Data Mining with Python (Working draft)

Finn Årup Nielsen

November 29, 2017

Contents

Contents **i**

List of Figures **vii**

List of Tables **ix**

1 Introduction **1**
- 1.1 Other introductions to Python? . . . 1
- 1.2 Why Python for data mining? . . . 1
- 1.3 Why not Python for data mining? . . . 2
- 1.4 Components of the Python language and software . . . 3
- 1.5 Developing and running Python . . . 5
  - 1.5.1 Python, pypy, IPython 5
  - 1.5.2 Jupyter Notebook . . . 6
  - 1.5.3 Python 2 vs. Python 3 . . . 6
  - 1.5.4 Editing . . . 7
  - 1.5.5 Python in the cloud . . . 7
  - 1.5.6 Running Python in the browser . . . 7

2 Python **9**
- 2.1 Basics . . . 9
- 2.2 Datatypes . . . 9
  - 2.2.1 Booleans (`bool`) . . . 9
  - 2.2.2 Numbers (`int`, `float`, `complex` and `Decimal`) . . . 10
  - 2.2.3 Strings (`str`) . . . 11
  - 2.2.4 Dictionaries (`dict`) . . . 11
  - 2.2.5 Dates and times . . . 12
  - 2.2.6 Enumeration . . . 13
  - 2.2.7 Other containers classes . . . 13
- 2.3 Functions and arguments . . . 14
  - 2.3.1 Anonymous functions with `lambdas` . . . 14
  - 2.3.2 Optional function arguments . . . 14
- 2.4 Object-oriented programming . . . 15
  - 2.4.1 Objects as functions . . . 17
- 2.5 Modules and import . . . 17
  - 2.5.1 Submodules . . . 18
  - 2.5.2 Globbing import . . . 19
  - 2.5.3 Coping with Python 2/3 incompatibility . . . 19
- 2.6 Persistency . . . 20
  - 2.6.1 Pickle and JSON . . . 20
  - 2.6.2 SQL . . . 21

2.6.3 NoSQL . . . 21
2.7 Documentation . . . 21
2.8 Testing . . . 22
2.8.1 Testing for type . . . 22
2.8.2 Zero-one-some testing . . . 23
2.8.3 Test layout and test discovery . . . 23
2.8.4 Test coverage . . . 24
2.8.5 Testing in different environments . . . 25
2.9 Profiling . . . 25
2.10 Coding style . . . 27
2.10.1 Where is `private` and `public`? . . . 28
2.11 Command-line interface scripting . . . 29
2.11.1 Distinguishing between module and script . . . 29
2.11.2 Argument parsing . . . 29
2.11.3 Exit status . . . 29
2.12 Debugging . . . 30
2.12.1 Logging . . . 31
2.13 Advices . . . 31

3 Python for data mining **33**
3.1 Numpy . . . 33
3.2 Plotting . . . 33
3.2.1 3D plotting . . . 34
3.2.2 Real-time plotting . . . 34
3.2.3 Plotting for the Web . . . 36
3.3 Pandas . . . 39
3.3.1 Pandas data types . . . 40
3.3.2 Pandas indexing . . . 40
3.3.3 Pandas joining, merging and concatenations . . . 42
3.3.4 Simple statistics . . . 43
3.4 SciPy . . . 44
3.4.1 `scipy.linalg` . . . 44
3.4.2 Fourier transform with `scipy.fftpack` . . . 44
3.5 Statsmodels . . . 45
3.6 Sympy . . . 47
3.7 Machine learning . . . 47
3.7.1 Scikit-learn . . . 49
3.8 Text mining . . . 50
3.8.1 Regular expressions . . . 50
3.8.2 Extracting from webpages . . . 51
3.8.3 NLTK . . . 52
3.8.4 Tokenization and part-of-speech tagging . . . 53
3.8.5 Language detection . . . 54
3.8.6 Sentiment analysis . . . 54
3.9 Network mining . . . 55
3.10 Miscellaneous issues . . . 56
3.10.1 Lazy computation . . . 56
3.11 Testing data mining code . . . 57

4 Case: Pure Python matrix library **59**
4.1 Code listing . . . 59

5 Case: Pima data set **65**
5.1 Problem description and objectives 65
5.2 Descriptive statistics and plotting 66
5.3 Statistical tests 67
5.4 Predicting diabetes type 69

6 Case: Data mining a database **71**
6.1 Problem description and objectives 71
6.2 Reading the data 71
6.3 Graphical overview on the connections between the tables 72
6.4 Statistics on the number of tracks sold 74

7 Case: Twitter information diffusion **75**
7.1 Problem description and objectives 75
7.2 Building a news classifier 75

8 Case: Big data **77**
8.1 Problem description and objectives 77
8.2 Stream processing of JSON 77
8.2.1 Stream processing of JSON Lines 78

Bibliography **81**

Index **85**

Preface

Python has grown to become one of the central languages in data mining offering both a general programming language and libraries specifically targeted numerical computations.

This book is continuously being written and grew out of course given at the Technical University of Denmark.

List of Figures

1.1 The Python hierarchy. 4

2.1 Overview of methods and attributes in the common Python 2 built-in data types plotted as a formal concept analysis lattice graph. Only a small subset of methods and attributes is shown. 16

3.1 Sklearn classes derivation. 49
3.2 Comorbidity for ICD-10 disease code (appendicitis). 55

5.1 Seaborn correlation plot on the Pima data set 68

6.1 Database tables graph 73

List of Tables

2.1 Basic built-in and Numpy and Pandas datatypes 10
2.2 Class methods and attributes 15
2.3 Testing concepts 22

3.1 Function for generation of Numpy data structures. 33
3.2 Some of the subpackages of SciPy. 44
3.3 Python machine learning packages 48
3.4 Scikit-learn methods 48
3.5 sklearn classifiers 49
3.6 Metacharacters and character classes 50
3.7 NLT submodules. 53

5.1 Variables in the Pima data set 65

Chapter 1

Introduction

1.1 Other introductions to Python?

Although we cover a bit of introductory Python programming in chapter 2 you should not regard this book as a Python introduction: Several free introductory ressources exist. First and foremost the official *Python Tutorial* at http://docs.python.org/tutorial/. Beginning programmers with no or little programming experience may want to look into the book *Think Python* available from http://www.greenteapress.com/thinkpython/ or as a book [1], while more experienced programmers can start with *Dive Into Python* available from http://www.diveintopython.net/.[1] Kevin Sheppard's presently 381-page *Introduction to Python for Econometrics, Statistics and Data Analysis* covers both Python basics and Python-based data analysis with Numpy, SciPy, Matplotlib and Pandas, — and it is not just relevant for econometrics [2]. Developers already well-versed in standard Python development but lacking experience with Python for data mining can begin with chapter 3. Readers in need of an introduction to machine learning may take a look in Marsland's *Machine learning: An algorithmic perspective* [3], that uses Python for its examples.

1.2 Why Python for data mining?

Researchers have noted a number of reasons for using Python in the data science area (data mining, scientific computing) [4, 5, 6]:

1. Programmers regard Python as a clear and simple language with a high **readability**. Even non-programmers may not find it too difficult. The simplicity exists both in the language itself as well as in the encouragement to write clear and simple code prevalent among Python programmers. See this in contrast to, e.g., Perl where short form variable names allow you to write condensed code but also requires you to remember nonintuitive variable names. A Python program may also be 2–5 shorter than corresponding programs written in Java, C++ or C [7, 8].

2. **Platform-independent**. Python will run on the three main desktop computing platforms Mac, Linux and Windows, as well as on a number of other platforms.

3. **Interactive program**. With Python you get an interactive prompt with REPL (read-eval-print loop) like in Matlab and R. The prompt facilitates exploratory programming convenient for many data mining tasks, while you still can develop complete programs in an edit-run-debug cycle. The Python-derivatives IPython and Jupyter Notebook are particularly suited for interactive programming.

4. **General purpose language**. Python is a general purpose language that can be used to a wide variety of tasks beyond data mining, e.g., user applications, system administration, gaming, web development psychological experiment presentations and recording. This is in contrast to Matlab and R.

[1]For further free website for learning Python see http://www.fromdev.com/2014/03/python-tutorials-resources.html.

Too see how well Python with its modern data mining packages compares with R take a look at Carl J. V.'s blog posts on *Will it Python?*[2] and his GitHub repository where he reproduces R code in Python based on R data analyses from the book *Machine Learning for Hackers*.

5. Python with its BSD license fall in the group of **free and open source software**. Although some large Python development environments may have associated license cost for commercial use, the basic Python development environment may be setup and run with no licensing cost. Indeed in some systems, e.g., many Linux distributions, basic Python comes readily installed. The Python Package Index provides a large set of packages that are also free software.

6. **Large community**. Python has a large community and has become more popular. Several indicators testify to this. Popularity of Language Index (PYPL) bases its programming language ranking on Google search volume provided by Google Trends and puts Python in the third position after Java and PHP. According to PYPL the popularity of Python has grown since 2004. TIOBE constructs another indicator putting Python in rank 6th. This indicator is "based on the number of skilled engineers world-wide, courses and third party vendors".[3] Also Python is among the leading programming language in terms of StackOverflow tags and GitHub projects.[4] Furthermore, in 2014 Python was the most popular programming language at top-ranked United States universities for teaching introductory programming [9].

7. **Quality**: The Coverity company finds that Python code has errors among its 400,000 lines of code, but that the error rate is very low compared to other open source software projects. They found a 0.005 defects per KLoC [10].

8. **Jupyter Notebook**: With the browser-based interactive notebook, where code, textual and plotting results and documentation may be interleaved in a cell-based environment, the Jupyter Notebook represents a interesting approach that you will typically not find in many other programming language. Exceptions are the commercial systems Maple and Mathematica that have notebook interfaces. IPython Notebooks runs locally on a Web-browser. The Notebook files are JSON files that can easily be shared and rendered on the Web.

   The obvious advantages with the Jupyter Notebook has led other language to use the environment. The Jupyter Notebook can be changed to use, e.g., the Julia language as the computational backend, i.e., instead of writing Python code in the code cells of the notebook you write Julia code. With appropriate extensions the Jupyter Notebook can intermix R code.

1.3 Why not Python for data mining?

Why shouldn't you use Python?

1. **Not well-suited to mobile phones and other portable devices**. Although Python surely can run on mobile phones and there exist a least one (dated) book for 'Mobile Python' [11], Python has not caught on for development of mobile apps. There exist several mobile app development frameworks with Kivy mentioned as leading contender. Developers can also use Python in mobile contexts for the backend of a web-based system and for data mining data collected at the backend.

2. **Does not run 'natively' in the browser**. Javascript entirely dominates as the language in web-browsers. Various ways exist to mix Python and webbrowser programming.[5] The Pyjamas project with its Python-to-Javascript compiler allows you to write webbrowser client code in Python and compile it to Javascript which the webbrowser then runs. There are several other of these stand-alone Javascript

[2]http://slendermeans.org/pages/will-it-python.html.
[3]http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.
[4]http://www.dataists.com/2010/12/ranking-the-popularity-of-programming-langauges/.
[5]See https://wiki.python.org/moin/WebBrowserProgramming

compilers in ‘various states of development’ as it is called: PythonJS, Pyjaco, Py2JS. Other frameworks use in-browser implementations, one of them being Brython, which enable the front-end engineer to write Python code in a HTML script tag if the page includes the `brython.js` Javascript library via the HTML script tag. It supports core Python modules and has access to the DOM API, but not, e.g., the scientific Python libraries written in C. Brython scripts run unfortunately considerable slower than scripts directly implemented Javascript or ordinary Python implementation execution [12].

3. **Concurrent programming**. Standard Python has no direct way of utilizing several CPUs in the language. Multithreading capabilities can be obtained with the `threading` package, but the individual threads will not run concurrently on different CPUs in the standard python implementation. This implementation has the so-called ‘Global Interpreter Lock’ (GIL), which only allows a single thread at a time. This is to ensure the integrity of the data. A way to get around the GIL is by spawning new process with the `multiprocessing` package or just the `subprocess` module.

4. **Installation friction**. You may run into problems when building, distributing and installing your software. There are various ways to bundle Python software, e.g., with `setuptools` package. Based on a configuration file, `setup.py`, where you specify, e.g., name, author and dependencies of your package, `setuptools` can build a file to distribute with the commands `python setup.py bdist` or `python setup.py bdist_egg`. The latter command will build a so-called Python Egg file containing all the Python files you specified. The user of your package can install your Python files based on the configuration and content of that file. It will still need to download and install the dependencies you have specified in the `setup.py` file, before the user of your software can use your code. If your user does not have Python, the installation tools and a C compiler installed it is likely that s/he find it a considerable task to install your program.

   Various tools exist to make the distribution easier by integrating the the distributed file to one self-contained downloadable file. These tools are called cx_Freeze, PyInstaller, py2exe for Window and py2app for OSX) and pynsist.

5. **Speed**. Python will typically perform slower than a compiled languages such as C++, and Python typically performs poorer than Julia, — the programming language designed for technical computing. Various Python implementations and extensions, such as `pypy`, `numba` and Cython, can speed up the execution of Python code, but even then Julia can perform faster: Andrew Tulloch has reported performance ratios between 1.1 and 300 in Julia’s favor for isotonic regression algorithms.[6] The slowness of Python means that Python libraries tends to be developed in C, while, e.g., well-performing Julia libraries may be developed in Julia itself.[7] Speeding up Python often means modifying Python code with, e.g., specialized decorators, but a proof-of-concept system, Bohrium, has shown that a Python extension may require only little change in ‘standard’ array-processing code to speed up Python considerably [13].

   It may, however, be worth to note that variability in a program’s performance can vary as much or more between programmers as between Python, Java and C++ [7].

1.4 Components of the Python language and software

1. **The Python language keywords**. At its most basic level Python contains a set of keywords, for definition (of, e.g., functions, anonymous function and classes with `def`, `lambda` and `class`, respectively), for control structures (e.g., `if` and `for`), exceptions, assertions and returning arguments (`yield` and `return`). If you want to have a peek at all the keywords, then the `keyword` module makes their names available in the `keyword.kwlist` variable.

   Python 2 has 31 keywords, while Python 3 has 33.

[6]http://tullo.ch/articles/python-vs-julia/
[7]Mike Innes, Performance matters more than you think.

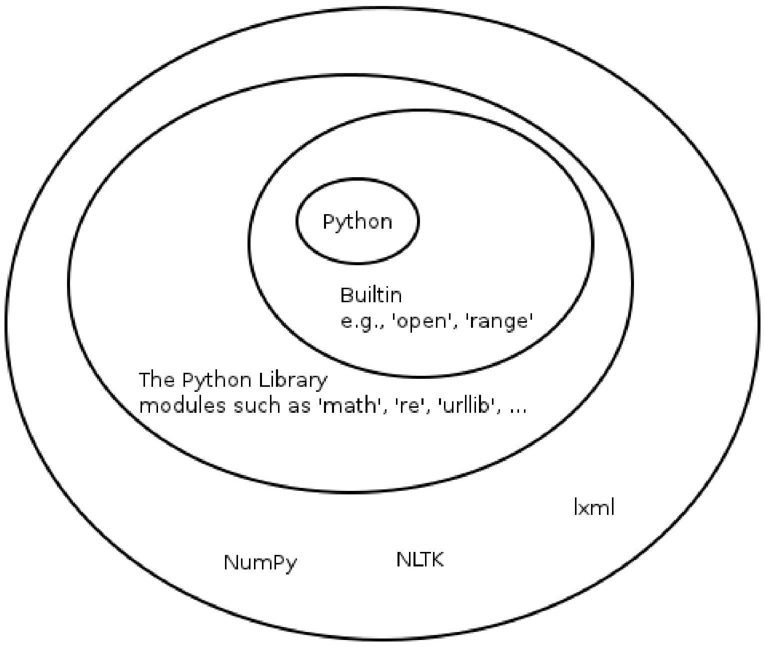

Figure 1.1: The Python hierarchy.

2. **Built-in classes and functions**. An ordinary implementation of Python makes a set of classes and functions available at program start without the need of module import. Examples include the function for opening files (`open`), classes for built-in data types (e.g., `float` and `str`) and data manipulation functions (e.g., `sum`, `abs` and `zip`). The `__builtins__` module makes these classes and functions available and you can see a listing of them with `dir(__builtins__)`.[8] You will find it non-trivial to get rid of the built-in functions, e.g., if you want to restrict the ability of untrusted code to call the `open` function, cf. sandboxing Python.

3. **Built-in modules**. Built-in modules contain extra classes and functions built into Python, — but not immediately accessible. You will need to import these with `import` to use them. The `sys` built-in module contains a list of all the built-in modules: `sys.builtin_module_names`. Among the built-in modules are the system-specific parameters and functions module (`sys`), a module with mathematical functions (`math`), the garbage collection module (`gc`) and a module with many handy iterator functions good to be acquited with (`itertools`).

   The set of built-in modules varies between implementations of Python. In one of my installations I count 46 modules, which include the `__builtins__` module and the current working module `__main__`.

4. **Python Standard Library** (PSL). An ordinary installation of Python makes a large set of modules with classes and functions available to the programmer without the need for extra installation. The programmer only needs to write a one line `import` statement to have access to exported classes, functions and constants in such a module.

   You can see which Python (byte-compiled) source file associates with the import via `__file__` property of the module, e.g., after `import os` you can see the filename with `os.__file__`. Built-in modules do not have this property set in the standard implementation of Python. On a typically Linux system you might find the PSL modules in a directories with names like `/usr/lib/python3.2/`.

   One of my installations has just above 200 PSL modules.

[8]There are some silly differences between `__builtin__` and `__builtins__`. For Python3 use `__builtins__`.

5. **Python Package Index** (PyPI) also known as the CheeseShop is the central archive for Python packages available from https://pypi.python.org.

   The index reports that it contains over 42393 packages as of April 2014. They range from popular packages such as `lxml` and `requests` over large web frameworks, such as Django to strange packages, such as `absolute`, — a package with the sole purpose of implementing a function that computes the absolute value of a number (this functionality is already built-in with the `abs` function).

   You will often need to install the packages unless you use one of the large development frameworks such as Enthought and Anaconda or if it is already installed via your system. If you have the `pip` program up and running then installation of packages from PyPI is relatively easy: From the terminal (outside Python) you write `pip install <packagename>`, which will download, possibly compile, install and setup the package. Unsure of the package, you can write `pip search <query>` and `pip` will return a list of packages matching the query. Once you have done installed the package you will be able to use the package in Python with `>>> import <packagename>`.

   If parts of the software you are installing are written in C, then the pip install will require a C compiler to build the library files. If a compiler is not readily available you can download and install a binary pre-compiled package, — if this is available. Otherwise some systems, e.g., Ubuntu and Debian will distribute a large set of the most common package from PyPI in their pre-compiled version, e.g., the Ubuntu/Debian name of `lxml` and `requests` are called `python-lxml` and `python-requests`.

   On a typical Linux system you will find the packages installed under directories, such as `/usr/lib/python2.7/dist-packages/`

6. Other Python components. From time to time you will find that not all packages are available from the Python Package Index. Often these packages comes with a `setup.py` that allows you to install the software.

   If the bundle of Python files does not even have a `setup.py` file, you can download it a put in your own self-selected directory. The python program will not be able to discover the path to the program, so you will need to tell it. In Linux and Windows you can set the environmental variable `PYTHONPATH` to a colon- or semicolon-separated list of directories with the Python code. Windows users may also set the `PYTHONPATH` from the 'Advanced' system properies. Alternatively the Python developer can set the `sys.path` attribute from within Python. This variable contains the paths as strings in a list and the developer can append a new directory to it.

GitHub user Vinta provides a good curated list of important Python frameworks, libraries and software from https://github.com/vinta/awesome-python.

1.5 Developing and running Python

1.5.1 Python, pypy, IPython ...

Various implementations for running or translating Python code exist: CPython, IPython, IPython notebook, PyPy, Pyston, IronPython, Jython, Pyjamas, Cython, Nuitka, Micro Python, etc. CPython is the standard reference implementation and the one that you will usually work with. It is the one you start up when you write `python` at the command-line of the operating system.

The PyPy implementation `pypy` usually runs faster than standard CPython. Unfortunately PyPy does not (yet) support some of the central Python packages for data mining, `numpy` and `scipy`, although some work on the issue has apparently gone on since 2012. If you do have code that does not contain parts not supported by PyPy and with critical timing performance, then `pypy` is worth looking into. Another jit-based (and LLVM-based) Python is Dropbox's Pyston. As of April 2014 it "'works', though doesn't support very much of the Python language, and currently is not very useful for end-users." and "seems to have better performance than CPython but lags behind PyPy."[9] Though interesting, these programs are not yet so relevant in data mining applications.

Some individuals and companies have assembled binary distributions of Python and many Python package together with an integrated development environment (IDE). These systems may be particularly relevant for users without a compiler to compile C-based Python packages, e.g., many Windows users. Python(x,y) is a Windows- and scientific-oriented Python distribution with the Spyder integrated development environment. WinPython is similar system. You will find many relevant data mining package included in the WinPython, e.g., `pandas`, IPython, `numexpr`, as well as a tool to install, uninstall and upgrade packages. Continuum Analytics distributes their Anaconda and Enthought their Enthought Canopy, — both systems targeted to scientists, engineers and other data analysts. Available for the Window, Linux and Mac platforms they include what you can almost expect of such data mining environments, e.g., `numpy`, `scipy`, `pandas`, `nltk`, `networkx`. Enthought Canopy is only free for academic use. The basic Anaconda is 'completely free', while the Continuum Analytics provides some 'add-ons' that are only free for academic use. Yet another prominent commercial grade distribution of Python and Python packages is ActivePython. It seems less geared towards data mining work. For Windows users not using these systems and who do not have the ability to compile C may take a look at Christoph Gohlke's large list of precompiled binaries assembled at `http://www.lfd.uci.edu/~gohlke/pythonlibs/`.

1.5.2 Jupyter Notebook

Jupyter Notebook (previously called IPython Notebook) is a system that intermix editor, Python interactive sessions and output, similar to *Mathematica*. It is browser-based and when you install newer versions of IPython you have it available and the ability to start it from the command-line outside Python with the command `jupyter notebook`. You will get a webserver running at your local computer with the default address `http://127.0.0.1:8888` with the IPython Notebook prompt available, when you point your browser to that address. You edit directly in the browser in what Jupyter Notebook calls 'cells', where you enter lines of Python code. The cells can readily be executed, e.g., via the shift+return keyboard shortcut. Plots either appear in a new window or if you set `%matplotlib online` they will appear in the same browser window as the code. You can intermix code and plot with cells of text in the Markdown format. The entire session with input, text and output will be stored in a special JSON file format with the `.ipynb` extension, ready for distribution. You can also export part of the session with the source code as an ordinary Python source `.py` file.

Although great for interactive data mining, Jupyter Notebook is perhaps less suitable to more traditional software development where you work with multiple reuseable modules and testing frameworks.

1.5.3 Python 2 vs. Python 3

Python is in a transition phase between the old Python version 2 and the new Python version 3 of the language. Python 2 is scheduled to survive until 2020 and yet in 2014 developers responded in a survey that the still wrote more 2.x code than 3.x code [14]. Python code written for one version may not necessarily work for the other version, as changes have occured in often used keywords, classes and functions such as `print`, `range`, `xrange`, `long`, `open` and the division operator. Check out http://python3wos.appspot.com/ to get an overview of which popular modules support Python 3. 3D scientific visualization lacks good Python 3 support. The central packages, `mayavi` and the VTK wrapper, are still not available for Python 3 as of March 2015.

Some Linux distributions still default to Python 2, while also enables the installation of Python 3 making it accessible as `python3` as according to PEP 394 [15]. Although many of the major data mining Python libraries are now available for Python 3, it might still be a good idea to stick with Python 2, while keeping Python 3 in mind, by not writing code that requires a major rewrite when porting to Python 3. The idea of writing in the subset of the intersection of Python 2 and Python 3 has been called 'Python X'.[10] One

[9]https://github.com/dropbox/pyston.

[10]Stephen A. Goss, Python 3 is killing Python, https://medium.com/@deliciousrobots/5d2ad703365d/.

part of this approach uses the `__future__` module importing relevant features, e.g., `__future__.division` and `__future__.print_function` like:

```
from __future__ import division, print_function, unicode_literals
```

This scheme will change Python 2's division operator '`/`' from integer division to floating point division and the `print` from a keyword to a function.

Python X adherrence might be particular inconvenient for string-based processing, but the module `six` provides further help on the issue. For testing whether a variable is a general string, in Python 2 you would test whether the variable is an instance of the `basestring` built-in type to capture both byte-based strings (Python 2 `str` type) and Unicode strings (Python 2 `unicode` type). However, Python 3 has no `basestring` by default. Instead you test with the Python 3 `str` class which contains Unicode strings. A constant in the `six` module, the `six.string_types` captures this difference and is an example how the `six` module can help writing portable code. The following code testing for string type for a variable will work in both Python 2 and 3:

```
if isinstance(my_variable, six.string_types):
    print('my_variable␣is␣a␣string')
else:
    print('my_variable␣is␣not␣a␣string')
```

1.5.4 Editing

For editing you should have a editor that understands the basic elements of the Python syntax, e.g., to help you make correct indentation which is an essential part of the Python syntax. A large number of Python-aware editors exists,[11] e.g., Emacs and the editors in the Spyder and Eric IDEs. Commercial IDEs, such as PyCharm and Wing IDE, also have good Python editors.

For autocompletion Python has a `jedi` module, which various editors can use through a plugin. Programmers can also call it directly from a Python program. IPython and spyder features autocompletion

For collorative programming—pair programming or physically separated programming—it is worth to note that the collaborative document editor Gobby has support for Python syntax highlighting and Pythonic indentation. It features chat, but has no features beyond simple editing, e.g., you will not find support for direct execution, style checking nor debugging, that you will find in Spyder. The Rudel plugin for Emacs supports the Gobby protocol.

1.5.5 Python in the cloud

A number of websites enable programmers to upload their Python code and run it from the website. Google App Engine is perhaps the most well-known. With Google App Engine Python SDK developers can develop and test web application locally before an upload to the Google site. Data persistency is handle by a specific Google App Engine datastore. It has an associated query language called GQL resembling SQL. The web application may be constructed with the Webapp2 framework and templating via Jinja2. Further information is available in the book *Programming Google App Engine* [16]. There are several other websites for running Python in the cloud: pythonanywhere, Heroku, PiCloud and StarCluster. Freemium service Pythonanywhere provides you, e.g., with a MySQL database and, the traditional data mining packages, the Flask web framework and web-access to the server access and error logs.

1.5.6 Running Python in the browser

Some systems allow you to run Python with the webbrowser without the need for local installation. Typically, the browser itself does not run Python, instead a webservice submits the Python code to a backend system that runs the code and return the result. Such systems may allow for quick and collaborative Python development.

[11] See https://stackoverflow.com/questions/81584/what-ide-to-use-for-python for an overview of features.

The company Runnable provides a such service through the URL http://runnable.com, where users may write Python code directly in the browser and let the system executes and returns the result. The cloud service Wakari (https://wakari.io/) let users work and share cloud-based Jupyter Notebook sessions. It is a cloud version of from Continuum Analytics' Anaconda.

The *Skulpt* implementation of Python runs in a browser and a demonstration of it runs from its homepage http://www.skulpt.org/. It is used by several other websites, e.g., *CodeSkulptor* http://www.codeskulptor.org. *Codecademy* is a webservice aimed at learning to code. Python features among the programming languages supported and a series of interactive introductory tutorials run from the URL http://www.codecademy.com/tracks/python. The *Online Python Tutor* uses its interactive environment to demonstrate with program visualization how the variables in Python changes as the program is executed [17]. This may serve well novices learning the Python, but also more experienced programmer when they debug. pythonanywhere (https://www.pythonanywhere.com) also has coding in the browser.

Code Golf from http://codegolf.com/ invites users to compete by solving coding problems with the smallest number of characters. The contestants cannot see each others contributions. Another Python code challenge website is Check IO, see http://www.checkio.org

Such services have less relevance for data mining, e.g., Runnable will not allow you to import `numpy`, but they may be an alternative way to learn Python. *CodeSkulptor* implementing a subset of Python 2 allows the programmer to import the modules `numeric`, `simplegui`, `simplemap` and `simpleplot` for rudimentary matrix computations and plotting numerical data. At *Plotly* (https://plot.ly) users can collaboratively construct plots, and Python coding with Numpy features as one of the methods to build the plots.

Chapter 2

Python

2.1 Basics

Two functions in Python are important to known: `help` and `dir`. `help` shows the documentation for the input argument, e.g., `help(open)` shows the documentation for the `open` built-in function, which reads and writes files. `help` works for most elements of Python: modules, classes, variables, methods, functions, . . . , — but not keywords. `dir` will show a list of methods, constants and attributes for a Python object, and since most elements in Python are objects (but not keywords) `dir` will work, e.g., `dir(list)` shows the methods associated with the built-in `list` datatype of Python. One of the methods in the list object is `append`. You can see its documentation with `help(list.append)`.

Indentation is important in Python, — actually essential: It is what determines the block structure, so indentation limits the scope of control structures as well as class and function definitions. Four spaces is the default indentation. Although the Python semantic will work with other number of spaces and tabs for indentation, you should generally stay with four spaces.

2.2 Datatypes

Table 2.1 displays Python's basic data types together with the central data types of the Numpy and Pandas modules. The data types in the first part of table are the built-in data types readily available when python starts up. The data types in the second part are Numpy data types discussed in chapter 3, specifically in section 3.1, while the data types in the third part of the table are from the Pandas package discussed in section 3.3. An instance of a data type is converted to another type by instancing the other class, e.g., turn the float `32.2` into a string `'32.2'` with `str(32.2)` or the string `'abc'` into the list `['a', 'b', 'c']` with `list('abc')`. Not all of the conversion combinations work, e.g., you cannot convert an integer to a list. It results in a `TypeError`.

2.2.1 Booleans (`bool`)

A Boolean `bool` is either `True` or `False`. The keywords `or`, `and` and `not` should be used with Python's Booleans, — not the bitwise operations `|`, `&` and `^`. Although the bitwise operators work for `bool` they evaluate the entire expression which fails, e.g., for this code `(len(s) > 2) & (s[2] == 'e')` that checks whether the third character in the string is an 'e': For strings shorter than 3 characters an indexing error is produced as the second part of the expression is evaluated regardless of the value of the first part of the expression. The expression should instead be written `(len(s) > 2) and (s[2] == 'e')`. Values of other types that evaluates to `False` are, e.g., `0`, `None`, `''` (the empty string), `[]`, `()` (the empty tuple), {}, `0.0` and `b'\x00'`, while values evaluating to `True` are, e.g., `1`, `-1`, `[1, 2]`, `'0'`, `[0]` and `0.000000000000001`.

Built-in type	Operator	Mutable	Example	Description
`bool`		No	`True`	Boolean
`bytearray`		Yes	`bytearray(b'\x01\x04')`	Array of bytes
`bytes`	`b''`	No	`b'\x00\x17\x02'`	
`complex`		No	`(1+4j)`	Complex number
`dict`	`{:}`	Yes	`{'a': True, 45: 'b'}`	Dictionary, indexed by, e.g., strings
`float`		No	`3.1`	Floating point number
`frozenset`		No	`frozenset({1, 3, 4})`	Immutable set
`int`		No	`17`	Integer
`list`	`[]`	Yes	`[1, 3, 'a']`	List
`set`	`{}`	Yes	`{1, 2}`	Set with unique elements
`slice`	:	No	`slice(1, 10, 2)`	Slice indices
`str`	`""` or `''`	No	`"Hello"`	String
`tuple`	`(,)`	No	`(1, 'Hello')`	Tuple
Numpy type	Char	Mutable	Example	
`array`		Yes	`np.array([1, 2])`	One-, two, or many-dimensional
`matrix`		Yes	`np.matrix([[1, 2]])`	Two-dimensional matrix
`bool_`		—	`np.array([1], 'bool_')`	Boolean, one byte long
`int_`		—	`np.array([1])`	Default integer, same as C's long
`int8`	b	—	`np.array([1], 'b')`	8-bit signed integer
`int16`	h	—	`np.array([1], 'h')`	16-bit signed integer
`int32`	i	—	`np.array([1], 'i')`	32-bit signed integer
`int64`	l, p, q	—	`np.array([1], 'l')`	64-bit signed integer
`uint8`	B	—	`np.array([1], 'B')`	8-bit unsigned integer
`float_`		—	`np.array([1.])`	Default float
`float16`	e	—	`np.array([1], 'e')`	16-bit half precision floating point
`float32`	f	—	`np.array([1], 'f')`	32-bit precision floating point
`float64`	d	—		64-bit double precision floating point
`float128`	g	—	`np.array([1], 'g')`	128-bit floating point
`complex_`		—		Same as `complex128`
`complex64`		—		Single precision complex number
`complex128`		—	`np.array([1+1j])`	Double precision complex number
`complex256`		—		2 128-bit precision complex number
Pandas type		Mutable	Example	Description
`Series`		Yes	`pd.Series([2, 3, 6])`	One-dimension (vector-like)
`DataFrame`		Yes	`pd.DataFrame([[1, 2]])`	Two-dimensional (matrix-like)
`Panel`		Yes	`pd.Panel([[[1, 2]]])`	Three-dimensional (tensor-like)
`Panel4D`		Yes	`pd.Panel4D([[[[1]]]])`	Four-dimensional

Table 2.1: Basic built-in and Numpy and Pandas datatypes. Here `import numpy as np` and `import pandas as pd`. Note that Numpy has a few more datatypes, e.g., time delta datatype.

2.2.2 Numbers (`int`, `float`, `complex` and `Decimal`)

In standard Python integer numbers are represented with the `int` type, floating-point numbers with `float` and complex numbers with `complex`. Decimal numbers can be represented via classes in the `decimal` module, particularly the `decimal.Decimal` class. In the `numpy` module there are datatypes where the number of bytes representing each number can be specified.

Numbers for `complex` built-in datatype can be written in forms such as `1j`, `2+2j`, `complex(1)` and `1.5j`.

The different packages of Python confusingly handle complex numbers differently. Consider three different implementations of the square root function in the `math`, `numpy` and `scipy` packages computing the square root of -1:

```
>>> import math, numpy, scipy
>>> math.sqrt(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
>>> numpy.sqrt(-1)
__main__:1: RuntimeWarning: invalid value encountered in sqrt
nan
>>> scipy.sqrt(-1)
1j
```

Here there is an exception for the `math.sqrt` function, `numpy` returns a NaN for the float input while `scipy` the imaginary number. The `numpy.sqrt` function may also return the imaginary number if—instead of the `float` input number it is given a `complex` number:

```
>>> numpy.sqrt(-1+0j)
1j
```

Python 2 has `long`, which is for long integers. In Python 2 `int(12345678901234567890)` will switch ton a variable with long datatype. In Python 3 `long` has been subsumed in `int`, so `int` in this version can represent arbitrary long integers, while the `long` type has been removed. A workaround to define long in Python 3 is simply `long = int`.

2.2.3 Strings (`str`)

Strings may be instanced with either single or double quotes. Multiline strings are instanced with either three single or three double quotes. The style of quoting makes no difference in terms of data type.

```
>>> s = "This␣is␣a␣sentence."
>>> t = 'This␣is␣a␣sentence.'
>>> s == t
True
>>> u = """This is a sentence."""
>>> s == u
True
```

The issue of multibyte Unicode and byte-strings yield complexity. Indeed Python 2 and Python 3 differ (unfortunately!) considerably in their definition of what is a Unicode strings and what is a byte strings.

The triple double quotes are by convention used for docstrings. When Python prints out a it uses single quotes, — unless the string itself contains a single quote.

2.2.4 Dictionaries (`dict`)

A dictionary (`dict`) is a mutable data structure where values can be indexed by a key. The value can be of any type, while the key should be *hashable*, which all immutable objects are. It means that, e.g., strings, integers, `tuple` and `frozenset` can be used as dictionary keys. Dictionaries can be instanced with `dict` or with curly braces:

```
>>> dict(a=1, b=2)                    # strings as keys, integers as values
{'a': 1, 'b': 2}
>>> {1: 'january', 2: 'february'}     # integers as keys
{1: 'january', 2: 'february'}
>>> a = dict()                        # empty dictionary
>>> a[('Friston', 'Worsley')] = 2     # tuple of strings as keys
>>> a
{('Friston', 'Worsley'): 2}
```

Dictionaries may also be created with dictionary comprehensions, here an example with a dictionary of lengths of method names for the float object:

```
>>> {name: len(name) for name in dir(float)}
{'__int__': 7, '__repr__': 8, '__str__': 7, 'conjugate': 9, ...
```

Iterations over the keys of the dictionary are immediately available via the object itself or via the `dict.keys` method. Values can be iterated with the `dict.values` method and both keys and values can be iterated with the `dict.items` method.

Dictionary access shares some functionality with object attribute access. Indeed the attributes are accessible as a dictionary in the `__dict__` attribute:

```
>>> class MyDict(dict):
...     def __init__(self):
...         self.a = None
>>> my_dict = MyDict()
>>> my_dict.a
>>> my_dict.a = 1
>>> my_dict.__dict__
{'a': 1}
>>> my_dict['a'] = 2
>>> my_dict
{'a': 2}
```

In the Pandas library (see section 3.3) columns in its `pandas.DataFrame` object can be accessed both as attributes and as keys, though only as attributes if the key name is a valid Python identifier, e.g., strings with spaces or other special characters cannot be attribute names. The `addict` package provides a similar functionality as in Pandas:

```
>>> from addict import Dict
>>> paper = Dict()
>>> paper.title = 'The␣functional␣anatomy␣of␣verbal␣initiation'
>>> paper.authors = 'Nathaniel-James,␣Fletcher,␣Frith'
>>> paper
{'authors': 'Nathaniel-James,␣Fletcher,␣Frith',
 'title': 'The␣functional␣anatomy␣of␣verbal␣initiation'}
>>> paper['authors']
'Nathaniel-James,␣Fletcher,␣Frith'
```

The advantage of accessing dictionary content as attributes is probably mostly related to ease of typing and readability.

2.2.5 Dates and times

There are only three options for
representing datetimes in data:
1) unix time 2) iso 8601
3) summary execution.
Alice Maz, 2015

There are various means to handle dates and times in Python. Python provides the `datetime` module with the `datetime.datetime` class (the class is confusingly called the same as the module). The `datetime.datetime` class records date, hours, minutes, seconds, microseconds and time zone information, while `datetime.date` only handles dates. As an example consider computing the number of days from 15 January 2001 to 24 September 2014. `datetime.date` makes such a computation relatively straightforward:

```
>>> from datetime import date
>>> date(2014, 9, 24) - date(2001, 1, 15)
datetime.timedelta(5000)
>>> str(date(2014, 9, 24) - date(2001, 1, 15))
'5000␣days,␣0:00:00'
```

i.e., 5000 days from the one date to the other. A function in the `dateutil` module converts from date and times represented as strings to `datetime.datetime` objects, e.g., `dateutil.parser.parse('2014-09-18')` returns `datetime.datetime(2014, 9, 18, 0, 0)`.

Numpy has also a datatype to handle dates, enabling easy date computation on multiple time data, e.g., below we compute the number of days for two given days given a starting date:

```
>>> import numpy as np
>>> start = np.array(['2014-09-01'], 'datetime64')
>>> dates = np.array(['2014-12-01', '2014-12-09'], 'datetime64')
>>> dates - start
array([91, 99], dtype='timedelta64[D]')
```

Here the computation defaults to represent the timing with respect to days.

A `datetime.datetime` object can be turned into a ISO 8601 string format with the `datetime.datetime.isoformat` method but simply using `str` may be easier:

```
>>> from datetime import datetime
>>> str(datetime.now())
'2015-02-13␣12:21:22.758999'
```

To get rid of the part with milliseconds use the `replace` method:

```
>>> str(datetime.now().replace(microsecond=0))
'2015-02-13␣12:22:52'
```

2.2.6 Enumeration

Python 3.4 has an enumeration datatype (symbolic members) with the `enum.Enum` class. In previous versions of Python enumerations were just implemented as integers, e.g., in the `re` regular expression module you would have a flag such as `re.IGNORECASE` set to the integer value 2. For older versions of Python the `enum34` pip package can be installed which contains an `enum` Python 3.4 compatible module.

Below is a class called `Grade` derived from `enum.Enum` and used as a label for the quality of an apple, where there are three fixed options for the quality:

```
from enum import Enum

class Grade(Enum):
    good = 1
    bad = 2
    ok = 3
```

After the definition

```
>>> apple = {'quality': Grade.good}
>>> apple['quality'] is Grade.good
True
```

2.2.7 Other containers classes

Outside the builtins the module `collections` provides a few extra interesting general container datatypes (classes). `collections.Counter` can, e.g., be used to count the number of times each word occur in a word list, while `collections.deque` can act as ring buffer.

2.3 Functions and arguments

Functions are defined with the keyword `def` and the `return` argument specifies which object the function should return, — if any. The function can be specified to have multiple, positional and keyword (named) input arguments and optional input arguments with default values can also be specified. As with control structures indentation marks the scope of the function definition.

Functions can be called recursively, but the are usually slower than their iterative counterparts and there is by default a recursion depth limit on 1000.

2.3.1 Anonymous functions with `lambdas`

One-line anonymous function can be defined with the `lambda` keyword, e.g., the definition of the polynomial $f(x) = 3x^2 - 2x - 2$ could be done with a compact definition like `f = lambda x: 3*x**2 - 2*x - 2`. The variable before the colon is the input argument and the expression after the colon is the returned value. After the definition we can call the function `f` like an ordinary function, e.g., `f(3)` will return `19`.

Functions can be manipulated like Python's other objects, e.g., we can return a function from a function. Below the `polynomial` function returns a function with fixed coefficients:

```
def polynomial(a, b, c):
    return lambda x: a*x**2 + b*x + c

f = polynomial(3, -2, -2)
f(3)
```

2.3.2 Optional function arguments

The `*` can be used to catch multiple optional positional and keyword arguments, where the standard names are `*args` and `**kwargs`. This trick is widely used in the Matplotlib plotting package. An example is shown below where a user function called `plot_dirac` is defined which calls the standard Matplotlib plotting function (`matplotlib.pyplot.plot` with the alias `plt.plot`), so that we can call `plot_dirac` with the `linewidth` keyword and pipe it further on to the Matplotlib function to control the line width of line that we are plotting:

```
import matplotlib.pyplot as plt

def plot_dirac(location, *args, **kwargs):
    print(args)
    print(kwargs)
    plt.plot([location, location], [0, 1], *args, **kwargs)

plot_dirac(2)
plt.hold(True)
plot_dirac(3, linewidth=3)
plot_dirac(-2, 'r--')
plt.axis((-4, 4, 0, 2))
plt.show()
```

In the first call to `plot_dirac` `args` and `kwargs` with be empty, i.e., an empty tuple and and empty dictionary. In the second called `print(kwargs)` will show `'linewidth': 3` and in the third call we get `('r--',)` from the `print(args)` statement.

The above `polynomial` function can be changed to accept a variable number of positional arguments so polynomials of any order can be returned from the polynomial construction function:

```
def polynomial(*args):
    expons = range(len(args))[::-1]
    return lambda x: sum([coef*x**expon for coef, expon in zip(args, expons)])
```

Method	Operator	Description
`__init__`	ClassName()	Constructor, called when an instance of a class is made
`__del__`	`del`	Destructor
`__call__`	object_name()	The method called when the object is a function, i.e., 'callable'
`__getitem__`	`[]`	Get element: `a.__getitem__(2)` the same as `a[2]`
`__setitem__`	`[] =`	Set element: `a.__setitem__(1, 3)` the same as `a[1] = 3`
`__contains__`	`in`	Determine if element is in container
`__str__`		Method used for `print` keyword/function
`__abs__`	`abs()`	Method used for absolute value
`__len__`	`len()`	Method called for the `len` (length) function
`__add__`	`+`	Add two objects, e.g., add two numbers or concatenate two strings
`__iadd__`	`+=`	Addition with assignment
`__div__`	`/`	Division (In Python 2 integer division for `int` by default)
`__floordiv__`	`//`	Integer division with floor rounding
`__pow__`	`**`	Power for numbers, e.g., `3 ** 4` $= 3^4 = 81$
`__and__`	`&`	Method called for and operator '&'
`__eq__`	`==`	Test for equality.
`__lt__`	`<`	Less than
`__le__`	`<=`	Less than or equal
`__xor__`	`^`	Exclusive or. Works bitwise for integers and binary for Booleans
...		
Attribute	—	Description
`__class__`		Class of object, e.g., `<type 'list'>` (Python 2), `<class 'list'>` (3)
`__doc__`		The documention string, e.g., used for `help()`

Table 2.2: Class methods and attributes. These names are available with the `dir` function, e.g., `an_integer = 3; dir(an_integer)`.

```
f = polynomial(3, -2, -2)
f(3)                        # Returned result is 19
f = polynomial(-2)
f(3)                        # Returned result is -2
```

2.4 Object-oriented programming

Almost everything in Python is an object, e.g., integer, strings and other data types, functions, class definitions and class methods are objects. These objects have associated methods and attributes, and some of the default methods and functions follow a specific naming pattern with pre- and postfixed double underscore. Table 2.2 gives an overview of some of the methods and attributes in an object. As always the `dir` function lists all the methods defined for the object. Figure 2.1 shows another overview of the method in the common built-in data types in a formal concept analysis lattice graph. The graph is constructed with the `concepts` module which uses the `graphviz` module and Graphviz program. The plot shows, e.g., that `int` and `bool` define the same methods (their implementations are of course different), that `__format__` and `__str__` are defined by all data types and that `__contains__` and `__len__` are available for `set`, `dict`, `list`, `tuple` and `str`, but not for `bool`, `int` and `float`.

Developers can define their own classes with the `class` keyword. The class definitions can take advantage of multiple inheritance. Methods of the defined class is added to the class with the `def` keyword in the indented block of the class. New classes may be derived from built-in data types, e.g., below a new integer

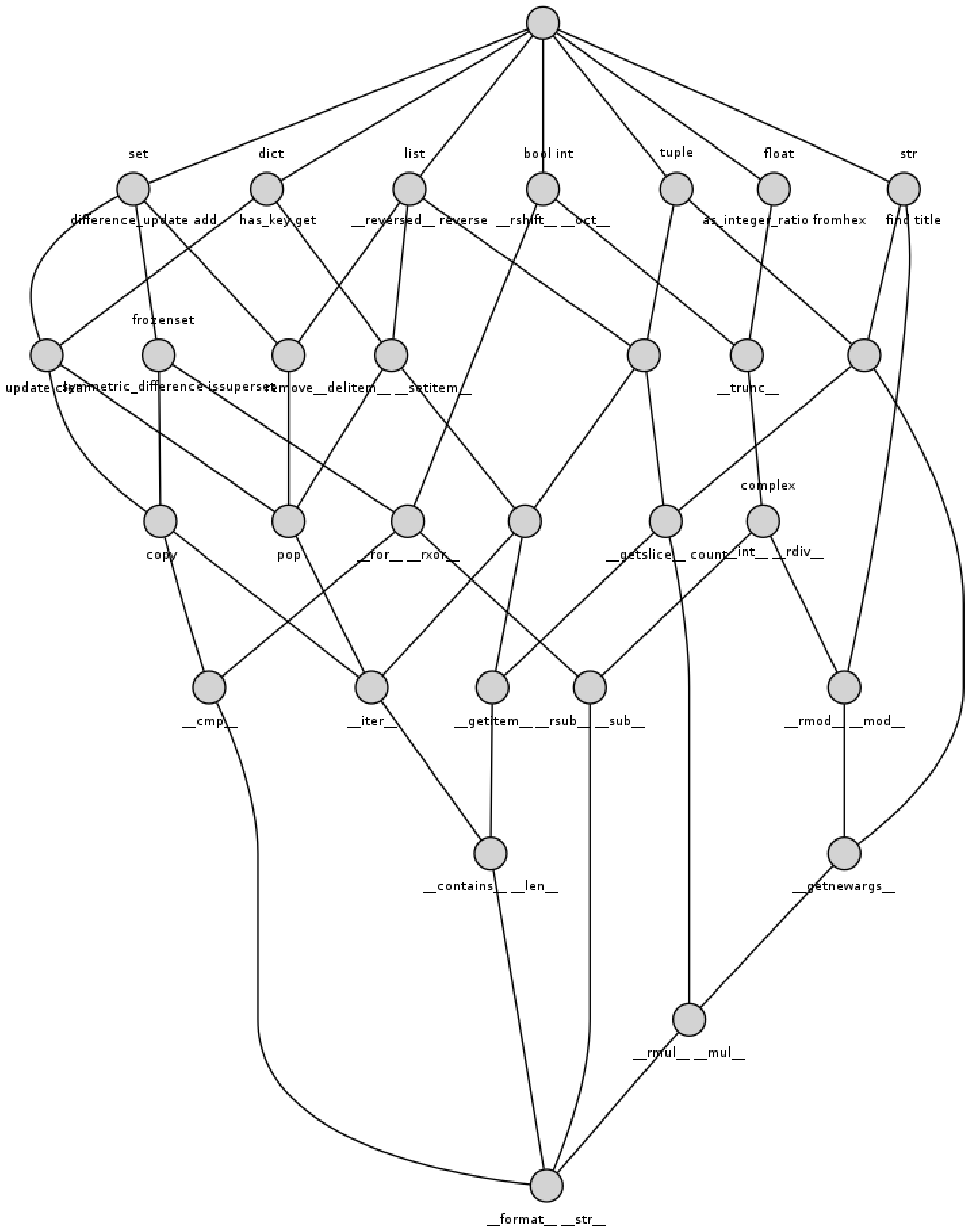

Figure 2.1: Overview of methods and attributes in the common Python 2 built-in data types plotted as a formal concept analysis lattice graph. Only a small subset of methods and attributes is shown.

class is defined with a definition for the length method:

```
>>> class Integer(int):
>>>     def __len__(self):
>>>         return 1
>>> i = Integer(3)
>>> len(i)
1
```

2.4.1 Objects as functions

Any object can be turned into a function by defining the `__call__` method. Here we derive a new class from the `str` data type/class defining the `__call__` method to split the string into words and return a word indexed by the input argument:

```
class WordsString(str):
    def __call__(self, index):
        return self.split()[index]
```

After instancing the `WordString` class with a string we can call the object to let it return, e.g., the fifth word:

```
>>> s = WordsString("To␣suppose␣that␣the␣eye␣will␣all␣its␣inimitable␣contrivances")
>>> s(4)
'eye'
```

Alternatively we could have defined an ordinary method with a name such as `word` and called the object as `s.word(4)`, — a slightly longer notation, but perhaps more readable and intuitive for the user of the class compared to the surprising use with the `__call__` method.

2.5 Modules and import

"A module is a file containing Python definitions and statements."[1] The file should have the extension `.py`. A Python developer should group classes, constants and functions into meaningful modules with meaningful names. To use a module in another Python script, module or interactive sessions they should be imported with the `import` statement.[2] For example, to import the `os` module write:

```
import os
```

The file associated with the module is available in the `__file__` attribute; in the example that would be `os.__file__`. While standard Python 2 (CPython) does not make this attribute available for builtin modules it is available in Python 3 and in this case link to the `os.py` file.

Individual classes, attributes and functions can be imported via the `from` keyword, e.g., if we only need the `os.listdir` function from the `os` module we could write:

```
from os import listdir
```

This import variation will make the `os.listdir` function available as `listdir`.

If the package contains submodules then they can be imported via the dot notation, e.g., if we want names from the tokenization part of the NLTK library we can include that submodule with:

```
import nltk.tokenize
```

The imported modules, class and functions can be renamed with the `as` keyword. By convention several data mining modules are aliased to specific names:

[1]6. Modules in The Python Tutorial
[2]Unless built-in.

```
import numpy as np
import matplotlib.pyplot as plt
import networkx as nx
import pandas as pd
import statsmodels.api as sm
import statsmodels.formula.api as smf
```

With these aliases Numpy's `sin` function will be avaiable under the name `np.sin`.

Import statements should occur before imported name is used. They are usually placed at the top of the file, but this is only a style convention. Import of names from the special `__future__` module should be at the very top. Style checking tool flake8 will help on checking conventions for imports, e.g., it will complain about unused import, i.e., if a module is imported but the names in it are never used in the importing module. The `flake8-import-order` flake8 extension even pedantically checks for the ordering of the imports.

2.5.1 Submodules

If a package contains of a directory tree then subdirectories can be used as submodules. For older versions of Python is it necessary to have a `__init__.py` file in each subdirectory before Python recognizes the subdirectories as submodules. Here is an example of a module, `imager`, which contains three submodules in two subdirectories:

```
/imager
    __init__.py
    /io
        __init__.py
        jpg.py
    /process
        __init__.py
        factorize.py
        categorize.py
```

Provided that the module `imager` is available in the path (`sys.path`) the `jpg` module will now be available for import as

```
import imager.io.jpg
```

Relative imports can be used inside the package. Relative import are specified with single or double dots in much the same way as directory navigation, e.g., a relative import of the `categorize` and `jpg` modules from the `factorize.py` file can read:

```
from . import categorize
from ..io import jpg
```

Some developers encourage the use of relative imports because it makes refactoring easier. On the other hand can relative imports cause problems if circular import dependencies between the modules appear. In this latter case absolute imports work around the problem.

Name clashes can appear: In the above case the `io` directory shares name with the `io` module of the standard library. If the file `imager/__init__.py` writes '`import io`' it is not immediately clear for the novice programmer whether it is the standard library version of `io` or the imager module version that Python imports. In Python 3 it is the standard library version. The same is the case in Python 2 if the '`from __future__ import absolute_import`' statement is used. To get the imager module version, `imager.io`, a relative import can be used:

```
from . import io
```

Alternatively, an absolute import with `import imager.io` will also work.

2.5.2 Globbing import

In interactive data mining one sometimes imports everything from the pylab module with 'from pylab import *'. pylab is actually a part of Matplotlib (as matplotlib.pylab) and it imports a large number of functions and class from the numerical and plotting packages of Python, i.e., numpy and matplotlib, so the definitions are readily available for use in the namespace without module prefix. Below is an example where a sinusoid is plotted with Numpy and Matplotlib functions:

```
from pylab import *

t = linspace(0, 10, 1000)
plot(t, sin(2 * pi * 3 * t))
show()
```

Some argue that the massive import of definitions with 'from pylab import *' pollutes the namespace and should not be used. Instead they argue you should use explicit import, like:

```
from numpy import linspace, pi, sin
from matplotlib.pyplot import plot, show

t = linspace(0, 10, 1000)
plot(t, sin(2 * pi * 3 * t))
show()
```

Or alternatively you should use prefix, here with an alias:

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

t = np.linspace(0, 10, 1000)
plt.plot(t, np.sin(2 * np.pi * 3 * t))
plt.show()
```

This last example makes it more clear where the individual functions comes from, probably making large Python code files more readable. With 'from pylab import *' it is not immediately clear the the load function comes from, — in this case the numpy.lib.npyio module which function reads pickle files. Similar named functions in different modules can have different behavior. Jake Vanderplas pointed to this nasty example:

```
>>> start = -1
>>> sum(range(5), start)
9
>>> from numpy import *
>>> sum(range(5), start)
10
```

Here the built-in sum function behaves differently than numpy.sum as their interpretations of the second argument differ.

2.5.3 Coping with Python 2/3 incompatibility

There is a number of modules that have changed their name between Python 2 and 3, e.g., ConfigParser/configparser, cPickle/pickle and cStringIO/StringIO/io. Exception handling and aliasing can be used to make code Python 2/3 compatible:

```
try:
    import ConfigParser as configparser
except ImportError:
    import configparser
```

```
try:
    from cStringIO import StringIO
except ImportError:
    try:
        from StringIO import StringIO
    except ImportError:
        from io import StringIO

try:
    import cPickle as pickle
except ImportError:
    import pickle
```

After these imports you will, e.g., have the configuration parser module available as `configparser`.

2.6 Persistency

How do you store data between Python sessions? You could write your own file reading and writing function or perhaps better rely on Python function in the many different modules, Python PSL, supports comma-separated values files (`csv` in PSL and `csvkit` that will handle UTF-8 encoded data) and JSON (`json`). PSL also has several XML modules, but developers may well prefer the faster `lxml` module, — not only for XML, but also for HTML [18].

2.6.1 Pickle and JSON

Python also has its own special serialization format called pickle. This format can store not only data but also objects with methods, e.g., it can store a trained machine learning classifier as an object and indeed you can discover that the `nltk` package stores a trained part-of-speech tagger as a pickled file. The power of pickle is also its downside: Pickle can embed dangerous code such as system calls that could erase your entire harddrive, and because of this issue the pickle format is only suitable for trusted code. Another downside is that it is a format mostly for Python with little support in other languages.[3] Also note that pickle comes with different protocols: If you store a pickle in Python 3 with the default setting you will not be able to load it with the standard tools in Python 2. The highest protocol version is 4 and featured in Python 3.4 [19]. Python 2 has two modules to deal with the pickle format, `pickle` and `cPickle`, where the latter is the prefered as it runs faster, and for compatibility reasons you would see imports like:

```
try:
    import cPickle as pickle
except ImportError:
    import pickle
```

where the slow pure Python-based is used as a fallback if the fast C-based version is not available. Python 3's `pickle` does this 'trick' automatically.

The open standard JSON (JavaScript Object Notation) has—as the name implies—its foundations in Javascript, but the format maps well to Python data types such as strings, numbers, list and dictionaries. JSON and Pickle modules have similar named functions: `load`, `loads`, `dump` and `dumps`. The `load` functions load objects from file-like objects into Python objects and `loads` functions load from string objects, while the `dump` and `dumps` functions 'save' to file-like objects and strings, respectively.

There are several JSON I/O modules for Python. Jonas Tärnström's `ujson` may perform more than twice as fast as Bob Ippolito's conventional `json/simplejson`. Ivan Sagalaev's `ijson` module provides a streaming-based API for reading JSON files, enabling the reading of very large JSON files which does not fit in memory.

[3] pickle-js, https://code.google.com/p/pickle-js/, is a Javascript implementation supporting a subset of primitive Python data types.

Note the few gotchas for the use of JSON in Python: while Python can use strings, Booleans, numbers, tuples and frozensets (i.e., hashable types) as keys in dictionaries, JSON can only handle strings. Python's `json` module converts numbers and Booleans to string representation in JSON, e.g., `json.loads(json.dumps({1: 1}))` returns the number used as key to a string: `{u'1': 1}`. A data type such as a tuple used as key will result in a `TypeError` when used to dump data to JSON. Numpy data type yields another JSON gotcha relevant in data mining. The `json` does not support, e.g., Numpy 32-bit floats, and with the following code you end up with a `TypeError`:

```
import json, numpy
json.dumps(numpy.float32(1.23))
```

Individual `numpy.float64` and `numpy.int` works with the `json` module, but Numpy arrays are not directly supported. Converting the array to a list may help

```
>>> json.dumps(list(numpy.array([1., 2.])))
'[1.0,␣2.0]'
```

Rather than `list` it is better to use the `numpy.array.tolist` method, which also works for arrays with dimensions larger than one:

```
>>> json.dumps(numpy.array([[1, 2], [3, 4]]).tolist())
'[[1,␣2],␣[3,␣4]]'
```

2.6.2 SQL

For interaction with SQL databases Python has specified a standard: The Python Database API Specification version 2 (DBAPI2) [20]. Several modules each implement the specification for individual database engines, e.g., SQLite (`sqlite3`), PostgreSQL (`psycopg2`) and MySQL (`MySQLdb`).

Instead of accessing the SQL databases directly through DBAPI2 you may use a object-relational mapping (ORM, aka object relation manager) encapsulating each SQL table with a Python class. Quite a number of ORM packages exist, e.g., `sqlobject`, `sqlalchemy`, `peewee` and `storm`. If you just want to read from an SQL database and perform data analysis on its content, then the `pandas` package provides a convenient SQL interface, where the `pandas.io.sql.read_frame` function will read the content of a table directly into a `pandas.DataFrame`, giving you basic Pythonic statistical methods or plotting just one method call away.

Greg Lamp's neat module, `db.py`, works well for exploring databases in data analysis applications. It comes with the Chinook SQLite demonstration database. Queries on the data yield `pandas.DataFrame` objects (see section 3.3).

2.6.3 NoSQL

Python can access NoSQL databases through modules for, e.g., MongoDB (`pymongo`). Such systems typically provide means to store data in a 'document' or schema-less way with JSON objects or Python dictionaries. Note that ordinary SQL RDMS can also store document data, e.g., FriendFeed has been storing data as zlib-compressed Python pickle dictionaries in a MySQL BLOB column.[4]

2.7 Documentation

Documentation features as an integral part of Python. If you setup the documentation correctly the Python execution environment has access to the documentation and may make the documentation available to the programmer/user in a variety of ways. Python can even use parts of the documentation, e.g., to test the code or produce functionality that the programmer would otherwise put in the code, examples include specifying an example use and return argument for automated testing with the `doctest` package or specifying script input argument schema parseable with the `docopt` module.

[4]http://backchannel.org/blog/friendfeed-schemaless-mysql.

Concept	Description
Unit testing	Testing each part of a system separately
Doctesting	Testing with small test snippets included in the documentation
Test discovery	Method, that a testing tools will use, to find which part of the code should be executed for testing.
Zero-one-some	Test a list input argument with zero, one and several elements
Coverage	Lines of codes tested compared to total number of lines of code

Table 2.3: Testing concepts

Programmers should not invent their own style of documentation but write to the standards of the Python documentation. PEP 257 documents the primary conventions for docstrings [21], and Vladimir Keleshev's `pydocstyle` tool (initially called `pep257`) will test if your documentation conforms to that standard. Numpy follows further docstring conventions which yield a standardized way to describe the input and return arguments, coding examples and description. It uses the reStructuredText text format. `pydocstyle` does not test for the Numpy convention.

Once (or while) your have documented your code properly you can translate it into several different formats with one of the several Python documentation generator tools, e.g., to HTML for an online help system. The Python Standard Library features the `pydoc` module, while Python Standard Library itself uses the popular *Sphinx* tool.

2.8 Testing

2.8.1 Testing for type

In data mining applications numerical list-like objects can have different types: list of integers, list of floats, list of booleans and Numpy arrays or Numpy matrices with different types of elements. Proper testing should cover all relevant input argument types. Below is an example where a `mean_diff` function is tested in the `test_mean_diff` function for both floats and integers:

```python
from numpy import max, min

def mean_diff(a):
    """Compute the mean difference in a sequence.

    Parameters
    ----------
    a : array_like

    """
    return float((max(a) - min(a)) / (len(a) - 1))

def test_mean_diff():
    assert mean_diff([1., 7., 3., 2., 5.]) == 1.5
    assert mean_diff([7, 3, 2, 1, 5]) == 1.5
```

The test fails in Python 2 because the parenthesis for the `float` class is not correct, so the division becomes an integer division. Either we need to move the parenthesis or we need to specify `from __future__ import division`. There are a range of other types we can test for in this case, e.g., should it work for Booleans and then what should be the result? Should it work for Numpy and Pandas data types? Should it work for higher order data types such as matrices, tensors and/or list of lists? A question is also what data type should be returned, — in this case it is always a float, but if the input was `[2, 4]` we could have returned an integer (2 rather than `2.0`).

2.8.2 Zero-one-some testing

The testing pattern *zero-one-some* attempts to ensure coverage for variables which may have multiple elements. The pattern says you should test with zeros elements, one elements and 'some' (2 or more) elements. The listing below shows the `test_mean` function testing an attempt on a `mean` function with the three zero-one-some cases:

```
def mean(x):
    return float(sum(x))/len(x)

import numpy as np

def test_mean():
    assert np.isnan(mean([]))
    assert mean([4.2]) == 4.2
    assert mean([1, 4.3, 4]) == 3.1
```

Here the code fails with a `ZeroDivisionError` already at the first `assert` as the `mean` function does not handle the case for a zero-element list.

A fix for the zero division uses exception handling catching the raised `ZeroDivisionError` and returning a Numpy not-a-number (`numpy.nan`). Below is the test included as a doctest in the docstring of the function implementation:

```
import numpy as np

def mean(x):
    """Compute mean of list of numbers.

    Examples
    --------
    >>> np.isnan(mean([]))
    True
    >>> mean([4.2])
    4.2
    >>> mean([1, 4.3, 4])
    3.1

    """
    try:
        return float(sum(x))/len(x)
    except ZeroDivisionError:
        return np.nan
```

If we call the file `doctestmean.py` we can then perform doctesting by invoking the `doctest` module on the file by `python -m doctest doctestmean.py`. This will report no output if no errors occur.

2.8.3 Test layout and test discovery

Python has a common practice for test layout and test discovery. Test layout is the schema for where you put and name you testing modules, classes and functions. Using a standard layout will help readers of your code to navigate and find the testing function and will ensure that testing tools can automatically identify classes, functions and method that should be executed to test your implementation, i.e., help test discovery.

`py.test` supports two test directory layouts, see http://pytest.org/latest/goodpractises.html. One where you put a `tests` directory on the same level as the package:

```
setup.py   # your distutils/setuptools Python package metadata
mypkg/
    __init__.py
    appmodule.py
    ...
tests/
    test_app.py
    ...
```

For the other layout, the 'inlining test directories', you put a `test` directory on the same level as the module:

```
setup.py   # your distutils/setuptools Python package metadata
mypkg/
    __init__.py
    appmodule.py
    ...
    test/
        test_app.py
        ...
```

This second method allows you to distribute the test together with the implementation, letting other developers use your tests as part of the application. In this case you should also add an `__init__.py` file to the `test` directory. For both layouts, files, methods and functions should be prefixed with `test_` for the test discovery, while test classes whould be prefixed with `Test`.

In data mining where you work with machine learning training and test set, you should be careful not to name your ordinary (non-testing) function with a pre- or postfix of 'test', as this may invoke testing when you run the test from the package level.

2.8.4 Test coverage

Test coverage tells you the fraction of code tested, and a developer would hope to reach 100% coverage. Provided you already have created the test function a Python tool exists for easy reporting of test coverage: the `coverage` package. Consider the following Python file `numerics.py` file with an obviously erroneous `compute` function tested with the function `test_compute`: We can test this with the standard `py.test` tools and find that it reports no errors:

```
> py.test numerics.py
============================ test session starts ==============================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5
collected 1 items

numerics.py .

=========================== 1 passed in 0.03 seconds ===========================
```

For some systems you will find that the `coverage` setup installs the central script as `python-coverage`. You can execute this script from the command-line, first calling it with the `run` command argument and the filename of the Python source, and then with the `report` command argument:

```
> python-coverage run numerics.py
> python-coverage report -m
Name                                     Stmts   Miss  Cover   Missing
----------------------------------------------------------------------
/usr/share/pyshared/coverage/collector     132    127     4%
3-229, 236-244, 248-292
/usr/share/pyshared/coverage/control       236    235     1%   3-355, 358-624
/usr/share/pyshared/coverage/execfile       35     16    54%
3-17, 42-43, 48, 54-65
numerics                                     8      1    88%   4
----------------------------------------------------------------------
TOTAL                                      411    379     8%
```

The -m optional command line argument reports the line numbers missing in the test. It reports that the test did not cover line 4 in numerics. This is because we did not test for the case with x = 2 so the block with the if condicional would be executed.

With the coverage module installed, the nose package can also report the coverage. Here we use the command line script nosetests with a optional input argument:

```
> nosetests --with-cover numerics.py
.
Name        Stmts   Miss  Cover   Missing
----------------------------------------
numerics        8      1    88%   4
----------------------------------------------------------------------
Ran 1 test in 0.003s

OK
```

A coverage plugin for py.test is also available, so the coverage for a module which contains test function may be measured with the --cov option:

```
> py.test --cov themodule
```

The specific lines that the test is missing to test can be show with an option to the report command:

```
> coverage report --show-missing
```

2.8.5 Testing in different environments

It is a ‘good thing’ if a module works in several different environments, e.g., different versions of Python. Virtual environments can be setup and tests executed in the environments. The tox program greatly simplifies the process allowing the developer to test the code in multiple versions of Python installations without much hassle after the tox initialization is setup.

If test functions and a setup.py package file are set up and py.text installed then tox will automatically create the virtual environments for testing and perform the actually testing in each of them depending on a specification in the tox.ini configuration file. After the one-time setup of setup.py and tox.ini any subsequent testing needs only to call the command-line program tox for the all the test to run.

Data mining application may require expensive compilation of Numpy an SciPy in each virtual environment. Tox can be setup so the virtual environment borrows the site packages, rather than installing new versions in each of the virtual environments.

2.9 Profiling

Various functions in the PSL time module allow the programmer to measure the timing performance of the code. One relevant function time.clock times ‘processor time’ on Unix-like system and elapsed wall-clock seconds since the first call to the function on Windows. Since version 3.3 Python has deprecated this function and instead encourages using the new functions time.perf_counter or time.process_time.

For short code snippets the time may not yield sufficient timing resolution, and the PSL timeit module will help the developer profiling such snippets by executing the code many times and time the total wall clock time with the timeit.timeit function.

The listing below applies timeit on code snippets taken from Google Python style guide example code [22]. The two different code snippets have similar functionality but implement it with for loops and list comprehension, respectively.

```
def function1():
    result = []
    for x in range(10):
        for y in range(5):
            if x * y > 10:
                result.append((x, y))
function1.name = "For␣loop␣version"

def function2():
    result = [(x, y) for x in range(10) for y in range(5) if x * y > 10]
function2.name = "List␣comprehension␣version"


import timeit

for func in [function1, function2]:
    print("{:26}␣=␣{:5.2f}".format(func.name, timeit.timeit(func)))
```

An execution of the code will show that the list comprehension version performs slightly faster:

```
$ python timeit_example.py
For loop version           = 10.14
List comprehension version =  8.67
```

By default `timeit.timeit` will execute the input argument one million times and report the total duration in seconds, thus in this case each execution of each code snippets takes around 10 microseconds.

Note that such control structure-heavy code can run considerably faster with the `pypy` Python implementation as opposed to the standard Python `python` implementation: `pypy` can gain a factor of around five.

```
$ pypy timeit_example.py
For loop version           =  2.26
List comprehension version =  1.93
```

`time` and `timeit` measure only a single number. If you want to measure timing performance for, e.g., each function called during an execution of a script then use `profile` or `cProfile` modules. `profile`, the pure Python implementation, has more overhead, so unless you find `cProfile` unavailable on your system, use `cProfile`. The associated `pstats` modules has method for displaying the result of the profiling from `profile` and `cProfile`. You can run these profiling tools both from within Python and by calling it from the shell command line. You will also find that some IDEs make the profiling functionality directly available, e.g., in Spyder profiling is just the F10 keyboard shortcut away. An example with profiling a module called `dynamatplotlib.py` with `cProfile` from the shell command line reads in one line:

```
$ python -m cProfile dynamatplotlib.py
```

It produces list with timing information for each invididual component of the program. It you wish to sort the list according to execution time of the individual parts then use the `-s` option:

```
$ python -m cProfile -s time dynamatplotlib.py
```

The report of the profiling may be long and difficult to get an overview of. The profiler can instead write the profiling report to a binary file which the `pstats` module can read and interact with. The follow combines `cProfile` and `pstats` for showing statistics about the ten longest running line in terms of cumulated time:

```
$ python -m cProfile -o dynamatplotlib.profile dynamatplotlib.py
$ python -m pstats dynamatplotlib.profile
dynamatplotlib.profile% sort cumulative
dynamatplotlib.profile% stats 10
```

The `pstats` module spawns a small command line utility where ‘help’, ‘sort’ and ‘stats’ are among the commands.

IPython has a magic function for quick and easy profiling of a function with `timeit`. The below IPython code tests with the `%timeit` magic function how well the scalar `sin` function from the `math` module performs against the vectorized `sin` version in the `numpy` module for a scalar input argument:

```
In [1]: from math import sin

In [2]: %timeit sin(3)
10000000 loops, best of 3: 89.6 ns per loop

In [3]: from numpy import sin

In [4]: %timeit sin(3)
100000 loops, best of 3: 2.51 µs per loop
```

In this case we see that the scalar `math.sin` function performs much faster on scalar variables compared to `numpy.sin`.

Beyond Python code timing profiling Python also has memory profiling, and several libraries exist: `meliae`, `PySizer`, Heapy from `guppy`, `pozer` and `memory_profiler` (with `psutil`), ... Such tools may come in handy if Python automated memory handling does not work as you intended. The garbage collection module `gc` can provide a more low-level approach to understanding memory allocation in Python, and the tool *valgrind* can—with some effort—also identify memory problems in Python programs.

2.10 Coding style

Python has a standard coding style embedded in the PEP 8 specification. The program `pep8` will enable you to check whether your code conforms to PEP 8. `flake8` wraps the `pep8` program together with the the `pyflakes` code analysis program and the `mccabe` tool to measure code complexity based on Thomas McCabe graph-theoretic complexity measure [23]. Another style checking tool is `pylint`, which is also integrated in the Spyder development environment, giving you an instant report when you press the F8 keyboard shortcut.

Besides wrapping `pep8` and `pyflakes`, `flake8` has several convenient plugins to catch common issues, e.g., `flake8-docstrings` for checking docstrings via `pydocstyle`. Not all of plugins are equally relevant, e.g., the two plugins, `flake8-double-quotes` and `flake8-quotes`, check for the consistent use of either single or double quotes, making the two together impossible to satisfy unless the code has no strings!

When you execute the `flake8` program you will likely discover it reports a few pedantic issues, e.g., 'trailing whitespace' (W291) and 'blank line contains whitespace' (W293), but usually the messages reported by are worth to consider, and if you start using the tools right away the will make you learn the coding style of Python. Good styled code should report no output for the `flake8` and `pydocstyle` tools, while the `pylint` may complain too much to care about, e.g., it may report errors not recognizing the shape attribute in `Pandas` data frame and note too short variable names. In numerical code it is quite common to give general matrix variables short names, such as 'X' and 'A', which `pylint` unfairly complains about. On the other hand it may be a good idea to run `pylint` and check its output, e.g., 'pointless statement' is caught by `pylint`, while one may find no such alert in `flake8`. The style checking tools have optional command line arguments to suppress types of messages, e.g., to suppress `flake8`'s "N803 argument name should be lowercase" and "N806 variable in function should be lowercase" messages from its naming plugin, `flake8` can be called with the `--ignore` argument:

```
$ flake8 --ignore=N803,N806 mypackage
```

Alternatively, the messages to ignore (and include) can be setup in a configuration file.

Do not use too long lines, with long `import` statements and URLs in comments as the only exceptions [22]. This should make the code easier to read. `pep8` checks for the width of the line and reports it with the error code E501. If you have long lines then break them up and use parentheses around the line elements. This method will work on both long expressions and long strings, e.g., convert `'long line ...'` to `('long ' 'line ...')`. Too long lines may also indicate a very nested program. In this case, you might want to consider breaking the part of the program up into subfunctions and methods.

The software engineer principle 'Don't repeat yourself' (DRY) says the code should not have redundancy. A few tools exists for spotting duplicate code, e.g,. `clonedigger`.

Some editors has the ability to setup the style checking tools to 'check-as-you-type'.

Coding style checking can be setup to run as part of the ordinary testing, e.g., it can be included as a `tox` test environment in the `tox.ini` file so that multi-version Python testing and style checking are run for an entire module when the `tox` program is run.

2.10.1 Where is `private` and `public`?

Python has no keyword for `private` and `public` and has no direct senses of private names (variables, functions, classes) in the language. Making a name private in Python requires a lot of work, and instead Python encourages a more open and flexible approach to programming interfaces: A 'private' name is indicated with a leading underscore. Such a name is still public, but programmers are expected to be 'gentlemen' that will not misuse the name by accessing and setting it directly.

Double leading underscores indicate 'really' private variables where the variable name gets 'name mangled' [24]. Programmers that use this construct usually comes from a Java background, having learned a hard private/public distinction, and do not understand the more open interface style in Python. It should often be avoided unless there are naming collisions.

On the module level Python has a facility to hide module variables from a globbing `import` (i.e., `from module import *`) via the `__all__` list, where you specify the names of the classes, functions, variables and constants you want for public export. Defining this variable as a tuple rather than as a list will help the `pydocstyle` docstring style checking program.

To control access to an attribute of an object you can use getter and setter methods, but it is regarded as more Pythonic to use properties with the `@property` decorator and its associated `.setter` decorator. In the example below, we want to ensure that the property `weight` is not set to a negative value. This is done via the "private" property `_weight` and two decorated functions:

```python
class WeightedText(object):

    def __init__(self, text, weight=1.0):
        self.text = text
        self._weight = weight

    @property
    def weight(self):
        return self._weight

    @weight.setter
    def weight(self, value):
        if value < 0:
            # Ensure weight is non-negative
            value = 0.0
        self._weight = float(value)


text = WeightedText('Hello')
text.weight            # = 1.0
text.weight = -10      # calls function with @weight.setter
text.weight            # = 0.0
```

With this framework, the `weight` property is no longer accessed as a method but rather as an attribute with no calling parentheses.

2.11 Command-line interface scripting

2.11.1 Distinguishing between module and script

A .py file may act as both a module and a script. To distinguish between code which python should execute when the file is to be regarded as a script rather than a module one usually use the `__name__ == '__main__'` 'trick', — an expression that will return true when the file is executed as a script and false when executed during import. Code in the main namespace (i.e., code at the top level) will get executed when a module gets imported, and to avoid having the (main part of) script running at the time of import the above conditional will help. As little Python code as possible should usually appear in the main namespace of the finished script, so usually one calls a defined '`main`' function right after the conditional:

```
def main():
    # Actual script code goes here

if __name__ == '__main__':
    main()
```

This pattern, encouraged by the Google Style Guide [22], allows one to use the script part of the file as a module by importing it with `import script` and call the function with `script.main()`. It also allows you to test most of the script gaining almost full test coverage using the usual Python unit testing frameworks. Otherwise, you could resort to the `scripttest` package to test your command-line script.

If a module contains a `__main__.py` file in the root directory then this file will be executed when the module is executed. Consider the following directory structure and files:

```
/mymodule
    __main__.py
    __init__.py
    mysubmodule.py
```

With `python -m mymodule` the `__main__.py` is executed.

2.11.2 Argument parsing

For specification and parsing of input arguments to the script (command-line options) use the `docopt` package, unless you are too worried about portability issues and want to stay with PSL modules. PSL has several input argument parsing modules (`argparse`, `getopt` and the deprecated `optparse`), but `docopt` provides means to specify the format for input arguments within the module docstring, helping you to document your script and enabling a format specification in a more human-readable format than the programmatical style provided by `argparse`. Contrary to `argparse`, `docopt` does not perform input validation, e.g., ensuring that the script can interprete an input argument specified to be an integer as an actual integer. To gain this functionality with docopt use the associated `schema` module. It allows you to validate complex input argument restrictions and convert strings to appropriate types, e.g., numeric values or file identifiers.

2.11.3 Exit status

In some systems it is a convention to let the script return an exit status depending on whether the program completed succesfully or whether and an error occurred. Python on Linux returns 0 if no exception occured and 1 if an exception was raised and not handled. Explicit setting of the exit status can be controlled with an argument to the the `sys.exit` function. The small script below, `print_one_arg.py`, shows the explicit setting of the exit status value to 2 with `sys.exit` if the number of input arguments to the script is not correct:

```
import sys

def main(args):
    if len(args) == 2:     # The first value in args is the program name
        print(args[1])
    else:
        sys.exit(2)

if __name__ == '__main__':
    main(sys.argv)
```

Here below is a Linux Bash shell session using the Python script, first with the correct number of input arguments and then with the wrong number of input arguments ('`$?`' is a Bash variable containing the exit code of the previous command):

```
$ python print_one_arg.py hello
hello
$ echo $?
0
$ python print_one_arg.py
$ echo $?
2
```

An alternative exit could raise the `SystemExit` exception with `raise SystemExit(2)`. Both `SystemExit` and `sys.exit` may take a string as input argument which are output from the script as a error message to the stderr. The exit status is then 1, unless the `code` attribute is set to another value in the raised exception object.

2.12 Debugging

If an error is not immediately obvious and you consider beginning a debugging session, you might instead want to run your script through one of the Python checker programs that can spot some common errors. `pylint`, `pyflakes` and `pychecker` will all do that. `pychecker` executes your program, while `pyflakes` does not do that, and thus considered 'safer', but checks for fewer issues. `pylint` and `flake8`, the latter wrapping `pyflakes` and `pep8`, will also perform code style checking, along with the code analysis. These tools can be called from outside Python on the command-line, e.g., '`pylint your_code.py`'. Yet another tool in this domain is PySonar, which can do type analysis. It may be run with something like '`python pysonarsq/scripts/run_analyzer.py your_code.py pysonar-output`'.

The Python checker programs do not necessarily catch all errors. Consider Bob Ippolito `nonsense.py` example:

```
from __future__ import print_function


def main():
    print(1 + "1")

if __name__ == '__main__':
    main()
```

The program generates a `TypeError` as the + operator sees both an integer and a string which it cannot handle. Neither `pep8` nor `pyflakes` may report any error, and `pylint` complains about the missing docstring, but not the type error.

The simplest run-time debugging puts in one or several `print` functions/command at the critical points in the code to examine the value of variables. `print` will usually not display a nested variable (e.g., a list of dicts of lists) in particular readable way, and here a function in the `pprint` module will come in handy: The `pprint.pprint` function will 'pretty print' nested variables with indentation.

The 'real' run-time debugging tool for Python programs is the command-line-based `pdb` and its graphical counterpart Winpdb. The name of the latter could trick you into believing that this was a Windows-based program only, but it is platform independent and once installed available in, e.g., Linux, as the `winpdb` command. In the simplest case of debugging with `pdb` you set a breakpoint in your code with

```
import pdb; pdb.set_trace()
```

and continue from there. You get a standard debugging prompt where you can examine and change variables, single step through the lines of the code or just continue with execution. Integrated development environment, such as Spyder, has convenient built-in debugging functionality with `pdb` and `winpdb` and keyboard shortcuts such as F12 for insertion of a break point and Ctrl+F10 for single step. Debugging with `pdb` can be combined with testing: `py.test` has an option that brings up the debugger command-line interface in case of an assertion error. The invocation of the testing could look like this: `py.test --pdb`.

A somewhat special tool is `pyringe` a "Python debugger capable of attaching to processes". More python debugging tools are displayed at https://wiki.python.org/moin/PythonDebuggingTools.

2.12.1 Logging

`print` (or `pprint`) should probably not occur in the finished code as means for logging. Instead you can use the PSL `logging` module. It provides methods for multiple modules logging with multiple logging levels, user-definable formats, and with a range of output options: standard out, file or network sockets. As one of the few examples of Python HOWTOs, you will find a Logging Cookbook with examples of, e.g., multiple module logging, configuration, logging across a network. A related module is `warnings` with the `warnings.warn` function. The Logging HOWTO suggests how you should distinguish the use of `logging` and `warnings` module: "`warnings.warn()`in library code if the issue is avoidable and the client application should be modified to eliminate the warning" and "`logging.warning()` if there is nothing the client application can do about the situation, but the event should still be noted."

The logging levels of `logging` are in increasing order of severity: debug, info, warn/warning, error, critical/fatal. These are associated with constants defined in the module, e.g., `logging.DEBUG`. The levels each has a function for logging with the same name. By default only error and critical/fatal log messages are outputted:

```
>>> import logging
>>> logging.info('Timeout␣on␣connection')
>>> logging.error('Timeout␣on␣connection')
ERROR:root:Timeout on connection
```

For customization a `logger.Logger` object can be acquired. It has a method for changing the log level:

```
>>> import logging
>>> logger = logging.getLogger(__name__)
>>> logger.setLevel(logging.INFO)
>>> logger.info('Timeout␣on␣connection')
INFO:__main__:Timeout on connection
```

The format of the outputted text message can be controlled with `logging.basicConfig`, such that, e.g., time information is added to the log. The stack trace from a raised exception can be written to the logger via the `exception` method and function.

2.13 Advices

1. Structure your code into module, classes and function.

2. Run your code through a style checker to ensure that it conforms to the standard, — and possible catch errors.

3. Do not make redundant code. `clonedigger` can check your code for redundancy.

4. Document your code according to standard. Check that it conforms to the standard with the `pydocstyle` tools. Follow the Numpy convention in reStructuredText format to document input arguments, returned output and other aspects of the functions and classes.

5. Test your code with one of the testing frameworks such as `py.test`. If there are code example in the documentation run these through doctests.

6. Measure test coverage with the `coverage` package. Ask yourself when you did not reach 100% coverage, — if you did not.

7. If you discover a bug, then write a test that tests for the specific bug before you change the code to fix the bug. Make sure that the test fails for the unpatched code, then fix the implementation and test that the implementation works.

Chapter 3

Python for data mining

3.1 Numpy

Numpy (`numpy`) dominates as the Python package for numerical computations in Python. Deprecated `numarray` and `numeric` packages has now only relevance for legacy code. Other data mining packages, such as SciPy, Matplotlib and Pandas, build upon Numpy. Numpy itself relies on numerical libraries. It can show which with the `numpy.__config__.show` function. It should show BLAS and LAPACK.

Function	Input	Description
`numpy.eye`	Number of rows	Identity matrix
`numpy.ones`	Shape	Array of elements set to one
`numpy.zeros`	Shape	Array of elements set to zero
`numpy.random.random`	Shape	Array of random values uniformly distributed between 0 and 1

Table 3.1: Function for generation of Numpy data structures.

While Numpy's primary container data type is the array, Numpy also contains the `numpy.matrix` class. The shape of the matrix is always two-dimensional (2D), meaning that any scalar indexing, such as `A[1, :]` will return a 2D structure. The matrix class defines the `*` operator as matrix multiplication, rather than elementwise multiplication as for `numpy.array`. Furthermore, the matrix class has complex conjugation (Hermitian) with the property `numpy.matrix.H` and the ordinary matrix inverse in the `numpy.matrix.I` property. The inversion will raise an exception in case the matrix is singular. For conversion back and forth between `numpy.array` and `numpy.matrix` the matrix class has the property `numpy.matrix.A` which converts to a 2D `numpy.array`.

3.2 Plotting

There is not quite a good all-embrassing plotting package in Python. There exists several libraries which each has its advantages and disadvantages: Matplotlib, Matplotlib toolkits with mplot3d, ggplot, seaborn, `mpltools`, Cairo, mayavi, PIL, Pillow, Pygame, `pyqtgraph`, `mpld3`, Plotly and `vincent`.

The primary plotting library associated with Numpy is `matplotlib`. Developers familiar with Matlab will find many of the functions quite similar to Matlab's plotting functions. Matplotlib has a confusing number of different backends. Often you do not need to worry about the backend.

Perhaps the best way to get an quick idea of the visual capabilities in Matplotlib is to go through a tour of Matplotlib galleries. The primary gallery is http://matplotlib.org/gallery.html, but there also alternatives, e.g., https://github.com/rasbt/matplotlib-gallery and J.R. Johansson's matplotlib - 2D and 3D plotting in Python Jupyter Notebook.

The young statistical data visualization module seaborn brings stylish and aesthetics plots to Python building on top of matplotlib. seaborn is pandas-aware, requires also scipy and furthermore recommends statsmodels. During import seaborn is usually aliased to sns or sb (e.g., import seaborn as sns). Among its capabilities you will find colorful annotated correlation plots (seaborn.corrplot) and regression plots (seaborn.regplot). A similar library is ggplot, which is heavily inspired by R's ggplot2 package. Other 'stylish' Matplotlib extensions, mpltools and prettyplotlib, adjust the style of plots, e.g., with respect to color, fonts or background.

3.2.1 3D plotting

3-dimensional (3D) plotting is unfortunately not implemented directly in the standard Matplotlib plotting functions, e.g., matplotlib.pyplot.plot will only take an x and y coordinate, — not a z coordinate, and generally Python's packages do not provide optimal functionality for 3D plotting.

Associated with Matplotlib is an extention call **mplot3d**, available as mpl_toolkits.mplot3d, that has a somewhat similar look and feel as Matplotlib and can be used as a 'remedy' for simple 3D visualization, such as a mesh plot. mplot3d has a Axes3D object with methods for plotting lines, 3D barplots and histograms, 3D contour plots, wireframes, scatter plots and triangle-based surfaces. mplot3d should not be use to render more complicated 3D models such as brain surfaces.

For more elaborate 3D scientific visualization the plotting library **Mayavi** might be better. Mayavi uses the Visualization Toolkit for its rendering and will accept Numpy arrays for input [25, 26]. The mayavi.mlab submodule, with a functional interface inspired from matplotlib, provides a number of 3D plotting functions, e.g., 3D contours. Mayavi visualization can be animated and the graphical user interface components can be setup to control the visualization. Mayavi has also a stand-alone program called mayavi2 for visualization. Mayavi relies on VTK (Visualization Toolkit). As of spring 2015 Python 3 has no support for VTK, thus Mayavi does not work with Python 3.

A newer 3D plotting option under development is OpenGL-based **Vispy**. It targets not only 3D visualization but also 2D plots. Developers behind Vispy has previously been involved in other Python visualization toolkits: pyqtgraph, also features 3D visualization and has methods for volume rendering, Visvis, Glumpy and Galry. The vispy.mpl_plot submodule is an experimental OpenGL-backend for matplotlib. Instead of import matplotlib.pyplot as plt one can write import vispy.mpl_plot as plt and get some of the same Matplotlib functionality from the functions in the plt alias. In the more low-level Vispy interface in vispy.gloo the developer should define the visual objects in a C-like language called GLSL.

3.2.2 Real-time plotting

Real-time plotting where a plot is constantly and smoothly updated may be implemented with Pygame and pyqtgraph. But also matplotlib may update smoothly. The listing below defines a generator for discrete unrestricted random walking filtered with an infinite impulse response filter (i.e., autoregression), stored in a (finite sized) ring buffer before being plotted with matplotlib in the animate_* methods of the Animator class. The collections.deque class is used as the ring buffer. At each visualization update a new data item is pulled from the random walk generator via the autoregression filter.

```python
import matplotlib.pyplot as plt
import random
from collections import deque


def random_walker():
    x = 0
    while True:
        yield x
        if random.random() > 0.5:
            x += 1
        else:
            x -= 1

def autoregressor(it):
    x = 0
    while True:
        x = 0.79 * x + 0.2 * it.next()
        yield x


class Animator():

    def __init__(self, window_width=100):
        self.random_walk = random_walker()
        self.autoregression = autoregressor(self.random_walk)
        self.data = deque(maxlen=window_width)
        self.fig = plt.figure()
        self.ax = self.fig.add_subplot(1, 1, 1)
        self.line, = self.ax.plot([], [], linewidth=5, alpha=0.5)

    def animate_step(self):
        self.data.append(self.autoregression.next())
        N = len(self.data)
        self.line.set_data(range(N), self.data)
        self.ax.set_xlim(0, N)
        abs_max = max(abs(min(self.data)), abs(max(self.data)))
        abs_max = max(abs_max, 1)
        self.ax.set_ylim(-abs_max, abs_max)

    def animate_infinitely(self):
        while True:
            self.animate_step()
            plt.pause(0.01)

animator = Animator(500)
animator.animate_infinitely()
```

This approach is not necessarily effective. Profiling of the script will show that most of the time is spend with Matplotlib drawing. The `matplotlib.animation` submodule has specialized classes for real-time plotting and animations. Its class `matplotlib.animation.FuncAnimation` takes a figure handle and an animation step function. An initialization plotting function might also be necessary to define and submit `FuncAnimation`. The flow of the plotting may be controlled by various parameters: number of frames, interval between plots, repetitions and repetition delays.

```
import matplotlib.pyplot as plt
from matplotlib import animation
import random
from collections import deque


def random_walker():
    x = 0
    while True:
        yield x
        if random.random() > 0.5:
            x += 1
        else:
            x -= 1
```

```
def autoregressor(it):
    x = 0
    while True:
        x = 0.70 * x + 0.2 * it.next()
        yield x


class Animator():

    def __init__(self, window_width=100):
        self.random_walk = random_walker()
        self.autoregression = autoregressor(self.random_walk)
        self.data = deque(maxlen=window_width)
        self.fig = plt.figure()
        self.ax = self.fig.add_subplot(1, 1, 1)
        self.line, = self.ax.plot([], [], linewidth=5, alpha=0.5)
        self.ax.set_xlim(0, window_width)
        self.ax.set_ylim(-80, 80)

    def init(self):
        return self.line,

    def step(self, n):
        self.data.append(self.autoregression.next())
        self.line.set_data(range(len(self.data)), self.data)
        return self.line,


animator = Animator(500)
anim = animation.FuncAnimation(animator.fig, animator.step,
                               init_func=animator.init,
                               interval=0, blit=True)
plt.show()
```

Here blitting is used with axes frozen in the constructor. The interval is set to zero giving as fast an update frequency as possible. Note that the initialization and step plotting method (`init` and `step`) must return an iterable, — not just a single graphics element handle: That is what the commas after `self.line` are for making the methods return a tuple. These dynamical plotting methods tie the visualization update to the data pull, thus the data frequency is the same as the visualization frequency (frame rate).

Instead of continuous plotting the dynamic visualization may be saved to an MPEG v4 file with the `save` method of the `FuncAnimation` object.

3.2.3 Plotting for the Web

Standard Python has no straightforward way for plotting to the web. Probably the most basic way would construct plots with `matplotlib`, save the plots as PNG or SVG image files and then serve the images as static files from the web server. `pygal` can construct of SVG files with plots from data. Other approaches gain help from Javascript, e.g., via `mpld3` and `vincent`. These packages are usually independent of the web framework (Django, Flask, ...) and may work with any of them. The examples listed below show only a few of the possible combinations between web plotting library and web frameworks

Embedding in HTML img tag

One way of dynamical web plotting is embedding a string encoded image in the 'src' attribute of the 'img' HTML tag. The condensed listing below uses the Flask web framework, where a 'saved' plot from Matplotlib is captured via the file-like `StringIO` string object. The binary data in the `sio` object can be read with the `getvalue` method and encoded to a ASCII string with the `encode` method in the 'base64' format.

Listing 3.1: Web plotting with Flask and an embedded imaged in img tag.

```
from flask import Flask
import matplotlib.pyplot as plt
from StringIO import StringIO

app = Flask(__name__)

def plot_example():
    plt.plot([1, 2, 5, 2, 7, 2, 1, 3])
    sio = StringIO()
    plt.savefig(sio)
    return sio.getvalue().encode('base64').strip()

@app.route('/')
def index():
    return """<html><body>
           <img src="data:image/png;base64,{0}">
           </body></html>""".format(plot_example())

if __name__ == '__main__':
    app.run()
```

When the web browser is pointed to the default Flask URL http://127.0.0.1:5000/ a plot should appear in the browser. The flask decorator, `@app.route('/')`, around the `index` function will tell Flask to call that function when the web client makes a request for the http://127.0.0.1:5000/ address.

Note that in Python 3 the `StringIO` object is moved to the `io` module, so the import should go like `from io import StringIO`. A Python 2/3 compatible version would put a `try` and `except` block around the `StringIO` imports.

Whether it is a good idea to put image data in the HTML file may depend on the application. In the present case the simple HTML file results in a over 40 KB large file that the server has to send to the requesting client at each non-cached page request. A similar binary-coded PNG file is somewhat smaller and the server can transmit it independently of the HTML. The Jupyter Notebook uses the img-tag encoding for its generated plots (which can be embedded on a web-page), so in the saved IPython Notebook session files you will find large blocks of string-encoded image data, and all data—code, annotation and plots—fit neatly into one file with no need to keep track of separate image data files when you move the session file around!

Vega and vincent

The condensed example in listing 3.2 uses the CherryPy web framework together with the `vincent` plotting library. `vincent` may output its plots in the Vega JSON format, — rather than image files. The Vega Javascript library (`vega.js`) can read a Vega JSON file and render it in the webbrowser. The `VegaExample.index` method in the listing serves the HTML scaffold taken and modified from the Trifacta's Vega Javascript library homepage. The served HTML file imports the necessary Javascript libraries. The `VegaExample.plot` method builds a barplot with five data points and outputs it as Vega JSON. When the script executes the CherryPy web server starts at the web address 127.0.0.1:8080, and when you point your web browser to that address you should see a barplot. Vincent is not restricted to CherryPy. Instead of CherryPy we could have used one of the many other web frameworks to serve the Vega JSON.

Listing 3.2: Webplotting with Vincent and Cherrypy.

```
import cherrypy
import vincent
```

```
# Vega Scaffold modified from https://github.com/trifacta/vega/wiki/Runtime
HTML = """
<html>
  <head>
    <script src="http://trifacta.github.io/vega/lib/d3.v3.min.js"></script>
    <script src="http://trifacta.github.io/vega/lib/d3.geo.projection.min.js"></script>
    <script src="http://trifacta.github.io/vega/vega.js"></script>
  </head>
  <body><div id="vis"></div></body>
<script type="text/javascript">
// parse a spec and create a visualization view
function parse(spec) {
  vg.parse.spec(spec, function(chart) { chart({el:"#vis"}).update(); });
}
parse("/plot");
</script>
</html>
"""

class VegaExample:
    @cherrypy.expose
    def index(self):
        return HTML

    @cherrypy.expose
    def plot(self):
        bar = vincent.Bar([2, 4, 2, 6, 3])
        return bar.to_json()

cherrypy.quickstart(VegaExample())
```

Plotly

Another web plotting approach use the freemium cloud service *Plotly* available from http://plot.ly. Users with an account created on the Plotly website may write Python plot commands on the Plotly website and render and share them online, but it is also possible to construct online plots from a local installation of Plotly. With the `plotly` Python package installed locally and the API key for Plotly available,[1] plotting a sinusoid online requires only few lines:

```
import plotly
import numpy as np

plotly.tools.set_credentials_file(username='fnielsen',
                                   api_key='The␣API␣key␣goes␣here',
                                   stream_ids=['a␣stream␣id',
                                               'another␣stream␣id'])

x = np.linspace(0, 10)
y = np.sin(x)
graph_url = plotly.plotly.plot([x, y])
```

By default the `plotly.plotly.plot` spawns a webbrowser with the `graph_url` which may be something like `https://plot.ly/~fnielsen/7`. The displayed webpage shows an interactive plot (in this case of the sinusoid) where the web user may zoom, pan and scroll. By default `plotly.plotly.plot` creates a world readable plot, i.e. public data files and public plot. The web developer using the plot in

[1]The API key may be found on https://plot.ly/python/getting-started/ or under the profile.

his/her web application can add the online plot as a frame on the webpage via an HTML iframe tag: `<iframe src="https://plot.ly/~fnielsen/7"/>`. Plotly has a range of chart types such as boxplots, bar charts, polar area chart, bubble chart, etc. with good control over the style of the plot elements. It also has the capability to continuously update the plot with the so-called streaming plots as well as some simple statistics functionality such as polynomial fitting. There are various ways to set up the plot. The above code called the plot command with a data set. It is also possible to use the standard Matplotlib to construct the plot and 'replot' the Matplotlib figure with the plotly function `plotly.plotly.plot_mpl` with the Matplotlib figure handle as the first input argument.

Plotly appears fairly easy to deal with. The downside is the reliance on the cloud service as a freemium service. The basic gratis plan provides unlimited number of public files (plots) and 20 private files.

D3

D3 is a JavaScript library for plotting on the web. There is a very large set of diverse visualization types possible with this library. There are also extensions to D3 for further refinement, e.g., NVD3. Python webservices can use D3 in two ways: Either by outputting data in a format that D3 can read and serve a HTML page with D3 JavaScript included, or by using a Python package that will take care of the translation from Python plot commands to D3 JavaScript and possible HTML. `mpld3` is an example of the latter, and the code below shows a small compact example on how it can be used in connection with CherryPy.

```
import matplotlib.pyplot as plt, mpld3, cherrypy

class Mpld3Example():
    @cherrypy.expose
    def index(self):
        fig = plt.figure()
        plt.plot([1, 2, 3, 2, 3, 1, 3])
        return mpld3.fig_to_html(fig)

cherrypy.quickstart(Mpld3Example())
```

The resulting webpage appears on CherryPy's default URL `http://127.0.0.1:8080/` and displays the line plot with axes. The `mpld3.fig_to_html` function will only output a fragment of an HTML document with 'style', 'div' and 'script' tag. The developer will need to add at least 'html' and 'body' tags to make it a full valid HTML document. `mpld3` includes JavaScript files from `https://mpld3.github.io/js/`.

Other visualizations

Various other visualization libraries exist, e.g., `bokeh`, `glue` and the OpenGL-based `vispy`.

```
from bokeh.plotting import *
import numpy as np

output_file('rand.html')
line(np.random.rand(10), np.random.rand(10))
show()

import webbrowser
webbrowser.open(os.path.join(os.getcwd(), 'rand.html'))
```

3.3 Pandas

`pandas`, a relatively new Python package for data analysis, features an R-like data frame structure, annotated data series, hierarchical indices, methods for easy handling of times and dates, pivoting as well as a range of other useful functions and methods for handling data. Together with the `statsmodels` packages it makes data loading, handling and ordinary linear statistical data analysis almost as simple as it can be written in R. The primary developer Wes McKinney has written the book *Python for Data Analysis* [27] explaining the package in detail. Here we will cover the most important elements of `pandas`. Often when the library is imported it is aliases to `pd` like "`import pandas as pd`".

3.3.1 Pandas data types

The primary data types (classes) of Pandas are the vector-like `pandas.Series`, the matrix-like `pandas.DataFrame` as well as the 3-dimensional and 4-dimensional tensor-like `pandas.Panel` and `pandas.Panel4D`. These data types can be thought of as annotated vectors, matrix and tensors with row and column header, e.g., if a `pandas.DataFrame` is used to store a data matrix with multiple objects across rows and where each column can contain a feature. Columns can then be indexed by features names and the rows indexed by object identifiers. The elements of the data structures can be heterogenous and are not restricted to be numerical. The data 'behind' the Pandas data type are available as `numpy.arrays` in the `values` attribute of the Pandas data types:

```
>>> C = pd.Series([1, 2, 3])
>>> C.values
array([1, 2, 3])
```

Note that the basic Numpy has a so-called 'structured array' also known as 'record array', which like Pandas' data frame can contain heterogeneous data, e.g., integers in one column and strings in another. The Pandas interface seems more succinct and convenient for the user, so in most situation a data miner will prefer Pandas' data frame to the record array. A Pandas data frame converts easily to a record array with `pandas.DataFrame.to_records` method and the data frame constructor will handle a record array as input, so translation back and forth are relatively easy.

3.3.2 Pandas indexing

Pandas has multiple ways of indexing its columns, rows and elements, but the bad news is that you cannot use the standard Numpy square-backet indexing directly. Parts of the pandas structure can be indexed both by numerical indexing ('integer position') as well as with the row index and column names ('label-based'), — or a mixture of the two! Indeed confusion arises if the label index is numerical. The indexing is supported by the `loc`, `iloc` and `ix` indexing objects that `pandas.Series`, `pandas.DataFrame` and `pandas.Panel` all implement. Lets take a confusing example with numerical label-based indices in a data frame, where the (row) indices are numercal while the columns (column indices) are strings:

$$\mathbf{A} = \begin{array}{c|ccc} \text{Index} & \text{a} & \text{b} & \text{c} \\ \hline 2 & 4 & 5 & \text{yes} \\ 3 & 6.5 & 7 & \text{no} \\ 6 & 8 & 9 & \text{ok} \end{array} \tag{3.1}$$

This data frame can be readily be represented with a `pandas.DataFrame`:

```
import pandas as pd

A = pd.DataFrame([[4, 5, 'yes'], [6.5, 7, 'no'], [8, 9, 'ok']],
                 index=[2, 3, 6], columns=['a', 'b', 'c'])
```

The row and column indices are available in the attributes `A.index` and `A.columns`, respectively. In this case they are list-like Pandas `Int64Index` and `Index` types.

For indexing the rows `loc`, `iloc` and `ix` indexing objects can be used (do not use the deprecated methods `irow`, `icol` and `iget_value`). For indexing individual rows the specified index should be an integer:

```
>>> A.loc[2, :]          # label-based (row where index=2)
a      4
```

```
b       5
c     yes
Name: 2, dtype: object
>>> A.iloc[2, :]         # position integer (3th row)
a      8
b      9
c     ok
Name: 6, dtype: object
>>> A.ix[2, :]           # label-based
a       4
b       5
c     yes
Name: 2, dtype: object
```

In all these cases the indexing methods return a `pandas.Series`. It is not necessary to index the columns with colon: In a more concise notation we can write `A.loc[2]`, `A.iloc[2]` or `A.ix[2]` and get the same rows returned. Note that in the above example that the `ix` method uses the label-based method, because the index contains integers. If instead, the index contains non-integers, e.g., strings, the `ix` method would fall back on position integer indexing as seen in the example here below (this ambiguity seems prone to bugs, so take care):

```
>>> B = pd.DataFrame([[4, 5, 'yes'], [6.5, 7, 'no'], [8, 9, 'ok']],
                     index=['e', 'f', 'g'], columns=['a', 'b', 'c'])
>>> B.ix[2, :]
a      8
b      9
c     ok
Name: g, dtype: object
```

Trying `B.iloc['f']` to address the second row will result in a `TypeError`, while `B.loc['f']` and `B.ix['f']` are ok.

The columns of the data frame may also be indexed. Here for the second column ('b') of the `A` matrix:

```
>>> A.loc[:, 'b']         # label-based
2     5
3     7
6     9
Name: b, dtype: int64
>>> A.iloc[:, 1]          # position-based
2     5
3     7
6     9
Name: b, dtype: int64
>>> A.ix[:, 1]            # fall back position-based
2     5
3     7
6     9
Name: b, dtype: int64
>>> A.ix[:, 'b']          # label-based
2     5
3     7
6     9
Name: b, dtype: int64
```

In all cases a `pandas.Series` is returned. The column may also be indexed directly as an item

```
>>> A['b']
2     5
3     7
```

```
6     9
Name: b, dtype: int64
```

If we want to get multiple rows or columns from `pandas.DataFrame`s the indices should be of the `slice` type or an iterable.

Combining position-based row indexing with label-based column indexing, e.g., getting the all rows from the second to the end row and the ‘c’ column of the **A** matrix, neither `A[1:, 'c']` nor `A.ix[1:, 'c']` work (the latter one returns all rows). Instead you need something like:

```
>>> A['c'][1:]
3     no
6     ok
Name: c, dtype: object
```

or somewhat confusingly, but more explicitly:

```
>>> A.loc[:, 'c'].iloc[1:, :]
3     no
6     ok
Name: c, dtype: object
```

3.3.3 Pandas joining, merging and concatenations

Pandas provides several functions and methods for rearranging data with database-like joining and concatenation operations. Consider the following two matrices with indices and column names which can be represented in a Pandas `DataFrame`:

$$\mathbf{A} = \begin{array}{c|cc} \text{Index} & \text{a} & \text{b} \\ \hline 1 & 4 & 5 \\ 2 & 6 & 7 \end{array} \qquad \mathbf{B} = \begin{array}{c|cc} \text{Index} & \text{a} & \text{c} \\ \hline 1 & 8 & 9 \\ 3 & 10 & 11 \end{array} \tag{3.2}$$

Note here that the two matrices/data frames has one overlapping row index (1) and two non-overlapping row indices (2 an 3) as well as one overlapping column name (a) and two non-overlapping column names (b and c). Also note that the one equivalent element in the two matrices (1, a) is inconsistent: 4 for **A** and 8 for **B**.

If we want to combine these two matrices into one there are multiple ways to do this. We can append the rows of **B** after the rows of **A**. We can match the columns of both matrices such that the a-column of **A** matches the a-column of **B**.

```
>>> import pandas as pd

>>> A = pd.DataFrame([[4, 5], [6, 7]], index=[1, 2], columns=['a', 'b'])
>>> B = pd.DataFrame([[8, 9], [10, 11]], index=[1, 3], columns=['a', 'c'])

>>> pd.concat((A, B))  # implicit outer join
    a    b    c
1   4    5  NaN
2   6    7  NaN
1   8  NaN    9
3  10  NaN   11

>>> pd.concat((A, B), join='inner')
    a
1   4
2   6
1   8
3  10
```

```
>>> A.merge(B, how='inner', left_index=True, right_index=True)
   a_x  b  a_y  c
1    4  5    8  9

>>> A.merge(B, how='outer', left_index=True, right_index=True)
   a_x   b  a_y   c
1    4   5    8   9
2    6   7  NaN NaN
3  NaN NaN   10  11

>>> A.merge(B, how='left', left_index=True, right_index=True)
   a_x  b  a_y   c
1    4  5    8   9
2    6  7  NaN NaN
```

3.3.4 Simple statistics

When data represented in Pandas series, data frame or panels methods in its class may compute simple summary statistics such as mean, standard deviations and quantiles. These are available as, e.g., the methods `pandas.Series.mean`, `pandas.Series.std`, `pandas.Series.kurtosis` and `pandas.Series.quantile`. The `describe` method of the Pandas classes computes a summary of count, mean, standard deviation, minimum, maximum and quantiles, so with an example data frame such (say, `df = pandas.DataFrame([4, 2, 6])`) you will get a quick overview with data columnwise with `df.describe()`.

For the computation of the standard deviation with the `std` methods care should be taken with the issue of biased/unbiased estimation. Pandas and Numpy compute the standard deviation differently(!):

```
import pandas, numpy

>>> x = [1, 2, 3]
>>> mean = numpy.mean(x)
>>> s_biased = numpy.sqrt(sum((x - mean)**2) / len(x))
>>> s_biased
0.81649658092772603
>>> s_unbiased = numpy.sqrt(sum((x - mean)**2) / (len(x) - 1))
>>> s_unbiased
1.0
>>> numpy.std(x)                    # Biased
0.81649658092772603
>>> numpy.std(x, ddof=1)            # Unbiased
1.0
>>> numpy.array(x).std()            # Biased
0.81649658092772603
>>> pandas.Series(x).std()          # Unbiased
1.0
>>> pandas.Series(x).values.std()   # Biased
0.81649658092772603
>>> df = pandas.DataFrame(x)
>>> df['dummy'] = numpy.ones(3)
>>> df.groupby('dummy').agg(numpy.std)   # Unbiased !!!
       0
dummy
1      1
```

Numpy computes by default the biased version of the standard deviation, but if the optional argument `ddof` is set to 1 it will compute the unbiased version. Contrary, Pandas computes by default the unbiased standard

Subpackage	Function examples	Description
`cluster`	`vq.keans`, `vq.vq`, `hierarchy.dendrogram`	Clustering algorithms
`fftpack`	`fft`, `ifft`, `fftfreq`, `convolve`	Fast Fourier transform, etc.
`optimize`	`fmin`, `fmin_cg`, `brent`	Function optimization
`spatial`	`ConvexHull`, `Voronoi`, `distance.cityblock`	Functions working with spatial data
`stats`	`nanmean`, `chi2`, `kendalltau`	Statistical functions

Table 3.2: Some of the subpackages of SciPy.

deviation. Perhaps the most surprising of the above examples is the case with aggregation method (`agg`) of the `DataFrameGroupBy` which will compute the unbiased estimate even when called with the `numpy.std` function! `pandas.Series.values` is a `numpy.array` and thus the `std` method will by default use the biased version.

3.4 SciPy

SciPy (Scientific Python) contains a number of numerical algorithms that work seamlessly with Numpy. SciPy contains functions for linear algebra, optimizations, sparse matrices, signal processing algorithms, statistical functions and special mathematical functions, see Table 3.2 for an overview of some of subpackages and their functions. Many of the SciPy function are made directly available by `pylab` with '`from pylab import *`'.

A number of the functions in `scipy` also exist in `numpy` for backwards compatibility. For instance, the packages makes eigenvalue decomposition available as both `numpy.linalg.linalg.eig` and `scipy.linalg.eig` and the Fourier transform as `numpy.fft.fft` and `scipy.fftpack.fft`. Usually the `scipy` version are preferable. They may be more flexible, e.g., be able to make inplace modification. In some instances one can also experience that `scipy` versions are faster. Note that the `pylab.eig` is the Numpy version.

3.4.1 `scipy.linalg`

The linear algebra part of SciPy (`scipy.linalg`) contains, e.g., singular value decomposition (`scipy.linalg.svd`), eigenvalue decomposition (`scipy.linalg.eig`). These common linear algebra methods are also implemented in Numpy and available with the `numpy.linalg` module. `scipy.linalg` has quite a large number of specialized linear algebra method, e.g., LU decomposision with `scipy.linalg.lu_factor`, which are not available in `numpy.linalg`.

Be careful with `scipy.linalg.eigh`. It will not check whether your matrix is symmetric and it looks only at the lower triangular matrix:

```
>>> eigh([[1, 0.5], [0.5, 1]])
(array([ 0.5,  1.5]), array([[-0.70710678,  0.70710678],
       [ 0.70710678,  0.70710678]]))
>>> eigh([[1, 1000], [0.5, 1]])
(array([ 0.5,  1.5]), array([[-0.70710678,  0.70710678],
       [ 0.70710678,  0.70710678]]))
```

3.4.2 Fourier transform with `scipy.fftpack`

Fourier transform is available in the `scipy.fftpack` subpackage. It handles real and complex multidimensional input and may make forward and inverse one-dimesional or higher-dimensional fast Fourier transform (FFT). The functions in the subpackage relies—as the name implies—on the FFTPACK Fortran library.[2]

[2] http://www.netlib.org/fftpack/

Numpy's and SciPy's Fourier transform may be considerably slow if the input is of a size corresponding to a prime number. In these cases the chirp-z transform algorithm can considerable cut processing time.

Outside `scipy` there exist wrappers for other efficient Fourier transform libraries beyond FFTPACK. The `pyFFTW` wraps the FFTW (Fastest Fourier Transform in the West) library.[3] A `scipy.fftpack` lookalike API in the `pyFFTW` package is available in the subpackage `pyfftw.interfaces.scipy_fftpack`. If the purpose with the Fourier transform is just to produce a spectrum then the `welch` function over in the `scipy.signal` subpackage may be another option. It calls the `scipy.fftpack.fft` function repeatedly on windows of the signal to be transformed.

3.5 Statsmodels

The `statsmodels` package provides some common statistical analysis methods with linear modeling and statistical tests. Part of the package is modeled after the R programming language where you write statistical formulas with the tilde notation enabling the specification of statistical tests in a quite compact format. With `pandas` and `statsmodels` imported, reading a data set from comma-separated files with multiple variables represented in columns, specifying the relevant test with dependent and independent variables, testing and reporting can be neatly done with just one single line of Python code.

Based on an example from the original statsmodels paper [4] we use a data set from James W. Longley that is included as part of the `statsmodels` package as one among over 25 reference data sets and accessible via the `datasets` submodule. The data is represented in the small comma-separated values file `longley.csv` placed in a `statsmodels` subdirectory. After adding an intercept column to the exogeneous variables (i.e., the independent variables) we instance an object from the ordinary least squares (OLS) class of `statsmodels` with the endogeneous variable (i.e., dependent variable) as the first argument to the constructor. Using the `fit` method of the object returns a result object which contain, e.g., parameter estimates. Its `summary` method produces a verbose text report with the fitted parameters, P-values, confidence intervals and diagnostics.

```
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> longley_model = sm.OLS(data.endog, sm.add_constant(data.exog))
>>> longley_results = longley_model.fit()
>>> print(longley_results.summary())
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.995
Model:                            OLS   Adj. R-squared:                  0.992
Method:                 Least Squares   F-statistic:                     330.3
Date:                Fri, 13 Feb 2015   Prob (F-statistic):           4.98e-10
Time:                        13:56:24   Log-Likelihood:                -109.62
No. Observations:                  16   AIC:                             233.2
Df Residuals:                       9   BIC:                             238.6
Df Model:                           6
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [95.0% Conf. Int.]
------------------------------------------------------------------------------
const      -3.482e+06    8.9e+05     -3.911      0.004       -5.5e+06 -1.47e+06
x1            15.0619     84.915      0.177      0.863       -177.029   207.153
x2            -0.0358      0.033     -1.070      0.313         -0.112     0.040
x3            -2.0202      0.488     -4.136      0.003         -3.125    -0.915
x4            -1.0332      0.214     -4.822      0.001         -1.518    -0.549
x5            -0.0511      0.226     -0.226      0.826         -0.563     0.460
x6          1829.1515    455.478      4.016      0.003        798.788  2859.515
==============================================================================
Omnibus:                        0.749   Durbin-Watson:                   2.559
Prob(Omnibus):                  0.688   Jarque-Bera (JB):                0.684
Skew:                           0.420   Prob(JB):                        0.710
```

[3] http://www.fftw.org/

```
Kurtosis:                       2.434   Cond. No.                     4.86e+09
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 4.86e+09. This might indicate that there are
strong multicollinearity or other numerical problems.
```

The original CSV file contained a column heading with the variable names: Obs, TOTEMP, GNPDEFL, GNP, UNEMP, ARMED, POP, YEAR. These name were lost when we used the `sm.datasets.longley.load()` method, as it only returned unannotated Numpy arrays. In the summary of the result the variable are referred to with the generic names x1, x2, ... With `statsmodels` `pandas` integration we can maintain the variable names:

```
>>> data = sm.datasets.longley.load_pandas()
>>> longley_model = sm.OLS(data.endog, sm.add_constant(data.exog))
>>> longley_results = longley_model.fit()
>>> print(longley_results.summary())
                            OLS Regression Results
==============================================================================
Dep. Variable:                 TOTEMP   R-squared:                       0.995
Model:                            OLS   Adj. R-squared:                  0.992
Method:                 Least Squares   F-statistic:                     330.3
Date:                Fri, 13 Feb 2015   Prob (F-statistic):           4.98e-10
Time:                        14:25:14   Log-Likelihood:                -109.62
No. Observations:                  16   AIC:                             233.2
Df Residuals:                       9   BIC:                             238.6
Df Model:                           6
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [95.0% Conf. Int.]
------------------------------------------------------------------------------
const      -3.482e+06    8.9e+05     -3.911      0.004     -5.5e+06 -1.47e+06
GNPDEFL       15.0619     84.915      0.177      0.863     -177.029   207.153
GNP           -0.0358      0.033     -1.070      0.313       -0.112     0.040
UNEMP         -2.0202      0.488     -4.136      0.003       -3.125    -0.915
ARMED         -1.0332      0.214     -4.822      0.001       -1.518    -0.549
POP           -0.0511      0.226     -0.226      0.826       -0.563     0.460
YEAR        1829.1515    455.478      4.016      0.003      798.788  2859.515
==============================================================================
Omnibus:                        0.749   Durbin-Watson:                   2.559
Prob(Omnibus):                  0.688   Jarque-Bera (JB):                0.684
Skew:                           0.420   Prob(JB):                        0.710
Kurtosis:                       2.434   Cond. No.                     4.86e+09
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 4.86e+09. This might indicate that there are
strong multicollinearity or other numerical problems.
```

Here we use the `load_pandas` method which returns Pandas objects so that `data.endog` is a `pandas.Series`, and `data.exog` is a `pandas.DataFrame` and now the result summary displays the variable names associated with the parameter estimates.

Using the R-like formula part of `statsmodels` available with the `statsmodels.formula.api` module the latter can also be written with:

```
>>> import statsmodels.api as sm
>>> import statsmodels.formula.api as smf
>>> data = sm.datasets.longley.load_pandas().data
>>> formula = 'TOTEMP␣~␣GNPDEFL␣+␣GNP␣+␣UNEMP␣+␣ARMED␣+␣POP␣+␣YEAR'
>>> longley_model = smf.ols(formula, data)
>>> longley_results = longley_model.fit()
>>> print(longley_results.summary())
```

Following the R-convention the variable on the left side of the tilde (in the `formula` variable) is the variable to be predicted (the dependent variable or the endogeneous variable), while the variables on the right side of the tilde are the independent variables. An intercept column is implicitly added to the independent variables. If the intercept column should not be added then the formula should be append with minus one:

```
formula = 'TOTEMP␣~␣GNPDEFL␣+␣GNP␣+␣UNEMP␣+␣ARMED␣+␣POP␣+␣YEAR␣-␣1'
```

3.6 Sympy

`sympy` is the symbolic mathematics package of Python. Symbolic variables can be setup to define equations and, e.g., functions may be differentiated and simplified. The below code sets up the equation $f = \sin(2\pi x) \times \exp(-x^2)$, differentiate it twice and plot the function and its derivatives as well as evaluates the second order derivative at $x = 0.75$:

```
import sympy

x = sympy.symbols('x')

f = sympy.sin(2*sympy.pi*x) * sympy.exp(-x**2)
df = f.diff()
ddf = df.diff()

p = sympy.plot(f, df, ddf, xlim=(-3, 3),
               adaptive=False, nb_of_points=1000)
p[0].line_color = (1, 0, 0)
p[1].line_color = (0, 1, 0)
p[2].line_color = (0, 0, 1)
p.show()

ddf.evalf(subs={'x': 0.75})
```

The last function call gives 22.3516785923350. A corresponding numerical differentiation with the `scipy.misc.derivative` function—that apply the central difference formula—gives around the same value:

```
>>> from scipy.misc import derivative
>>> from math import sin, exp, pi

>>> f_numerical = lambda x: sin(2*pi*x) * exp(-x**2)
>>> derivative(f_numerical, 0.75, dx=0.001, n=2, order=7)
22.351678592706648
```

The instancing of the numerical function in the line with the lambda keyword may also be implemented via the above-defined Sympy symbolic variables and the `lambdify` function in Sympy:

```
>>> from sympy.utilities.lambdify import lambdify
>>> f_numerical = lambdify(x, f)
>>> derivative(f_numerical, 0.75, dx=0.001, n=2, order=7)
22.351678592706648
```

It may be worth to note that the user can initialize IPython with a 'sympy' profile with the command `ipython --profile=sympy`. It sets up a few variables as `sympy` symbols (x, y, z, t, k, m, n, f, g, h) and imports all exported names from the `sympy` module.

3.7 Machine learning

Rather than reinventing the wheel by programming machine learning algorithms from scratch the Python developer can take advantage of the multiple algorithms available in the machine learning packages already

Name	Module	KLoC	GS-cites	Reference
SciPy.linalg	`scipy.linalg`			
Statsmodels	`statsmodels`	92	141	[4]
Scikit-learn	`sklearn`	427	8.830	[28]
PyMVPA		136	100 + 45	[29, 30]
Orange	`orange`	286	56	[31]
Mlpy: machine learning Python		75	8	[32]
Modular toolkit for Data Processing (MDP)		31	58	[33]
PyBrain	`pybrain`	36	128	[34]
Pylearn2			61	
Bob	`bob`		31	
Gensim	`gensim`	9	699	[35]
Natural Language Toolkit (NLTK)	`nltk`	215	2.590	[36]
PyPR	`pypr`	?	—	—
Caffe				
...				

Table 3.3: Some of the Python machine learning packages. KLoC denotes 1,000 lines of code and 'GS-cites' the number of citations as reported by Google Scholar (Note: Numbers not necessarily up to date).

Name	Input	Description
`get_params`	—	Get parameter
`set_params`	Parameters	Set parameters
`decision_function`	X	
`fit`	X, y	Estimate model parameters
`fit_predict`	X	Performs clustering and return cluster labels
`fit_transform`	X, (y)	Fit and transform
`inverse_transform`	Y	Opposite operation of `transform`
`predict`	X	Estimate output
`predict_proba`	X	
`score`	X, y	Coefficient of determination
`transform`	X	Transform data, e.g., through dimensionality reduction

Table 3.4: Scikit-learn methods. Whether the methods are define for the class depends on its algorithmic types, e.g., classifiers should have the `predict` defined.

available, see Table 3.3 for a list. Here we will cover the *Scikit-learn* package, which probably is the machine learning package with the largest momentum as of 2014 both according to the lines of code and the number of scientific citations to the primary research paper for the package [28], — if we disregard NLTK as a machine learning package.

There are a number of other package: The functions in `scipy.linalg` can be used to estimate linear models. The relevant methods are, e.g., the pseudo-inverse of a matrix `scipy.linalg.pinv` and the least square solution to the equation $\mathbf{Ax} = \mathbf{b}$ with `scipy.linalg.lstsq` and, for square matrices, `scipy.linalg.solve`. For optimizing a machine learning cost function where the parameters enter in a non-linear fashion the functions `scipy.optimize` can be used. The primary general function in that module is `scipy.optimize.minimize`. NLTK is primarily a toolbox for processing text, see section 3.8, but it also contains some classifiers in the `nltk.classify` module. The newest version of the NLTK toolbox contains an interface to the Scikit-learn classifiers.

Type	Class name	Parameter examples
K-nearest neighbor	`sklearn.neighbors.KNeighborsClassifier`	
Linear discriminant analysis	`sklearn.lda.LDA`	
Support vector machine	`sklearn.svm.SVC`	kernel
Principal component analysis	`sklearn.decomposition.PCA`	Number of components
Non-negative matrix factorization	`sklearn.decomposition.NMF`	

Table 3.5: sklearn classifiers

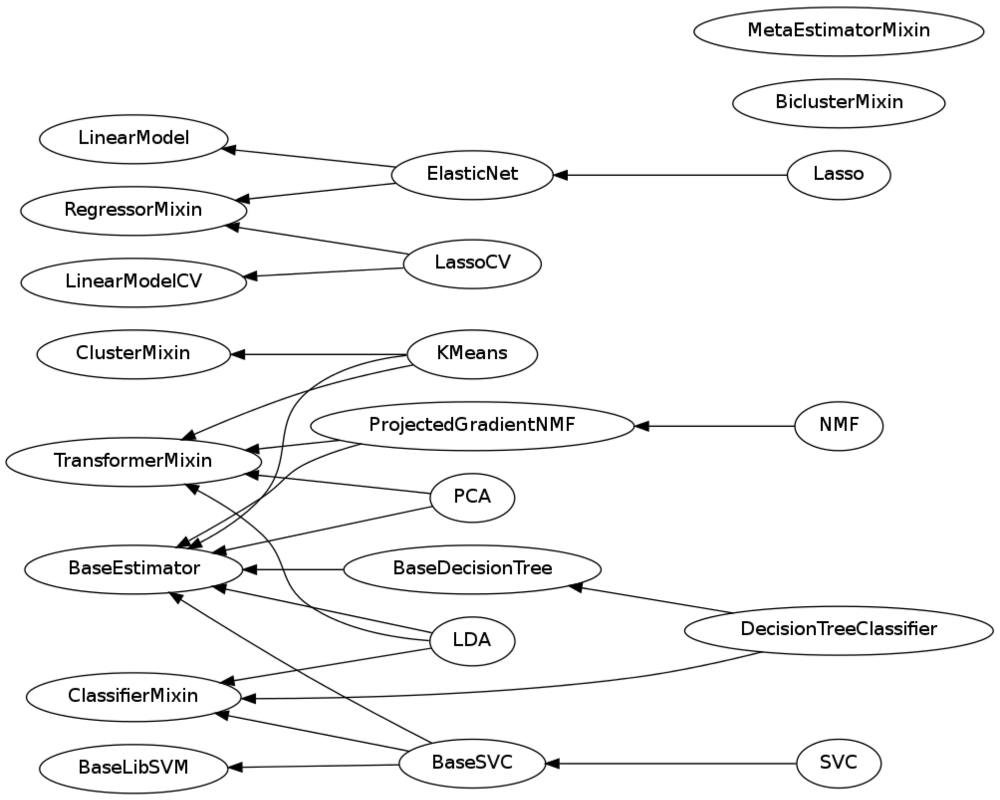

Figure 3.1: Sklearn classes derivation.

3.7.1 Scikit-learn

PyPI calls the package `scikit-learn` while the name of the main module is called `sklearn` and should be imported as such.

`sklearn` has a plentora of classifiers, decomposition and clustering algorithms and other machine learning algorithms all implemented as individual Python classes. The methods in the classes follow a naming pattern, see Table 3.4. The method for parameter estimation or training is called `fit`, while the method for prediction is called `predict`, — if prediction is relevant for the algorithm. The parameters of the model are available as class attributes with names that has a postfixed underscore, e.g., `coef_` or `alpha_`. The uniform interface to the classifiers means that it does not take that much extra effort to use several classifiers compared to a single classifier.

	Description	Example	Example matches
`.`	Any character	`.`	a, b, .
`*`	Zero or more of the preceding group	`.*`	a, ab, abab, ‘’ (empty string)
`^`	Beginning of string	`^b.*`	b, baaaa
`$`	End of string	`b.*b$`	bb, baaaab, bab
`[ ]`	Match any one in a set of characters	`[a-cz]`	a, b, c, z
`[^ ]`	Set of characters	`[^a]`	b, c, 1, 2
`( )`	Captured subexpression	`(a.*)`	a, abb
`{m, n}`	Match at least m and at most n of preceding group	`a{2,4}`	aa, aaa, aaaa
`|`	Or, alternation, either one or the other	`a|b`	a, b
`+`	One or more of the proceeding group	`a+`	a, aa, aaa, aaaaaaaaa
`?`	Zero or one	`a?`	‘’ (empty string), a
`\d`	Digit	`\d`	1, 5, 0
`\D`	Non-digit	`\D`	a, b,)
`\s`	Whitespace		
`\S`	Non-whitespace		
`\w`	Word character		
`\W`	Non-word character		
`\b`	Word boundary		

Table 3.6: Metacharacters and character classes of Python’s regular expressions in the `re` module. The metacharacters in the first group are the POSIX metacharacters, while the second group features are the extended POSIX metacharacters. The third group has the Perl-like character sets.

3.8 Text mining

The ordinary string object of Python (`str` or Python 2’s `unicode`) has a range of methods for simple text processing, e.g., `str.split` and `str.rsplit` split a string at a specified separator while `str.splitlines` splits at line break. `str.lower`, `str.upper`, `title` and `str.capitalize` change letter case, while `str.replace` can replace a substring within a string. Some methods, returning a Boolean, test for various conditions, e.g., `str.isdigit` and `str.isspace`.

Python has a range of modules and packages for more elaborate extracting and processing of texts: `re`, `lxml`, BeautifulSoup, html5lib, Scrapy, Portia, NLTK, Redshift, `textblob`, `pattern` [37], Orange-Text, spaCy, Gensim, etc. Furthermore, there are several wrapper for the Java-based *Stanford CoreNLP* tools.

If you need to read a special format you might be lucky in finding a specialized Python module to do the job. MediaWiki-based wikis, such as Wikipedia use their own idiosyncratic markup language. If you want to strip the markup and only retain the text, you could very well be tempted to build regular expression patterns that tries to match the nested constructs in the syntax, but you are probably better off using the `mwparserfromhell` package, which will easily do the job with `mwparserfromhell.parse(wikitext).strip_code()`.

3.8.1 Regular expressions

Python provides regular expression functionality through the PSL module `re`. Its regular expressions are modeled after the powerful regular expressions in Perl and simple regular expressions from Perl can be used in Python, but not necessarily the more complicated ones.

Python `re` (and Perl) implements the POSIX regular expressions metacharacters, except reference to subexpressions matches with ‘\\n’. For escaping the metacharacters, use the backslash. Python defines a set of character classes similar to Perl, e.g., `\d` means any digit corresponding to the character set `[0-9]` in POSIX notation. Python (and Perl) defines the complement set with upper case letters, e.g., `\D` means any non-digit character or `[^0-9]` in POSIX notation, see Table 3.6 for the list of character classes. Note that the so-called word character referenced by `\w` (that matches letter, digit or the underscore) may match letters beyond ASCII's a–z, such as ø, æ, å and ß. They can be used to match all international letters that the character class `[a-zA-Z]` will not do, by using the complement of the complement of word characters excluding digits and the underscore: `[^\W_0-9]`. This trick will be able to identify words with international letters, here in Python 2: `re.findall('[^\W_0-9]+', u'Årup Sø Straße 2800', flags=re.UNICODE)`, which will return a list with 'Årup', 'Sø' and 'Straße' while avoiding the number.

One application of regular expressions is tokenization: Finding meaningful entities (e.g., words) in a text. Word tokenization in informal texts are not necessarily easy. Regard the following difficult invented micropost "`@fnielsen Pråblemer!..Øreaftryk i Århus..:)`" where there seems to be two ellipses and smiley as well as international characters:

```
# Ordinary string split() does only split at whitespace
text.split()

# Username @fnielsen and smiley lost
re.findall('\w+', text, re.UNICODE)
re.findall('[^\W_\d]+', text, re.UNICODE)

# @fnielsen ok now, but smiley still not tokenized
re.findall('@\w+␣|␣[^\W_\d]+', text, re.UNICODE | re.VERBOSE)

# All tokens catched except ellipses
re.findall('@\w+␣|␣[^\W_\d]+␣|␣:\)', text, re.UNICODE | re.VERBOSE)

# Also ellipses
re.findall('@\w+␣|␣[^\W_\d]+␣|␣:\)␣|␣\.\.+', text, re.UNICODE | re.VERBOSE)
```

The last two regular expressions catch the smiley, but it will not catch, e.g, `:(`, `:-)` or the full `:))`. In the above code `re.VERBOSE` will ignore whitespaces in the definition of the regular expression making it more readable. `re.UNICODE` ensures that Python 2 will handle Unicode characters, e.g., `re.findall('\w+', text)` without `re.UNICODE` will not work as it splits the string at the Danish characters.

For more information about Python regular expressions see the Python's regular expression HOWTO[4] or chapter 7 in *Dive into Python*[5]. For some cases the manual page for Perl regular expressions (perlre) may also be of some help, but the docstring of the `re` module, available with `help(re)`, also has a good overview of the special characters in regular expressions patterns.

3.8.2 Extracting from webpages

To extract parts of a webpage basic regular expressions with the `re` can be used or an approach with BeautifulSoup. XPath functionality that is found in the `lxml` package may also be used. XPath is its own idiosyncratic language to specify elements in XML and HTML. Here is an example with a partial extraction of editor names from a World Wide Web Consortium (W3C) specification. We fetch the HTML text with the `requests` library that makes the byte-based content[6] available in the `content` attribute of the response object return via the `request.get` function:

```
import requests
from lxml import etree

url = 'http://www.w3.org/TR/2009/REC-skos-reference-20090818/'
response = requests.get(url)
tree = etree.HTML(response.content)
```

[4]https://docs.python.org/2/howto/regex.html
[5]http://www.diveintopython.net/regular_expressions/
[6]That is not in Unicode. In Python 2 it has the type 'str', while in Python 3 it has the type 'bytes'.

```
# The title in first-level header:
title = tree.xpath("//h1")[0].text

# Find the elements where editors are defined
editor_tree = tree.xpath("//dl/dt[contains(., 'Editors')]")[0]

# Get the names from the text between the HTML tags
names = [name.strip() for name in editor_tree.getnext().itertext()]
```

W3C seems to have no consistent formatting of the editor names for its many specifications, so you will need to do further processing of the `names` list of names to extract real names. In this case the editors end up in a list split between given name and surname and contain affiliation as well: "`['Alistair', '', 'Miles', ', STFC\n Rutherford Appleton Laboratory / University of Oxford', 'Sean', '', 'Bechhofer', ',\n University of Manchester']`"

Note that Firefox has an 'Inspector' in the Web Developer tool (F12 keyboard shortcut) which helps navigating the tag hierarchy and identify suitable tags for the XPath specification.

Below is another implementation of the W3C technical report editor extraction with a low-level rather 'dump' use of the `re` module

```
import re
import requests

url = 'http://www.w3.org/TR/2009/REC-skos-reference-20090818/'
response = requests.get(url)
editors = re.findall('Editors:(.*?)</dl>', response.text,
                     flags=re.UNICODE | re.DOTALL)[0]
editor_list = re.findall('<a .*?>(.+?)</a>', editors)

# Strip remaining HTML 'span' tags
names = [re.sub('</?span>', '', text, flags=re.UNICODE) for text in editor_list]
```

Here the `names` variables contains a list with each element as a name: "`[u'Alistair Miles', u'Sean Bechhofer']`, but this version unfortunately does not necessarily work with other W3C pages, e.g., it fails with http://www.w3.org/TR/2013/REC-css-style-attr-20131107/.

We may also use BeautifulSoup and its `find_all` and `find_next` methods:

```
from bs4 import BeautifulSoup
import re
import requests

url = 'http://www.w3.org/TR/2009/REC-skos-reference-20090818/'
response = requests.get(url)
soup = BeautifulSoup(response.content)
names = soup.find_all('dt', text=re.compile('Editors?:'))[0].find_next('dd').text
```

Here the result is returned in a string with both names and affiliation. A regular expression using the `re` module matches text to find the `dt` HTML tag containing the word 'Editor' or 'Editors' followed by a colon. After BeautifulSoup has found the the relevant `dt` HTML tag, it identifies the text of the following `dt` tag with the `find_next` method of the BeautifulSoup object.

3.8.3 NLTK

NLTK (*Natural Language Processing Toolkit*) is one of the leading natural language processing packages for Python. It is described in depth by the authors of the package in the book *Natural Language Processing with Python* [36], available online. There are many submodules in NLTK, some of them displayed in Table 3.7. Associated with the package is a range of standard natural language processing corpora which each and all

Name	Description	Example
`nltk.app`	Miscellaneous application, e.g., a WordNet browser	`nltk.app.wordnet`
`nltk.book`	Example texts associated with the book [36]	`nltk.book.sent7`
`nltk.corpus`	Example texts, some of them annotated	`nltk.corpus.shakespeare`
`nltk.text`	Representation and text	
`nltk.tokenize`	Word and sentence segmentation	`nltk.tokenize.sent_tokenize`

Table 3.7: NLT submodules.

can be downloaded with the `nltk.download` interactive function. Once downloaded, the corpora are made available by functions in the `nltk.corpus` submodule.

3.8.4 Tokenization and part-of-speech tagging

Tokenization separates a text into tokens (desired constituent parts), usually either sentences or words. NLTK has two functions in each basic namespace: `nltk.sent_tokenize` and `nltk.word_tokenize`.

For social media-style texts ordinary sentence and word segmentation and part-of-speech tagging might work poorly. One common problem is the handling of URLs that standard word tokenizers typically splits into multiple tokens. Christopher Potts has implemented a specialized tokenizer for Twitter messages available for noncommercial applications in the `happyfuntokenizing.py` file. Another tokenizer for Twitter is Brendan O'Connor's `twokenize.py` from TweetMotif [38]. Myle Ott distributes a newer version of the twokenize.py.

Specialized Twitter part-of-speech (POS) tags, an POS-annotated corpus and a system for POS-tagging have been developed [39]. Though originally developed for Java a wrapper exists for Python.

NLTK makes POS tagging available out of the box. Here we define a small text and let NLTK POS tag it to find all nouns in singular form:

```
>>> text = ("To suppose that the eye with all its inimitable contrivances "
            "for adjusting the focus to different distances, for admitting "
            "different amounts of light, and for the correction of spherical "
            "and chromatic aberration, could have been formed by natural "
            "selection, seems, I freely confess, absurd in the highest degree. "
            "When it was first said that the sun stood still and the world "
            "turned round, the common sense of mankind declared the doctrine "
            "false; but the old saying of Vox populi, vox Dei, as every "
            "philosopher knows, cannot be trusted in science.")

>>> import nltk
>>> pos_tags = [nltk.pos_tag(nltk.word_tokenize(sent))
                for sent in nltk.sent_tokenize(text)]
>>> pos_tags[0][:5]
[('To', 'TO'), ('suppose', 'VB'), ('that', 'IN'), ('the', 'DT'), ('eye', 'NN')]
>>> [word for sent in pos_tags for word, tag in sent if tag == 'NN'] # Nouns
['eye', 'focus', 'admitting', 'correction', 'aberration', 'selection', 'degree',
'sun', 'world', 'round', 'sense', 'mankind', 'doctrine', 'false',
'populi', 'vox', 'philosopher', 'science']
```

Note that we have word tokenized and POS-tagged each sentence individually.

The efficient Cython-based natural language processing toolkit spaCy also has POS tagging for English texts. With the above text as Unicode the identification of singular nouns in the text may look like:

```
>>> from __future__ import unicode_literals
>>> from spacy.en import English
>>> nlp = English()
>>> tokens = nlp(text)
>>> [(token.orth_, token.tag_) for token in tokens][:4]
```

```
[(u'To', u'TO'), (u'suppose', u'VB'), (u'that', u'IN'), (u'the', u'DT')]
>>> [token.orth_ for token in tokens if token.tag_ == 'NN']
[u'eye', u'focus', u'light', u'correction', u'aberration',
u'selection', u'confess', u'absurd', u'degree', u'sun', u'world',
u'round', u'sense', u'mankind', u'doctrine', u'false', u'saying',
u'populi', u'philosopher', u'science']
```

Note the differencies in POS-tagging between NLTK and spaCy in words such as 'admitting' and 'light'. In its documentation spaCy claims to have both more accuracy and much faster execution than NLTK's POS-tagging.

3.8.5 Language detection

The `langid` may detect language. Here is a Danish text correctly classified:

```
>>> import langid
>>> langid.classify(u'Det␣er␣ikke␣godt␣håndværk.')
('da', 0.9681243715129888)
```

Another language detector is Chromium Compact Language Detector. The `cld` module makes a single function available:

```
>>> import cld
>>> cld.detect(u'Det␣er␣ikke␣godt␣håndværk.'.encode('utf-8'))
('DANISH', 'da', False, 30, [('DANISH', 'da', 63, 49.930651872399444),
                             ('NORWEGIAN', 'nb', 37, 26.410564225690276)])
```

Here the input is not Unicode, but rather UTF-8.

The `textblob` module also has a language detector:

```
>>> from textblob import TextBlob
>>> TextBlob(u'Det␣er␣ikke␣godt␣håndværk.').detect_language()
u'da'
```

The language detection in this module has been using the Google Translate service for the detection. Although this seems to offer quite good results, any repeated use could presumable be blocked by Google.

3.8.6 Sentiment analysis

Sentiment analysis methods can be grouped in wordlist-based methods and methods based on a trained classifier. The perhaps simplest Pythonic sentiment analysis is included in the `textblob` module. The sentiment analysis is readily available as an attribute to the `textblob.TextBlob` object:

```
>>> from textblob import TextBlob
>>> TextBlob('This␣is␣bad.').sentiment.polarity
-0.6999999999999998
>>> TextBlob('This␣is␣way␣worse␣than␣bad.').sentiment.polarity
-0.5499999999999999
>>> TextBlob('This␣is␣not␣bad.').sentiment.polarity
0.3499999999999999
```

The base sentiment analyzer uses an English word-based sentiment analyzer and process the text so it will handle a few cases of negations. The `textblob` base sentiment analyzer comes from the `pattern` module. The interface in the `pattern` library is different:

```
>>> from pattern.en import sentiment
>>> sentiment('This␣is␣way␣worse␣than␣bad.')
(-0.5083333333333333, 0.6333333333333333)
```

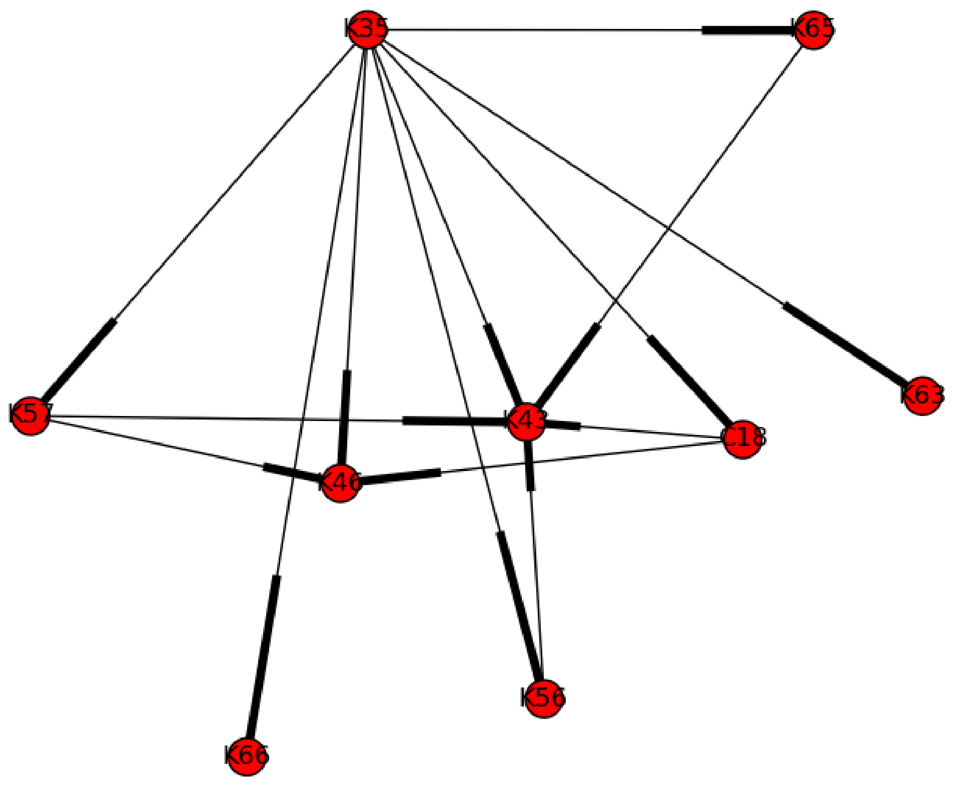

Figure 3.2: Comorbidity for ICD-10 disease code (appendicitis).

The returned values are polarity and subjectivity as for the `textblob` method.

Both `pattern` and `textblob` rely (in their default setup) on the `en-sentiment.xml` file containing over 2,900 English words where WordNet identifier, POS tag, polarity, subjectivity, subjectivity, intensity and confidence are encoded for each word. Numerous other wordlists for sentiment analysis exist, e.g., my AFINN wordlist [40]. Good wordlist-based sentiment analyzer often use multiple wordlists.

3.9 Network mining

The NetworkX package (`networkx`, [41]) has positioned itself as the primary Python package for small-scale network mining. It contains classes for representing networks: undirected, directed as well as multigraphs and multidigraphs (graphs with multiple edges between nodes). A series of classic simple graphs may be set up with functions in the package, but graphs can also be setup by the user. NetworkX makes a large number of network analysis functions available and graphs can directly be plotted with a Matplotlib interface. NetworkX is usually aliases with `import networkx as nx`.

Below is an example of network plotting temporal disease cooccurences (comorbidity) from a published paper. Data is read as the supplementary material to a published paper stored as an Microsoft Excel spreadsheet. We let `requests` fetch the file and send it to `pandas.read_excel` to setup a Pandas data frame. Extracting the two columns with ICD-10 disease codes we build up the directed graph as a `networkx.DiGraph` with diseases as nodes and comorbidity as edges. Finally we draw a part of the graph as the 'ego-graph' around the node K35, which is the disease code for appendicitis:

```
import matplotlib.pyplot as plt
import networkx as nx
import pandas as pd
import requests
from StringIO import StringIO

URL = ('http://www.nature.com/ncomms/2014/140617/'
       'ncomms5022/extref/ncomms5022-s2.zip')

data = pd.read_excel(StringIO(requests.get(URL).content),
                     'Supplementary␣Data␣1', skiprows=3, header=4)

disease_graph = nx.DiGraph()
disease_graph.add_edges_from(data[['Code', 'Code.1']].itertuples(index=False))
nx.draw(nx.ego_graph(disease_graph, 'K35'))
plt.show()
```

Figure 3.2 displays the resulting plot after we call the `matplotlib.pyplot.show` function and after we have extracted the ego-graph with `networkx.ego_graph` and plotted it with `networkx.draw`.

3.10 Miscellaneous issues

3.10.1 Lazy computation

For methods or attributes that take some time to compute you might want to store the result for any subsequent calls, — if the original input data does not change. The below listing uses a decorator from the `lazy` module, to make a 'lazy attribute' out of a function, so that only in the first invocation of the attribute the value is computed. In this case it is the left singular vector of a matrix that is computed using the singular value decomposition function from the `scipy.linalg` module. In any subsequent calls a stored value is used for `X.U`.

Listing 3.3: Lazy attribute for a slow computation

```
from lazy import lazy
from numpy import matrix
from scipy.linalg import svd

class Matrix(matrix):
    @lazy
    def U(self):
        U, s, Vh = svd(self, full_matrices=False)
        return U


import numpy.random as npr

# Initialize with a large random matrix
X = Matrix(npr.random((2000, 300)))

X.U    # Slow - first call computes the value and stores it
X.U    # Fast - second call uses the cached value and is much faster
```

The first call to the attribute may take several hundred milliseconds to execute, while the second call takes in the order of microseconds. Note that the decorator changes the method `Matrix.U()` so that the computed values should not be accessed as a function (`X.U()`) but rather as an attribute (`X.U`) without calling parentheses. If we wanted to access the right singular values (`Vh`) we would need to implement a second method with computation of the singular values. In this case we will not take advantage of that the same computation occurs both for `Matrix.U` and `Matrix.Vh`.

It is possible to move the computation to the constructor:

```
class Matrix(matrix):
    def __init__(self, *args, **kwargs):
        matrix.__init__(self, *args, **kwargs)
        self._U, self._s, self._Vh = svd(self, full_matrices=False)
    @property
    def U(self):
        return self._U
```

In this case the singular vectors are only computed once, but it also means that the computation will always happen, even though the attribute with the singular vectors is never accessed. However, in this case we only perform one single computation for `Matrix.U`, and `Matrix.Vh` if we implemented that attribute.

It is possible to only compute the singular vectors when they are needed and only compute the singular value decomposition once by moving the computation to a separate method and have multiple attributes calling it.

```
class Matrix(matrix):
    def __init__(self, *args, **kwargs):
        matrix.__init__(self, *args, **kwargs)
        self._U, self._s, self._Vh = None, None, None
    def svd(self):
        self._U, self._s, self._Vh = svd(self)
        return self._U, self._s, self._Vh
    @property
    def U(self):
        if self._U is None:
            self.svd()
        return self._U
    @property
    def s(self):
        if self._s is None:
            self.svd()
        return self._s
    @property
    def Vh(self):
        if self._Vh is None:
            self.svd()
        return self._Vh
```

Applying this class:

```
>>> X = Matrix(npr.random((3000, 200)))   # Fast
>>> X.Vh     # Slow - computing the SVD
>>> X.Vh     # Fast
>>> X.U      # Fast
```

3.11 Testing data mining code

Testing in data mining code will present you with at least three issues that to some degree distinguishes it ordinary software engineering testing:

1. The test should work on data structures, such as arrays where multiple elements should be checked.

2. Numerical precision in the computation means that computed results are different from expected ‘exact’ results.

3. For machine learning algorithms you may have no idea what the result should be, and indeed the task of machine learning is to develop an algorithm that performs well.

Numpy has developed a testing framework to deal with the first two issues which is available in the `numpy.testing` module.

Chapter 4

Case: Pure Python matrix library

4.1 Code listing

Below is a listing of an example of a Python module partly implementing a pure Python matrix class with a Numpy-like interface. Note that docstrings are used together with small doctests for some of the functions.

Listing 4.1: Matrix

```
"""Matrix."""

import logging
from logging import NullHandler

log = logging.getLogger(__name__)

# Avoid "No handlers" message if no logger
log.addHandler(NullHandler())


class Matrix(object):

    """Numerical matrix."""

    def __init__(self, obj):
        """Initialize matrix object."""
        log.debug("Constructing␣matrix␣object")
        self._matrix = obj

    def __getitem__(self, indices):
        """Get element in matrix.

        Examples
        --------
        >>> m = Matrix([[1, 2], [3, 4]])
        >>> m[0, 1]
        2

        """
        return self._matrix[indices[0]][indices[1]]

    def __setitem__(self, indices, value):
        """Set element in matrix.
        Examples
        --------
        >>> m = Matrix([[1, 2], [3, 4]])
        >>> m[0, 1]
        2
        >>> m[0, 1] = 5
        >>> m[0, 1]
        5

        """
        self._matrix[indices[0]][indices[1]] = value

    @property
    def shape(self):
        """Return shape of matrix.

        Examples
        --------
        >>> m = Matrix([[1, 2], [3, 4], [5, 6]])
        >>> m.shape
        (3, 2)

        """
        rows = len(self._matrix)
        if rows == 0:
            rows = 1
            columns = 0
        else:
            columns = len(self._matrix[0])
        return (rows, columns)

    def __abs__(self):
        """Return the absolute value.

        Examples
        --------
        >>> m = Matrix([[1, -1]])
        >>> m_abs = abs(m)
        >>> m_abs[0, 1]
        1

        """
        result = Matrix([[abs(element) for element in row]
                         for row in self._matrix])
        return result

    def __add__(self, other):
        """Add number to matrix.

        Parameters
        ----------
        other : integer or Matrix

        Returns
        -------
        m : Matrix
            Matrix of the same size as the original matrix

        Examples
        --------
        >>> m = Matrix([[1, 2], [3, 4]])
        >>> m = m + 1
        >>> m[0, 0]
        2

        >>> m = m + Matrix([[5, 6], [7, 8]])
        >>> m[0, 0]
        7

        """
        if isinstance(other, int) or isinstance(other, float):
            result = [[element + other for element in row]
                      for row in self._matrix]
        elif isinstance(other, Matrix):
            result = [[self[m, n] + other[m, n]
                       for n in range(self.shape[1])]
                      for m in range(self.shape[0])]
        else:
            raise TypeError
        return Matrix(result)

    def __mul__(self, other):
        """Multiply number to matrix.

        Parameters
        ----------
        other : integer, float

        Returns
        -------
        m : Matrix
            Matrix with multiplication result

        Examples
        --------
        >>> m = Matrix([[1, 2], [3, 4]])
        >>> m = m * 2
        >>> m[0, 0]
        2

        """
        if isinstance(other, int) or isinstance(other, float):
            result = [[element * other for element in row]
                      for row in self._matrix]
        else:
            raise TypeError
        return Matrix(result)

    def __pow__(self, other):
        """Compute power of element with ‘other‘ as the exponent.
        Parameters
        ----------
        other : integer, float

        Returns
        -------
        m : Matrix
            Matrix with multiplication result

        Examples
        --------
        >>> m = Matrix([[1, 2], [3, 4]])
        >>> m = m ** 3
        >>> m[0, 1]
        8

        """
        if isinstance(other, int) or isinstance(other, float):
            result = [[element ** other for element in row]
                      for row in self._matrix]
        else:
            raise TypeError
        return Matrix(result)

    def __str__(self):
        """Return string representation of matrix."""
        return str(self._matrix)

    def transpose(self):
        """Return transposed matrix.

        Examples
        --------
        >>> m = Matrix([[1, 2], [3, 4]])
        >>> m = m.transpose()
        >>> m[0, 1]
        3

        """
        log.debug("Transposing")
        # list necessary for Python 3 where zip is a generator
        return Matrix(list(zip(*self._matrix)))

    @property
    def T(self):
        """Transposed of matrix.

        Returns
        -------
        m : Matrix
            Copy of matrix

        Examples
        --------
        >>> m = Matrix([[1, 2], [3, 4]])
        >>> m = m.T
```

```
>>> m[0, 1]
3

"""
log.debug("Calling␣transpose()")
return self.transpose()
```

Chapter 5

Case: Pima data set

5.1 Problem description and objectives

Pima Indians, a community of American Indians in Arizona, have an unusual high incidence and prevalence of diabetes mellitus, with one study finding analmost 20-fold greater incidence compared to a predominately white population in Minnesota [42, 43]. The extraordinary high rate has given rise to a considerable body of research, e.g., the U.S. National Institute of Diabetes and Digestive and Kidney Diseases has studied the community,[1] and they have assemble a data set, which Vincent Sigillito in 1990 donated to the Machine Learning Repository at the Center for Machine Learning and Intelligent Systems, University of California, Irvine. Since an initial machine learning study of the data [44], researchers have used it as a benchmark data set for demonstrating and testing data mining algorithms. The data set has 9 variables, see Table 5.1, and the typical machine learning task is to predict the class variable of whether the the subject has been diagnosed as diabetic or not.

1.	Number of times pregnant
2.	Plasma glucose concentration
3.	Diastolic blood pressure (mm Hg)
4.	Triceps skin fold thickness (mm)
5.	[2-Hour serum insulin (mu U/ml)]
6.	Body mass index (weight in kg/(height in m)^2)
7.	Diabetes pedigree function
8.	Age (years)
9.	Diabetes (No or Yes)

Table 5.1: Variables in the Pima data set. From description at UCI Machine Learning Repository.

Python packages seems not yet to have incorporated the data set, so we will leave Python and take the data set from R, which has the data set available in its `MASS` library, writing it to comma-separated values (CVS) data files:

```
> R
> library(MASS)
> write.csv(Pima.tr, 'pima.tr.csv')
> write.csv(Pima.te, 'pima.te.csv')
```

Now we have two files ready for reading with Python. An examination of the first four lines of the `pima.tr.csv` file yields:

```
"","npreg","glu","bp","skin","bmi","ped","age","type"
"1",5,86,68,28,30.2,0.364,24,"No"
"2",7,195,70,33,25.1,0.163,55,"Yes"
"3",5,77,82,41,35.8,0.156,35,"No"
```

Here the first column is the row index: Note that the last column ('type') does not contain numerical values in its cells, but rather a string for the categorical column, thus we cannot read it directly into a `numpy.array`.

[1] http://diabetes.niddk.nih.gov/dm/pubs/pima/pathfind/pathfind.htm

This data set has left out one of the variables (the measurement of serum insulin) and has also excluded subjects that had missing data for some of the variables. Furthermore the data set is split into a training set and test set, with 200 and 332 subjects respectively. The full data set represents data from 768 female subjects.

5.2 Descriptive statistics and plotting

For reading the comma-separated values data, rather than using the function from the PSL `csv` module, the `csvkit` or SciPy's `scipy.loadtxt` function, we will use the function from pandas: `pandas.read_csv`. It automatically handles the column and row headers as well as handle the categorical value in the last column, — issues that would require considerable more code should we have used `csv` or `scipy.loadtxt`.

```
import pandas as pd

pima_tr = pd.read_csv('pima.tr.csv', index_col=0)
pima_te = pd.read_csv('pima.te.csv', index_col=0)
```

`pandas.read_csv` handles the column header by default, setting the `column` variable of the returned object based on the first line in the CSV file, while for the row header (here the first column) we need to explicitly set it with the `index_col=0` input argument. We now have access, e.g., to the number of pregnancies column in the training set with `pima_tr.npreg` or `pima_tr['npreg']`.

If we would like to read in the full data set with the insulin serum measurement and with subject having missing data we can grab the dat from the UCI repository:

```
url = ('http://ftp.ics.uci.edu/pub/machine-learning-databases/'
       'pima-indians-diabetes/pima-indians-diabetes.data')
pima = pd.read_csv(url, names=['npreg', 'glu', 'bp', 'skin',
                               'ins', 'bmi', 'ped', 'age', 'type'])
```

Note here that the `pandas.read_csv` are able to download the data from the Internet and that there is no header in the data set, which is why the columns are named explicit with the `names` argument.

There are various ways to get an overview of the data. The Pandas data frame object has, e.g., `mean`, `std`, `min` methods. These can all be displayed with the `describe` data frame method:

```
 >>> pima_tr.describe()
            npreg         glu          bp        skin         bmi         ped  \
count  200.000000  200.000000  200.000000  200.000000  200.000000  200.000000
mean     3.570000  123.970000   71.260000   29.215000   32.310000    0.460765
std      3.366268   31.667225   11.479604   11.724594    6.130212    0.307225
min      0.000000   56.000000   38.000000    7.000000   18.200000    0.085000
25%      1.000000  100.000000   64.000000   20.750000   27.575000    0.253500
50%      2.000000  120.500000   70.000000   29.000000   32.800000    0.372500
75%      6.000000  144.000000   78.000000   36.000000   36.500000    0.616000
max     14.000000  199.000000  110.000000   99.000000   47.900000    2.288000

              age
count  200.000000
mean    32.110000
std     10.975436
min     21.000000
25%     23.000000
50%     28.000000
75%     39.250000
max     63.000000
```

Here we see that the maximum number of pregnancies in the data set for an Indian women is 14 ('max' row and 'npreg' column), the maximum BMI is 47.9 and the age ranges between 21 and 63 with an average of 32.11. Note that the describe method ignored the categorical 'type' column.

With the grouping functionality using the `groupby` method of the Pandas data frame we can get the summary statistics based on the rows grouped into sets depending on the value of the specified column When we used the 'type' column for the grouping operation the two set become 'No' and 'Yes':

```
>>> pima_tr.groupby('type').mean()
         npreg         glu         bp       skin        bmi       ped  \
type
No    2.916667  113.106061  69.545455  27.204545  31.074242  0.415485
Yes   4.838235  145.058824  74.588235  33.117647  34.708824  0.548662

            age
type
No    29.234848
Yes   37.691176
```

The `groupby` method returns a Pandas object called `DataFrameGroupBy`. Like the `DataFrame` object it has summary statistics methods and the above listing showed an example with the `mean` method. It indicates to us that the women with a diagnose of diabetes mellitus have on average a higher number of pregnancies (4.8 against 2.9), a higher BMI value and higher age.

With standard Pandas functionality we can also get an overview of the correlation between the variables with the `corr` method of the data frame:

```
>>> pima_tr.corr()
          npreg       glu        bp      skin       bmi       ped       age
npreg  1.000000  0.170525  0.252061  0.109049  0.058336 -0.119473  0.598922
glu    0.170525  1.000000  0.269381  0.217597  0.216790  0.060710  0.343407
bp     0.252061  0.269381  1.000000  0.264963  0.238821 -0.047400  0.391073
skin   0.109049  0.217597  0.264963  1.000000  0.659036  0.095403  0.251926
bmi    0.058336  0.216790  0.238821  0.659036  1.000000  0.190551  0.131920
ped   -0.119473  0.060710 -0.047400  0.095403  0.190551  1.000000 -0.071410
age    0.598922  0.343407  0.391073  0.251926  0.131920 -0.071410  1.000000
```

It shows skin fold thickness and BMI to be quite correlated with a correlation of 0.659036 and that age and number of pregnancies are also quite correlated with 0.598922.

The `seaborn` package has a nice correlation plot function which works with with Pandas, corresponding to the Pandas data frame `corr` method:

```
import seaborn as sns
import matplotlib.pyplot as plt

sns.corrplot(pima_tr)
plt.show()
```

Seaborn produces a color-coded correlation plot which also displays the variable names, the numerical correlation coefficients and the result of statistical tests for the correlation coefficient, see Figure 5.1. The `statsmodels` also has a correlation plot function hidden as `statsmodels.graphics.correlation.plot_corr`, but it does not yield as informative a plot.

5.3 Statistical tests

Functions from `statsmodels` can be used for statistical tests. We can model the 'type' column as the dependent variable using the other columns as independent variables in a generalized linear model with a logit link function and an intercept column automatically added by default:

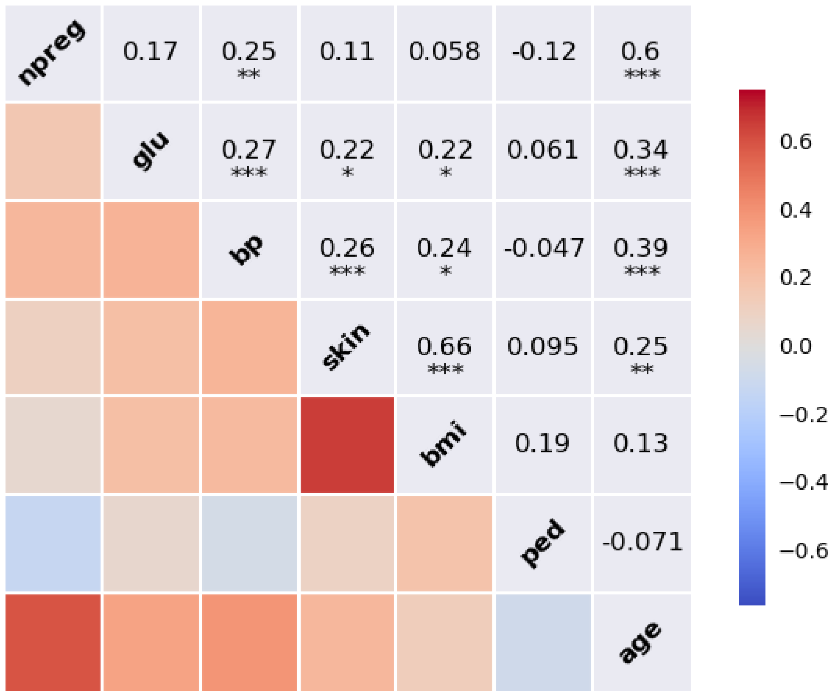

Figure 5.1: Seaborn correlation plot on the Pima data set constructed with the `seaborn.corrplot` function. The diagonal displays the variables names, the upper right triangle the numerical correlation coefficients together with starts indicating statistical significance and with the lower left triangle color-coded according to correlation.

```
import statsmodels.api as sm
import statsmodels.formula.api as smf

model = smf.glm('type␣~␣npreg␣+␣glu␣+␣bp␣+␣skin␣+␣bmi␣+␣ped␣+␣age',
                data=pima_tr, family=sm.families.Binomial()).fit()
print(model.summary())
```

The last line will print out the result of the fitting of the model, with the fitted parameter values, their standard errors, their t-values, the two-sided P-values and the 95% confidence intervals:

```
                 Generalized Linear Model Regression Results
==============================================================================
Dep. Variable:     ['type[No]', 'type[Yes]']   No. Observations:          200
Model:                                   GLM   Df Residuals:              192
Model Family:                       Binomial   Df Model:                    7
Link Function:                         logit   Scale:                     1.0
Method:                                 IRLS   Log-Likelihood:        -89.195
Date:                       Fri, 17 Oct 2014   Deviance:               178.39
Time:                               22:43:41   Pearson chi2:             177.
No. Iterations:                            7
```

```
==============================================================================
                 coef    std err          z      P>|z|      [95.0% Conf. Int.]
------------------------------------------------------------------------------
Intercept      9.7731      1.770      5.520      0.000         6.303    13.243
npreg         -0.1032      0.065     -1.595      0.111        -0.230     0.024
glu           -0.0321      0.007     -4.732      0.000        -0.045    -0.019
bp             0.0048      0.019      0.257      0.797        -0.032     0.041
skin           0.0019      0.022      0.085      0.932        -0.042     0.046
bmi           -0.0836      0.043     -1.953      0.051        -0.168     0.000
ped           -1.8204      0.666     -2.735      0.006        -3.125    -0.516
age           -0.0412      0.022     -1.864      0.062        -0.084     0.002
==============================================================================
```

The fitted parameters which are displayed in the 'coef' column are also available in the `model.params` attribute. This variable has the `pandas.Series` data type, so we, e.g., can access float value (actually `numpy.float64`) of the parameter for the intercept with `model.params.Intercept`. The other numerical data displayed with the `print` function are also available, e.g., the *t*-values from the 't' columns appear in the `model.tvalues` attribute.

5.4 Predicting diabetes type

Now we will want to make machine learning classifiers that can predict the diagnostic label on the test set based on training model parameters on the training set. First we will make a Python class for a baseline machine learning classifier, — one that just predict the most common label in the training set:

```
class NoClassifier():
    """Classifier that predict all data as "No"."""
    def predict(self, x):
        return pd.Series(["No"] * x.shape[0])
```

To evaluate the performance of the classifier we will need a function that compares the predicted value with the true value. Here we define an `accuracy` function function that computes the fraction of correctly predicted labels:

```
def accuracy(truth, predicted):
    if len(truth) != len(predicted):
        raise Exception("Wrong␣sizes␣...")
    total = len(truth)
    if total == 0:
        return 0
    hits = len(filter(lambda (x, y): x == y, zip(truth, predicted)))
    return float(hits)/total
```

Unsurprisingly scikit-learn has also an accuracy function. It is available as `sklearn.metrics.accuracy_score`, and we could have used that instead.

```
from scipy.linalg import pinv
from numpy import asarray, hstack, mat, ones, where

class LinearClassifier():
    """ y = X*b and b = pinv(X) * y """
    def __init__(self):
        self._parameters = None

    def from_labels(self, y):
        return mat(where(y=="No", -1, 1)).T
    def to_labels(self, y):
        return pd.Series(asarray(where(y<0, "No", "Yes")).flatten())

    def fit(self, x, y):
        intercept = ones((x.shape[0], 1))
        self._parameters = pinv(hstack((mat(x), intercept))) * self.from_labels(y)

    def predict(self, x):
        intercept = ones((x.shape[0], 1))
        y_estimated = hstack((mat(x), intercept)) * self._parameters
        return self.to_labels(y_estimated)
```

Note that the `from_labels` and `to_labels` methods use no data from the class, so we could have made these methods static with the `@staticmethod` decorator.

Chapter 6

Case: Data mining a database

6.1 Problem description and objectives

We have a relational database accessible via SQL and we would like to find 'interesting' patterns in it of any sort. What should we do?

First we will need to identify a relevant database. Here we will use the Chinook Database, — a readily available demonstration database which models a digital media store with customer data and transactions as well as metadata about the music—artists, albums and media tracks. The database is distributed from Microsoft's CodePlex cloud service, but also available out of the box from the `db.py` package.

The database creators made some parts of the data from real-life data, while other parts are made up. A classical data mining exercise on transaction data is market basket analysis, but given that some data are made up it might not produce interesting results. Instead we will focus on making data mining that can answer whether the data is made up or not. The answer to this problem is readily available on the Chinook homepage, and we will see to which extent we can establish the authenticity of the data by pure data mining techniques.

6.2 Reading the data

Since the `db.py` package makes the Chinook Database available we can trivally connect to the database using its `db.DemoDB` class:

```
>>> from db import DemoDB
>>> db = DemoDB()
```

Instancing the `DemoDB` will read the data and setup it up for access in the `db` object. Note that here we—somewhat confusingly—use the same name for the database object as the module name. An overview of the tables are available with the `tables` attribute. Here we have a shortened output:

```
>>> db.tables
+---------------+------------------------------------...
| Table         | Columns
+---------------+------------------------------------...
| Album         | AlbumId, Title, ArtistId
| Artist        | ArtistId, Name
| Customer      | CustomerId, FirstName, LastName, ...
| Employee      | EmployeeId, LastName, FirstName, Title, ...
...
```

Examining the schema of the individual tables is likewise straightforward as the individual tables are accessible as attributes to the `tables` atttribute:

```
>>> db.tables.Album
+--------------------------------------------------------------+
|                            Album                             |
+----------+---------------+-----------------+-----------------+
| Column   | Type          | Foreign Keys    | Reference Keys  |
+----------+---------------+-----------------+-----------------+
| AlbumId  | INTEGER       |                 | Track.AlbumId   |
| Title    | NVARCHAR(160) |                 |                 |
| ArtistId | INTEGER       | Artist.ArtistId |                 |
+----------+---------------+-----------------+-----------------+
```

These tables have a specific object type

```
>>> type(db.tables.InvoiceLine)
<class 'db.db.Table'>
```

For accessing the actual data in the database we can use the `select`, `all`, `sample` or `head` methods of the `db.db.Table` object.

```
>>> db.tables.InvoiceLine.head()
   InvoiceLineId  InvoiceId  TrackId  UnitPrice  Quantity
0              1          1        2       0.99         1
1              2          1        4       0.99         1
2              3          2        6       0.99         1
3              4          2        8       0.99         1
4              5          2       10       0.99         1
5              6          2       12       0.99         1
```

The returned object is a `pandas.DataFrame`, such that further data analysis with the functions and methods of Pandas is straightforward.

Data returned as Pandas' data frame can also be obtained via the `db.query` function where SQL statements can be formulated, e.g., the following two statements return the same data:

```
>>> db.tables.Album.head(3)
   AlbumId                                  Title  ArtistId
0        1  For Those About To Rock We Salute You         1
1        2                      Balls to the Wall         2
2        3                      Restless and Wild         2
>>> db.query('select␣*␣from␣Album␣limit␣␣3')
   AlbumId                                  Title  ArtistId
0        1  For Those About To Rock We Salute You         1
1        2                      Balls to the Wall         2
2        3                      Restless and Wild         2
```

6.3 Graphical overview on the connections between the tables

We can get another overview of the database by plotting the tables and their connections. Figure 6.1 shows the tables as nodes and references among the tables as edges. The references are etablished by examining the foreign keys in each table. The size of the nodes are set as a monotone function of the number of rows in each table, — here the square root. The graph is construted with NetworkX. This default rendering in NetworkX does not reveal connections within a table, e .g., the 'Employee' table has a foreign key to itself to store which employee reports to which other employee (his/her superior).

```
from db import DemoDB
import networkx as nx
import numpy as np
import matplotlib.pyplot as plt
```

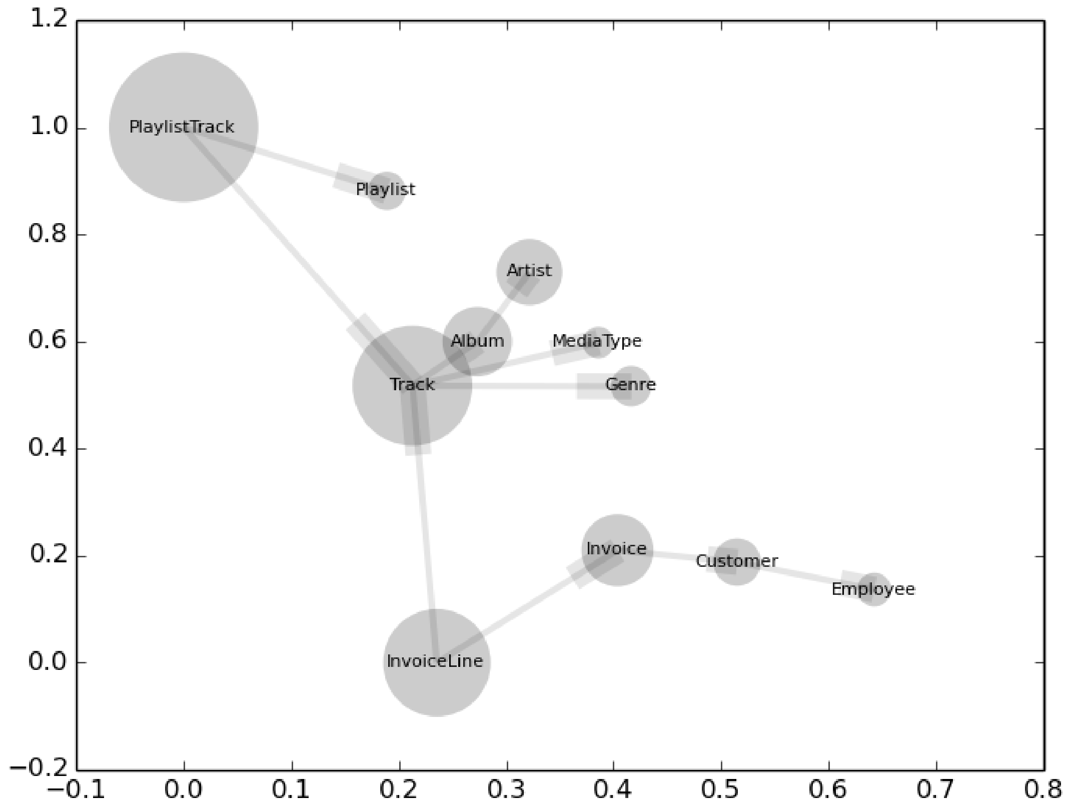

Figure 6.1: Database tables graph with the Chinook database where nodes are tables and edges indicate foreign keys connections.

```
# Load Chinook database
db = DemoDB()

# Construct graph
graph = nx.MultiDiGraph()
for table in db.tables:
    graph.add_node(table.name, number_of_rows=len(table.all()))
    for key in table.foreign_keys:
        graph.add_edge(table.name, key.table)

# Position and size of nodes
pos = nx.layout.fruchterman_reingold_layout(graph)
sizes = 100 + 50 * np.sqrt([attrs['number_of_rows']
                             for node, attrs in graph.nodes(data=True)])

# Draw the components of the graph
nx.draw_networkx_edges(graph, pos=pos, node_color='k', alpha=0.1, width=3)
nx.draw_networkx_nodes(graph, pos=pos, node_color='k', alpha=0.2, linewidths=0.,
                         node_size=sizes)
nx.draw_networkx_labels(graph, pos=pos, font_color='k', font_size=8)
plt.show()
```

6.4 Statistics on the number of tracks sold

Joining the data from the `Track` and the `InvoiceLine` tables allow us to create a track chart with listing the most sold tracks. With the `db.query` we formulate an SQL query with the SQL join between the two tables and join them by the `TrackId` identifier available in both tables:

```
>>> sql = """
select * from Track
    left outer join
      (select TrackId, sum(quantity) as Sold
           from InvoiceLine
           group by TrackId) as Count
      on Track.TrackId = Count.TrackId
    order by Sold desc
"""
sold_per_track = db.query(sql).fillna(0)
>>> sold_per_track.head()[['Name', 'Composer', 'Sold']]
                Name                                   Composer  Sold
0  Balls to the Wall                                       None     2
1   Inject The Venom  Angus Young, Malcolm Young, Brian Johnson     2
2         Snowballed  Angus Young, Malcolm Young, Brian Johnson     2
3           Overdose                                      AC/DC     2
4    Deuces Are Wild                 Steven Tyler, Jim Vallance     2
```

It appears that not all tracks in the database have been sold so with the left outer join we end up with some tracks with no entry in the 'Sold' column. When the data is returned as a `pandas.DataFrame` these missing entries have the value `NaN`. As they should be interpreted as zero we exchange NaN with zero using the `fillna` method of the data frame object.

We can get an overview of the number of sold track by plotting the histogram:

```
>>> import matplotlib.pyplot as plt
>>> sold_per_track['Sold'].hist()
>>> plt.xlabel('Number␣of␣items␣sold␣per␣track')
>>> plt.ylabel('Frequency')
>>> plt.show()
```

Here we should get suspicious if this part of Chinook was based on real-life data: Following the idea of the long tail [45], we should expect a few hit tracks selling a large number of items, while the most of the tracks should sell few.

Chapter 7

Case: Twitter information diffusion

7.1 Problem description and objectives

What features of a message determine whether it diffuses? News media have long been aware of what factors influence the ‘flow of news’. One obvious feature is the circulation of the media: If a newspaper with a large number of subscribers carries a story it is more likely to be read more than if the story appeared in a newspaper with a small number of subscribers. A classic study from 1965 on news criteria mentioned—as one of the features—negativity [46]. Several studies have looked on Twitter and analyzed how the posts (tweets) spread as reposted (retweeted) [47, 48]. One study examined the features: appearance of a hashtag with #, appearance of a mentioning of another user with @, the appearance of a URL, the number other users a user follows and the number of follows a user has, the number of tweets the user have made, the number of favorites [47]. A following study added the affective valence of the tweet and the ‘newsness’ as features [48], and we will here attempt to reproduce this study.

7.2 Building a news classifier

To determine the ‘newsness’ we will build a classifier. With the `nltk` package comes several corpora. The installation of `ntlk` will not install the corpora, instead the user needs to download them via the interactive GUI program `nltk.download`, that eventually will make the corpora available via the `nltk.corpus` module. Here we will only use the Brown corpus, — a 3.2 MB sized corpus, which as a number of text labeled with the categories, e.g., news, reviews, romance, humor, etc.

First we will read in modules that we later will use

```
try:
    import cPickle as pickle
except ImportError:
    import pickle
from nltk.corpus import brown
from nltk.classify import apply_features, NaiveBayesClassifier
from nltk.tokenize import word_tokenize
```

We read in all the words from the corpus and find the unique words in the lowercase version. After that we read in all sentences into two data sets: one with sentence labeled as ‘news’, the other labeled with any of the other category:

```
unique_words = set([word.lower() for word in brown.words()])

news_sentences = brown.sents(categories='news')
other_sentences = brown.sents(categories=set(brown.categories()) - set(['news']))
```

To train a classifier we will use one-gram word features, i.e., indicate with a Boolean variable whether a word is present or not in each sentence.

```
def word_features(sentence):
    features = {word: False for word in unique_words}
    for word in sentence:
        if word.isalpha():
            features[word.lower()] = True
    return features

featuresets = apply_features(word_features, (
                 [(sent, 'news') for sent in news_sentences] +
                 [(sent, 'other') for sent in other_sentences]))
```

We can do a sampling test on whether the features are set up correctly, e.g., the word 'county' appears in the first sentence and in the `featureset` variable the associated value should be true, whereas the word 'city' occurs in the second sentence but not the first:

```
>>> news_sentences[0][:8]
['The', 'Fulton', 'County', 'Grand', 'Jury', 'said', 'Friday', 'an']
>>> featuresets[0][0]['county']
True
>>> featuresets[0][0]['city']
False
```

The actual training is done by instancing an object of the `nltk.classify.NaiveBayesClassifier` class with the `featuresets` as input to the `train` method:

```
classifier = NaiveBayesClassifier.train(featuresets)
```

The estimation takes some time, so when it finally finishes we save the trained classifier in the pickle format via the `pickle` module:

```
pickle.dump(classifier, open('news_classifier.pck', 'w'))
```

The pickle file will allow us to load the classifier in another Python session, rather than training the classifier again. The `pickle` module will load the classifier with `classifier = pickle.load(open('news_classifier.pck'))`.

We can make a couple of random tests displaying the estimated probability of being a news sentence:

```
>>> news = 'senate␣tax␣overhaul␣gains␣steam␣as␣floor␣debate␣awaits'
>>> classifier.prob_classify(word_features(news.split())).prob('news')
0.7145304532017633

>>> other = 'when␣are␣they␣going␣to␣let␣you␣back␣in␣the␣usa'
>>> classifier.prob_classify(word_features(other.split())).prob('news')
3.5592820175844e-05

>>> news2 = 'war␣criminal␣dies␣after␣taking␣poison␣in␣court'
>>> classifier.prob_classify(word_features(news2.split())).prob('news')
0.011451627538375269
```

While the two first sentences yield a reasonable classification, the third sentences is bad. When interpreting the probability, one should keep in mind that the dataset is quite unbalanced with only 8% of the sentences being news sentences:

```
>>> from __future__ import division
>>> len(news_sentences) / len(featuresets)
0.08062434600627834
```

Chapter 8

Case: Big data

8.1 Problem description and objectives

The data set size has always been recognized as an issue in machine learning, but data scientists have increasingly recognized the importance of large data sets in obtaining good useful performance from machine learning systems with quotes such as "the unreasonable effectiveness of data" [49], "No data like more data" [50] and "all models are wrong, and increasingly you can succeed without them" (when you have massive amounts of data) [51].

One approach to managing massive amounts of data is to regard it as an infinite stream of data where your algorithm handles each item/sample or subset/window of your data at a time: You do not read in the full data set to your algorithm at once, but continuously work on part of the data updating your model.

8.2 Stream processing of JSON

Ordinary Python modules for JSON might not handle large JSON files well, unable to work on part of the data. Ivan Sagalaev's `ijson` module provides a streaming API to JSON files with the potential to work on infinite data. As an example lets begin with a 4-line JSON string with a list of dictionaries processed by `ijson`:

```
import ijson
from StringIO import StringIO

json_string = """[
{"id": 1, "content": "hello"},
{"id": 2, "content": "world"}]
"""

sio = StringIO(json_string)
objects = ijson.items(sio, 'item')

for obj in objects:
    print(obj)
```

The `print` line displays the dictionary as the `ijson.items` returns a generator which can be iterated with the for loop. The second input argument to `ijson.items` tells which part of the JSON object the function should yield at each iteration. In this case an object is yielded at every JSON list item, and the variable `object` has the data type `dict`.

Lets scale the simple 4-line example up to a 3.1 gigabyte compressed JSON file (20140721.json.gz) provided by the Wikidata project currently available from http://dumps.wikimedia.org/other/wikidata/ and which contains over 15 million items in multiple languages. The `gzip` library will uncompress the file on-the-fly with the `gzip.open` function returning a file-like handle that directly can be feed into the `ijson.items` function.

```
import collections
import gzip
import ijson
import os.path


filename = os.path.expanduser('~/data/wikidata/20140721.json.gz')
id_company = 783794 # https://www.wikidata.org/wiki/Q783794

def get_instance_of_ids(subject):
    """Return numeric ids for 'instance of' (P31) object for subject."""
    ids = []
    if 'claims' in subject and 'P31' in subject['claims']:
        for statement in subject['claims']['P31']:
            try:
                id = statement['mainsnak']['datavalue']['value']['numeric-id']
                ids.append(id)
            except KeyError:
                pass
    return ids

objects = ijson.items(gzip.open(filename), 'item')
labels = collections.defaultdict(str)

for obj in objects:
    for language in ['ro', 'de', 'en']:
        if 'labels' in obj and language in obj['labels']:
            labels[obj['id']] = obj['labels'][language]['value']
            break
    ids = get_instance_of_ids(obj)
    if id_company in ids:
        print(labels[obj['id']])
```

In this case we print company names when an item in the Wikidata is annotated as an instance of a company (https://www.wikidata.org/wiki/Q783794), in the present case the generated output starts with: EADS, Sako, SABMiller, Berliet, Aixam, The Walt Disney Company, ... The company names are printed with their Romanian ('ro') names with fallback to German ('de') and English ('en') names.

8.2.1 Stream processing of JSON Lines

The JSON Lines format are newline-delimited JSON. This format is straightforward to read with standard Python functions and the ordinary `json` module.

```
from __future__ import print_function
import json
from six import StringIO

json_string = """{"id": 1, "content": "hello"}
{"id": 2, "content": "world"}
"""

sio = StringIO(json_string)
for line in sio:
```

```
obj = json.loads(line)
print(obj)
```

Bibliography

[1] Allen B. Downey. Think Python. O'Reilly Media, first edition, August 2012.

[2] Kevin Sheppard. Introduction to Python for econometrics, statistics and data analysis. Self-published, University of Oxford, version 2.1 edition, February 2014.

[3] Stephen Marsland. Machine learning: An algorithmic perspective. Chapman & Hall/CRC, 2009.

[4] Skipper Seabold and Josef Perktold. Statsmodels: econometric and statistical modeling with python. In *Proceedings of the 9th Python in Science Conference*, 2010.

Annotation: Description of the statsmodels package for Python.

[5] Florian Krause and Oliver Lindemann. Expyriment: A Python library for cognitive and neuroscientific experiments. *Behavior Research Methods*, 46(2):416–428, June 2013.

Annotation: Initial publication for the Expyriment Python package for stimulus presentation, response collection and recording in psychological experiments.

[6] Jeffrey M. Perkel. Programming: pick up python. *Nature*, 518(7537):125–126, February 2015.

Annotation: Report on the increasing use of Python programming in science explaining it as due to its simple syntax and scientific toolkits and online resources.

[7] Lutz Prechelt. An empirical comparison of C, C++, Java, Perl, Python, Rexx and Tcl. *Computer*, 33(10):23–29, October 2000.

[8] Sebastian Nanz and Carlo A. Furia. A comparative study of programming languages in Rosetta Code. ArXiv, September 2014.

Annotation: Compares C, C#, F#, Go, Haskell, Java, Python and Ruby in terms of lines of code, size of executable and running time.

[9] Philip Guo. Python is now the most popular introductory teaching language at top U.S. universities. BLOG@CACM, July 2014.

[10] Coverity. Coverity finds Python sets new level of quality for open source software. Press release, August 2013.

[11] Jürgen Scheible and Ville Tuulos. Mobile python: Rapid prototyping of applications on the mobile platform. Wiley, 1st edition, October 2007.

[12] Susan Tan. Python in the browser: Intro to Brython. YouTube, April 2014.

[13] Mads Ruben Burgdorff Kristensen, Simon Andreas Frimann Lund, Troels Blum, Kenneth Skovhede, and Brian Vinter. Bohrium: unmodified NumPy code on CPU, GPU, and cluster. In *Python for High Performance and Scientific Computing*, November 2013.

[14] Sue Gee. Python 2.7 to be maintained until 2020. I Programmer, April 2014.

[15] Kerrick Staley and Nick Coghlan. The "python" command on Unix-like systems. PEP 394, Python Software Foundation, 2011.

[16] Dan Sanderson. Programming Google App Engine. O'Reilly, Sebastopol, California, USA, second edition edition, October 2012.

[17] Philip J. Guo. Online Python Tutor: embeddable web-based program visualization for CS education. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 579–584, New York, NY, USA, March 2013. Association for Computing Machinery.

Annotation: A description of an interactive online program visualization service for the Python programming language.

[18] Ian Bicking. Python HTML parser performance. Ian Bicking: a blog, March 2008.

[19] Antoine Pitrou. Pickle protocol version 4. PEP 3154, Python Software Foundation, August 2011.

[20] Marc-André Lemburg. Python database API specification v2.0. PEP 249, Python Software Foundation, Beaverton, Oregon, USA, JNovember 2012.

[21] David Goodger and Guido van Rossum. Docstring conventions. PEP 257, Python Software Foundation, Beaverton, Oregon, USA, June 2001.

[22] Amit Patel, Antoine Picard, Eugene Jhong, Gregory P. Smith, Jeremy Hylton, Matt Smart, Mike Shields, and Shane Liebling. Google Python style guide, 2013.

Annotation: Coding style guide for Python.

[23] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[24] Guido van Rossum, Barry Warsaw, and Nick Coglan. Style guide for python code. Python Enhancement Proposals 8, Python Software Foundation, August 2013.

[25] Prabhu Ramachandran and Gaël Varoquaux. Mayavi: making 3D data visualization reusable. In Gaël Varoquaux, T. Vaught, and J. Millman, editors, *Proceedings of the 7th Python in Science Conference (SciPy 2008)*, pages 51–57, 2008.

[26] Prabhu Ramachandran and Gaël Varoquaux. Mayavi: 3D visualization of scientific data. *Computing in Science & Engineering*, 13(2):40–50, March-April 2011.

Annotation: Introduction to the Mayavi Python 3D scientific visualization package.

[27] Wes McKinney. Python for data analysis. O'Reilly, Sebastopol, California, first edition, October 2012.

Annotation: Book on data analysis with Python introducing the Pandas library.

[28] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[29] Michael Hanke, Yaroslav O. Halchenko, Per B. Sederberg, Stephen José Hanson, James V. Haxby, and Stefan Pollmann. PyMVPA: a Python toolbox for multivariate pattern analysis of fMRI data. *Neuroinformatics*, 7(1):37–53, March 2009.

[30] Michael Hanke, Yaroslav O. Halchenko, Per B. Sederberg, Emanuele Olivetti, Ingo Frund, Jochem W. Rieger, Christoph S. Herrmann, James V. Haxby, Stephen Jose Hanson, and Stefan Pollmann. PyMVPA: a unifying approach to the analysis of neuroscientific data. *Frontiers in neuroinformatics*, 3:3, 2009.

[31] Janez Demšar, Tomaž Curk, Aleš Erjavec, Črt Gorup, Tomaž Hočevar, Mitar Milutinovi, Martin Možina, Matija Polajnar, Marko Toplak, Anže Stari, Miha Štajdohar, Lam Umek, Lan Žagar, Jure Žbontar, Marinka Žitnik, and Blaž Zupan. Orange: data mining toolbox in Python. *Journal of Machine Learning Research*, 14:2349–2353, August 2013.

[32] Davide Albanese, Roberto Visintainer, Stefano Merler, Samantha Riccadonna, Giuseppe Jurman, and Cesare Furlanello. `mlpy`: machine learning Python. ArXiv, March 2012.

[33] T. Zito, N. Wilbert, L. Wiskott, and P. Berkes. Modular toolkit for data processing (mdp): a python data processing framework. *Frontiers in Neuroinformatics*, 2:8, 2008.

[34] Tom Schaul, Justin Bayer, Daan Wierstra, Yi Sun, Martin Felder, Frank Sehnke, Thomas Rückstie, and Jürgen Schmidhuber. Pybrain. *Journal of Machine Learning Research*, 11:743—746, February 2010.

Annotation: Presents the PyBrain Python machine learning package.

[35] Radim Řehůřek and Petr Sojka. Software framework for topic modelling with large corpora. In *Proceedings of LREC 2010 workshop New Challenges for NLP Frameworks*, 2010.

[36] Steven Bird, Ewan Klein, and Edward Loper. Natural language processing with Python. O'Reilly, Sebastopol, California, June 2009.

Annotation: The canonical book for the NLTK package for natural language processing in the Python programming language. Corpora, part-of-speech tagging and machine learning classification are among the topics covered.

[37] Tom De Smedt and Walter Daelemans. Pattern for Python. *Journal of Machine Learning Research*, 13:2063–2067, 2012.

Annotation: Describes the Pattern module written in the Python programming language for data, web, text and network mining.

[38] Brendan O'Connor, Michel Krieger, and David Ahn. TweetMotif: exploratory search and topic summarization for Twitter. In *Proceedings of the International AAAI Conference on Weblogs and Social Media*, 2010.

[39] Kevin Gimpel, Nathan Schneider, Brendan O'Connor, Dipanjan Das, Daniel Mills, Jacob Eisenstein, Michael Heilman, Dani Yogatama, Jeffrey Flanigan, and Noah A. Smith. Part-of-speech tagging for Twitter: annotation, features, and experiments. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies: Short Papers*, volume 2, pages 42–47. Association for Computational Linguistics, 2011.

[40] Finn Årup Nielsen. A new ANEW: evaluation of a word list for sentiment analysis in microblogs. In Matthew Rowe, Milan Stankovic, Aba-Sah Dadzie, and Mariann Hardey, editors, *Proceedings of the ESWC2011 Workshop on 'Making Sense of Microposts': Big things come in small packages*, volume 718 of *CEUR Workshop Proceedings*, pages 93–98, May 2011.

Annotation: Initial description and evaluation of the AFINN word list for sentiment analysis.

[41] Aric Hagberg, Pieter Swart, and Daniel S. Chult. Exploring network structure, dynamics, and function using NetworkX. In Gäel Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11–16, 2008.

[42] Peter H. Bennett, Thomas A. Burch, and Max Miller. Diabetes mellitus in American (Pima) Indians. *Lancet*, 2(7716):125–128, July 1971.

[43] W. C. Knowler, P. H. Bennett, R. F. Hamman, and M. Miller. Diabetes incidence and prevalence in Pima Indians: a 19-fold greater incidence than in Rochester, Minnesota. *American Journal of Epidemiology*, 108(6):497–495, December 1978.

[44] Jack W. Smith, J. E. Everhart, W. C. Dickson, W. C. Knowler, and R. S. Johannes. Using the ADAP learning algorithm to forecast the onset of diabetes mellitus. In *Proceedings of the Annual Symposium on Computer Application in Medical Care*, pages 261–265. American Medical Informatics Association, 1988.

[45] Chris Anderson. The long tail. *Wired*, 12(10), October 2004.

[46] Johan Galtung and Mari Holmboe Ruge. The structure of foreign news: the presentation of the Congo, Cuba and Cyprus crises in four Norwegian newspapers. *Journal of Peace Research*, 2(1):64–91, 1965.

[47] Bongwon Suh, Lichan Hong, Peter Pirolli, and Ed H. Chi. Want to be retweeted? large scale analytics on factors impacting retweet in Twitter network. In *2010 IEEE International Conference on Social Computing (SocialCom10)*. IEEE, 2010.

[48] Lars Kai Hansen, Adam Arvidsson, Finn Årup Nielsen, Elanor Colleoni, and Michael Etter. Good friends, bad news — affect and virality in Twitter. In James J. Park, Laurence T. Yang, and Changhoon Lee, editors, *Future Information Technology*, volume 185 of *Communications in Computer and Information Science*, pages 34–43, Berlin, 2011. Springer.

[49] Alon Halevy, Peter Norvig, and Fernando Pereira. The unreasonable effectiveness of data. *IEEE Intelligent Systems*, pages 8–12, March/April 2009.

[50] Frederick Jelinek. Some of my best friends are linguists. Talk at LREC 2004, May 2008.

[51] Chris Anderson. The end of theory: The data deluge makes the scientific method obsolete. *Wired*, June 2008.

Index

", 10
'', 10
( ), 50
(), 15
(,), 10
*, 14, 33, 50
**, 15
**kwargs, 14
*args, 14
+, 15, 50
+=, 15
., 50
.py, 17
/, 7, 15
//, 15
<, 15
<=, 15
==, 15
?, 50
[ ], 50
[^], 50
[], 10, 15
[] =, 15
$, 50
%timeit, 27
&, 9, 15
^, 9, 15, 50
__abs__, 15
__add__, 15
__all__, 28
__and__, 15
__builtin__, 4
__builtins__, 4
__call__, 15, 17
__class__, 15
__contains__, 15
__del__, 15
__dict__, 12
__div__, 15
__doc__, 15
__eq__, 15
__file__, 4, 17
__floordiv__, 15
__format__, 15
__future__, 7, 18
__future__.division, 7
__future__.print_function, 7
__getitem__, 15
__iadd__, 15
__init__, 15
__init__.py, 18, 24
__le__, 15
__len__, 15
__lt__, 15
__main__, 4
__main__, 29
__main__.py, 29
__pow__, 15
__setitem__, 15
__str__, 15
__xor__, 15
\D, 50
\S, 50
\W, 50
\b, 50
\d, 50
\s, 50
\w, 50
{:}, 10
{}, 10
{m, n}, 50

abs, 4, 5
abs, 15
absolute, 5
accuracy, 69
ActivePython, 6
addict, 12
addition, 15
agg, 44
Anaconda, 6, 8
and, 9

animation, 35, 36
append, 9
argparse, 29
args, 14
array, 10
as, 17
assert, 23
autocompletion, 7
autoregression, 34
Axes3D, 34

b'', 10
barplot, 37
basestring, 7
BeautifulSoup, 50–52
big data, 77
bitwise, 9
bob, 48
Bohrium, 3
bokeh, 39
bool, 9, 10, 15
bool_, 10
brent, 44
Brython, 3
brython.js, 3
built-in, 4
bytearray, 10
bytes, 10

Cairo, 33
Canopy, 6
central difference, 47
CheeseShop, 5
CherryPy, 37, 39
chi2, 44
Chromium Compact Language Detector, 54
class, 3, 15
classifier, 76
cld, 54
clonedigger, 28, 31
cluster, 44
Code Golf, 8
Codecademy, 8
collaborative programming, 7
collections, 13
collections.Counter, 13
collections.deque, 13, 34
command-line interface, 29, 30
comorbidity, 55
complex, 10, 11
complex128, 10
complex256, 10
complex64, 10
complex_, 10
concepts, 15
confidence interval, 68
ConfigParser, 19
configparser, 19, 20
Continuum Analytics, 6
ConvexHull, 44
convolve, 44
corr, 67
correlation, 67, 68
coverage, 22, 24, 25, 32
coverage, 24, 25, 32
Coverity, 2
cPickle, 19, 20
cProfile, 26
cStringIO, 19
csv, 20, 66
csvkit, 20, 66
cx_Freeze, 3
Cython, 3, 53

D3, 39
data frame, 39, 40
data type
    testing, 22
DataFrame, 10, 42, 67
DataFrameGroupBy, 44, 67
datasets, 45
date, 12, 13
datetime, 12
datetime.date, 12
datetime.datetime, 12, 13
datetime.datetime.isoformat, 13
dateutil, 13
db.db.Table, 72
db.DemoDB, 71
db.py, 21, 71
db.query, 72, 74
Debian, 5
debugging, 30, 31
Decimal, 10
decimal, 10
decimal.Decimal, 10
decision_function, 48
decorator, 37, 70
def, 3, 14, 15
del, 15
describe, 43
diabetes, 65
dict, 10, 11, 15, 77
dict.items, 12

dict.keys, 12
dict.values, 12
differentiation, 47
dir, 9, 15
distance.cityblock, 44
division, 15
  integer, 15
Django, 5
docopt, 21, 29
docstring, 11, 59
doctest, 23, 59
doctest, 21, 23
doctesting, 22
documentation, 21, 22
DOM, 3
don't repeat yourself, 28
DRY, 28
dump, 20
dumps, 20

ego-graph, 56
Emacs, 7
encode, 37
Enthought, 6
enum, 13
enum.Enum, 13
enum34, 13
Eric, 7
Excel, 55
except, 37
exception, 31
exception, 31
exit status, 29, 30
exploratory programming, 1

False, 9
fft, 44
fftfreq, 44
fftpack, 44
fillna, 74
fit, 45, 48, 49
fit_predict, 48
fit_transform, 48
flake8, 18
flake8, 27, 30
flake8-docstrings, 27
flake8-double-quotes, 27
flake8-import-order, 18
flake8-quotes, 27
Flask, 7, 36
float, 4, 10, 11, 15, 22
float128, 10
float16, 10
float32, 10
float64, 10
float_, 10
fmin, 44
fmin_cg, 44
for, 3
formal concept analysis, 15, 16
Fourier transform, 44, 45
FriendFeed, 21
from, 17
frozenset, 10, 11
function, 14, 15, 17

gc, 4, 27
generator, 77
gensim, 48
gentlemen, 28
get_params, 48
getopt, 29
getvalue, 37
ggplot, 33
ggplot, 34
GIL, 3
Global Interpreter Lock, 3
globbing, 19
GLSL, 34
glue, 39
Gobby, 7
Google App Engine, 7
Google Scholar, 48
graphviz, 15
groupby, 67
guppy, 27
gzip, 78
gzip.open, 78

happyfuntokenizing.py, 53
hashable, 11
help, 9, 15
hierarchy.dendrogram, 44
HTML, 36, 39, 50, 51

icol, 40
IDE, *see* integrated development environment
if, 3
ifft, 44
iget_value, 40
ijson, 20, 77
ijson.items, 77, 78
iloc, 40
import, 17–20
  globbing, 19

relative, 18
import, 4, 17, 27, 28
in, 15
indentation, 9
Index, 40
indexing, 40, 42
infinite impulse response, 34
information diffusion, 75
int, 10, 11, 15
int16, 10
int32, 10
int64, 10
Int64Index, 40
int8, 10
int_, 10
integrated development environment, 6, 7
inverse_transform, 48
io, 18, 19, 37
IPython, 27, 47
IPython Notebook, 6
irow, 40
ISO 8601, 13
itertools, 4
ix, 40, 41

JavaScript, 39
Javascript, 2, 3, 20
jedi, 7
Jinja2, 7
JSON, 20, 21, 77
json, 20, 21, 78
Julia, 2, 3
Jupyter, 2
Jupyter Notebook, 6, 37

Keleshev, Vladimir, 22
kendalltau, 44
keyboard shortcut, 31
keyword, 3
keyword.kwlist, 3
Kivy, 2
kwargs, 14

lambda, 3, 14
lambdas, 14
lambdify, 47
langid, 54
language detection, 54
lazy, 56
lazy attribute, 56
len, 15
len(), 15
length, 15
linewidth, 14
list, 9, 10, 15, 21
listdir, 17
load, 19, 20
loads, 20
loc, 40
logger.Logger, 31
logging, 31
logging, 31
logging.basicConfig, 31
logging.DEBUG, 31
logging.warning, 31
long, 6, 11
long line, 27
lxml, 5, 20, 50, 51

machine learning, 47–49
magic function, 27
main, 29
Maple, 2
Markdown, 6
math, 4, 11, 27
math.sin, 27
math.sqrt, 11
Mathematica, 2, 6
Matplotlib, 19, 33–36, 39, 55
matplotlib, 19, 33, 34, 36
matplotlib.animation, 35
matplotlib.animation.FuncAnimation, 35
matplotlib.pylab, 19
matplotlib.pyplot.plot, 14, 34
matplotlib.pyplot.show, 56
matrix, 10
Mayavi, 34
mayavi, 6
mayavi.mlab, 34
mccabe, 27
mean, 67
MediaWiki, 50
meliae, 27
memory profiling, 27
memory_profiler, 27
Micro Python, 5
Microsoft Excel, 55
Mlpy, 48
module, 17, 19, 20
mpl_toolkits.mplot3d, 34
mpld3, 33, 36, 39
mpld3.fig_to_html, 39
mpltools, 33, 34
multiprocessing, 3
mwparserfromhell, 50

MySQL, 7
MySQLdb, 21

NaN, 74
nanmean, 44
network mining, 55, 56
NetworkX, 55, 72, 74
networkx, 6, 55
networkx.DiGraph, 55
networkx.draw, 56
networkx.ego_graph, 56
NLTK, 52, 53
nltk, 6, 20, 48, 75
nltk.app, 53
nltk.app.wordnet, 53
nltk.book, 53
nltk.classify, 48
nltk.classify.NaiveBayesClassifier, 76
nltk.corpus, 53, 75
nltk.download, 53, 75
nltk.sent_tokenize, 53
nltk.text, 53
nltk.tokenize, 53
nltk.tokenize.sent_tokenize, 53
nltk.word_tokenize, 53
nose, 25
nosetests, 25
NoSQL, 21
not, 9
np.sin, 18
ntlk, 75
numarray, 33
numba, 3
numeric, 8, 33
numexpr, 6
Numpy, 9, 13, 19, 22, 23, 33, 40, 44, 59
numpy, 5, 6, 8, 10, 11, 19, 27, 33, 44
numpy.__config__.show, 33
numpy.array, 33, 40, 44, 65
numpy.array.tolist, 21
numpy.eye, 33
numpy.fft.fft, 44
numpy.float64, 21, 69
numpy.int, 21
numpy.lib.npyio, 19
numpy.linalg, 44
numpy.linalg.linalg.eig, 44
numpy.matrix, 33
numpy.matrix.A, 33
numpy.matrix.H, 33
numpy.matrix.I, 33
numpy.nan, 23
numpy.ones, 33
numpy.random.random, 33
numpy.sin, 27
numpy.sqrt, 11
numpy.std, 44
numpy.sum, 19
numpy.testing, 58
numpy.zeros, 33
NVD3, 39
nx, 55

object-relational mapping, 21
Online Python Tutor, 8
open, 4, 6, 9
optimization, 48
optimize, 44
optparse, 29
or, 9
Orange, 48
orange, 48
os, 17
os.__file__, 4
os.listdir, 17

pair programming, 7
Pandas, 9, 33, 39, 40, 42–44, 72, 74
Pandas, 27
pandas, 6, 21, 34, 39, 40, 45, 46
pandas.DataFrame, 12, 21, 40, 42, 46, 72, 74
pandas.DataFrame.to_records, 40
pandas.io.sql.read_frame, 21
pandas.Panel, 40
pandas.Panel4D, 40
pandas.read_csv, 66
pandas.read_excel, 55
pandas.Series, 40, 41, 46, 69
pandas.Series.kurtosis, 43
pandas.Series.mean, 43
pandas.Series.quantile, 43
pandas.Series.std, 43
pandas.Series.values, 44
Panel, 10
Panel4D, 10
part-of-speech tagging, 53, 54
pattern, 50, 54, 55
pd, 40
pdb, 30, 31
peewee, 21
pep257, 22
pep8, 27, 30
Perl, 1
persistency, 20

pickle, 20
pickle, 19, 20, 76
PIL, 33
Pillox, 33
Pima data set, 65–67, 69, 70
Pima Indians, 65
pip, 5
Plotly, 8, 33, 38
plotly, 38
plotly.plotly.plot, 38
plotly.plotly.plot_mpl, 39
plotting, 33, 34, 36–39
    3D, 34
    network, 55, 56
    web, 36–39
plt, 34
plt.plot, 14
pozer, 27
pprint, 30, 31
pprint.pprint, 30
predict, 48, 49
predict_proba, 48
pretty printing, 30
prettyplotlib, 34
print, 6, 7, 30, 31, 77
private, 28
profile, 26
profiling, 25, 27
PSL, *see* Python Standard Library
pstats, 26
psutil, 27
psycopg2, 21
public, 28
py.test, 23–25, 31, 32
py.text, 25
py2exe, 3
PyBrain, 48
pybrain, 48
PyCharm, 7
pychecker, 30
pydoc, 22
pydocstyle, 22, 27, 28, 32
pyFFTW, 45
pyfftw.interfaces.scipy_fftpack, 45
pyflakes, 27, 30
pygal, 36
Pygame, 34
pygame, 33
PyInstaller, 3
pylab, 19, 44
pylab.eig, 44
pylint, 27, 30
pymongo, 21
PyMVPA, 48
pynsist, 3
PyPI, 5
PyPR, 48
pypr, 48
pypy, 3, 5, 26
pyqtgraph, 33, 34
pyringe, 31
PySizer, 27
PySonar, 30
Pyston, 5
python, 5, 26
Python Egg, 3
Python Package Index, 5
Python Standard Library, 4, 20, 25, 29, 31, 50, 66
Python(x,y), 6
python-coverage, 24
python3, 6
Pythonanywhere, 7
PYTHONPATH, 5

quantile, 43
quote, 11

R, 2, 39
random walk, 34
range, 6
re, 13, 50–52
re.IGNORECASE, 13
re.UNICODE, 51
re.VERBOSE, 51
read-eval-print loop, 1
record array, 40
regular expression, 50
REPL, 1
request.get, 51
requests, 5, 51, 55
return, 3, 14
ring buffer, 13, 34
rounding, 15
Rudel, 7
Runnable, 8

sandboxing, 4
sb, 34
schema, 29
scikit-learn, 49
SciPy, 33, 44, 66
scipy, 5, 6, 11, 34, 44, 45
scipy.fftpack, 44, 45
scipy.fftpack.fft, 44, 45
scipy.linalg, 44, 48, 56

scipy.linalg.eig, 44
scipy.linalg.eigh, 44
scipy.linalg.lstsq, 48
scipy.linalg.lu_factor, 44
scipy.linalg.pinv, 48
scipy.linalg.solve, 48
scipy.linalg.svd, 44
scipy.loadtxt, 66
scipy.misc.derivative, 47
scipy.optimize, 48
scipy.optimize.minimize, 48
scipy.signal, 45
score, 48
scripttest, 29
seaborn, 33
seaborn, 34, 67
seaborn.corrplot, 34, 68
seaborn.regplot, 34
sentiment analysis, 54, 55
Series, 10
set, 10, 15
set_params, 48
setup.py, 3, 5, 25
setuptools, 3
simplegui, 8
simplejson, 20
simplemap, 8
simpleplot, 8
sin, 18, 27
singular value decomposition, 44, 56, 57
six, 7
six.string_types, 7
sklearn, 48, 49
sklearn.decomposition.NMF, 49
sklearn.decomposition.PCA, 49
sklearn.lda.LDA, 49
sklearn.metrics.accuracy_score, 69
sklearn.neighbors.KNeighborsClassifier, 49
sklearn.svm.SVC, 49
Skulpt, 8
slice, 10, 42
sns, 34
spaCy, 50, 53
spatial, 44
Sphinx, 22
spreadsheet, 55
Spyder, 6, 7, 26, 27, 31
SQL, 7, 21
sqlalchemy, 21
sqlite3, 21
sqlobject, 21
standard deviation, 43
Stanford CoreNLP, 50
static method, 70
stats, 44
statsmodels, 34, 39, 45, 46, 48, 67
statsmodels.formula.api, 46
statsmodels.graphics.correlation.plot_corr, 67
std, 43, 44
storm, 21
str, 4, 7, 10, 11, 13, 15, 17, 50
str.capitalize, 50
str.isdigit, 50
str.isspace, 50
str.lower, 50
str.replace, 50
str.rsplit, 50
str.split, 50
str.splitlines, 50
str.upper, 50
StringIO, 19, 37
structured array, 40
style, 27, 28
subprocess, 3
sum, 4, 19
summary, 45
Sympy, 47
sympy, 47
sys, 4
sys.builtin_module_names, 4
sys.exit, 29, 30
sys.path, 5, 18
SystemExit, 30

t-values, 69
Test, 24
test coverage, 24
test discovery, 23, 24
test layout, 23, 24
test_, 24
testing, 22
text discovery, 22
text mining, 50, 55
textblob, 50, 54, 55
textblob.TextBlob, 54
threading, 3
time, 25, 26
time.clock, 25
time.perf_counter, 25
time.process_time, 25
timeit, 25–27
timeit.timeit, 25, 26
title, 50

tokenization, 51, 53
`tox`, 25, 28
`tox.ini`, 25, 28
`TrackId`, 74
`transform`, 48
Trifacta, 37
`True`, 9, 10
`try`, 37
`tuple`, 10, 11, 15
Twitter, 75
  part-of-speech tagging, 53
  tokenization, 53
`twokenize.py`, 53
`TypeError`, 9, 21, 30, 41

Ubuntu, 5
`uint8`, 10
`ujson`, 20
underscore, 28
`unicode`, 7, 50

valgrind, 27
Vega, 37
`vincent`, 33, 36, 37
Vispy, 34
`vispy`, 39
`vispy.gloo`, 34
`vispy.mpl_plot`, 34
volume rendering, 34
`Voronoi`, 44
`vq.keans`, 44
`vq.vq`, 44
VTK, 34

`warnings`, 31
`warnings.warn`, 31
`welch`, 45
Wikidata, 77, 78
Wikipedia, 50
Windows, 5, 6
Wing IDE, 7
Winpdb, 30
`winpdb`, 31
WinPython, 6
WordNet, 55

XML, 20, 51
XPath, 51, 52
`xrange`, 6

`yield`, 3

zero-one-sum, 23
`ZeroDivisionError`, 23
`zip`, 4