steps from data to knowledge; we call this process, the data analysis process and we will introduce it in the next section:

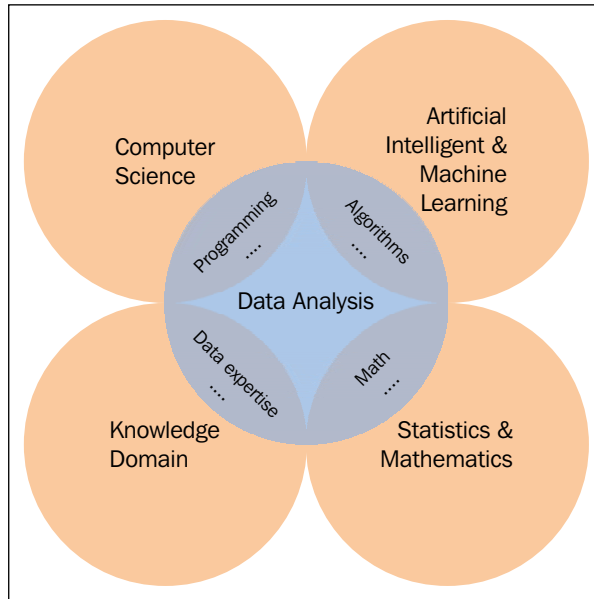


In this chapter, we will cover the following topics:

- Data analysis and process
- An overview of libraries in data analysis using different programming languages
- Common Python data analysis libraries

Data analysis and processing

Data is getting bigger and more diverse every day. Therefore, analyzing and processing data to advance human knowledge or to create value is a big challenge. To tackle these challenges, you will need domain knowledge and a variety of skills, drawing from areas such as computer science, **artificial intelligence (AI)** and **machine learning (ML)**, statistics and mathematics, and knowledge domain, as shown in the following figure:



Let's go through data analysis and its domain knowledge:

- **Computer science:** We need this knowledge to provide abstractions for efficient data processing. Basic Python programming experience is required to follow the next chapters. We will introduce Python libraries used in data analysis.
- **Artificial intelligence and machine learning:** If computer science knowledge helps us to program data analysis tools, artificial intelligence and machine learning help us to model the data and learn from it in order to build smart products.
- **Statistics and mathematics:** We cannot extract useful information from raw data if we do not use statistical techniques or mathematical functions.
- **Knowledge domain:** Besides technology and general techniques, it is important to have an insight into the specific domain. What do the data fields mean? What data do we need to collect? Based on the expertise, we explore and analyze raw data by applying the above techniques, step by step.

Data analysis is a process composed of the following steps:

- **Data requirements:** We have to define what kind of data will be collected based on the requirements or problem analysis. For example, if we want to detect a user's behavior while reading news on the internet, we should be aware of visited article links, dates and times, article categories, and the time the user spends on different pages.
- **Data collection:** Data can be collected from a variety of sources: mobile, personal computer, camera, or recording devices. It may also be obtained in different ways: communication, events, and interactions between person and person, person and device, or device and device. Data appears whenever and wherever in the world. The problem is how we can find and gather it to solve our problem? This is the mission of this step.
- **Data processing:** Data that is initially obtained must be processed or organized for analysis. This process is performance-sensitive. How fast can we create, insert, update, or query data? When building a real product that has to process big data, we should consider this step carefully. What kind of database should we use to store data? What kind of data structure, such as analysis, statistics, or visualization, is suitable for our purposes?
- **Data cleaning:** After being processed and organized, the data may still contain duplicates or errors. Therefore, we need a cleaning step to reduce those situations and increase the quality of the results in the following steps. Common tasks include record matching, deduplication, and column segmentation. Depending on the type of data, we can apply several types of data cleaning. For example, a user's history of visits to a news website might contain a lot of duplicate rows, because the user might have refreshed certain pages many times. For our specific issue, these rows might not carry any meaning when we explore the user's behavior so we should remove them before saving it to our database. Another situation we may encounter is click fraud on news—someone just wants to improve their website ranking or sabotage a website. In this case, the data will not help us to explore a user's behavior. We can use thresholds to check whether a visit page event comes from a real person or from malicious software.
- **Exploratory data analysis:** Now, we can start to analyze data through a variety of techniques referred to as exploratory data analysis. We may detect additional problems in data cleaning or discover requests for further data. Therefore, these steps may be iterative and repeated throughout the whole data analysis process. Data visualization techniques are also used to examine the data in graphs or charts. Visualization often facilitates understanding of data sets, especially if they are large or high-dimensional.

- **Modelling and algorithms:** A lot of mathematical formulas and algorithms may be applied to detect or predict useful knowledge from the raw data. For example, we can use similarity measures to cluster users who have exhibited similar news-reading behavior and recommend articles of interest to them next time. Alternatively, we can detect users' genders based on their news reading behavior by applying classification models such as the **Support Vector Machine (SVM)** or linear regression. Depending on the problem, we may use different algorithms to get an acceptable result. It can take a lot of time to evaluate the accuracy of the algorithms and choose the best one to implement for a certain product.
- **Data product:** The goal of this step is to build data products that receive data input and generate output according to the problem requirements. We will apply computer science knowledge to implement our selected algorithms as well as manage the data storage.

An overview of the libraries in data analysis

There are numerous data analysis libraries that help us to process and analyze data. They use different programming languages, and have different advantages and disadvantages of solving various data analysis problems. Now, we will introduce some common libraries that may be useful for you. They should give you an overview of the libraries in the field. However, the rest of this book focuses on Python-based libraries.

Some of the libraries that use the Java language for data analysis are as follows:

- **Weka:** This is the library that I became familiar with the first time I learned about data analysis. It has a graphical user interface that allows you to run experiments on a small dataset. This is great if you want to get a feel for what is possible in the data processing space. However, if you build a complex product, I think it is not the best choice, because of its performance, sketchy API design, non-optimal algorithms, and little documentation (<http://www.cs.waikato.ac.nz/ml/weka/>).

- **Mallet:** This is another Java library that is used for statistical natural language processing, document classification, clustering, topic modeling, information extraction, and other machine-learning applications on text. There is an add-on package for Mallet, called GRMM, that contains support for inference in general, graphical models, and training of **Conditional random fields (CRF)** with arbitrary graphical structures. In my experience, the library performance and the algorithms are better than Weka. However, its only focus is on text-processing problems. The reference page is at <http://mallet.cs.umass.edu/>.
- **Mahout:** This is Apache's machine-learning framework built on top of Hadoop; its goal is to build a scalable machine-learning library. It looks promising, but comes with all the baggage and overheads of Hadoop. The homepage is at <http://mahout.apache.org/>.
- **Spark:** This is a relatively new Apache project, supposedly up to a hundred times faster than Hadoop. It is also a scalable library that consists of common machine-learning algorithms and utilities. Development can be done in Python as well as in any JVM language. The reference page is at <https://spark.apache.org/docs/1.5.0/mllib-guide.html>.

Here are a few libraries that are implemented in C++:

- **Vowpal Wabbit:** This library is a fast, out-of-core learning system sponsored by Microsoft Research and, previously, Yahoo! Research. It has been used to learn a tera-feature (1012) dataset on 1,000 nodes in one hour. More information can be found in the publication at <http://arxiv.org/abs/1110.4198>.
- **MultiBoost:** This package is a multiclass, multi label, and multitask classification boosting software implemented in C++. If you use this software, you should refer to the paper published in 2012 in the *Journal Machine Learning Research*, *MultiBoost: A Multi-purpose Boosting Package*, D.Benbouzid, R. Busa-Fekete, N. Casagrande, F.-D. Collin, and B. Kégl.
- **MLpack:** This is also a C++ machine-learning library, developed by the **Fundamental Algorithmic and Statistical Tools Laboratory (FASTLab)** at Georgia Tech. It focusses on scalability, speed, and ease-of-use, and was presented at the BigLearning workshop of NIPS 2011. Its homepage is at <http://www.mlpack.org/about.html>.
- **Caffe:** The last C++ library we want to mention is Caffe. This is a deep learning framework made with expression, speed, and modularity in mind. It is developed by the **Berkeley Vision and Learning Center (BVLC)** and community contributors. You can find more information about it at <http://caffe.berkeleyvision.org/>.

Other libraries for data processing and analysis are as follows:

- **Statsmodels:** This is a great Python library for statistical modeling and is mainly used for predictive and exploratory analysis.
- **Modular toolkit for data processing (MDP):** This is a collection of supervised and unsupervised learning algorithms and other data processing units that can be combined into data processing sequences and more complex feed-forward network architectures (<http://mdp-toolkit.sourceforge.net/index.html>).
- **Orange:** This is an open source data visualization and analysis for novices and experts. It is packed with features for data analysis and has add-ons for bioinformatics and text mining. It contains an implementation of self-organizing maps, which sets it apart from the other projects as well (<http://orange.biolab.si/>).
- **Mirador:** This is a tool for the visual exploration of complex datasets, supporting Mac and Windows. It enables users to discover correlation patterns and derive new hypotheses from data (<http://orange.biolab.si/>).
- **RapidMiner:** This is another GUI-based tool for data mining, machine learning, and predictive analysis (<https://rapidminer.com/>).
- **Theano:** This bridges the gap between Python and lower-level languages. Theano gives very significant performance gains, particularly for large matrix operations, and is, therefore, a good choice for deep learning models. However, it is not easy to debug because of the additional compilation layer.
- **Natural language processing toolkit (NLTK):** This is written in Python with very unique and salient features.

Here, I could not list all libraries for data analysis. However, I think the above libraries are enough to take a lot of your time to learn and build data analysis applications. I hope you will enjoy them after reading this book.

Python libraries in data analysis

Python is a multi-platform, general-purpose programming language that can run on Windows, Linux/Unix, and Mac OS X, and has been ported to Java and .NET virtual machines as well. It has a powerful standard library. In addition, it has many libraries for data analysis: Pylearn2, Hebel, Pybrain, Pattern, MontePython, and MILK. In this book, we will cover some common Python data analysis libraries such as Numpy, Pandas, Matplotlib, PyMongo, and scikit-learn. Now, to help you get started, I will briefly present an overview of each library for those who are less familiar with the scientific Python stack.

NumPy

One of the fundamental packages used for scientific computing in Python is Numpy. Among other things, it contains the following:

- A powerful N-dimensional array object
- Sophisticated (broadcasting) functions for performing array computations
- Tools for integrating C/C++ and Fortran code
- Useful linear algebra operations, Fourier transformations, and random number capabilities

Besides this, it can also be used as an efficient multidimensional container of generic data. Arbitrary data types can be defined and integrated with a wide variety of databases.

Pandas

Pandas is a Python package that supports rich data structures and functions for analyzing data, and is developed by the PyData Development Team. It is focused on the improvement of Python's data libraries. Pandas consists of the following things:

- A set of labeled array data structures; the primary of which are Series, DataFrame, and Panel
- Index objects enabling both simple axis indexing and multilevel/hierarchical axis indexing
- An intergraded group by engine for aggregating and transforming datasets
- Date range generation and custom date offsets
- Input/output tools that load and save data from flat files or PyTables/HDF5 format
- Optimal memory versions of the standard data structures
- Moving window statistics and static and moving window linear/panel regression

Due to these features, Pandas is an ideal tool for systems that need complex data structures or high-performance time series functions such as financial data analysis applications.

Matplotlib

Matplotlib is the single most used Python package for 2D-graphics. It provides both a very quick way to visualize data from Python and publication-quality figures in many formats: line plots, contour plots, scatter plots, and Basemap plots. It comes with a set of default settings, but allows customization of all kinds of properties. However, we can easily create our chart with the defaults of almost every property in Matplotlib.

PyMongo

MongoDB is a type of NoSQL database. It is highly scalable, robust, and perfect to work with JavaScript-based web applications, because we can store data as JSON documents and use flexible schemas.

PyMongo is a Python distribution containing tools for working with MongoDB. Many tools have also been written for working with PyMongo to add more features such as MongoKit, Humongolus, MongoAlchemy, and Ming.

The scikit-learn library

The scikit-learn is an open source machine-learning library using the Python programming language. It supports various machine learning models, such as classification, regression, and clustering algorithms, interoperated with the Python numerical and scientific libraries NumPy and SciPy. The latest scikit-learn version is 0.16.1, published in April 2015.

Summary

In this chapter, we presented three main points. Firstly, we figured out the relationship between raw data, information and knowledge. Due to its contribution to our lives, we continued to discuss an overview of data analysis and processing steps in the second section. Finally, we introduced a few common supported libraries that are useful for practical data analysis applications. Among those, in the next chapters, we will focus on Python libraries in data analysis.

Practice exercise

The following table describes users' rankings on Snow White movies:

UserID	Sex	Location	Ranking
A	Male	Philips	4
B	Male	VN	2
C	Male	Canada	1
D	Male	Canada	2
E	Female	VN	5
F	Female	NY	4

Exercise 1: What information can we find in this table? What kind of knowledge can we derive from it?

Exercise 2: Based on the data analysis process in this chapter, try to define the data requirements and analysis steps needed to predict whether user B likes Maleficent movies or not.

2

NumPy Arrays and Vectorized Computation

NumPy is the fundamental package supported for presenting and computing data with high performance in Python. It provides some interesting features as follows:

- Extension package to Python for multidimensional arrays (`ndarrays`), various derived objects (such as masked arrays), matrices providing vectorization operations, and broadcasting capabilities. Vectorization can significantly increase the performance of array computations by taking advantage of **Single Instruction Multiple Data (SIMD)** instruction sets in modern CPUs.
- Fast and convenient operations on arrays of data, including mathematical manipulation, basic statistical operations, sorting, selecting, linear algebra, random number generation, discrete Fourier transforms, and so on.
- Efficiency tools that are closer to hardware because of integrating C/C++/Fortran code.

NumPy is a good starting package for you to get familiar with arrays and array-oriented computing in data analysis. Also, it is the basic step to learn other, more effective tools such as Pandas, which we will see in the next chapter. We will be using NumPy version 1.9.1.

NumPy arrays

An array can be used to contain values of a data object in an experiment or simulation step, pixels of an image, or a signal recorded by a measurement device. For example, the latitude of the Eiffel Tower, Paris is 48.858598 and the longitude is 2.294495. It can be presented in a NumPy array object as `p`:

```
>>> import numpy as np
>>> p = np.array([48.858598, 2.294495])
>>> p
Output: array([48.858598, 2.294495])
```

This is a manual construction of an array using the `np.array` function. The standard convention to import NumPy is as follows:

```
>>> import numpy as np
```

You can, of course, put `from numpy import *` in your code to avoid having to write `np`. However, you should be careful with this habit because of the potential code conflicts (further information on code conventions can be found in the *Python Style Guide*, also known as **PEP8**, at <https://www.python.org/dev/peps/pep-0008/>).

There are two requirements of a NumPy array: a fixed size at creation and a uniform, fixed data type, with a fixed size in memory. The following functions help you to get information on the `p` matrix:

```
>>> p.ndim      # getting dimension of array p
1
>>> p.shape     # getting size of each array dimension
(2,)
>>> len(p)      # getting dimension length of array p
2
>>> p.dtype     # getting data type of array p
dtype('float64')
```

Data types

There are five basic numerical types including Booleans (`bool`), integers (`int`), unsigned integers (`uint`), floating point (`float`), and complex. They indicate how many bits are needed to represent elements of an array in memory. Besides that, NumPy also has some types, such as `intc` and `intp`, that have different bit sizes depending on the platform.

See the following table for a listing of NumPy's supported data types:

Type	Type code	Description	Range of value
bool		Boolean stored as a byte	True/False
intc		Similar to C int (int32 or int 64)	
intp		Integer used for indexing (same as C size_t)	
int8, uint8	i1, u1	Signed and unsigned 8-bit integer types	int8: (-128 to 127) uint8: (0 to 255)
int16, uint16	i2, u2	Signed and unsigned 16-bit integer types	int16: (-32768 to 32767) uint16: (0 to 65535)
int32, uint32	i4, u4	Signed and unsigned 32-bit integer types	int32: (-2147483648 to 2147483647) uint32: (0 to 4294967295)
int64, uint64	i8, u8	Signed and unsigned 64-bit integer types	int64: (-9223372036854775808 to 9223372036854775807) uint64: (0 to 18446744073709551615)
float16	f2	Half precision float: sign bit, 5 bits exponent, and 10b bits mantissa	
float32	f4 / f	Single precision float: sign bit, 8 bits exponent, and 23 bits mantissa	
float64	f8 / d	Double precision float: sign bit, 11 bits exponent, and 52 bits mantissa	
complex64, complex128, complex256	c8, c16, c32	Complex numbers represented by two 32-bit, 64-bit, and 128-bit floats	
object	O	Python object type	
string_	S	Fixed-length string type	Declare a string dtype with length 10, using S10
unicode_	U	Fixed-length Unicode type	Similar to string_ example, we have 'U10'

We can easily convert or cast an array from one `dtype` to another using the `astype` method:

```
>>> a = np.array([1, 2, 3, 4])
>>> a.dtype
dtype('int64')
>>> float_b = a.astype(np.float64)
>>> float_b.dtype
dtype('float64')
```



The `astype` function will create a new array with a copy of data from an old array, even though the new `dtype` is similar to the old one.

Array creation

There are various functions provided to create an array object. They are very useful for us to create and store data in a multidimensional array in different situations.

Now, in the following table we will summarize some of NumPy's common functions and their use by examples for array creation:

Function	Description	Example
<code>empty</code> , <code>empty_like</code>	Create a new array of the given shape and type, without initializing elements	<pre>>>> np.empty([3,2], dtype=np.float64) array([[0., 0.], [0., 0.], [0., 0.]]) >>> a = np.array([[1, 2], [4, 3]]) >>> np.empty_like(a) array([[0, 0], [0, 0]])</pre>
<code>eye</code> , <code>identity</code>	Create a NxN identity matrix with ones on the diagonal and zero elsewhere	<pre>>>> np.eye(2, dtype=np.int) array([[1, 0], [0, 1]])</pre>
<code>ones</code> , <code>ones_like</code>	Create a new array with the given shape and type, filled with 1s for all elements	<pre>>>> np.ones(5) array([1., 1., 1., 1., 1.]) >>> np.ones(4, dtype=np.int) array([1, 1, 1, 1]) >>> x = np.array([[0,1,2], [3,4,5]]) >>> np.ones_like(x) array([[1, 1, 1], [1, 1, 1]])</pre>

Function	Description	Example
zeros, zeros_like	This is similar to ones, ones_like, but initializing elements with 0s instead	<pre>>>> np.zeros(5) array([0., 0., 0., 0., 0.]) >>> np.zeros(4, dtype=np.int) array([0, 0, 0, 0]) >>> x = np.array([[0, 1, 2], [3, 4, 5]]) >>> np.zeros_like(x) array([[0, 0, 0], [0, 0, 0]])</pre>
arange	Create an array with even spaced values in a given interval	<pre>>>> np.arange(2, 5) array([2, 3, 4]) >>> np.arange(4, 12, 5) array([4, 9])</pre>
full, full_like	Create a new array with the given shape and type, filled with a selected value	<pre>>>> np.full((2,2), 3, dtype=np.int) array([[3, 3], [3, 3]]) >>> x = np.ones(3) >>> np.full_like(x, 2) array([2., 2., 2.])</pre>
array	Create an array from the existing data	<pre>>>> np.array([[1.1, 2.2, 3.3], [4.4, 5.5, 6.6]]) array([1.1, 2.2, 3.3], [4.4, 5.5, 6.6])</pre>
asarray	Convert the input to an array	<pre>>>> a = [3.14, 2.46] >>> np.asarray(a) array([3.14, 2.46])</pre>
copy	Return an array copy of the given object	<pre>>>> a = np.array([[1, 2], [3, 4]]) >>> np.copy(a) array([[1, 2], [3, 4]])</pre>
fromstring	Create 1-D array from a string or text	<pre>>>> np.fromstring('3.14 2.17', dtype=np.float, sep=' ') array([3.14, 2.17])</pre>

Indexing and slicing

As with other Python sequence types, such as lists, it is very easy to access and assign a value of each array's element:

```
>>> a = np.arange(7)
>>> a
array([0, 1, 2, 3, 4, 5, 6])
>>> a[1], a[4], a[-1]
(1, 4, 6)
```



In Python, array indices start at 0. This is in contrast to Fortran or Matlab, where indices begin at 1.

As another example, if our array is multidimensional, we need tuples of integers to index an item:

```
>>> a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> a[0, 2]          # first row, third column
3
>>> a[0, 2] = 10
>>> a
array([[1, 2, 10], [4, 5, 6], [7, 8, 9]])
>>> b = a[2]
>>> b
array([7, 8, 9])
>>> c = a[:2]
>>> c
array([[1, 2, 10], [4, 5, 6]])
```

We call `b` and `c` as array slices, which are views on the original one. It means that the data is not copied to `b` or `c`, and whenever we modify their values, it will be reflected in the array `a` as well:

```
>>> b[-1] = 11
>>> a
array([[1, 2, 10], [4, 5, 6], [7, 8, 11]])
```



We use a colon (`:`) character to take the entire axis when we omit the index number.

Fancy indexing

Besides indexing with slices, NumPy also supports indexing with Boolean or integer arrays (masks). This method is called **fancy indexing**. It creates copies, not views.

First, we take a look at an example of indexing with a Boolean mask array:

```
>>> a = np.array([3, 5, 1, 10])
>>> b = (a % 5 == 0)
>>> b
array([False,  True, False,  True], dtype=bool)
>>> c = np.array([[0, 1], [2, 3], [4, 5], [6, 7]])
>>> c[b]
array([[2, 3], [6, 7]])
```

The second example is an illustration of using integer masks on arrays:

```
>>> a = np.array([[1, 2, 3, 4],
                  [5, 6, 7, 8],
                  [9, 10, 11, 12],
                  [13, 14, 15, 16]])
>>> a[[2, 1]]
array([[9, 10, 11, 12], [5, 6, 7, 8]])
>>> a[[-2, -1]]          # select rows from the end
array([[ 9, 10, 11, 12], [13, 14, 15, 16]])
>>> a[[2, 3], [0, 1]]    # take elements at (2, 0) and (3, 1)
array([9, 14])
```



The mask array must have the same length as the axis that it's indexing.



Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Numerical operations on arrays

We are getting familiar with creating and accessing `ndarrays`. Now, we continue to the next step, applying some mathematical operations to array data without writing any for loops, of course, with higher performance.

Scalar operations will propagate the value to each element of the array:

```
>>> a = np.ones(4)
>>> a * 2
array([2., 2., 2., 2.])
>>> a + 3
array([4., 4., 4., 4.])
```

All arithmetic operations between arrays apply the operation element wise:

```
>>> a = np.ones([2, 4])
>>> a * a
array([[1., 1., 1., 1.], [1., 1., 1., 1.]])
>>> a + a
array([[2., 2., 2., 2.], [2., 2., 2., 2.]])
```

Also, here are some examples of comparisons and logical operations:

```
>>> a = np.array([1, 2, 3, 4])
>>> b = np.array([1, 1, 5, 3])
>>> a == b
array([True, False, False, False], dtype=bool)

>>> np.array_equal(a, b)      # array-wise comparison
False

>>> c = np.array([1, 0])
>>> d = np.array([1, 1])
>>> np.logical_and(c, d)      # logical operations
array([True, False])
```

Array functions

Many helpful array functions are supported in NumPy for analyzing data. We will list some part of them that are common in use. Firstly, the transposing function is another kind of reshaping form that returns a view on the original data array without copying anything:

```
>>> a = np.array([[0, 5, 10], [20, 25, 30]])
>>> a.reshape(3, 2)
array([[0, 5], [10, 20], [25, 30]])
>>> a.T
array([[0, 20], [5, 25], [10, 30]])
```

In general, we have the `swapaxes` method that takes a pair of axis numbers and returns a view on the data, without making a copy:

```
>>> a = np.array([[[0, 1, 2], [3, 4, 5]],
                  [[6, 7, 8], [9, 10, 11]]])
>>> a.swapaxes(1, 2)
array([[[0, 3],
        [1, 4],
        [2, 5]],
       [[6, 9],
        [7, 10],
        [8, 11]]])
```

The transposing function is used to do matrix computations; for example, computing the inner matrix product $X^T \cdot X$ using `np.dot`:

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> np.dot(a.T, a)
array([[17, 22, 27],
       [22, 29, 36],
       [27, 36, 45]])
```

Sorting data in an array is also an important demand in processing data. Let's take a look at some sorting functions and their use:

```
>>> a = np.array ([[6, 34, 1, 6], [0, 5, 2, -1]])

>>> np.sort(a)      # sort along the last axis
array([[1, 6, 6, 34], [-1, 0, 2, 5]])

>>> np.sort(a, axis=0)  # sort along the first axis
array([[0, 5, 1, -1], [6, 34, 2, 6]])

>>> b = np.argsort(a)   # fancy indexing of sorted array
>>> b
array([[2, 0, 3, 1], [3, 0, 2, 1]])
>>> a[b[0][b[0]]]
array([1, 6, 6, 34])

>>> np.argmax(a)      # get index of maximum element
1
```

See the following table for a listing of array functions:

Function	Description	Example
sin, cos, tan, cosh, sinh, tanh, arcsin, arctan, deg2rad	Trigonometric and hyperbolic functions	<pre>>>> a = np.array([0., 30., 45.]) >>> np.sin(a * np.pi / 180) array([0., 0.5, 0.7071678])</pre>
around, round, rint, fix, floor, ceil, trunc	Rounding elements of an array to the given or nearest number	<pre>>>> a = np.array([0.34, 1.65]) >>> np.round(a) array([0., 2.])</pre>
sqrt, square, exp, expm1, exp2, log, log10, log1p, logaddexp	Computing the exponents and logarithms of an array	<pre>>>> np.exp(np.array([2.25, 3.16])) array([9.4877, 23.5705])</pre>

Function	Description	Example
add, negative, multiply, divide, power, subtract, mod, modf, remainder	Set of arithmetic functions on arrays	<pre>>>> a = np.arange(6) >>> x1 = a.reshape(2,3) >>> x2 = np.arange(3) >>> np.multiply(x1, x2) array([[0,1,4],[0,4,10]])</pre>
greater, greater_equal, less, less_equal, equal, not_equal	Perform elementwise comparison: >, >=, <, <=, ==, !=	<pre>>>> np.greater(x1, x2) array([[False, False, False], [True, True, True]], dtype = bool)</pre>

Data processing using arrays

With the NumPy package, we can easily solve many kinds of data processing tasks without writing complex loops. It is very helpful for us to control our code as well as the performance of the program. In this part, we want to introduce some mathematical and statistical functions.

See the following table for a listing of mathematical and statistical functions:

Function	Description	Example
sum	Calculate the sum of all the elements in an array or along the axis	<pre>>>> a = np.array([[2,4], [3,5]]) >>> np.sum(a, axis=0) array([5, 9])</pre>
prod	Compute the product of array elements over the given axis	<pre>>>> np.prod(a, axis=1) array([8, 15])</pre>
diff	Calculate the discrete difference along the given axis	<pre>>>> np.diff(a, axis=0) array([[1,1]])</pre>
gradient	Return the gradient of an array	<pre>>>> np.gradient(a) (array([[1., 1.], [1., 1.]]), array([[2., 2.], [2., 2.]])</pre>
cross	Return the cross product of two arrays	<pre>>>> b = np.array([[1,2], [3,4]]) >>> np.cross(a,b) array([0, -3])</pre>

Function	Description	Example
std, var	Return standard deviation and variance of arrays	<pre>>>> np.std(a) 1.1180339 >>> np.var(a) 1.25</pre>
mean	Calculate arithmetic mean of an array	<pre>>>> np.mean(a) 3.5</pre>
where	Return elements, either from x or y, that satisfy a condition	<pre>>>> np.where([[True, True], [False, True]], [[1,2], [3,4]], [[5,6], [7,8]]) array([[1,2], [7, 4]])</pre>
unique	Return the sorted unique values in an array	<pre>>>> id = np.array(['a', 'b', 'c', 'c', 'd']) >>> np.unique(id) array(['a', 'b', 'c', 'd'], dtype=' S1')</pre>
intersect1d	Compute the sorted and common elements in two arrays	<pre>>>> a = np.array(['a', 'b', 'a', 'c', 'd', 'c']) >>> b = np.array(['a', 'xyz', 'klm', 'd']) >>> np.intersect1d(a,b) array(['a', 'd'], dtype=' S3')</pre>

Loading and saving data

We can also save and load data to and from a disk, either in text or binary format, by using different supported functions in NumPy package.

Saving an array

Arrays are saved by default in an uncompressed raw binary format, with the file extension `.npy` by the `np.save` function:

```
>>> a = np.array([[0, 1, 2], [3, 4, 5]])
>>> np.save('test1.npy', a)
```



The library automatically assigns the `.npy` extension, if we omit it.

If we want to store several arrays into a single file in an uncompressed `.npz` format, we can use the `np.savez` function, as shown in the following example:

```
>>> a = np.arange(4)
>>> b = np.arange(7)
>>> np.savez('test2.npz', arr0=a, arr1=b)
```

The `.npz` file is a zipped archive of files named after the variables they contain. When we load an `.npz` file, we get back a dictionary-like object that can be queried for its lists of arrays:

```
>>> dic = np.load('test2.npz')
>>> dic['arr0']
array([0, 1, 2, 3])
```

Another way to save array data into a file is using the `np.savetxt` function that allows us to set format properties in the output file:

```
>>> x = np.arange(4)
>>> # e.g., set comma as separator between elements
>>> np.savetxt('test3.out', x, delimiter=',')
```

Loading an array

We have two common functions such as `np.load` and `np.loadtxt`, which correspond to the saving functions, for loading an array:

```
>>> np.load('test1.npy')
array([[0, 1, 2], [3, 4, 5]])
>>> np.loadtxt('test3.out', delimiter=',')
array([0., 1., 2., 3.])
```

Similar to the `np.savetxt` function, the `np.loadtxt` function also has a lot of options for loading an array from a text file.

Linear algebra with NumPy

Linear algebra is a branch of mathematics concerned with vector spaces and the mappings between those spaces. NumPy has a package called **linalg** that supports powerful linear algebra functions. We can use these functions to find eigenvalues and eigenvectors or to perform singular value decomposition:

```
>>> A = np.array([[1, 4, 6],
                  [5, 2, 2],
                  [-1, 6, 8]])
>>> w, v = np.linalg.eig(A)
>>> w                                # eigenvalues
array([-0.111 + 1.5756j, -0.111 - 1.5756j, 11.222+0.j])
>>> v                                # eigenvector
array([[ -0.0981 + 0.2726j, -0.0981 - 0.2726j, 0.5764+0.j],
       [ 0.7683+0.j, 0.7683-0.j, 0.4591+0.j],
       [-0.5656 - 0.0762j, -0.5656 + 0.00763j, 0.6759+0.j]])
```

The function is implemented using the geev Lapack routines that compute the eigenvalues and eigenvectors of general square matrices.

Another common problem is solving linear systems such as $Ax = b$ with A as a matrix and x and b as vectors. The problem can be solved easily using the `numpy.linalg.solve` function:

```
>>> A = np.array([[1, 4, 6], [5, 2, 2], [-1, 6, 8]])
>>> b = np.array([[1], [2], [3]])
>>> x = np.linalg.solve(A, b)
>>> x
array([[ -1.77635e-16], [2.5], [-1.5]])
```

The following table will summarise some commonly used functions in the `numpy.linalg` package:

Function	Description	Example
<code>dot</code>	Calculate the dot product of two arrays	<pre>>>> a = np.array([[1, 0], [0, 1]]) >>> b = np.array([[4, 1], [2, 2]]) >>> np.dot(a,b) array([[4, 1], [2, 2]])</pre>

Function	Description	Example
<code>inner, outer</code>	Calculate the inner and outer product of two arrays	<pre>>>> a = np.array([1, 1, 1]) >>> b = np.array([3, 5, 1]) >>> np.inner(a,b) 9</pre>
<code>linalg.norm</code>	Find a matrix or vector norm	<pre>>>> a = np.arange(3) >>> np.linalg.norm(a) 2.23606</pre>
<code>linalg.det</code>	Compute the determinant of an array	<pre>>>> a = np.array([[1,2],[3,4]]) >>> np.linalg.det(a) -2.0</pre>
<code>linalg.inv</code>	Compute the inverse of a matrix	<pre>>>> a = np.array([[1,2],[3,4]]) >>> np.linalg.inv(a) array([[-2., 1.], [1.5, -0.5]])</pre>
<code>linalg.qr</code>	Calculate the QR decomposition	<pre>>>> a = np.array([[1,2],[3,4]]) >>> np.linalg.qr(a) (array([[0.316, 0.948], [0.948, 0.316]]), array([[3.162, 4.427], [0., 0.632]]))</pre>
<code>linalg.cond</code>	Compute the condition number of a matrix	<pre>>>> a = np.array([[1,3],[2,4]]) >>> np.linalg.cond(a) 14.933034</pre>
<code>trace</code>	Compute the sum of the diagonal element	<pre>>>> np.trace(np.arange(6)). reshape(2,3) 4</pre>

NumPy random numbers

An important part of any simulation is the ability to generate random numbers. For this purpose, NumPy provides various routines in the submodule `random`. It uses a particular algorithm, called the Mersenne Twister, to generate pseudorandom numbers.

First, we need to define a seed that makes the random numbers predictable. When the value is reset, the same numbers will appear every time. If we do not assign the seed, NumPy automatically selects a random seed value based on the system's random number generator device or on the clock:

```
>>> np.random.seed(20)
```

An array of random numbers in the `[0.0, 1.0]` interval can be generated as follows:

```
>>> np.random.rand(5)
array([0.5881308, 0.89771373, 0.89153073, 0.81583748,
       0.03588959])
>>> np.random.rand(5)
array([0.69175758, 0.37868094, 0.51851095, 0.65795147,
       0.19385022])
```

```
>>> np.random.seed(20)      # reset seed number
>>> np.random.rand(5)
array([0.5881308, 0.89771373, 0.89153073, 0.81583748,
       0.03588959])
```

If we want to generate random integers in the half-open interval `[min, max]`, we can use the `randint(min, max, length)` function:

```
>>> np.random.randint(10, 20, 5)
array([17, 12, 10, 16, 18])
```

NumPy also provides for many other distributions, including the Beta, binomial, chi-square, Dirichlet, exponential, F, Gamma, geometric, or Gumbel.

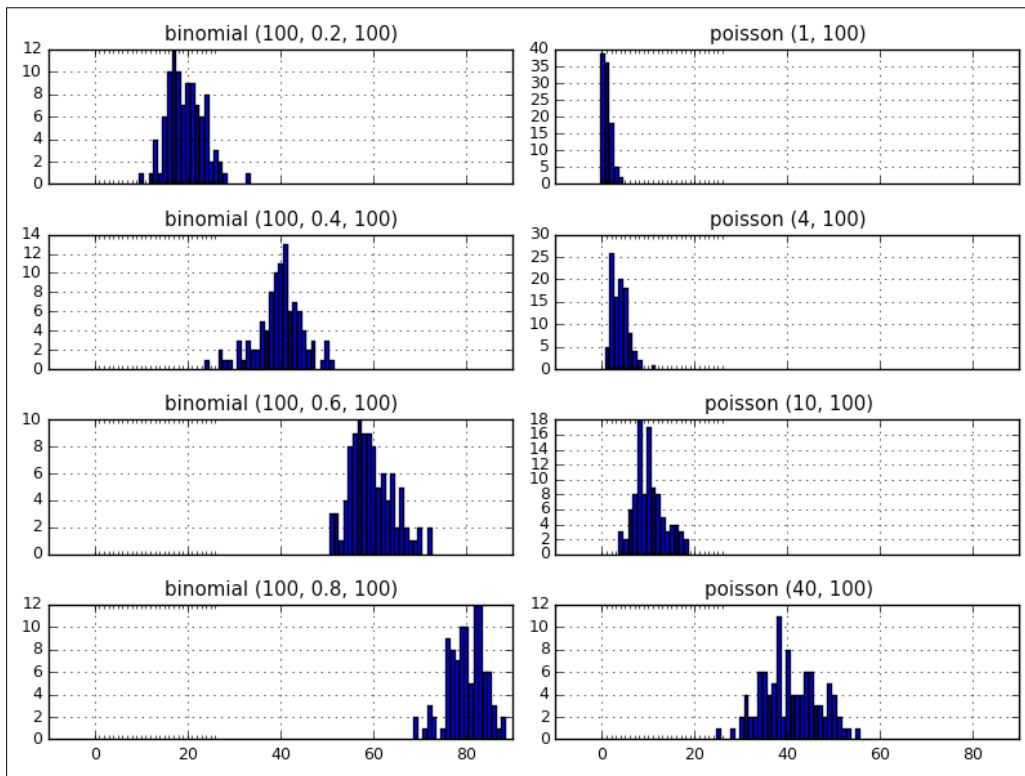
The following table will list some distribution functions and give examples for generating random numbers:

Function	Description	Example
binomial	Draw samples from a binomial distribution (n: number of trials, p: probability)	<pre>>>> n, p = 100, 0.2 >>> np.random.binomial(n, p, 3) array([17, 14, 23])</pre>
dirichlet	Draw samples using a Dirichlet distribution	<pre>>>> np.random.dirichlet(alpha=(2,3), size=3) array([[0.519, 0.480], [0.639, 0.36], [0.838, 0.161]])</pre>
poisson	Draw samples from a Poisson distribution	<pre>>>> np.random.poisson(lam=2, size=2) array([4, 1])</pre>
normal	Draw samples using a normal Gaussian distribution	<pre>>>> np.random.normal(loc=2.5, scale=0.3, size=3) array([2.4436, 2.849, 2.741])</pre>
uniform	Draw samples using a uniform distribution	<pre>>>> np.random.uniform(low=0.5, high=2.5, size=3) array([1.38, 1.04, 2.19])</pre>

We can also use the random number generation to shuffle items in a list. Sometimes this is useful when we want to sort a list in a random order:

```
>>> a = np.arange(10)
>>> np.random.shuffle(a)
>>> a
array([7, 6, 3, 1, 4, 2, 5, 0, 9, 8])
```

The following figure shows two distributions, `binomial` and `poisson`, side by side with various parameters (the visualization was created with `matplotlib`, which will be covered in *Chapter 4, Data Visualization*):



Summary

In this chapter, we covered a lot of information related to the NumPy package, especially commonly used functions that are very helpful to process and analyze data in `ndarray`. Firstly, we learned the properties and data type of `ndarray` in the NumPy package. Secondly, we focused on how to create and manipulate an `ndarray` in different ways, such as conversion from other structures, reading an array from disk, or just generating a new array with given values. Thirdly, we studied how to access and control the value of each element in `ndarray` by using indexing and slicing.

Then, we are getting familiar with some common functions and operations on `ndarray`.

And finally, we continue with some advance functions that are related to statistic, linear algebra and sampling data. Those functions play important role in data analysis.

However, while NumPy by itself does not provide very much high-level data analytical functionality, having an understanding of it will help you use tools such as Pandas much more effectively. This tool will be discussed in the next chapter.

Practice exercises

Exercise 1: Using an array creation function, let's try to create arrays variable in the following situations:

- Create `ndarray` from the existing data
- Initialize `ndarray` which elements are filled with ones, zeros, or a given interval
- Loading and saving data from a file to an `ndarray`

Exercise 2: What is the difference between `np.dot(a, b)` and `(a*b)`?

Exercise 3: Consider the vector `[1, 2, 3, 4, 5]` building a new vector with four consecutive zeros interleaved between each value.

Exercise 4: Taking the data example file `chapter2-data.txt`, which includes information on a system log, solves the following tasks:

- Try to build an `ndarray` from the data file
- Statistic frequency of each device type in the built matrix
- List unique OS that appears in the data log
- Sort user by `provinceID` and count the number of users in each province

3

Data Analysis with Pandas

In this chapter, we will explore another data analysis library called Pandas. The goal of this chapter is to give you some basic knowledge and concrete examples for getting started with Pandas.

An overview of the Pandas package

Pandas is a Python package that supports fast, flexible, and expressive data structures, as well as computing functions for data analysis. The following are some prominent features that Pandas supports:

- Data structure with labeled axes. This makes the program clean and clear and avoids common errors from misaligned data.
- Flexible handling of missing data.
- Intelligent label-based slicing, fancy indexing, and subset creation of large datasets.
- Powerful arithmetic operations and statistical computations on a custom axis via axis label.
- Robust input and output support for loading or saving data from and to files, databases, or HDF5 format.

Related to Pandas installation, we recommend an easy way, that is to install it as a part of Anaconda, a cross-platform distribution for data analysis and scientific computing. You can refer to the reference at <http://docs.continuum.io/anaconda/> to download and install the library.

After installation, we can use it like other Python packages. Firstly, we have to import the following packages at the beginning of the program:

```
>>> import pandas as pd
>>> import numpy as np
```

The Pandas data structure

Let's first get acquainted with two of Pandas' primary data structures: the Series and the DataFrame. They can handle the majority of use cases in finance, statistic, social science, and many areas of engineering.

Series

A Series is a one-dimensional object similar to an array, list, or column in table. Each item in a Series is assigned to an entry in an index:

```
>>> s1 = pd.Series(np.random.rand(4),
                    index=['a', 'b', 'c', 'd'])

>>> s1
a    0.6122
b    0.98096
c    0.3350
d    0.7221
dtype: float64
```

By default, if no index is passed, it will be created to have values ranging from 0 to $N-1$, where N is the length of the Series:

```
>>> s2 = pd.Series(np.random.rand(4))
>>> s2
0    0.6913
1    0.8487
2    0.8627
3    0.7286
dtype: float64
```

We can access the value of a Series by using the index:

```
>>> s1['c']
0.3350
>>> s1['c'] = 3.14
>>> s1['c', 'a', 'b']
c    3.14
a    0.6122
b    0.98096
```

This accessing method is similar to a Python dictionary. Therefore, Pandas also allows us to initialize a Series object directly from a Python dictionary:

```
>>> s3 = pd.Series({'001': 'Nam', '002': 'Mary',
                    '003': 'Peter'})

>>> s3
001    Nam
002    Mary
003    Peter
dtype: object
```

Sometimes, we want to filter or rename the index of a Series created from a Python dictionary. At such times, we can pass the selected index list directly to the initial function, similarly to the process in the above example. Only elements that exist in the index list will be in the Series object. Conversely, indexes that are missing in the dictionary are initialized to default NaN values by Pandas:

```
>>> s4 = pd.Series({'001': 'Nam', '002': 'Mary',
                    '003': 'Peter'}, index=[
                    '002', '001', '024', '065'])

>>> s4
002    Mary
001    Nam
024    NaN
065    NaN
dtype: object
```


The library also supports functions that detect missing data:

```
>>> pd.isnull(s4)
002    False
001    False
024    True
065    True
dtype: bool
```

Similarly, we can also initialize a Series from a scalar value:

```
>>> s5 = pd.Series(2.71, index=['x', 'y'])
>>> s5
x    2.71
y    2.71
dtype: float64
```

A Series object can be initialized with NumPy objects as well, such as `ndarray`. Moreover, Pandas can automatically align data indexed in different ways in arithmetic operations:

```
>>> s6 = pd.Series(np.array([2.71, 3.14]), index=['z', 'y'])
>>> s6
z    2.71
y    3.14
dtype: float64
>>> s5 + s6
x    NaN
y    5.85
z    NaN
dtype: float64
```

The DataFrame

The DataFrame is a tabular data structure comprising a set of ordered columns and rows. It can be thought of as a group of Series objects that share an index (the column names). There are a number of ways to initialize a DataFrame object. Firstly, let's take a look at the common example of creating DataFrame from a dictionary of lists:

```
>>> data = {'Year': [2000, 2005, 2010, 2014],
            'Median_Age': [24.2, 26.4, 28.5, 30.3],
```

```

        'Density': [244, 256, 268, 279]}
>>> df1 = pd.DataFrame(data)
>>> df1
   Density  Median_Age  Year
0    244         24.2   2000
1    256         26.4   2005
2    268         28.5   2010
3    279         30.3   2014

```

By default, the DataFrame constructor will order the column alphabetically. We can edit the default order by passing the column's attribute to the initializing function:

```

>>> df2 = pd.DataFrame(data, columns=['Year', 'Density',
                                     'Median_Age'])
>>> df2
   Year  Density  Median_Age
0   2000     244         24.2
1   2005     256         26.4
2   2010     268         28.5
3   2014     279         30.3
>>> df2.index
Int64Index([0, 1, 2, 3], dtype='int64')

```

We can provide the index labels of a DataFrame similar to a Series:

```

>>> df3 = pd.DataFrame(data, columns=['Year', 'Density',
                                     'Median_Age'], index=['a', 'b', 'c', 'd'])
>>> df3.index
Index([u'a', u'b', u'c', u'd'], dtype='object')

```

We can construct a DataFrame out of nested lists as well:

```

>>> df4 = pd.DataFrame([
    ['Peter', 16, 'pupil', 'TN', 'M', None],
    ['Mary', 21, 'student', 'SG', 'F', None],
    ['Nam', 22, 'student', 'HN', 'M', None],
    ['Mai', 31, 'nurse', 'SG', 'F', None],
    ['John', 28, 'lawyer', 'SG', 'M', None]],
    columns=['name', 'age', 'career', 'province', 'sex', 'award'])

```

Columns can be accessed by column name as a Series can, either by dictionary-like notation or as an attribute, if the column name is a syntactically valid attribute name:

```
>>> df4.name      # or df4['name']
0    Peter
1    Mary
2    Nam
3    Mai
4    John
Name: name, dtype: object
```

To modify or append a new column to the created DataFrame, we specify the column name and the value we want to assign:

```
>>> df4['award'] = None
>>> df4
   name age  career province sex award
0  Peter  16   pupil      TN    M  None
1   Mary  21  student      SG    F  None
2   Nam  22  student      HN    M  None
3   Mai  31   nurse      SG    F  None
4   John  28   lawer      SG    M  None
```

Using a couple of methods, rows can be retrieved by position or name:

```
>>> df4.ix[1]
name      Mary
age        21
career    student
province    SG
sex        F
award      None
Name: 1, dtype: object
```

A DataFrame object can also be created from different data structures such as a list of dictionaries, a dictionary of Series, or a record array. The method to initialize a DataFrame object is similar to the examples above.

Another common case is to provide a DataFrame with data from a location such as a text file. In this situation, we use the `read_csv` function that expects the column separator to be a comma, by default. However, we can change that by using the `sep` parameter:

```
# person.csv file
name,age,career,province,sex
Peter,16,pupil,TN,M
Mary,21,student,SG,F
Nam,22,student,HN,M
Mai,31,nurse,SG,F
John,28,lawer,SG,M
# loading person.csv into a DataFrame
>>> df4 = pd.read_csv('person.csv')
>>> df4
```

	name	age	career	province	sex
0	Peter	16	pupil	TN	M
1	Mary	21	student	SG	F
2	Nam	22	student	HN	M
3	Mai	31	nurse	SG	F
4	John	28	lawyer	SG	M

While reading a data file, we sometimes want to skip a line or an invalid value. As for Pandas 0.16.2, `read_csv` supports over 50 parameters for controlling the loading process. Some common useful parameters are as follows:

- `sep`: This is a delimiter between columns. The default is comma symbol.
- `dtype`: This is a data type for data or columns.
- `header`: This sets row numbers to use as the column names.
- `skiprows`: This skips line numbers to skip at the start of the file.
- `error_bad_lines`: This shows invalid lines (too many fields) that will, by default, cause an exception, such that no DataFrame will be returned. If we set the value of this parameter as `false`, the bad lines will be skipped.

Moreover, Pandas also has support for reading and writing a DataFrame directly from or to a database such as the `read_frame` or `write_frame` function within the Pandas module. We will come back to these methods later in this chapter.

The essential basic functionality

Pandas supports many essential functionalities that are useful to manipulate Pandas data structures. In this book, we will focus on the most important features regarding exploration and analysis.

Reindexing and altering labels

Reindex is a critical method in the Pandas data structures. It confirms whether the new or modified data satisfies a given set of labels along a particular axis of Pandas object.

First, let's view a `reindex` example on a Series object:

```
>>> s2.reindex([0, 2, 'b', 3])
0    0.6913
2    0.8627
b    NaN
3    0.7286
dtype: float64
```

When reindexed labels do not exist in the data object, a default value of `NaN` will be automatically assigned to the position; this holds true for the DataFrame case as well:

```
>>> df1.reindex(index=[0, 2, 'b', 3],
                 columns=['Density', 'Year', 'Median_Age', 'C'])
   Density  Year  Median_Age    C
0       244  2000        24.2  NaN
2       268  2010        28.5  NaN
b       NaN   NaN         NaN  NaN
3       279  2014        30.3  NaN
```

We can change the NaN value in the missing index case to a custom value by setting the `fill_value` parameter. Let us take a look at the arguments that the `reindex` function supports, as shown in the following table:

Argument	Description
<code>index</code>	This is the new labels/index to conform to.
<code>method</code>	This is the method to use for filling holes in a reindexed object. The default setting is <code>unfill</code> gaps. <code>pad/ffill</code> : fill values forward <code>backfill/bfill</code> : fill values backward <code>nearest</code> : use the nearest value to fill the gap
<code>copy</code>	This return a new object. The default setting is <code>true</code> .
<code>level</code>	The matches index values on the passed multiple index level.
<code>fill_value</code>	This is the value to use for missing values. The default setting is <code>NaN</code> .
<code>limit</code>	This is the maximum size gap to fill in <code>forward</code> or <code>backward</code> method.

Head and tail

In common data analysis situations, our data structure objects contain many columns and a large number of rows. Therefore, we cannot view or load all information of the objects. Pandas supports functions that allow us to inspect a small sample. By default, the functions return five elements, but we can set a custom number as well. The following example shows how to display the first five and the last three rows of a longer Series:

```
>>> s7 = pd.Series(np.random.rand(10000))
>>> s7.head()
0    0.631059
1    0.766085
2    0.066891
3    0.867591
4    0.339678
```

```
dtype: float64
>>> s7.tail(3)
9997    0.412178
9998    0.800711
9999    0.438344
dtype: float64
```

We can also use these functions for DataFrame objects in the same way.

Binary operations

Firstly, we will consider arithmetic operations between objects. In different indexes objects case, the expected result will be the union of the index pairs. We will not explain this again because we had an example about it in the above section (`s5 + s6`). This time, we will show another example with a DataFrame:

```
>>> df5 = pd.DataFrame(np.arange(9).reshape(3,3),0
                        columns=['a','b','c'])

>>> df5
   a  b  c
0  0  1  2
1  3  4  5
2  6  7  8

>>> df6 = pd.DataFrame(np.arange(8).reshape(2,4),
                        columns=['a','b','c','d'])

>>> df6
   a  b  c  d
0  0  1  2  3
1  4  5  6  7

>>> df5 + df6
   a  b  c  d
0  0  2  4 NaN
1  7  9 11 NaN
2  NaN NaN NaN NaN
```

The mechanisms for returning the result between two kinds of data structure are similar. A problem that we need to consider is the missing data between objects. In this case, if we want to fill with a fixed value, such as 0, we can use the arithmetic functions such as `add`, `sub`, `div`, and `mul`, and the function's supported parameters such as `fill_value`:

```
>>> df7 = df5.add(df6, fill_value=0)
>>> df7
```

	a	b	c	d
0	0	2	4	3
1	7	9	11	7
2	6	7	8	NaN

Next, we will discuss comparison operations between data objects. We have some supported functions such as **equal** (`eq`), **not equal** (`ne`), **greater than** (`gt`), **less than** (`lt`), **less equal** (`le`), and **greater equal** (`ge`). Here is an example:

```
>>> df5.eq(df6)
```

	a	b	c	d
0	True	True	True	False
1	False	False	False	False
2	False	False	False	False

Functional statistics

The supported statistics method of a library is really important in data analysis. To get inside a big data object, we need to know some summarized information such as mean, sum, or quantile. Pandas supports a large number of methods to compute them. Let's consider a simple example of calculating the `sum` information of `df5`, which is a DataFrame object:

```
>>> df5.sum()
```

a	9
b	12
c	15

`dtype: int64`

When we do not specify which axis we want to calculate `sum` information, by default, the function will calculate on index axis, which is axis 0:

- **Series:** We do not need to specify the axis.
- **DataFrame:** Columns (`axis = 1`) or index (`axis = 0`). The default setting is `axis 0`.

We also have the `skipna` parameter that allows us to decide whether to exclude missing data or not. By default, it is set as `true`:

```
>>> df7.sum(skipna=False)
```

```
a    13
b    18
c    23
d   NaN
dtype: float64
```

Another function that we want to consider is `describe()`. It is very convenient for us to summarize most of the statistical information of a data structure such as the Series and DataFrame, as well:

```
>>> df5.describe()
```

	a	b	c
count	3.0	3.0	3.0
mean	3.0	4.0	5.0
std	3.0	3.0	3.0
min	0.0	1.0	2.0
25%	1.5	2.5	3.5
50%	3.0	4.0	5.0
75%	4.5	5.5	6.5
max	6.0	7.0	8.0

We can specify percentiles to include or exclude in the output by using the `percentiles` parameter; for example, consider the following:

```
>>> df5.describe(percentiles=[0.5, 0.8])
```

	a	b	c
count	3.0	3.0	3.0
mean	3.0	4.0	5.0
std	3.0	3.0	3.0

```

min      0.0   1.0   2.0
50%      3.0   4.0   5.0
80%      4.8   5.8   6.8
max      6.0   7.0   8.0

```

Here, we have a summary table for common supported statistics functions in Pandas:

Function	Description
<code>idxmin(axis)</code> , <code>idxmax(axis)</code>	This compute the index labels with the minimum or maximum corresponding values.
<code>value_counts()</code>	This compute the frequency of unique values.
<code>count()</code>	This return the number of non-null values in a data object.
<code>mean()</code> , <code>median()</code> , <code>min()</code> , <code>max()</code>	This return mean, median, minimum, and maximum values of an axis in a data object.
<code>std()</code> , <code>var()</code> , <code>sem()</code>	These return the standard deviation, variance, and standard error of mean.
<code>abs()</code>	This gets the absolute value of a data object.

Function application

Pandas supports function application that allows us to apply some functions supported in other packages such as NumPy or our own functions on data structure objects. Here, we illustrate two examples of these cases, firstly, using `apply` to execute the `std()` function, which is the standard deviation calculating function of the NumPy package:

```

>>> df5.apply(np.std, axis=1)      # default: axis=0
0      0.816497
1      0.816497
2      0.816497
dtype: float64

```

Secondly, if we want to apply a formula to a data object, we can also use `apply` function by following these steps:

1. Define the function or formula that you want to apply on a data object.
2. Call the defined function or formula via `apply`. In this step, we also need to figure out the axis that we want to apply the calculation to:

```
>>> f = lambda x: x.max() - x.min()      # step 1
>>> df5.apply(f, axis=1)                  # step 2
0      2
1      2
2      2
dtype: int64

>>> def sigmoid(x):
    return 1/(1 + np.exp(x))
>>> df5.apply(sigmoid)
      a      b      c
0  0.500000  0.268941  0.119203
1  0.047426  0.017986  0.006693
2  0.002473  0.000911  0.000335
```

Sorting

There are two kinds of sorting method that we are interested in: sorting by row or column index and sorting by data value.

Firstly, we will consider methods for sorting by row and column index. In this case, we have the `sort_index()` function. We also have `axis` parameter to set whether the function should sort by row or column. The `ascending` option with the `true` or `false` value will allow us to sort data in ascending or descending order. The default setting for this option is `true`:

```
>>> df7 = pd.DataFrame(np.arange(12).reshape(3,4),
                        columns=['b', 'd', 'a', 'c'],
                        index=['x', 'y', 'z'])

>>> df7
      b  d  a  c
x  0  1  2  3
y  4  5  6  7
z  8  9 10 11
```

```
>>> df7.sort_index(axis=1)
      a  b  c  d
x     2  0  3  1
y     6  4  7  5
z    10  8 11  9
```

Series has a method `order` that sorts by value. For NaN values in the object, we can also have a special treatment via the `na_position` option:

```
>>> s4.order(na_position='first')
024      NaN
065      NaN
002     Mary
001      Nam
dtype: object
>>> s4
002     Mary
001      Nam
024      NaN
065      NaN
dtype: object
```

Besides that, Series also has the `sort()` function that sorts data by value. However, the function will not return a copy of the sorted data:

```
>>> s4.sort(na_position='first')
>>> s4
024      NaN
065      NaN
002     Mary
001      Nam
dtype: object
```

If we want to apply sort function to a DataFrame object, we need to figure out which columns or rows will be sorted:

```
>>> df7.sort(['b', 'd'], ascending=False)
   b  d  a  c
z  8  9 10 11
y  4  5  6  7
x  0  1  2  3
```

If we do not want to automatically save the sorting result to the current data object, we can change the setting of the `inplace` parameter to `False`.

Indexing and selecting data

In this section, we will focus on how to get, set, or slice subsets of Pandas data structure objects. As we learned in previous sections, Series or DataFrame objects have axis labeling information. This information can be used to identify items that we want to select or assign a new value to in the object:

```
>>> s4[['024', '002']]      # selecting data of Series object
024      NaN
002      Mary
dtype: object
>>> s4[['024', '002']] = 'unknown' # assigning data
>>> s4
024      unknown
065      NaN
002      unknown
001      Nam
dtype: object
```

If the data object is a DataFrame structure, we can also proceed in a similar way:


```
>>> df5[['b', 'c']]
   b  c
0  1  2
1  4  5
2  7  8
```

For label indexing on the rows of DataFrame, we use the `ix` function that enables us to select a set of rows and columns in the object. There are two parameters that we need to specify: the `row` and `column` labels that we want to get. By default, if we do not specify the selected column names, the function will return selected rows with all columns in the object:

```
>>> df5.ix[0]
a    0
b    1
c    2
Name: 0, dtype: int64
>>> df5.ix[0, 1:3]
b    1
c    2
Name: 0, dtype: int64
```

Moreover, we have many ways to select and edit data contained in a Pandas object. We summarize these functions in the following table:

Method	Description
<code>icol, irow</code>	This selects a single row or column by integer location.
<code>get_value, set_value</code>	This selects or sets a single value of a data object by row or column label.
<code>xs</code>	This selects a single column or row as a Series by label.

 Pandas data objects may contain duplicate indices. In this case, when we get or set a data value via index label, it will affect all rows or columns that have the same selected index name.

Computational tools

Let's start with correlation and covariance computation between two data objects. Both the Series and DataFrame have a `cov` method. On a DataFrame object, this method will compute the covariance between the Series inside the object:

```
>>> s1 = pd.Series(np.random.rand(3))
>>> s1
```

```
0    0.460324
1    0.993279
2    0.032957
dtype: float64
>>> s2 = pd.Series(np.random.rand(3))
>>> s2
0    0.777509
1    0.573716
2    0.664212
dtype: float64
>>> s1.cov(s2)
-0.024516360159045424

>>> df8 = pd.DataFrame(np.random.rand(12).reshape(4,3),
                        columns=['a', 'b', 'c'])
>>> df8
      a      b      c
0  0.200049  0.070034  0.978615
1  0.293063  0.609812  0.788773
2  0.853431  0.243656  0.978057
0.985584  0.500765  0.481180
>>> df8.cov()
      a      b      c
a  0.155307  0.021273 -0.048449
b  0.021273  0.059925 -0.040029
c -0.048449 -0.040029  0.055067
```

Usage of the correlation method is similar to the covariance method. It computes the correlation between Series inside a data object in case the data object is a DataFrame. However, we need to specify which method will be used to compute the correlations. The available methods are `pearson`, `kendall`, and `spearman`. By default, the function applies the `spearman` method:

```
>>> df8.corr(method = 'spearman')
      a      b      c
a  1.0  0.4 -0.8
b  0.4  1.0 -0.8
c -0.8 -0.8  1.0
```

We also have the `corrwith` function that supports calculating correlations between Series that have the same label contained in different DataFrame objects:

```
>>> df9 = pd.DataFrame(np.arange(8).reshape(4,2),
                        columns=['a', 'b'])

>>> df9
   a  b
0  0  1
1  2  3
2  4  5
3  6  7

>>> df8.corrwith(df9)
a    0.955567
b    0.488370
c         NaN
dtype: float64
```

Working with missing data

In this section, we will discuss missing, NaN, or null values, in Pandas data structures. It is a very common situation to arrive with missing data in an object. One such case that creates missing data is reindexing:

```
>>> df8 = pd.DataFrame(np.arange(12).reshape(4,3),
                        columns=['a', 'b', 'c'])

   a  b  c
0  0  1  2
1  3  4  5
2  6  7  8
3  9 10 11

>>> df9 = df8.reindex(columns = ['a', 'b', 'c', 'd'])

   a  b  c  d
0  0  1  2 NaN
1  3  4  5 NaN
2  6  7  8 NaN
4  9 10 11 NaN
```



```
>>> df10 = df8.reindex([3, 2, 'a', 0])
      a    b    c
3     9   10   11
2     6    7    8
a  NaN  NaN  NaN
0     0    1    2
```

To manipulate missing values, we can use the `isnull()` or `notnull()` functions to detect the missing values in a Series object, as well as in a DataFrame object:

```
>>> df10.isnull()
      a      b      c
3  False  False  False
2  False  False  False
a    True   True   True
0  False  False  False
```

On a Series, we can drop all null data and index values by using the `dropna` function:

```
>>> s4 = pd.Series({'001': 'Nam', '002': 'Mary',
                    '003': 'Peter'},
                    index=['002', '001', '024', '065'])

>>> s4
002    Mary
001     Nam
024     NaN
065     NaN
dtype: object

>>> s4.dropna()      # dropping all null value of Series object
002    Mary
001     Nam
dtype: object
```

With a DataFrame object, it is a little bit more complex than with Series. We can tell which rows or columns we want to drop and also if all entries must be `null` or a single `null` value is enough. By default, the function will drop any row containing a missing value:

```
>>> df9.dropna()      # all rows will be dropped
Empty DataFrame
Columns: [a, b, c, d]
Index: []
>>> df9.dropna(axis=1)
   a  b  c
0  0  1  2
1  3  4  5
2  6  7  8
3  9 10 11
```

Another way to control missing values is to use the supported parameters of functions that we introduced in the previous section. They are also very useful to solve this problem. In our experience, we should assign a fixed value in missing cases when we create data objects. This will make our objects cleaner in later processing steps. For example, consider the following:

```
>>> df11 = df8.reindex([3, 2, 'a', 0], fill_value = 0)
>>> df11
   a  b  c
3  9 10 11
2  6  7  8
a  0  0  0
0  0  1  2
```

We can also use the `fillna` function to fill a custom value in missing values:

```
>>> df9.fillna(-1)
   a  b  c  d
0  0  1  2 -1
1  3  4  5 -1
2  6  7  8 -1
3  9 10 11 -1
```

Advanced uses of Pandas for data analysis

In this section we will consider some advanced Pandas use cases.

Hierarchical indexing

Hierarchical indexing provides us with a way to work with higher dimensional data in a lower dimension by structuring the data object into multiple index levels on an axis:

```
>>> s8 = pd.Series(np.random.rand(8), index=[['a','a','b','b','c','c','d','d'], [0, 1, 0, 1, 0, 1, 0, 1]])
>>> s8
a  0    0.721652
   1    0.297784
b  0    0.271995
   1    0.125342
c  0    0.444074
   1    0.948363
d  0    0.197565
   1    0.883776
dtype: float64
```

In the preceding example, we have a Series object that has two index levels. The object can be rearranged into a DataFrame using the `unstack` function. In an inverse situation, the `stack` function can be used:

```
>>> s8.unstack()
           0         1
a  0.549211  0.420874
b  0.051516  0.715021
c  0.503072  0.720772
d  0.373037  0.207026
```

We can also create a DataFrame to have a hierarchical index in both axes:

```
>>> df = pd.DataFrame(np.random.rand(12).reshape(4,3),
                        index=[['a', 'a', 'b', 'b'],
                              [0, 1, 0, 1]],
                        columns=[['x', 'x', 'y'], [0, 1, 0]])

>>> df
```

		x		y
		0	1	0
a	0	0.636893	0.729521	0.747230
	1	0.749002	0.323388	0.259496
b	0	0.214046	0.926961	0.679686
	1	0.013258	0.416101	0.626927

```
>>> df.index
MultiIndex(levels=[['a', 'b'], [0, 1]],
            labels=[[0, 0, 1, 1], [0, 1, 0, 1]])

>>> df.columns
MultiIndex(levels=[['x', 'y'], [0, 1]],
            labels=[[0, 0, 1], [0, 1, 0]])
```

The methods for getting or setting values or subsets of the data objects with multiple index levels are similar to those of the nonhierarchical case:

```
>>> df['x']
```

		0	1
a	0	0.636893	0.729521
	1	0.749002	0.323388
b	0	0.214046	0.926961
	1	0.013258	0.416101

```
>>> df[[0]]
```

		x
		0
a	0	0.636893
	1	0.749002
b	0	0.214046
	1	0.013258

```
>>> df.ix['a', 'x']
           0           1
0  0.636893  0.729521
0.749002  0.323388
>>> df.ix['a', 'x'].ix[1]
0    0.749002
1    0.323388
Name: 1, dtype: float64
```

After grouping data into multiple index levels, we can also use most of the descriptive and statistics functions that have a level option, which can be used to specify the level we want to process:

```
>>> df.std(level=1)
           x           y
           0           1           0
0  0.298998  0.139611  0.047761
0.520250  0.065558  0.259813
>>> df.std(level=0)
           x           y
           0           1           0
a  0.079273  0.287180  0.344880
b  0.141979  0.361232  0.037306
```

The Panel data

The Panel is another data structure for three-dimensional data in Pandas. However, it is less frequently used than the Series or the DataFrame. You can think of a Panel as a table of DataFrame objects. We can create a Panel object from a 3D ndarray or a dictionary of DataFrame objects:

```
# create a Panel from 3D ndarray
>>> panel = pd.Panel(np.random.rand(2, 4, 5),
                      items = ['item1', 'item2'])

>>> panel
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 4 (major_axis) x 5 (minor_axis)
Items axis: item1 to item2
Major_axis axis: 0 to 3
```

Minor_axis axis: 0 to 4

```
>>> df1 = pd.DataFrame(np.arange(12).reshape(4, 3),
                        columns=['a', 'b', 'c'])

>>> df1
   a  b  c
0  0  1  2
1  3  4  5
2  6  7  8
3  9 10 11

>>> df2 = pd.DataFrame(np.arange(9).reshape(3, 3),
                        columns=['a', 'b', 'c'])

>>> df2
   a  b  c
0  0  1  2
1  3  4  5
2  6  7  8

# create another Panel from a dict of DataFrame objects
>>> panel2 = pd.Panel({'item1': df1, 'item2': df2})
>>> panel2
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 4 (major_axis) x 3 (minor_axis)
Items axis: item1 to item2
Major_axis axis: 0 to 3
Minor_axis axis: a to c
```

Each item in a Panel is a DataFrame. We can select an item, by item name:

```
>>> panel2['item1']
   a  b  c
0  0  1  2
1  3  4  5
2  6  7  8
3  9 10 11
```

Alternatively, if we want to select data via an axis or data position, we can use the `ix` method, like on Series or DataFrame:

```
>>> panel2.ix[:, 1:3, ['b', 'c']]
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 3 (major_axis) x 2 (minor_axis)
Items axis: item1 to item2
Major_axis axis: 1 to 3
Minor_axis axis: b to c
>>> panel2.ix[:, 2, :]
```

	item1	item2
a	6	6
b	7	7
c	8	8

Summary

We have finished covering the basics of the Pandas data analysis library. Whenever you learn about a library for data analysis, you need to consider the three parts that we explained in this chapter. Data structures: we have two common data object types in the Pandas library; Series and DataFrames. Method to access and manipulate data objects: Pandas supports many way to select, set or slice subsets of data object. However, the general mechanism is using index labels or the positions of items to identify values. Functions and utilities: They are the most important part of a powerful library. In this chapter, we covered all common supported functions of Pandas which allow us compute statistics on data easily. The library also has a lot of other useful functions and utilities that we could not explain in this chapter. We encourage you to start your own research, if you want to expand your experience with Pandas. It helps us to process large data in an optimized way. You will see more of Pandas in action later in this book.

Until now, we learned about two popular Python libraries: NumPy and Pandas. Pandas is built on NumPy, and as a result it allows for a bit more convenient interaction with data. However, in some situations, we can flexibly combine both of them to accomplish our goals.

Practice exercises

The link https://www.census.gov/2010census/csv/pop_change.csv contains an US census dataset. It has 23 columns and one row for each US state, as well as a few rows for macro regions such as North, South, and West.

- Get this dataset into a Pandas DataFrame. Hint: just skip those rows that do not seem helpful, such as comments or description.
- While the dataset contains change metrics for each decade, we are interested in the population change during the second half of the twentieth century, that is between, 1950 and 2000. Which region has seen the biggest and the smallest population growth in this time span? Also, which US state?

Advanced open-ended exercise:

- Find more census data on the internet; not just on the US but on the world's countries. Try to find GDP data for the same time as well. Try to align this data to explore patterns. How are GDP and population growth related? Are there any special cases. such as countries with high GDP but low population growth or countries with the opposite history?