Introduction to Python for Econometrics, Statistics and Data Analysis

Kevin Sheppard
University of Oxford

Monday 25th November, 2013

Change Log

Version 2.02

- Changed the Anaconda install to use both create and install, which shows how to install additional packages.

- Fixed some missing packages in the direct install.

- Changed the configuration of IPython to reflect best practices.

- Added subsection covering IPython profiles.

Version 2.01

- Updated Anaconda to 1.8 and added some additional packages to the installation for Spyder.

- Small section about Spyder as a good starting IDE.

Notes to the 2nd Edition

This edition includes the following changes from the first edition (March 2012):

- The preferred installation method is now Continuum Analytics' Anaconda. Anaconda is a complete scientific stack and is available for all major platforms.

- New chapter on pandas. pandas provides a simple but powerful tool to manage data and perform basic analysis. It also greatly simplifies importing and exporting data.

- New chapter on advanced selection of elements from an array.

- Numba provides just-in-time compilation for numeric Python code which often produces large performance gains when pure NumPy solutions are not available (e.g. looping code).

- Dictionary, set and tuple comprehensions

- Numerous typos

- All code has been verified working against Anaconda 1.7.0.

Contents

Chapter 1

Introduction

1.1 Background

These notes are designed for someone new to statistical computing wishing to develop a set of skills necessary to perform original research using Python. They should also be useful for students, researchers or practitioners who require a versatile platform for econometrics, statistics or general numerical analysis (e.g. numeric solutions to economic models or model simulation).

Python is a popular general purpose programming language which is well suited to a wide range of problems.[1] Recent developments have extended Python's range of applicability to econometrics, statistics and general numerical analysis. Python – with the right set of add-ons – is comparable to domain-specific languages such as MATLAB and R. If you are wondering whether you should bother with Python (or another language), a very incomplete list of considerations includes:

You might want to consider R if:

- You want to apply statistical methods. The statistics library of R is second to none, and R is clearly at the forefront in new statistical algorithm development – meaning you are most likely to find that new(ish) procedure in R.

- Performance is of secondary importance.

- Free is important.

You might want to consider MATLAB if:

- Commercial support, and a clean channel to report issues, is important.

- Documentation and organization of modules is more important than raw routine availability.

- Performance is more important than scope of available packages. MATLAB has optimizations, such as Just-in-Time (JIT) compilation of loops, which is not automatically available in most other packages.

Having read the reasons to choose another package, you may wonder why you should consider Python.

---

[1] According to the ranking on http://www.tiobe.com/, Python is the 8th most popular language. http://langpop.corger.nl/ ranks Python as 5th or 6th, and on http://langpop.com/, Python is 6th.

- You need a language which can act as an end-to-end solution so that everything from accessing web-based services and database servers, data management and processing and statistical computation can be accomplished in a single language. Python can even be used to write apps for desktop-class operating systems with graphical user interfaces as well as tablets and phones (IOS and Android).

- Data handling and manipulation – especially cleaning and reformatting – is an important concern. Data set construction is substantially more capable at than either R or MATLAB.

- Performance is a concern, but not at the top of the list.[2]

- Free is an important consideration – Python can be freely deployed, even to 100s of servers in a compute cluster or in the cloud (e.g. Amazon Web Services or Azure).

- Knowledge of Python, as a general purpose language, is complementary to R/MATLAB/Ox/GAUSS/Julia.

1.2 Conventions

These notes will follow two conventions.

1. Code blocks will be used throughout.

```python
"""A docstring
"""

# Comments appear in a different color

# Reserved keywords are highlighted
and as assert break class continue def del elif else
except exec finally for from global if import in is
lambda not or pass print raise return try while with yield

# Common functions and classes are highlighted in a
# different color. Note that these are not reserved,
# and can be used although best practice would be
# to avoid them if possible
array matrix xrange list True False None

# Long lines are indented
some_text = 'This is a very, very, very, very, very, very, very, very, very, very, very
    , very long line.'
```

2. When a code block contains >>>, this indicates that the command is running an interactive IPython session. Output will often appear after the console command, and will *not* be preceded by a command indicator.

---

[2]Python performance can be made arbitrarily close to C using a variety of methods, including Numba (pure python), Cython (C/Python creole language) or directly calling C code. Moreover, recent advances have substantially closed the gap with respect to other Just-in-Time compiled languages such as MATLAB.

```
>>> x = 1.0
>>> x + 2
3.0
```

If the code block does not contain the console session indicator, the code contained in the block is intended to be executed in a standalone Python file.

```python
from __future__ import print_function
import numpy as np

x = np.array([1,2,3,4])
y = np.sum(x)
print(x)
print(y)
```

1.3   Important Components of the Python Scientific Stack

1.3.1   Python

Python 2.7.6 (or later, but in the Python 2.7.x family) is required. This provides the core Python interpreter.

1.3.2   NumPy

NumPy provides a set of array and matrix data types which are essential for statistics, econometrics and data analysis.

1.3.3   SciPy

SciPy contains a large number of routines needed for analysis of data. The most important include a wide range of random number generators, linear algebra routines and optimizers. SciPy depends on NumPy.

1.3.4   IPython

IPython provides an interactive Python environment which enhances productivity when developing code or performing interactive data analysis.

1.3.5   matplotlib

matplotlib provides a plotting environment for 2D plots, with limited support for 3D plotting.

1.3.6   pandas

pandas provides high-performance data structures.

1.3.7   Performance Modules

A number of modules are available to help with performance. These include Cython and Numba. Cython is a Python module which facilitates using a simple Python-derived creole to write functions that can be compiled to native (C code) Python extensions. Numba uses a method of just-in-time compilation to translate a subset of Python to native code using Low-Level Virtual Machine (LLVM).

1.4   Setup

The recommended method to install the Python scientific stack is to use Continuum Analytics' Anaconda. Instructions are also provided for directly installing Python and the required modules if it isn't possible to install Anaconda.

1.4.1   Continuum Analytics' Anaconda

Anaconda, a free product of Continuum Analytics (`www.continuum.io`), is a virtually complete scientific stack for Python. It includes both the core Python interpreter and standard libraries as well as most modules required for data analysis. Anaconda is free to use and modules for accelerating the performance of linear algebra on Intel processors using the Math Kernel Library (MKL) are available (free to academic users and for a small cost to non-academic users). Continuum Analytics also provides other high-performance modules for reading large data files or using the GPU to further accelerate performance for an additional, modest charge. Most importantly, installation is extraordinarily easy on Windows, Linux and OSX. Anaconda is also simple to update to the latest version using

```
conda update conda
conda update anaconda
```

Windows

Installation on Windows requires downloading the installer and running. These instructions use ANA-CONDA to indicate the Anaconda installation directory (e.g. the default is C:\Anaconda). Once the setup has completed, open a command prompt (cmd.exe) and run

```
cd ANACONDA
conda update conda
conda update anaconda
conda create -n econometrics numpy scipy ipython-qtconsole ipython-notebook spyder pandas
    statsmodels matplotlib
conda install -n econometrics cython distribute lxml nose numba numexpr pep8 pip psutil
    pyflakes pytables pywin32 rope openpyxl sphinx statsmodels xlrd xlwt
conda install -n econometrics mkl
```

which will first ensure that Anaconda is up-to-date and then create a virtual environment named econometrics. The virtual environment provides a set of components which will not change even if anaconda is updated. Using a virtual environment is a best practice and is important since component updates can lead to errors in otherwise working programs due to backward incompatible changes in a module. The long list of modules in the `conda create` command includes the core modules. The `first conda install`

contains the remaining packages, and is shown as an example of how to add packages to a virtual environment after it has been created. The second `conda install` installs the Intel Math Kernel library linked-modules which provide large performance gains in Intel systems – this package requires a license from Continuum which is is free to academic users (and low cost otherwise). I recommend acquiring a license as the performance gains are substantial, even on dual core machines. If you will not be purchasing a license, this line should be omitted. It is also possible to install all available packages using the command `conda create -n econometrics anaconda`.

The econometrics environment must be activated before use. This is accomplished by running

```
ANACONDA\Scripts\activate.bat econometrics
```

from the command prompt, which prepends [econometrics] to the prompt as an indication that virtual environment is active. Activate the econometrics environment and then run

```
pip install pylint beautifulsoup4 html5lib
```

which installs one package not directly available in Anaconda.

Linux and OSX

Installation on Linux requires executing

```
bash Anaconda-x.y.z-Linux-ISA.sh
```

where `x.y.z` will depend on the version being installed and ISA will be either x86 or more likely x86_64. The OSX installer is available either in a GUI installed (pkg format) or as a bash installer which is installed in an identical manner to the Linux installation. After installation completes, change to the folder where Anaconda installed (written here as ANACONDA, default ~/anaconda) and execute

```
cd ANACONDA
cd bin
./conda update conda
./conda update anaconda
./conda create -n econometrics numpy scipy ipython-qtconsole ipython-notebook spyder pandas
    statsmodels matplotlib
./conda install -n econometrics cython distribute lxml nose numba numexpr pep8 pip psutil
    pyflakes pytables rope openpyxl sphinx statsmodels xlrd xlwt
./conda install -n econometrics mkl
```

which will first ensure that Anaconda is up-to-date and then create a virtual environment named econometrics with the required packages. `conda create` creates the environment and `conda install` installs additional packages to the existing environment. The second invocation of `conda install` is used to install the Intel Math Kernel library-linked modules, which provide substantial performance improvements – this package requires a license which is free to academic users and low cost to others. If acquiring a license is not possible, omit this line. `conda install` can be used later to install other packages that may be of interest. To activate the newly created environment, run

```
source ANACONDA/bin/activate econometrics
```

and then run the command

```
pip install pylint beautifulsoup4 html5lib
```

to install one package not included in Anaconda.

1.4.2 Installation without Anaconda

Anaconda greatly simplifies installing the scientific Python stack. However, there may be situations where installing Anaconda is not possible, and so (substantially more complicated) instructions are included for both Windows and Linux.

Windows

The list of required windows binary packages, along with the version and Windows installation file, required for these notes include:

Package	Version	File name
Python	2.7.5	python-2.7.5.amd64
Setuptools	1.3.2	setuptools-1.3.2.win-amd64-py2.7
Pip	1.4.1	pip-1.4.1.win-amd64-py2.7
Virtualenv	1.10.1	virtualenv-1.10.1.win-amd64-py2.7
pywin32	218.4	pywin32-218.4.win-amd64-py2.7
Jinja2	2.7.1	Jinja2-2.7.1.win-amd64-py2.7.exe
Tornado	3.1.1	tornado-3.1.1.win-amd64-py2.7.exe
PyCairo	1.10.0	pycairo-1.10.0.win-amd64-py2.7
PyZMQ	14.0.0	pyzmq-13.1.0.win-amd64-py2.7
PyQt	4.9.6-1	PyQt-Py2.7-x64-gpl-4.9.6-1
NumPy	1.7.1	numpy-MKL-1.7.1.win-amd64-py2.7
SciPy	0.13.0	scipy-0.13.0.win-amd64-py2.7
MatplotLib	1.3.1	matplotlib-1.3.0.win-amd64-py2.7
pandas	0.12.0	pandas-0.12.0.win-amd64-py2.7
IPython	1.1.0	ipython-1.1.0.win-amd64-py2.7
statsmodels	0.5.0	statsmodels-0.5.0.win-amd64-py2.7
PyTables	3.0.0	tables-3.0.0.win-amd64-py2.7
lxml	3.2.4	lxml-3.2.4.win-amd64-py2.7
psutil	1.1.3	psutil-1.1.3.win-amd64-py2.7

These remaining packages are optional and are only discussed in the final chapters related to performance.

Package	Version	File name
Performance		
Cython		
Cython	0.19.2	Cython-0.19.2.win-amd64-py2.7
Numba		
LLVMPy	0.12.1	llvmpy-0.12.1.win-amd64-py2.7
LLVMMath	0.1.2	llvmmath-0.1.2.win-amd64-py2.7
Numba	0.11.1	numba-0.11.1.win-amd64-py2.7
pandas (Optional)		
Bottleneck	0.7.0	Bottleneck-0.7.0.win-amd64-py2.7
NumExpr	2.2.2	numexpr-2.2.1.win-amd64-py2.7

Begin by installing Python, setuptools, pip and virtualenv. After these four packages are installed, open an elevated command prompt (cmd.exe with administrator privileges) and initialized the virtual environment using the command:

```
cd C:\Dropbox
virtualenv econometrics
```

I prefer to use my Dropbox as the location for virtual environments and have named the virtual environment econometrics. The virtual environment can be located anywhere (although best practice is to use a path without spaces) and can have a different name. Throughout the remainder of this section, VIRTUALENV will refer to the complete directory containing the virtual environment (e.g. C:\Dropbox\econometrics).

Once the virtual environment setup is complete, run

```
cd VIRTUALENV\Scripts
activate.bat
pip install beautifulsoup4 html5lib meta nose openpyxl patsy pep8 pyflakes pygments pylint
    pylint pyparsing pyreadline python-dateutil pytz==2013d rope sphinx spyder wsgiref xlrd
    xlwt
```

which activates the virtual environment and installs some additional required packages. Finally, before installing the remaining packages, it is necessary to register the virtual environment as the default Python environment by running the script register_python.py[3], which is available on the website. Once the correct version of Python is registered, install the remaining packages in order, including any optional packages. Finally, run one final command in the prompt.

```
xcopy c:\Python27\tcl VIRTUALENV\tcl /S /E /I
```

Linux (Ubuntu 12.04 LTS)

To install on Ubuntu 12.04 LTS, begin by updating the system using

---

[3]This file registers the virtual environment as the default python in Windows. To restore the main Python installation (normally C:\Python27) run register_python.py with the main Python interpreter (normally C:\Python27\python.exe) in an elevated command prompt.

```
sudo apt-get update
sudo apt-get upgrade
```

Next, install the system packages required using

```
sudo apt-get install python-pip libzmq-dev python-all-dev build-essential gfortran libatlas-
    base-dev pyqt4-dev-tools libfreetype6-dev libpng12-dev python-qt4 python-qt4-dev python-
    cairo python-cairo-dev hdf5-tools libhdf5-serial-dev texlive-full dvipng pandoc
```

Finally, install virtualenv using

```
sudo pip install virtualenv
```

The next step is to initialize the virtual environment, which is assumed to be in your home directory and named econometrics.

```
cd ~
virtualenv econometrics
```

The virtual environment can be activated using

```
source ~/econometrics/bin/activate
```

Once the virtual environment has been initialized, the remaining packages can be installed using the commands

```
mkdir ~/econometrics/lib/python2.7/site-packages/PyQt4/
mkdir ~/econometrics/lib/python2.7/site-packages/cairo/
cp -r /usr/lib/python2.7/dist-packages/PyQt4/* ~/econometrics/lib/python2.7/site-packages/
    PyQt4/
cp -r /usr/lib/python2.7/dist-packages/cairo/* ~/econometrics/lib/python2.7/site-packages/
    cairo/
cp /usr/lib/python2.7/dist-packages/sip* ~/econometrics/lib/python2.7/site-packages/
pip install Cython
pip install numpy
pip install scipy
pip install matplotlib
pip install ipython[/*all*/]
pip install beautifulsoup4 html5lib lxml openpyxl pytz==2013d xlrd xlwt
pip install patsy bottleneck numexpr
pip install tables
pip install pandas
pip install statsmodels
pip install distribute meta rope pep8 pexpect pylint pyflakes psutil sphinx spyder
```

The three cp lines copy files from the default Python installation which are more difficult to build using pip. Next, if interested in Numba, a package which can be used to enhance the performance of Python, enter the following commands. *Note*: The correct version of llvm might change as llvmpy and numba progress.

```
wget http://llvm.org/releases/3.2/llvm-3.2.src.tar.gz
tar -zxf llvm-3.2.src.tar.gz
cd llvm-3.2.src
./configure --enable-optimizations --prefix=/home/username/llvm
REQUIRES_RTTI=1 make
```

```
make install
cd ..
LLVM_CONFIG_PATH=/home/username/llvm/bin/llvm-config pip install llvmpy
pip install llvmmath
pip install numba
```

1.5 Using Python

Python can be programmed using an interactive session using IPython or by directly executing Python scripts – text files that end in the extension .py – using the Python interpreter.

1.5.1 Python and IPython

Most of this introduction focuses on interactive programming, which has some distinct advantages when learning a language. The standard Python interactive console is very basic and does not support useful features such as tab completion. IPython, and especially the QtConsole version of IPython, transforms the console into a highly productive environment which supports a number of useful features:

- Tab completion - After entering 1 or more characters, pressing the tab button will bring up a list of functions, packages and variables which match the typed text. If the list of matches is large, pressing tab again allows the arrow keys can be used to browse and select a completion.

- "Magic" function which make tasks such as navigating the local file system (using `%cd ~/directory/` or just `cd ~/directory/` assuming that `%automagic` is on) or running other Python programs (using `run program.py`) simple. Entering `%magic` inside and IPython session will produce a detailed description of the available functions. Alternatively, `%lsmagic` produces a succinct list of available magic commands. The most useful magic functions are

    - `cd` - change directory
    - `edit` *filename* - launch an editor to edit *filename*
    - `ls` or `ls` *pattern* - list the contents of a directory
    - `run` filename - run the Python file *filename*
    - `timeit` - time the execution of a piece of code or function

- Integrated help - When using the QtConsole, calling a function provides a view of the top of the help function. For example, entering `mean(` will produce a view of the top 20 lines of its help text.

- Inline figures - The QtConsole can also display figure inline which produces a tidy, self-contained environment. (when using the `--pylab=inline` switch when starting, or when using the configuration option `_c.IPKernelApp.pylab="inline"`).

- The special variable `_` contains the last result in the console, and so the most recent result can be saved to a new variable using the syntax `x = _`.

- Support for profiles, which provide further customization of sessions.

1.5.2 IPython Profiles

IPython supports using profiles which allows for alternative environments (at launch), either in appearance or in terms of packages which have been loaded into the IPython session. Profiles are configured using a set of files located in

%USERPROFILE%\.ipython\

on Windows and and

~/.config/ipython/

on OSX or Linux. There should be one directory in this location, profile_default, that is mostly empty. To configure a profile, activate the environment using

```
ANACONDA\Scripts\activate.bat econometrics
```

on Windows or

```
source ANACONDA/bin/activate econometrics
```

on OSX or Linux, and then run

```
ipython profile create econometrics
```

This will create a directory named profile_econometrics and populate it with 4 files:

File	Purpose
ipython_config.py	General IPython setting for all IPython sessions
ipython_nbconvert_config.py	Settings used by the Notebook converter
ipython_notebook_config.py	Settings specific to IPython Notebook (browser) sessions
ipython_qtconsole_config.py	Settings specific to QtConsole sessions

The two most important are ipython_config and ipython_qtconsole_config. Opening these files in a text editor will reveal a vast array of options, all which are commented out using #. A full discussion of these files would require a chapter or more, and so please refer to the online IPython documentation for details about a specific setting (although most settings have a short comment containing an explanation and possible values).

ipython_config

The settings in this file apply to all IPython sessions using this profile, irrespective of whether they are in the terminal, QtConsole or Notebook. One of the most useful settings is

```
c.InteractiveShellApp.exec_lines
```

which allows commands to be executed each time an IPython session is open. This is useful, for example, to import specific packages commonly used in a project. Another useful configuration options is

```
c.InteractiveShellApp.pylab
```

which can be used to load pylab in the session, and is identical to launching an IPython session using the command line switch --pylab=*backend*. An alternative is to use

```
c.InteractiveShellApp.matplotlib
```

which will only load matplotlib and not the rest of pylab.

ipython_qtconsole_config

The settings in this file only apply to QtConsole sessions, and the most useful affect the appearance of the console. The first two can be used to set the font size (a number) and font family (a string, containing the name of the font).

```
c.IPythonWidget.font_size
```

```
c.IPythonWidget.font_family
```

The next setting sets the model for pylab, which can in particular be set to `"inline"` which is identical to using the command line switch `--pylab=inline` when starting IPython using the QtConsole. This setting is similar to the previous pylab setting, but since this is specific to QtConsole sessions, it will override the general setting (only) in using QtConsole, and so it is possible to use, for example, `"qt4"`, for terminal-based IPython sessions, and to use `"inline"` for QtConsole sessions.

```
c.IPKernelApp.pylab
```

This final setting is identical to the command-line switch `--colors` and can be set to `"linux"` to produce a console with a dark background and light characters.

```
c.ZMQInteractiveShell.colors
```

1.5.3 Configuring IPython

These notes assume that two imports are made when running code in IPython or as stand-alone Python programs. These imports are

```python
from __future__ import print_function, division
```

which imports the future versions of `print` and / (division). Open ipython_config.py in the directory profile_econometrics and set the values

```python
c.InteractiveShellApp.exec_lines=["from __future__ import print_function, division",
                                  "import os",
                                  "os.chdir('c:\\dir\\to\\start\\in')"]
```

and

```python
c.InteractiveShellApp.pylab="qt4"
```

This code does two things. First, it imports two "future" features (which are standard in Python 3.x+), the print function and division, which are useful for numerical programming.

- In Python 2.7, `print` is not a standard function and is used like `print 'string to print'`. Python 3.x changes this behavior to be a standard function call, `print('string to print')`. I prefer the latter since it will make the move to 3.x easier, and find it more coherent with other function in Python.

- In Python 2.7, division of integers always produces an integer so that the result is truncated (i.e. `9/5=1`). In Python 3.x, division of integers does not produce an integer if the integers are not even multiples (i.e. `9/5=1.8`). Additionally, Python 3.x uses the syntax `9//5` to force integer division with truncation (i.e. `11/5=2.2`, while `11//5=2`).

11

Second, pylab will be loaded by default using the qt4 backend.

Changing settings in ipython_qtconsole_config.py is optional, although I recommend using

```
c.IPythonWidget.font_size=11
c.IPythonWidget.font_family="Bitstream Vera Sans Mono"
c.IPKernelApp.pylab="inline"
c.ZMQInteractiveShell.colors="linux"
```

These commands assume that the Bitstream Vera fonts have been locally installed, which are available from http://ftp.gnome.org/pub/GNOME/sources/ttf-bitstream-vera/1.10/.

1.5.4 Launching IPython

OSX and Linux

Starting IPython requires activating the virtual environment and then starting IPython with the correct profile.

```
source ANACONDA/bin/activate econometrics
ipython --profile=econometrics
```

Starting IPython using the QtConsole is virtually identical.

```
source ANACONDA/bin/activate econometrics
ipython qtconsole --profile=econometrics
```

A single line launcher on OSX or Linux can be constructed using

```
bash -c "source ANACONDA/bin/activate econometrics && ipython qtconsole --profile=
    econometrics"
```

This single line launcher can be saved as *filename*.command where *filename* is a meaningful name (e.g. IPython-Terminal) to create a launcher on OSX by entering the command

```
chmod 755 /FULL/PATH/TO/filename.command
```

The same command can to create a Desktop launcher on Ubuntu by running

```
sudo apt-get install --no-install-recommends gnome-panel
gnome-desktop-item-edit ~/Desktop/ --create-new
```

and then using the command as the Command in the dialog that appears.

Note that if Python was directly installed, launching IPython is identical only replacing the Anaconda virtual environment activation line with the activation line for the directly created virtual environment, as in

```
source VIRTUALENV/bin/activate econometrics
ipython qtconsole --profile=econometrics
```

Windows (Anaconda)

Starting IPython requires activating the virtual environment and the starting IPython with the correct profile using cmd.
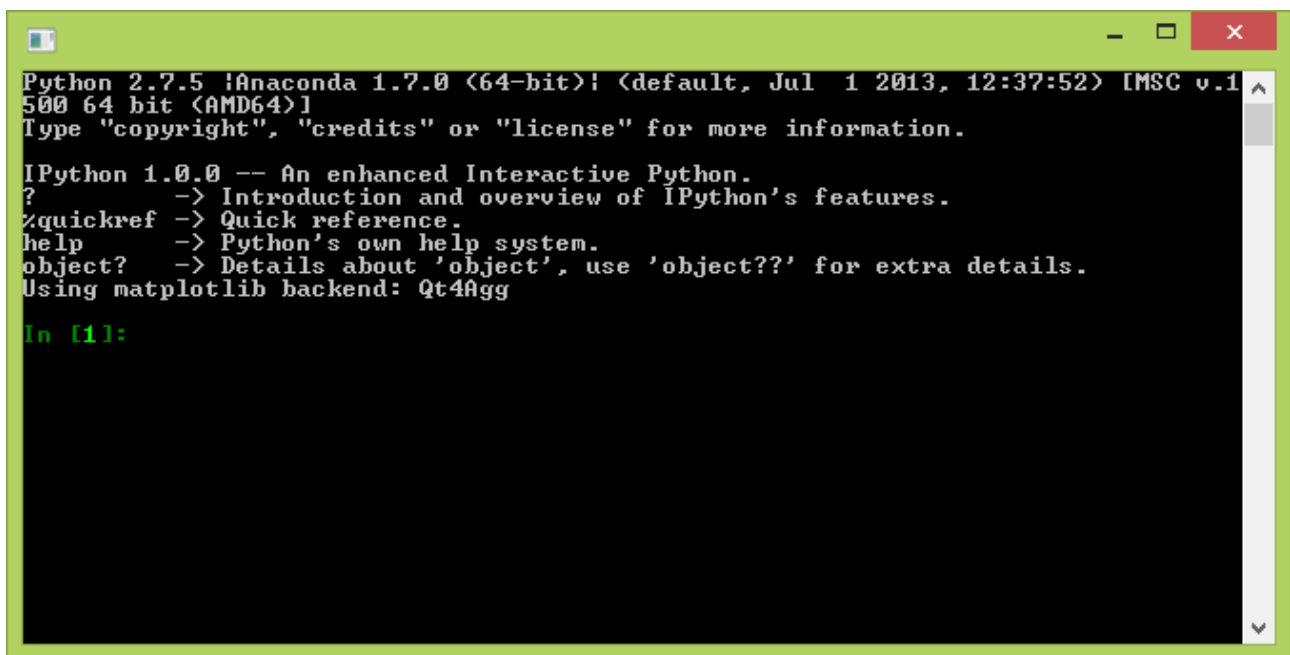
Figure 1.1: IPython running in the standard Windows console (cmd.exe).

```
ANACONDA/Scripts/activate.bat econometrics
ipython --profile=econometrics
```

Starting using the QtConsole is similar.

```
ANACONDA/Scripts/activate.bat econometrics
ipython qtconsole --profile=econometrics
```

Launchers can be created for the both the virtual environment and the IPython interactive Python console. First, open a text editor, enter

```
cmd /k "ANACONDA\Scripts\activate econometrics"
```

and save the file as ANACONDA\envs\econometrics\python-econometrics.bat. The batch file will open a command prompt in the econometrics virtual environment. Right click on the batch file and select Send To, Desktop (Create Shortcut) which will place a shortcut on the desktop. Next, create a launcher to run IPython in the standard Windows cmd.exe console. Open a text editor enter
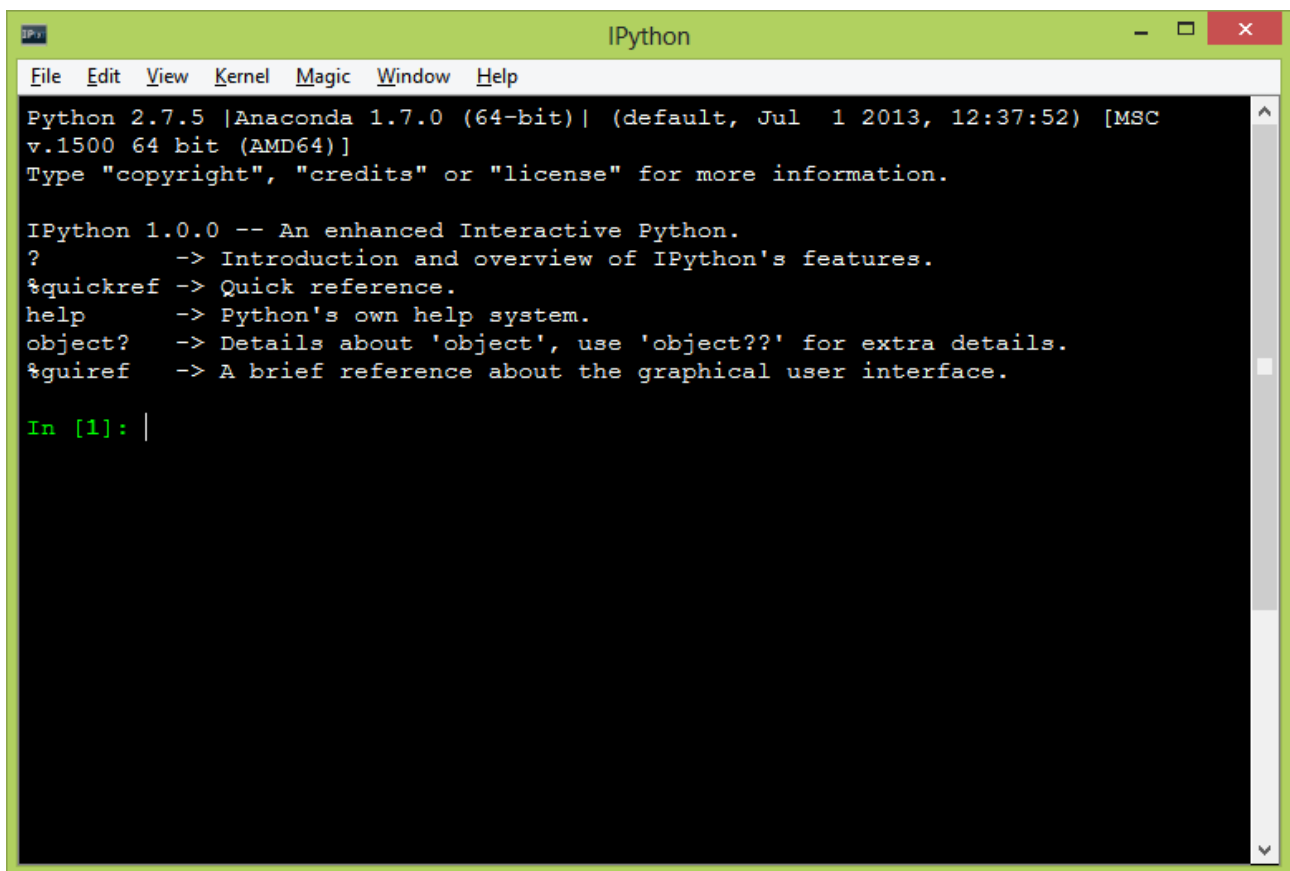
```
cmd "/c ANACONDA\Scripts\activate econometrics && start "" "ipython.exe" --profile=
    econometrics"
```

and save the file as ANACONDA\envs\econometrics\ipython-plain.bat. Finally, right click on ipython-plain.bat select Sent To, Desktop (Create Shortcut). The icon of the shortcut will be generic, and if you want a more meaningful icon, select the properties of the shortcut, and then Change Icon, and navigate to c:\Anaconda\envs\econometrics\Menu\ and select IPython.ico. Opening the batch file should create a window similar to that in figure 1.1.

Launching the QtConsole is similar. Start by entering the following command in a text editor

```
cmd "/c ANACONDA\Scripts\activate econometrics &&  start "" "pythonw" ANACONDA\envs\
    econometrics\Scripts\ipython-script.py qtconsole --profile=econometrics"
```

and then saving the file as ANACONDA\envs\econometrics\ipython-qtconsole.bat. Create a shortcut for this batch file, and change the icon if desired. Opening the batch file should create a window similar to that in figure 1.2 (although the appearance might differ).

13

Figure 1.2: IPython running in a QtConsole session.

Windows (Direct)

If using the direct installation method on Windows, open a text editor, enter the following text

```
cmd "/c VIRTUALENV\Scripts\activate.bat && start "" "python" VIRTUALENV\Scripts\
    ipython-script.py --profile=econometrics"
```

and save the file in VIRTUALENV as ipython.bat. Right-click on ipython.bat and Send To, Desktop (Create Shortcut). The icon of the shortcut will be generic, and if you want a nice icon, select the properties of the shortcut, and then Change Icon, and navigate to VIRTUALENV\Scripts\ and select IPython.ico.

The QtConsole can be configured to run by entering

```
cmd "/c VIRTUALENV\Scripts\activate.bat && start "" "pythonw" VIRTUALENV\Scripts\
    ipython-script.py qtconsole --profile=econometrics"
```

saving the file as VIRTUALENV\ipython-qtconsole.bat and finally right-click and Sent To, Desktop (Create Shortcut). The icon can be changed using the same technique as the basic IPython shell. Launching IPython QtConsole will produce a window similar to that depicted in figure 1.2.

1.5.5 Getting Help

Help is available in IPython sessions using help(*function*). Some functions (and modules) have very long help files. When using IPython, these can be paged using the command ?*function* or *function*? so that the text can be scrolled using page up and down and q to quit. ??*function* or *function*?? can be used to type the entire function including both the docstring and the code.

1.5.6 Running Python programs

While interactive programing is useful for learning a language or quickly developing some simple code, complex projects require the use of complete programs. Programs can be run either using the IPython magic work %run program.py or by directly launching the Python program using the standard interpreter using python program.py. The advantage of using the IPython environment is that the variables used in the program can be inspected after the program run has completed. Directly calling Python will run the program and then terminate, and so it is necessary to output any important results to a file so that they can be viewed later.[4]

To test that you can successfully execute a Python program, input the code in the block below into a text file and save it as firstprogram.py.

```python
# First Python program
from __future__ import print_function, division
import time

print('Welcome to your first Python program.')
raw_input('Press enter to exit the program.')
print('Bye!')
time.sleep(2)
```

---

[4]Programs can also be run in the standard Python interpreter using the command:
`exec(compile(open('filename.py').read(),'filename.py','exec'))`
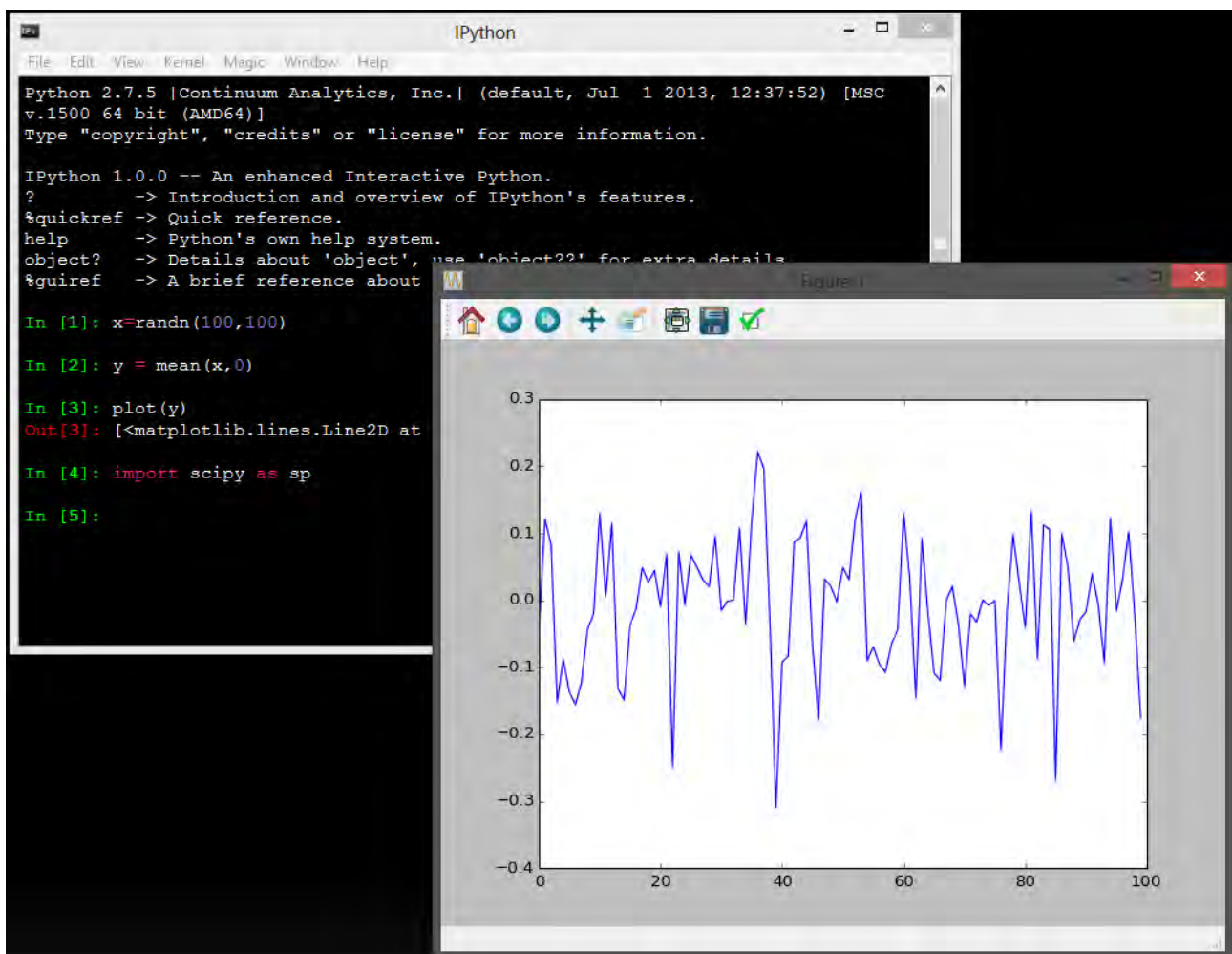
Figure 1.3: A successful test that matplotlib, IPython, NumPy and SciPy were all correctly installed.

Once you have saved this file, open the console, activate the virtual environment, navigate to the directory you saved the file and run `python firstprogram.py`. Finally, run the program in IPython by first launching IPython, and the using `%cd` to change to the location of the program, and finally executing the program using `%run firstprogram.py`.

1.5.7 Testing the Environment

To make sure that you have successfully installed the required components, run IPython using the shortcut previously created on windows, or by running `ipython --pylab` or `ipython qtconsole --pylab` in a Unix terminal window. Enter the following commands, one at a time (the meaning of the commands will be covered later in these notes).

```
>>> x = randn(100,100)
>>> y = mean(x,0)
>>> plot(y)
>>> import scipy as sp
```

If everything was successfully installed, you should see something similar to figure 1.3.

16

1.5.8 IPython Notebook

IPython notebooks are a useful method to share code with others. Notebooks allow for a fluid synthesis of formatted text, typeset mathematics (using LaTeX via MathJax) and Python. The primary method for using IPython notebooks is through a web interface. The web interface allow creation, deletion, export and interactive editing of notebooks. Before running IPython Notebook for the first time, it is useful to open IPython and run the following two commands.

```
>>> from IPython.external.mathjax import install_mathjax
>>> install_mathjax()
```

These commands download a local copy of MathJax, a Javascript library for typesetting LaTeX math on web pages.

To launch the IPython notebook server on Anaconda/Windows, open a text editor, enter

```
cmd "/c ANACONDA\Scripts\activate econometrics && start "" "ipython.exe" notebook
    --matplotlib='inline' --notebook-dir=u'c:\\PATH\\TO\\NOTEBOOKS\\'"
```

and save the file as ipython-notebook.bat.

If using Linux or OSX, run

```
source ANACONDA/bin/activate econometrics
ipython notebook --matplotlib='inline' --notebook-dir='/PATH/TO/NOTEBOOKS/'
```

The command uses two optional argument. `--matplotlib='inline'` launches IPython with inline figures so that they show in the browser, and is highly recommended. `--notebook-dir='/PATH/TO/NOTEBOOKS/'` allows the default path for storing the notebooks to be set. This can be set to any location, and if not set, a default value is used. Note that both of these options can be set in ipython_notebook_config.py in profile_econometrics using

```
c.IPKernelApp.matplotlib = 'inline'
c.FileNotebookManager.notebook_dir = '/PATH/TO/NOTEBOOKS/'
```

and then the notebook should be started using only `--profile=econometrics`.

These commands will start the server and open the default browser which should be a modern version of Chrome (preferable) Chromium or Firefox. If the default browser is Safari, Internet Explorer or Opera, the URL can be copied into the Chrome address bar. The first screen that appears will look similar to figure 1.4, except that the list of notebooks will be empty. Clicking on New Notebook will create a new notebook, which, after a bit of typing, can be transformed to resemble figure 1.5. Notebooks can be imported by dragging and dropping and exported from the menu inside a notebook.

1.5.9 Integrated Development Environments

As you progress in Python and begin writing more sophisticated programs, you will find that using an Integrated Development Environment (IDE) will increase your productivity. Most contain productivity enhancements such as built-in consoles, code completion (or intellisense, for completing function names) and integrated debugging. Discussion of IDEs is beyond the scope of these notes, although Spyder is a
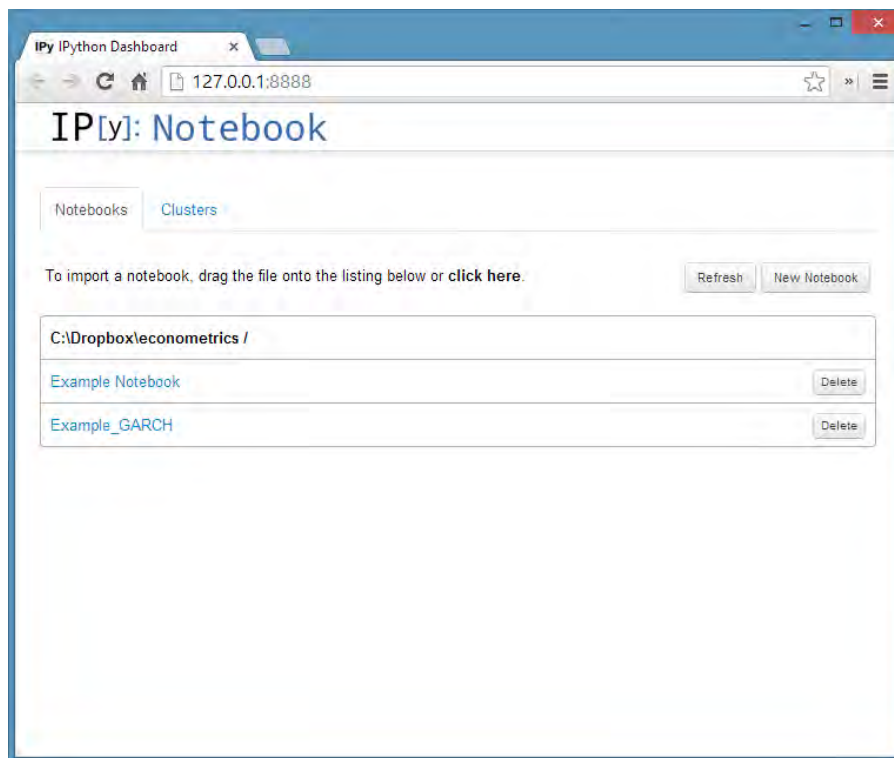
Figure 1.4: The default IPython Notebook screen showing two notebooks.
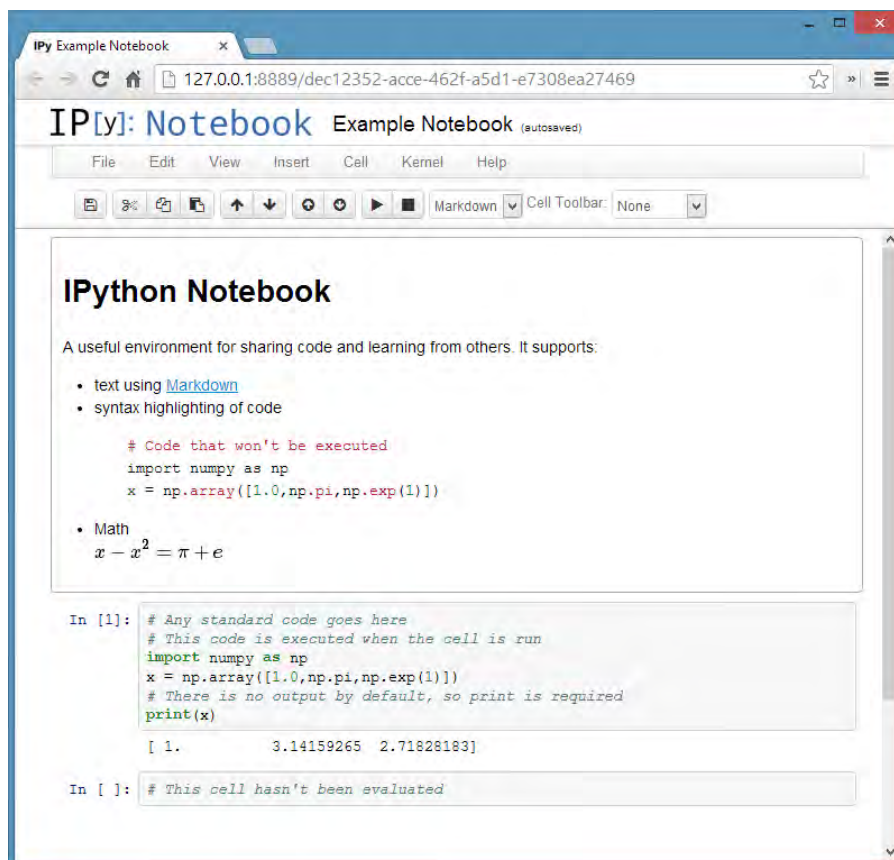
Figure 1.5: An IPython notebook showing formatted markdown, LaTeX math and cells containing code.

reasonable choice (free, cross-platform). Aptana Studio is another free alternative. My preferred IDE is PyCharm, which has a community edition that is free for use (the professional edition is low cost for academics).

Spyder

Spyder is an IDE specialized for use in scientific application rather than for general purpose Python application development. This is both an advantage and a disadvantage when compared to more full featured IDEs such as PyCharm, PyDev or Aptana Studio. The main advantage is that many powerful but complex features are not integrated into Spyder, and so the learning curve is much shallower. The disadvantage is similar - in more complex projects, or if developing something that is not straight scientific Python, Spyder is less capable. However, netting these two, Spyder is almost certainly the IDE to use when starting Python, and it is always relatively simple to migrate to a sophisticated IDE if needed.

Launching Spyder requires activating the virtual environment and then entering the command spyder. A window similar to that in figure 1.6 should appear. The main components are the the the editor (1), the object inspector (2), which dynamically will show help for functions that are used in the editor, and the console (3). By default Spyder opens a standard Python console, although it also supports using the more powerful IPython console. The object inspector window, by default, is grouped with a variable explorer, which shows the variables that are in memory and the file explorer, which can be used to navigate the file system. The console is grouped with an IPython console window (needs to be activated first using the Interpreters menu along the top edge), and the history log which contains a list of commands executed. The buttons along the top edge facilitate saving code, running code and debugging.

1.6 Exercises

1. Install Python.

2. Test the installation using the code in section 1.5.7.

3. Configure IPython using the start-up script in section 1.5.3.

4. Customize IPython QtConsole using a font or color scheme. More customizations can be found by running ipython -h.

5. Explore tab completion in IPython by entering a*<TAB>* to see the list of functions which start with a and are loaded by pylab. Next try i*<TAB>*, which will produce a list longer than the screen – press ESC to exit the pager.

6. Launch IPython Notebook and run code in the testing section.

7. Open Spyder and explore its features.

1.A register_python.py

A complete listing of register_python.py is included in this appendix.
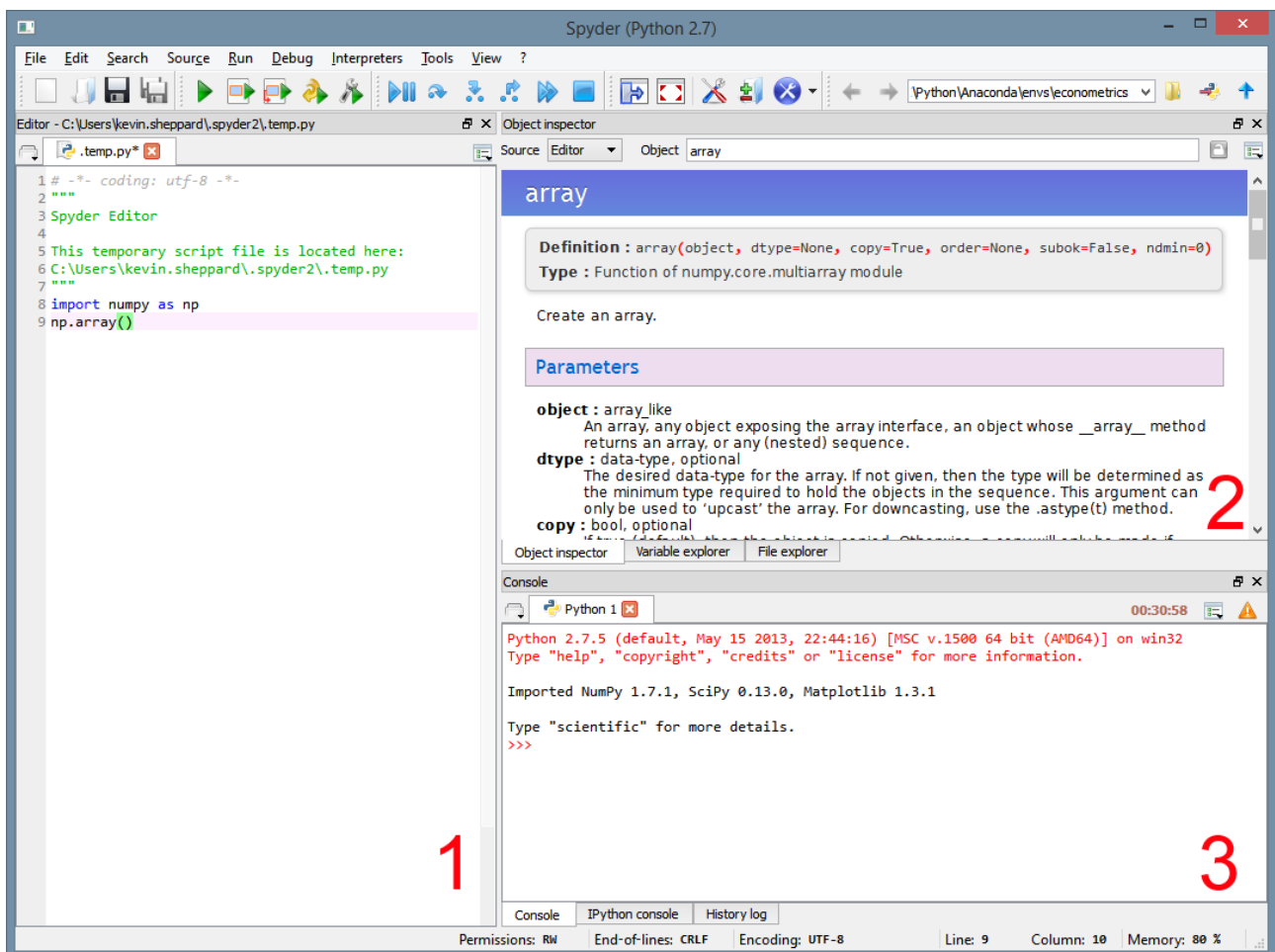
Figure 1.6: The default Spyder IDE on Windows.

```python
# -*- encoding: utf-8 -*-
#
# Script to register Python 2.0 or later for use with win32all
# and other extensions that require Python registry settings
#
# Adapted by Ned Batchelder from a script
# written by Joakim Law for Secret Labs AB/PythonWare
#
# source:
# http://www.pythonware.com/products/works/articles/regpy20.htm

import sys
from _winreg import *

# tweak as necessary
version = sys.version[:3]
installpath = sys.prefix

regpath = "SOFTWARE\\Python\\Pythoncore\\%s\\" % (version)
installkey = "InstallPath"
pythonkey = "PythonPath"
pythonpath = "%s;%s\\Lib\\;%s\\DLLs\\" % (
    installpath, installpath, installpath
)

def RegisterPy():
    try:
        reg = OpenKey(HKEY_LOCAL_MACHINE, regpath)
    except EnvironmentError:
        try:
            reg = CreateKey(HKEY_LOCAL_MACHINE, regpath)
        except Exception, e:
            print "*** Unable to register: %s" % e
            return

    SetValue(reg, installkey, REG_SZ, installpath)
    SetValue(reg, pythonkey, REG_SZ, pythonpath)
    CloseKey(reg)
    print "--- Python %s at %s is now registered!" % (version, installpath)

if __name__ == "__main__":
    RegisterPy()
```

Chapter 2

Python 2.7 vs. 3 (and the rest)

Python comes in a number of flavors which may be suitable for econometrics, statistics and numerical analysis. This chapter explains why 2.7 was chosen for these notes and highlights some of the available alternatives.

2.1 Python 2.7 vs. 3

Python 2.7 is the final version of the Python 2.x line – all future development work will focus on Python 3. It may seem strange to learn an "old" language. The reasons for using 2.7 are:

- There are more modules available for Python 2.7. While all of the core python modules are available for both Python 2.7 and 3, some of the more esoteric modules are either only available for 2.7 or have not been extensively tested in Python 3. Over time, many of these modules will be available for Python 3, but they aren't ready today.

- The language changes relevant for numerical computing are *very* small – and these notes explicitly minimize these so that there should few changes needed to run against Python 3+ in the future (ideally none).

- Configuring and installing 2.7 is easier.

- Anaconda defaults to 2.7 and the selection of packages available for Python 3 is limited.

Learning Python 3 has some advantages:

- No need to update in the future.

- Some improved out-of-box behavior for numerical applications.

2.2 Intel Math Kernel Library and AMD Core Math Library

Intel's MKL and AMD's CML provide optimized linear algebra routines. The functions in these libraries execute faster than basic those in linear algebra libraries and are, by default, multithreaded so that a many linear algebra operations will automatically make use all of the processors on your system. Most standard builds of NumPy do not include these, and so it is important to use a Python distribution built with an

appropriate linear algebra library (especially if computing inverses or eigenvalues of large matrices). The three primary methods to access NumPy built with the Intel MKL are:

- Use Anaconda on any platform and secure a license for MKL (free for academic use, otherwise $29 at the time of writing).

- Use the pre-built NumPy binaries made available by Christoph Gohlke for Windows.

- Follow instructions for building NumPy on Linux with MKL, which is free on Linux.

There are no pre-built libraries using AMD's CML, and so it is necessary to build NumPy from scratch if using an AMD processor (or buy and Intel system, which is an easier solution).

2.3 Other Variants

Some other variants of the recommended version of Python are worth mentioning.

2.3.1 Enthought Canopy

Enthought Canopy is an alternative to Anaconda. It is available for Windows, Linux and OS X. Canopy is regularly updated and is currently freely available in its basic version. The full version is also freely available to academic users. Canopy is built using MKL, and so matrix algebra performance is very fast.

2.3.2 IronPython

IronPython is a variant which runs on the Common Language Runtime (CLR , aka Windows .NET). The core modules – NumPy and SciPy – are available for IronPython, and so it is a viable alternative for numerical computing, especially if already familiar with the C# or interoperation with .NET components is important. Other libraries, for example, matplotlib (plotting) are not available, and so there are some important limitations.

2.3.3 Jython

Jython is a variant which runs on the Java Runtime Environment (JRE). NumPy is not available in Jython which severely limits Jython's usefulness for numeric work. While the limitation is important, one advantage of Python over other languages is that it is possible to run (mostly unaltered) Python code on a JVM and to call other Java libraries.

2.3.4 PyPy

PyPy is a new implementation of Python which uses Just-in-time compilation to accelerate code, especially loops (which are common in numerical computing). It may be anywhere between 2 - 500 times faster than standard Python. Unfortunately, at the time of writing, the core library, NumPy is only partially implemented, and so it is not ready for use. Current plans are to have a version ready in the near future, and if so, PyPy may quickly become the preferred version of Python for numerical computing.

2.A   Relevant Differences between Python 2.7 and 3

Most differences between Python 2.7 and 3 are not important for using Python in econometrics, statistics and numerical analysis. I will make three common assumptions which will allow 2.7 and 3 to be used interchangeable. The configuration instructions in the previous chapter for IPython will produce the expected behavior when run interactively. Note that these differences *are* important in stand-alone Python programs.

2.A.1   `print`

`print` is a function used to display test in the console when running programs. In Python 2.7, `print` is a keyword which behaves differently from other functions. In Python 3, `print` behaves like most functions. The standard use in Python 2.7 is

```python
print 'String to Print'
```

while in Python 3 the standard use is

```python
print('String to Print')
```

which resembles calling a standard function. Python 2.7 contains a version of the Python 3 `print`, which can be used in any program by including

```python
from __future__ import print_function
```

at the top of the file. I prefer the Python 3 version of `print`, and so I assume that all programs will include this statement.

2.A.2   `division`

Python 3 changes the way integers are divided. In Python 2.7, the ratio of two integers was always an integer, and so results are truncated towards 0 if the result was fractional. For example, in Python 2.7, 9/5 is 1. Python 3 gracefully converts the result to a floating point number, and so in Python 3, 9/5 is 1.8. When working with numerical data, automatically converting ratios avoids some rare errors. Python 2.7 can use the Python 3 behavior by including

```python
from __future__ import division
```

at the top of the program. I assume that all programs will include this statement.

2.A.3   `range` and `xrange`

It is often useful to generate a sequence of number for use when iterating over the some data. In Python 2.7, the best practice is to use the keyword `xrange` to do this, while in Python 3, this keyword has been renamed `range`. I will always use `xrange` and so it is necessary to replace `xrange` with `range` if using Python 3.

2.A.4   Unicode strings

Unicode is an industry standard for consistently encoding text. The computer alphabet was originally limited to 128 characters which is insufficient to contain the vast array of characters in all written languages.

Unicode expands the possible space to be up to 2^{31} characters (depending on encoding). Python 3 treats all strings as unicode unlike Python 2.7 where characters are a single byte, and unicode strings require the special syntax `u'unicode string'` or `unicode('unicode string')`. In practice this is unlikely to impact most numeric code written in Python except possibly when reading or writing data. If working in a language where characters outside of the standard but limited 128 character set are commonly encountered, it may be useful to use

```
from __future__ import unicode_literals
```

to will help with future compatibility when moving to Python 3.

Chapter 3

Built-in Data Types

Before diving into Python for analyzing data or running Monte Carlos, it is necessary to understand some basic concepts about the core Python data types. Unlike domain-specific languages such as MATLAB or R, where the default data type has been chosen for numerical work, Python is a general purpose programming language which is also well suited to data analysis, econometrics and statistics. For example, the basic numeric type in MATLAB is an array (using double precision, which is useful for *floating point* mathematics), while the basic numeric data type in Python is a 1-dimensional scalar which may be either an integer or a double-precision floating point, depending on the formatting of the number when input.

3.1   Variable Names

Variable names can take many forms, although they can only contain numbers, letters (both upper and lower), and underscores (_). They must begin with a letter or an underscore and are `CaSe SeNsItIve`. Additionally, some words are reserved in Python and so cannot be used for variable names (e.g. `import` or `for`). For example,

```
x = 1.0
X = 1.0
X1 = 1.0
X1 = 1.0
x1 = 1.0
dell = 1.0
dellreturns = 1.0
dellReturns = 1.0
_x = 1.0
x_ = 1.0
```

are all legal and distinct variable names. Note that names which begin or end with an underscore, while legal, are not normally used since by convention these convey special meaning.[1] Illegal names do not follow these rules.

---

[1] Variable names with a single leading underscores, for example `_some_internal_value`, indicate that the variable is for internal use by a module or class. While indicated to be private, this variable will generally be accessible by calling code. Double leading underscores, for example `__some_private_value` indicate that a value is actually private and is not accessible. Variable names with trailing underscores are used to avoid conflicts with reserved Python words such as `class_` or `lambda_`. Double leading and trailing underscores are reserved for "magic" variable (e.g. `__init__`) , and so should be avoided except when specifically accessing a feature.

```
# Not allowed
x: = 1.0
1X = 1
X-1 = 1
for = 1
```

Multiple variables can be assigned on the same line using commas,

```
x, y, z = 1, 3.1415, 'a'
```

3.2 Core Native Data Types

3.2.1 Numeric

Simple numbers in Python can be either integers, floats or complex. Integers correspond to either 32 bit or 64-bit integers, depending on whether the python interpreter was compiled for a 32-bit or 64-bit operating system, and floats are always 64-bit (corresponding to doubles in C/C++). Long integers, on the other hand, do not have a fixed size and so can accommodate numbers which are larger than maximum the basic integer type can handle. This chapter does not cover all Python data types, and instead focuses on those which are most relevant for numerical analysis, econometrics and statistics. The byte, bytearray and memoryview data types are not described.

3.2.1.1 Floating Point (float)

The most important (scalar) data type for numerical analysis is the float. Unfortunately, not all non-complex numeric data types are floats. To input a floating data type, it is necessary to include a . (period, dot) in the expression. This example uses the function type() to determine the data type of a variable.

```
>>> x = 1
>>> type(x)
int

>>> x = 1.0
>>> type(x)
float

>>> x = float(1)
>>> type(x)
float
```

This example shows that using the expression that x = 1 produces an integer-valued variable while x = 1.0 produces a float-valued variable. Using integers can produce unexpected results and so it is important to include ".0" when expecting a float.

3.2.1.2 Complex (complex)

Complex numbers are also important for numerical analysis. Complex numbers are created in Python using j or the function complex().

```
>>> x = 1.0
>>> type(x)
float

>>> x = 1j
>>> type(x)
complex

>>> x = 2 + 3j
>>> x
(2+3j)

>>> x = complex(1)
>>> x
(1+0j)
```

Note that $a+b$j is the same as `complex`(a, b), while `complex`(a) is the same as $a+0$j.

3.2.1.3 Integers (int and long)

Floats use an approximation to represent numbers which may contain a decimal portion. The integer data type stores numbers using an exact representation, so that no approximation is needed. The cost of the exact representation is that the integer data type cannot express anything that isn't an integer, rendering integers of limited use in most numerical work.

Basic integers can be entered either by excluding the decimal (see float), or explicitly using the `int()` function. The `int()` function can also be used to convert a float to an integer by round towards 0.

```
>>> x = 1
>>> type(x)
int

>>> x = 1.0
>>> type(x)
float

>>> x = int(x)
>>> type(x)
int
```

Integers can range from -2^{31} to $2^{31} - 1$. Python contains another type of integer, known as a long integer, which has no effective range limitation. Long integers are entered using the syntax x = 1L or by calling `long()`. Additionally python will automatically convert integers outside of the standard integer range to long integers.

```
>>> x = 1
>>> x
1

>>> type(x)
int
```

```
>>> x = 1L
>>> x
1L

>>> type(x)
long

>>> x = long(2)
>>> type(x)
long

>>> y = 2
>>> type(y)
int

>>> x = y ** 64  # ** is denotes exponentiation, y^64 in TeX
>>> x
18446744073709551616L
```

The trailing `L` after the number indicates that it is a long integer, rather than a standard integer.

3.2.2   Boolean (bool)

The Boolean data type is used to represent true and false, using the reserved keywords `True` and `False`. Boolean variables are important for program flow control (see Chapter 13) and are typically created as a result of logical operations (see Chapter 11), although they can be entered directly.

```
>>> x = True
>>> type(x)
bool

>>> x = bool(1)
>>> x
True

>>> x = bool(0)
>>> x
False
```

Non-zero, non-empty values generally evaluate to true when evaluated by `bool()`. Zero or empty values such as `bool(0)`, `bool(0.0)`, `bool(0.0j)`, `bool(None)`, `bool('')` and `bool([])` are all false.

3.2.3   Strings (str)

Strings are not usually important for *numerical* analysis, although they are frequently encountered when dealing with data files, especially when importing or when formatting output for human consumption. Strings are delimited using `''` or `""` but not using combination of the two delimiters (i.e. do not try `'"`) in a single string, except when used to express a quotation.

```
>>> x = 'abc'
>>> type(x)
```

```
str
```

```
>>> y = '"A quotation!"'
>>> print(y)
"A quotation!"
```

String manipulation is further discussed in Chapter 21.

3.2.3.1 Slicing Strings

Substrings within a string can be accessed using *slicing*. Slicing uses [] to contain the indices of the characters in a string, where the first index is 0, and the last is $n - 1$ (assuming the string has n letters). The following table describes the types of slices which are available. The most useful are s[i], which will return the character in position i, s[:i], which return the leading characters from positions 0 to $i - 1$, and s[i:] which returns the trailing characters from positions i to $n - 1$. The table below provides a list of the types of slices which can be used. The second column shows that slicing can use negative indices which essentially index the string backward.

Slice	Behavior	Slice	Behavior
s[:]	Entire string	s[i]	Characters $n - i$
s[i]	Characters$i$	s[$-i$]	Characters$n - i, \ldots, n - 1$
s[i:]	Characters$i, \ldots, n - 1$	s[$-i$:]	Characters$0, \ldots, n - i$
s[:i]	Characters$0, \ldots, i - 1$	s[:$-i$]	Characters $n - j, \ldots, n - i$
s[i:j]	Characters$i, \ldots, j - 1$	s[$-j$:$-i$]	Characters $j, j - 1, \ldots, i + 1$
s[i:j:m]	Characters$i, i + m, \ldots i + m \lfloor \frac{j-i-1}{m} \rfloor$	s[$-j$:$-i$:m]	Characters $j, j - m, \ldots, j - m \lfloor \frac{j-i-1}{m} \rfloor$

```
>>> text = 'Python strings are sliceable.'
>>> text[0]
'P'

>>> text[10]
'i'

>>> L = len(text)
>>> text[L] # Error
IndexError: string index out of range

>>> text[L-1]
'.'

>>> text[:10]
'Python str'

>>> text[10:]
'ings are sliceable.'
```

3.2.4 Lists (list)

Lists are a built-in data type which require other data types to be useful. A list is a collection of other *objects* – floats, integers, complex numbers, strings or even other lists. Lists are essential to Python programming and are used to store collections of other values. For example, a list of floats can be used to express a vector (although the NumPy data types `array` and `matrix` are better suited). Lists also support *slicing* to retrieve one or more elements. Basic lists are constructed using square braces, `[]`, and values are separated using commas.

```
>>> x = []
>>> type(x)
builtins.list

>>> x=[1,2,3,4]
>>> x
[1,2,3,4]

# 2-dimensional list (list of lists)
>>> x = [[1,2,3,4], [5,6,7,8]]
>>> x
[[1, 2, 3, 4], [5, 6, 7, 8]]

# Jagged list, not rectangular
>>> x = [[1,2,3,4] , [5,6,7]]
>>> x
[[1, 2, 3, 4], [5, 6, 7]]

# Mixed data types
>>> x = [1,1.0,1+0j,'one',None,True]
>>> x
[1, 1.0, (1+0j), 'one', None, True]
```

These examples show that lists can be regular, nested and can contain any mix of data types including other lists.

3.2.4.1 Slicing Lists

Lists, like strings, can be sliced. Slicing is similar, although lists can be sliced in more ways than strings. The difference arises since lists can be multi-dimensional while strings are always $1 \times n$. Basic list slicing is identical to slicing strings, and operations such as `x[:]`, `x[1:]`, `x[:1]` and `x[-3:]` can all be used. To understand slicing, assume x is a 1-dimensioanl list with n elements and $i \geq 0, j > 0, i < j, m \geq 1$. Python uses 0-based indices, and so the n elements of x can be thought of as $x_0, x_1, \ldots, x_{n-1}$.

Slice	Behavior,	Slice	Behavior
x[:]	Return all x	x[i]	Return x_i
x[i]	Return x_i	x[$-i$]	Returns x_{n-i} except when $i = -0$
x[i:]	Return $x_i, \ldots x_{n-1}$	x[$-i$:]	Return $x_{n-i}, \ldots, x_{n-1}$
x[:i]	Return $x_0, \ldots, x_{i-1}$	x[:$-i$]	Return $x_0, \ldots, x_{n-i}$
x[i:j]	Return $x_i, x_{i+1}, \ldots x_{j-1}$	x[$-j$:$-i$]	Return $x_{n-j}, \ldots, x_{n-i}$
x[i:j:m]	Returns $x_i, x_{i+m}, \ldots x_{i+m\lfloor \frac{j-i-1}{m} \rfloor}$	x[$-j$:$-i$:m]	Returns $x_j, x_{j-m}, \ldots, x_{j-m\lfloor \frac{j-i-1}{m} \rfloor}$

The default list slice uses a unit stride (step size of one) . It is possible to use other strides using a third input in the slice so that the slice takes the form x[i:j:m] where i is the index to start, j is the index to end (exclusive) and m is the stride length. For example x[::2] will select every second element of a list and is equivalent to x[0:n:2] where n = len(x). The stride can also be negative which can be used to select the elements of a list in reverse order. For example, x[::-1] will reverse a list and is equivalent to x[0:n:-1] .

Examples of accessing elements of 1-dimensional lists are presented below.

```
>>> x = [0,1,2,3,4,5,6,7,8,9]
>>> x[0]
0
>>> x[5]
5
>>> x[10] # Error
IndexError: list index out of range
>>> x[4:]
[4, 5, 6, 7, 8, 9]
>>> x[:4]
[0, 1, 2, 3]
>>> x[1:4]
[1, 2, 3]
>>> x[-0]
0
>>> x[-1]
9
>>> x[-10:-1]
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

List can be multidimensional, and slicing can be done directly in higher dimensions. For simplicity, consider slicing a 2-dimensional list x = [[1,2,3,4], [5,6,7,8]]. If single indexing is used, x[0] will return the first (inner) list, and x[1] will return the second (inner) list. Since the list returned by x[0] is sliceable, the inner list can be directly sliced using x[0][0] or x[0][1:4].

```
>>> x = [[1,2,3,4], [5,6,7,8]]
>>> x[0]
[1, 2, 3, 4]
>>> x[1]
[5, 6, 7, 8]
>>> x[0][0]
1
```

```
>>> x[0][1:4]
[2, 3, 4]
>>> x[1][-4:-1]
[5, 6, 7]
```

3.2.4.2 List Functions

A number of functions are available for manipulating lists. The most useful are

Function	Method	Description
list.append(x, *value*)	x.append(*value*)	Appends *value* to the end of the list.
len(x)	–	Returns the number of elements in the list.
list.extend(x, *list*)	x.extend(*list*)	Appends the values in *list* to the existing list.[2]
list.pop(x, *index*)	x.pop(*index*)	Removes the value in position *index* and returns the value.
list.remove(x, *value*)	x.remove(*value*)	Removes the first occurrence of *value* from the list.
list.count(x, *value*)	x.count(*value*)	Counts the number of occurrences of *value* in the list.
del x[*slice*]		Deletes the elements in *slice*.

```
>>> x = [0,1,2,3,4,5,6,7,8,9]
>>> x.append(0)
>>> x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0]

>>> len(x)
11

>>> x.extend([11,12,13])
>>> x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 11, 12, 13]

>>> x.pop(1)
1

>>> x
[0, 2, 3, 4, 5, 6, 7, 8, 9, 0, 11, 12, 13]

>>> x.remove(0)
>>> x
[2, 3, 4, 5, 6, 7, 8, 9, 0, 11, 12, 13]
```

Elements can also be deleted from lists using the keyword del in combination with a slice.

```
>>> x = [0,1,2,3,4,5,6,7,8,9]
>>> del x[0]
>>> x
[1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> x[:3]
[1, 2, 3]
```

```
>>> del x[:3]
>>> x
[4, 5, 6, 7, 8, 9]

>>> del x[1:3]
>>> x
[4, 7, 8, 9]

>>> del x[:]
>>> x
[]
```

3.2.5 Tuples (tuple)

A tuple is in many ways like a list – tuple contain multiple pieces of data which may contain a mix of data types. Aside from using a different syntax to construct a tuple, they are close enough to lists to ignore the difference *except that tuples are immutable*. Immutability means that the elements the comprise a tuple cannot be changed. It is not possible to add or remove elements form a tuple. However, if a tuple contains a mutable data type, for example a tuple that contains a list, the contents mutable data type can change.

Tuples are constructed using parentheses (()), rather than square braces ([]) of lists. Tuples can be sliced in an identical manner as lists. A list can be converted into a tuple using `tuple()` (Similarly, a tuple can be converted to list using `list()`).

```
>>> x =(0,1,2,3,4,5,6,7,8,9)
>>> type(x)
tuple

>>> x[0]
0

>>> x[-10:-5]
(0, 1, 2, 3, 4)

>>> x = list(x)
>>> type(x)
list

>>> x = tuple(x)
>>> type(x)
tuple

>>> x= ([1,2],[3,4])
>>> x[0][1] = -10
>>> x # Contents can change, elements cannot
([1, -10], [3, 4])
```

Note that tuples containing a single element must a comma when created, so that x = (2,) is assign a tuple to x, while x=(2) will assign 2 to x. The latter interprets the parentheses as if they are part of a mathematical formula rather than being used to construct a tuple. x = `tuple`([2]) can also be used to create

a single element tuple. Lists do not have this issue since square brackets do not have this ambiguity.

```
>>> x =(2)
>>> type(x)
int

>>> x = (2,)
>>> type(x)
tuple

>>> x = tuple([2])
>>> type(x)
tuple
```

3.2.5.1 Tuple Functions

Tuples are immutable, and so only have the methods `index` and `count`, which behave in an identical manner to their list counterparts.

3.2.6 Xrange (xrange)

A `xrange` is a useful data type which is most commonly encountered when using a `for` loop. `xrange(a,b,i)` creates the sequences that follows the pattern $a, a+i, a+2i, \ldots, a+(m-1)i$ where $m = \lceil \frac{b-a}{i} \rceil$. In other words, it find all integers x starting with a such $a \leq x < b$ and where two consecutive values are separated by i. `xrange` can be called with 1 or two parameters – `xrange(a,b)` is the same as `xrange(a,b,1)` and `xrange(b)` is the same as `xrange(0,b,1)`.

```
>>> x = xrange(10)
>>> type(x)
xrange

>>> print(x)
xrange(0, 10)

>>> list(x)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> x = xrange(3,10)
>>> list(x)
[3, 4, 5, 6, 7, 8, 9]

>>> x = xrange(3,10,3)
>>> list(x)
[3, 6, 9]

>>> y = range(10)
>>> type(y)
list

>>> y
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

xrange is not technically a list, which is why the statement `print(x)` returns `xrange(0,10)`. Explicitly converting with `list` produces a list which allows the values to be printed. Technically `xrange` is an iterator which does not actually require the storage space of a list. This can be seen in the differences between using `y = range(10)`, which returns a list and `y=xrange(10)` which returns an `xrange` object. Best practice is to use `xrange` instead of `range`.[3]

3.2.7 Dictionary (dict)

Dictionaries are encountered far less frequently than then any of the previously described data types in numerical Python. They are, however, commonly used to pass options into other functions such as optimizers, and so familiarity with dictionaries is important. Dictionaries in Python are composed of keys (words) and values (definitions). Dictionaries keys must be unique primitive data types (e.g. strings, the most common key), and values can contain any valid Python data type.[4] Values are accessed using keys.

```
>>> data = {'age': 34, 'children' : [1,2], 1: 'apple'}
>>> type(data)
dict

>>> data['age']
34
```

Values associated with an existing key can be updated by making an assignment to the key in the dictionary.

```
>>> data['age'] = 'xyz'
>>> data['age']
'xyz'
```

New key-value pairs can be added by defining a new key and assigning a value to it.

```
>>> data['name'] = 'abc'
>>> data
{1: 'apple', 'age': 'xyz', 'children': [1, 2], 'name': 'abc'}
```

Key-value pairs can be deleted using the reserved keyword `del`.

```
>>> del data['age']
>>> data
{1: 'apple', 'children': [1, 2], 'name': 'abc'}
```

3.2.8 Sets (set, frozenset)

Sets are collections which contain all *unique* elements of a collection. `set` and `frozenset` only differ in that the latter is immutable (and so has higher performance), and so `set` is similar to a unique `list` while `frozenset` is similar to a unique `tuple` . While sets are generally not important in numerical analysis, they can be very useful when working with messy data – for example, finding the set of unique tickers in a long list of tickers.

---

[3]`xrange` has been removed in Python 3 and `range` is always an iterator.

[4]Formally dictionary keys must support the __hash__ function, equality comparison and it must be the case that different keys have different hashes.

3.2.8.1 Set Functions

A number of methods are available for manipulating sets. The most useful are

Function	Method	Description
`set`.add(x, *element*)	x.add(*element*)	Appends *element* to a set.
`len`(x)	–	Returns the number of elements in the set.
`set`.difference(x, *set*)	x.difference(*set*)	Returns the elements in x which are not in *set*.
`set`.intersection(x, *set*)	x.intersection(*set*)	Returns the elements of x which are also in *set*.
`set`.remove(x, *element*)	x.remove(*element*)	Removes *element* from the set.
`set`.union(x, *set*)	x.union(*set*)	Returns the set containing all elements of x and *set*.

The code below demonstrates the use of `set`. Note that `'MSFT'` is repeated in the list used to initialize the set, but only appears once in the set since all elements must be unique.

```
>>> x = set(['MSFT','GOOG','AAPL','HPQ','MSFT'])
>>> x
{'AAPL', 'GOOG', 'HPQ', 'MSFT'}

>>> x.add('CSCO')
>>> x
{'AAPL', 'CSCO', 'GOOG', 'HPQ', 'MSFT'}

>>> y = set(['XOM', 'GOOG'])
>>> x.intersection(y)
{'GOOG'}

>>> x = x.union(y)
>>> x
{'AAPL', 'CSCO', 'GOOG', 'HPQ', 'MSFT', 'XOM'}

>>> x.remove('XOM')
{'AAPL', 'CSCO', 'GOOG', 'HPQ', 'MSFT'}
```

A `frozenset` supports the same methods except `add` and `remove`.

3.3 Python and Memory Management

Python uses a highly optimized memory allocation system which attempts to avoid allocating unnecessary memory. As a result, when one variable is assigned to another (e.g. to y = x), these will actually point to the same data in the computer's memory. To verify this, id() can be used to determine the unique identification number of a piece of data.[5]

```
>>> x = 1
>>> y = x
>>> id(x)
82970264L

>>> id(y)
```

---

[5]The ID numbers on your system will likely differ from those in the code listing.

```
82970264L

>>> x = 2.0
>>> id(x)
93850568L

>>> id(y)
82970264L
```

In the above example, the initial assignment of y = x produced two variables with the same ID. However, once x was changed, its ID changed while the ID of y did not, indicating that the data in each variable was stored in different locations. This behavior is both safe and efficient, and is common to the basic Python *immutable* types: int, long, float, complex, string, tuple, frozenset and xrange.

3.3.1 Example: Lists

Lists are mutable and so assignment does not create a copy , and so changes to either variable affect both.

```
>>> x = [1, 2, 3]
>>> y = x
>>> y[0] = -10
>>> y
[-10, 2, 3]

>>> x
[-10, 2, 3]
```

Slicing a list creates a copy of the list and *any immutable* types in the list – but not mutable elements in the list.

```
>>> x = [1, 2, 3]
>>> y = x[:]
>>> id(x)
86245960L

>>> id(y)
86240776L
```

To see that the inner lists are not copied, consider the behavior of changing one element in a nested list.

```
>>> x=[[0,1],[2,3]]
>>> y = x[:]
>>> y
[[0, 1], [2, 3]]

>>> id(x[0])
117011656L

>>> id(y[0])
117011656L

>>> x[0][0]
```

```
0.0

>>> id(x[0][0])
30390080L

>>> id(y[0][0])
30390080L

>>> y[0][0] = -10.0
>>> y
[[-10.0, 1], [2, 3]]

>>> x
[[-10.0, 1], [2, 3]]
```

When lists are nested or contain other mutable objects (which do not copy), slicing copies the outermost list to a new ID, but the inner lists (or other objects) are still linked. In order to copy nested lists, it is necessary to explicitly call deepcopy(), which is in the module copy.

```
>>> import copy as cp
>>> x=[[0,1],[2,3]]
>>> y = cp.deepcopy(x)
>>> y[0][0] = -10.0
>>> y
[[-10.0, 1], [2, 3]]

>>> x
[[0, 1], [2, 3]]
```

3.4 Exercises

1. Enter the following into Python, assigning each to a unique variable name:

    (a) `4`
    (b) `3.1415`
    (c) `1.0`
    (d) `2+4j`
    (e) `'Hello'`
    (f) `'World'`

2. What is the type of each variable? Use `type` if you aren't sure.

3. Which of the 6 types can be:

    (a) Added +
    (b) Subtracted –
    (c) Multiplied *

    (d) Divided /

4. What are the types of the output (when an error is not produced) in the above operations?

5. Input the variable `ex = 'Python is an interesting and useful language for numerical computing!'`. Using slicing, extract:

    (a) `Python`

    (b) `!`

    (c) `computing`

    (d) `in`

        Note: There are multiple answers for all.

    (e) `!gnitupmoc laciremun rof egaugnal lufesu dna gnitseretni na si nohtyP'` (Reversed)

    (f) `nohtyP`

    (g) `Pto sa neetn n sfllnug o ueia optn!`

6. What are the direct 2 methods to construct a tuple that has only a single item? How many ways are there to construct a list with a single item?

7. Construct a nested list to hold the matrix

$$\begin{bmatrix} 1 & .5 \\ .5 & 1 \end{bmatrix}$$

so that item `[i][j]` corresponds to the position in the matrix (Remember that Python uses 0 indexing).

8. Assign the matrix you just created first to `x`, and then assign `y=x`. Change `y[0][0]` to `1.61`. What happens to `x`?

9. Next assign `z=x[:]` using a simple slice. Repeat the same exercise using `y[0][0] = 1j`. What happens to `x` and `z`? What are the `id`s of `x`, `y` and `z`? What about `x[0]`, `y[0]` and `z[0]`?

10. How could you create `w` from `x` so that `w` can be changed without affecting `x`?

11. Initialize a list containing `4`, `3.1415`, `1.0`, `2+4j`, `'Hello'`, `'World'`. How could you:

    (a) Delete `1.0` if you knew its position? What if you didn't know its position?

    (b) How can the list `[1.0, 2+4j, 'Hello']` be added to the existing list?

    (c) How can the list be reversed?

    (d) In the extended list, how can you count the occurrence of `'Hello'`?

12. Construct a dictionary with the keyword-value pairs: `'alpha'` and `1.0`, `'beta'` and `3.1415`, `'gamma'` and `-99`. How can the value of `alpha` be retrieved?

13. Convert the final list at the end of problem 11 to a `set`. How is the set different from the list?

Chapter 4

Arrays and Matrices

NumPy provides the most important data types for econometrics, statistics and numerical analysis – arrays and matrices. The difference between these two data types are:

- Arrays can have 1, 2, 3 or more dimensions, and matrices always have 2 dimensions. This means that a 1 by n vector stored as an array has 1 dimension and n elements, while the same vector stored as a matrix has 2-dimensions where the sizes of the dimensions are 1 and n (in either order).

- Standard mathematical operators on arrays operate *element-by-element.* This is not the case for matrices, where multiplication (`*`) follows the rules of linear algebra. 2-dimensional arrays can be multiplied using the rules of linear algebra using `dot`. Similarly, the function `multiply` can be used on two matrices for element-by-element multiplication.

- Arrays are more common than matrices, and all functions are thoroughly tested with arrays. Functions *should also* work with matrices, but an occasional strange result may be encountered.

- Arrays can be quickly treated as a matrix using either `asmatrix` or `mat` without copying the underlying data.

The best practice is to use arrays and to use the `asmatrix` view when writing linear algebra-heavy code. It is also important to test any custom function with both arrays and matrices to ensure that false assumptions about the behavior of multiplication have not been made.

4.1 Array

Arrays are the base data type in NumPy, are are arrays in some ways similar to lists since they both contain collections of elements. The focus of this section is on homogeneous arrays containing numeric data – that is, an array where all elements have the same numeric type (heterogeneous arrays are covered in Chapters 16 and 17). Additionally, arrays, unlike lists, are always rectangular so that all rows have the same number of elements.

Arrays are initialized from lists (or tuples) using `array`. Two-dimensional arrays are initialized using lists of lists (or tuples of tuples, or lists of tuples, etc.), and higher dimensional arrays can be initialized by further nesting lists or tuples.

```
>>> x = [0.0, 1, 2, 3, 4]
>>> y = array(x)
>>> y
array([ 0.,  1.,  2.,  3.,  4.])

>>> type(y)
numpy.ndarray
```

Two (or higher) -dimensional arrays are initialized using nested lists.

```
>>> y = array([[0.0, 1, 2, 3, 4], [5, 6, 7, 8, 9]])
>>> y
array([[ 0.,  1.,  2.,  3.,  4.],
       [ 5.,  6.,  7.,  8.,  9.]])

>>> shape(y)
(2L, 5L)

>>> y = array([[[1,2],[3,4]],[[5,6],[7,8]]])
>>> y
array([[[1, 2],
        [3, 4]],

       [[5, 6],
        [7, 8]]])

>>> shape(y)
(2L, 2L, 2L)
```

4.1.1 Array dtypes

Homogeneous arrays can contain a variety of numeric data types. The most useful is 'float64', which corresponds to the python built-in data type of float (and C/C++ double). By default, calls to array will preserve the type of the input, if possible. If an input contains all integers, it will have a dtype of 'int32' (similar to the built in data type `int`). If an input contains integers, floats, or a mix of the two, the array's data type will be `float64`. If the input contains a mix of integers, floats and complex types, the array will be initialized to hold complex data.

```
>>> x = [0, 1, 2, 3, 4] # Integers
>>> y = array(x)
>>> y.dtype
dtype('int32')

>>> x = [0.0, 1, 2, 3, 4] # 0.0 is a float
>>> y = array(x)
>>> y.dtype
dtype('float64')

>>> x = [0.0 + 1j, 1, 2, 3, 4] # (0.0 + 1j) is a complex
>>> y = array(x)
```

```
>>> y
array([ 0.+1.j,  1.+0.j,  2.+0.j,  3.+0.j,  4.+0.j])

>>> y.dtype
dtype('complex128')
```

NumPy attempts to find the smallest data type which can represent the data when constructing an array. It is possible to force NumPy to select a particular dtype by using the keyword argument dtype=*datetype* when initializing the array.

```
>>> x = [0, 1, 2, 3, 4] # Integers
>>> y = array(x)
>>> y.dtype
dtype('int32')

>>> y = array(x, dtype='float64') # String dtype
>>> y.dtype
dtype('float64')

>>> y = array(x, dtype=float64) # NumPy type dtype
>>> y.dtype
dtype('float64')
```

4.2 Matrix

Matrices are essentially a subset of arrays, and behave in a virtually identical manner. The two important differences are:

- Matrices always have 2 dimensions

- Matrices follow the rules of linear algebra for *

1- and 2-dimensional arrays can be copied to a matrix by calling matrix on an array. Alternatively, calling mat or asmatrix provides a faster method where an array can behave like a matrix without copying any data.

```
>>> x = [0.0,1, 2, 3, 4] # 1 Float makes all float
>>> y = array(x)
>>> type(y)
numpy.ndarray

>>> y * y # Element-by-element
array([1, 4])

>>> z = asmatrix(x)
>>> type(z)
numpy.matrixlib.defmatrix.matrix

>>> z * z # Error
ValueError: objects are not aligned
```

4.3 1-dimensional Arrays

A vector

$$x = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \end{bmatrix}$$

is entered as a 1-dimensional array using

```
>>> x=array([1.0,2.0,3.0,4.0,5.0])
array([ 1.,  2.,  3.,  4.,  5.])

>>> ndim(x)
1
```

If an array with 2-dimensions is required, it is necessary to use a trivial nested list.

```
>>> x=array([[1.0,2.0,3.0,4.0,5.0]])
array([[ 1.,  2.,  3.,  4.,  5.]])

>>> ndim(x)
2
```

A matrix is always 2-dimensional and so a nested list is not required to initialize a a row matrix

```
>>> x=matrix([1.0,2.0,3.0,4.0,5.0])
>>> x
matrix([[ 1.,  2.,  3.,  4.,  5.]])

>>> ndim(x)
2
```

Notice that the output matrix representation uses nested lists ([[ ]]) to emphasize the 2-dimensional structure of all matrices. The column vector,

$$x = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}$$

is entered as a matrix or 2-dimensional array using a set of nested lists

```
>>> x=matrix([[1.0],[2.0],[3.0],[4.0],[5.0]])
>>> x
matrix([[ 1.],
        [ 2.],
        [ 3.],
        [ 4.],
        [ 5.]])

>>> x = array([[1.0],[2.0],[3.0],[4.0],[5.0]])
>>> x
array([[ 1.],
```

```
           [ 2.],
           [ 3.],
           [ 4.],
           [ 5.]])
```

4.4 2-dimensional Arrays

Matrices and 2-dimensional arrays are rows of columns, and so

$$x = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix},$$

is input by enter the matrix one row at a time, each in a list, and then encapsulate the row lists in another list.

```
>>> x = array([[1.0,2.0,3.0],[4.0,5.0,6.0],[7.0,8.0,9.0]])
>>> x
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.],
       [ 7.,  8.,  9.]])
```

4.5 Multidimensional Arrays

Higher dimensional arrays are useful when tracking matrix valued processes through time, such as a time-varying covariance matrices. Multidimensional (N-dimensional) arrays are available for N up to about 30, depending on the size of each matrix dimension. Manually initializing higher dimension arrays is tedious and error prone, and so it is better to use functions such as `zeros((2, 2, 2))` or `empty((2, 2, 2))`.

4.6 Concatenation

Concatenation is the process by which one vector or matrix is appended to another. Arrays and matrices can be concatenation horizontally or vertically. For example, suppose

$$x = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \text{ and } y = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix};$$

and

$$z = \begin{bmatrix} x \\ y \end{bmatrix}.$$

needs to be constructed. This can be accomplished by treating x and y as elements of a new matrix and using the function `concatenate` to join them. The inputs to `concatenate` must be grouped in a tuple and the keyword argument `axis` specifies whether the arrays are to be vertically (`axis = 0`) or horizontally (`axis = 1`) concatenated.

```
>>> x = array([[1.0,2.0],[3.0,4.0]])
>>> y = array([[5.0,6.0],[7.0,8.0]])
>>> z = concatenate((x,y),axis = 0)
>>> z
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.],
       [ 7.,  8.]])

>>> z = concatenate((x,y),axis = 1)
>>> z
array([[ 1.,  2.,  5.,  6.],
       [ 3.,  4.,  7.,  8.]])
```

Concatenating is the code equivalent of block-matrix forms in standard matrix algebra. Alternatively, the functions vstack and hstack can be used to vertically or horizontally stack arrays, respectively.

```
>>> z = vstack((x,y)) # Same as z = concatenate((x,y),axis = 0)
>>> z = hstack((x,y)) # Same as z = concatenate((x,y),axis = 1)
```

4.7 Accessing Elements of an Array

Four methods are available for accessing elements contained within an array: scalar selection, slicing, numerical indexing and logical (or Boolean) indexing. Scalar selection and slicing are the simplest and so are presented first. Numerical indexing and logical indexing both depends on specialized functions and so these methods are discussed in Chapter 12.

4.7.1 Scalar Selection

Pure scalar selection is the simplest method to select elements from an array, and is implemented using $[i]$ for 1-dimensional arrays, $[i,j]$ for 2-dimensional arrays and $[i_1,i_2,\ldots,i_n]$ for general n-dimensional arrays.

```
>>> x = array([1.0,2.0,3.0,4.0,5.0])
>>> x[0]
1.0

>>> x = array([[1.0,2,3],[4,5,6]])
>>> x[1,2]
6.0

>>> type(x[1,2])
numpy.float64
```

Pure scalar selection always returns a single element which is *not* an array. The data type of the selected element matches the data type of the array used in the selection. Scalar selection can also be used to assign values in an array.

```
>>> x = array([1.0,2.0,3.0,4.0,5.0])
>>> x[0] = -5
>>> x
array([-5.,  2.,  3.,  4.,  5.])
```

4.7.2 Array Slicing

Arrays, like lists and tuples, can be sliced. Arrays slicing is virtually identical to lists slicing except that a simpler slicing syntax is available since arrays are explicitly multidimensional and rectangular. Arrays are sliced using the syntax [:,:,...,:] (where the number of dimensions of the arrays determines the size of the slice).[1] Recall that the slice notation $a{:}b{:}s$ will select every s^{th} element where the indices i satisfy $a \leq i < b$ so that the starting value a is always included in the list and the ending value b is always excluded. Additionally, a number of shorthand notations are commonly encountered

- : and :: are the same as 0:n:1 where n is the length of the array (or list).

- a: and a:n are the same as a:n:1 where n is the length of the array (or list).

- :b is the same as 0:b:1.

- ::s is the same as 0:n:s where n is the length of the array (or list).

Basic slicing of 1-dimensional arrays is identical to slicing a simple list, and the returned type of all slicing operations matches the array being sliced.

```
>>> x = array([1.0,2.0,3.0,4.0,5.0])
>>> y = x[:]
array([ 1.,  2.,  3.,  4.,  5.])

>>> y = x[:2]
array([ 1.,  2.])

>>> y = x[1::2]
array([ 2.,  4.])
```

In 2-dimensional arrays, the first dimension specifies the row or rows of the slice and the second dimension specifies the the column or columns. Note that the 2-dimensional slice syntax y[a:b,c:d] is the same as y[a:b,:][:,c:d] or y[a:b][:,c:d], although clearly the shorter form is preferred. In the case where only row slicing in needed y[a:b], which is the equivalent to y[a:b,:], is the shortest syntax.

```
>>> y = array([[0.0, 1, 2, 3, 4],[5, 6, 7, 8, 9]])
>>> y
array([[ 0.,  1.,  2.,  3.,  4.],
       [ 5.,  6.,  7.,  8.,  9.]])

>>> y[:1,:] # Row 0, all columns
array([[ 0.,  1.,  2.,  3.,  4.]])
```

---

[1] It is not necessary to include all trailing slice dimensions, and any omitted trailing slices are set to select all elements (the slice :). For example, if x is a 3-dimensional array, x[0:2] is the same as x[0:2,:,:] and x[0:2,0:2] is the same as x[0:2,0:2,:].

```
>> y[:1] # Same as y[:1,:]
array([[ 0.,  1.,  2.,  3.,  4.]])

>>> y[:,:1] # all rows, column 0
array([[ 0.],
       [ 5.]])

>>> y[:1,0:3] # Row 0, columns 0 to 2
array([[ 0.,  1.,  2.]])

>>> y[:1][:,0:3] # Same as previous
array([[ 0.,  1.,  2.]])

>>> y[:,3:] # All rows, columns 3 and 4
array([[ 3.,  4.],
       [ 8.,  9.]])

>>> y = array([[[1.0,2],[3,4]],[[5,6],[7,8]]])
>>> y[:1,:,:] # Panel 0 of 3D y
array([[[1, 2],
        [3, 4]]])
```

In the previous examples, slice notation was always used even when only selecting 1 row or column. This was done to emphasize the difference between using slice notation, which always returns an array with the same dimension and using a scalar selector which will perform dimension reduction.

4.7.3 Mixed Selection using Scalar and Slice Selectors

When arrays have more than 1-dimension, it is often useful to mix scalar and slice selectors to select an entire row, column or panel of a 3-dimensional array. This is similar to pure slicing with one important caveat – dimensions selected using scalar selectors are eliminated. For example, if x is a 2-dimensional array, then x[0,:] will select the first row. However, unlike the 2-dimensional array constructed using the slice x[:1,:], x[0,:] will be a 1-dimensional array.

```
>>> x = array([[1.0,2],[3,4]])
>>> x[:1,:] # Row 1, all columns, 2-dimensional
array([[ 1.,  2.]])

>>> x[0,:] # Row 1, all columns, dimension reduced
array([ 1.,  2.])
```

While these two selections appear similar, the first produces a 2-dimensional array (note the [[ ]] syntax) while the second is a 1-dimensional array. In most cases where a single row or column is required, using scalar selectors such as y[0,:] is the best practice. It is important to be aware of the dimension reduction since scalar selections from a 2-dimensional arrays will no longer have 2-dimensions. This type of dimension reduction may matter when evaluating linear algebra expression.

The principle adopted by NumPy is that slicing should always preserve the dimension of the underlying array, while scalar indexing should always collapse the dimension(s). This is consistent with x[0,0]

returning a scalar (or 0-dimensional array) since both selections are scalar. This is demonstrated in the next example which highlights the differences between pure slicing, mixed slicing and pure scalar selection. Note that the function `ndim` returns the number of dimensions of an array.

```
>>> x = array([[0.0, 1, 2, 3, 4],[5, 6, 7, 8, 9]])
>>> x[:1,:] # Row 0, all columns, 2-dimensional
array([[ 0.,  1.,  2.,  3.,  4.]])

>>> ndim(x[:1,:])
2

>>> x[0,:] # Row 0, all column, dim reduction to 1-d array
array([ 0.,  1.,  2.,  3.,  4.])

>>> ndim(x[0,:])
1

>>> x[0,0] # Top left element, dim reduction to scalar (0-d array)
0.0

>>> ndim(x[0,0])

>>> x[:,0] # All rows, 1 column, dim reduction to 1-d array
array([ 0., 5.])
```

4.7.4 Assignment using Slicing

Slicing and scalar selection can be used to assign arrays that have the same dimension as the slice.[2]

```
>>> x = array([[0.0]*3]*3)  # *3 repeats the list 3 times
>>> x
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])

>>> x[0,:] = array([1.0, 2.0, 3.0])
>>> x
array([[ 1.,  2.,  3.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])

>>> x[::2,::2] =  array([[-99.0,-99],[-99,-99]]) # 2 by 2
>>> x
array([[-99.,   2., -99.],
       [  0.,   0.,   0.],
       [-99.,   0., -99.]])

>>> x[1,1] = pi
```

---

[2]Formally, the array to be assigned must be broadcastable to the size of the slice. Broadcasting is described in Chapter 5, and assignment using broadcasting is discussed in Chapter 12.

```
>>> x
array([[-99.       ,    2.       , -99.      ],
       [  0.       ,    3.14159265,   0.      ],
       [-99.       ,    0.       , -99.      ]])
```

NumPy attempts to automatic (silent) data type conversion if an element with one data type is inserted into an array wit a different data type. For example, if an array has an integer data type, place a float into the array results in the float being truncated and stored as an integer. This is dangerous, and so in most cases, arrays should be initialized to contain floats unless a considered decision is taken to use a different data type.

```
>>> x = [0, 1, 2, 3, 4] # Integers
>>> y = array(x)
>>> y.dtype
dtype('int32')

>>> y[0] = 3.141592
>>> y
array([3, 1, 2, 3, 4])

>>> x = [0.0,1, 2, 3, 4] # 1 Float makes all float
>>> y = array(x)
>>> y.dtype
dtype('float64')

>>> y[0] = 3.141592
>>> y
array([ 3.141592, 1.       , 2.       , 3.       , 4.       ])
```

4.7.5 Linear Slicing using `flat`

Data in matrices is stored in *row-major order* – elements are indexed by first counting across rows and then down columns. For example, in the matrix

$$x = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

the first element of x is 1, the second element is 2, the third is 3, the fourth is 4, and so on.

In addition to slicing using the $[:,:,\ldots,:]$ syntax, k-dimensional arrays can be linear sliced. Linear slicing assigns an index to each element of the array, starting with the first (0), the second (1), and so on until the final element $(n-1)$. In 2-dimensions, linear slicing works by first counting across rows, and then down columns. To use linear slicing, the method or function `flat` must first be used.

```
>>> y = reshape(arange(25.0),(5,5))
>>> y
array([[  0.,   1.,   2.,   3.,   4.],
       [  5.,   6.,   7.,   8.,   9.],
       [ 10.,  11.,  12.,  13.,  14.],
       [ 15.,  16.,  17.,  18.,  19.],
```

```
      [ 20.,  21.,  22.,  23.,  24.]])

>>> y[0] # Same as y[0,:], first row
array([ 0.,  1.,  2.,  3.,  4.])

>>> y.flat[0] # Scalar slice, flat is 1-dimensional
0

>>> y[6] # Error
IndexError: index out of bounds

>>> y.flat[6] # Element 6
6.0

>>> y.flat[12:15]
array([ 12.,  13.,  14.])

>>> y.flat[:] # All element slice
array([[  0.,   1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.,
         11.,  12.,  13.,  14.,  15.,  16.,  17.,  18.,  19.,  20.,  21.,
         22.,  23.,  24.]])
```

Note that arange and reshape are useful functions are described in later chapters.

4.8 Slicing and Memory Management

Unlike lists, slices of arrays are do not copy the underlying data. Instead a slice of an array returns a view of the array which shares the data in the sliced array. This is important since changes in slices will propagate to the underlying array and to any other slices which share the same element.

```
>>> x = reshape(arange(4.0),(2,2))
>>> x
array([[ 0.,  1.],
       [ 2.,  3.]])

>>> s1 = x[0,:] # First row
>>> s2 = x[:,0] # First column
>>> s1[0] = -3.14 # Assign first element
>>> s1
array([-3.14,  1.  ])

>>> s2
array([-3.14,  2.  ])

>>> x
array([[-3.14,  1.  ],
       [ 2.  ,  3.  ]])
```

If changes should not propagate to parent and sibling arrays, it is necessary to call copy on the slice. Alternatively, they can also be copied by calling array on arrays, or matrix on matrices.

```
>>> x = reshape(arange(4.0),(2,2))
>>> s1 = copy(x[0,:]) # Function copy
>>> s2 = x[:,0].copy() # Method copy
>>> s3 = array(x[0,:]) # Create a new array
>>> s1[0] = -3.14
>>> s1
array([-3.14,  1.])

>>> s2
array([ 0.,  2.])

>>> s3
array([0.,  1.])

>>> x[0,0]
array([[ 0.,  1.],
       [ 2.,  3.]])
```

There is one notable exception to this rule – when using pure scalar selection the (scalar) value returned is always a copy.

```
>>> x = arange(5.0)
>>> y = x[0] # Pure scalar selection
>>> z = x[:1] # A pure slice
>>> y = -3.14
>>> y # y Changes
-3.14

>>> x # No propagation
array([ 0.,  1.,  2.,  3.,  4.])

>>> z  # No changes to z either
array([ 0.])

>>> z[0] = -2.79
>>> y # No propagation since y used pure scalar selection
-3.14

>>> x # z is a view of x, so changes propagate
array([-2.79,  1.  ,  2.  ,  3.  ,  4.  ])
```

Finally, assignments from functions which change values will automatically create a copy of the underlying array.

```
>>> x = array([[0.0, 1.0],[2.0,3.0]])
>>> y = x
>>> print(id(x),id(y)) # Same
 129186368 129186368
>>> y = x + 1.0
>>> y
array([[ 1.,  2.],
       [ 3.,  4.]])
```

```
>>> print(id(x),id(y)) # Different
129186368 129183104

>>> x # Unchanged
array([[ 0.,  1.],
       [ 2.,  3.]])

>>> y = exp(x)
>>> print(id(x),id(y)) # Also Different
129186368 129185120
```

Even trivial function such as `y = x + 0.0` create a copy of `x`, and so the only scenario where explicit copying is required is when `y` is directly assigned using a slice of `x`, and changes to `y` should not propagate to `x`.

4.9 `import` and Modules

Python, by default, only has access to a small number of built-in types and functions. The vast majority of functions are located in modules, and before a function can be accessed, the module which contains the function must be imported. For example, when using `ipython --pylab` (or any variants), a large number of modules are automatically imported, including NumPy and matplotlib. This is style of importing useful for learning and interactive use, but care is needed to make sure that the correct module is imported when designing more complex programs.

`import` can be used in a variety of ways. The simplest is to use `from` *module* `import` `*` which imports all functions in *module*. This method of using `import` can dangerous since if you use it more than once, it is possible for functions to be hidden by later imports. A better method is to just import the required functions. This still places functions at the top level of the namespace, but can be used to avoid conflicts.

```
from pylab import log2 # Will import log2 only
from scipy import log10 # Will not import the log2 from SciPy
```

The functions `log2` and `log10` can both be called in subsequent code. An alternative, and more common, method is to use `import` in the form

```
import pylab
import scipy
import numpy
```

which allows functions to be accessed using dot-notation and the module name, for example `scipy.log2`. It is also possible to rename modules when imported using `as`

```
import pylab as pl
import scipy as sp
import numpy as np
```

The only difference between these two is that `import scipy` is implicitly calling `import scipy as scipy`. When this form of import is used, functions are used with the "as" name. For example, the load provided by NumPy is accessed using `sp.log2`, while the pylab load is `pl.log2` – and both can be used where appropriate. While this method is the most general, it does require slightly more typing.

4.10 Calling Functions

Functions calls have different conventions than most other expressions. The most important difference is that functions can take more than one input and return more than one output. The generic structure of a function call is *out1, out2, out3, …* = `functionname`(*in1, in2, in3, …*). The important aspects of this structure are

- If multiple outputs are returned, but only one output variable is provided, the output will (generally) be a tuple.

- If more than one output variable is given in a function call, the number of output must match the number of output provided by the function. It is not possible to ask for two output if a function returns three – using an incorrect number of outputs results in `ValueError: too many values to unpack`.

- Both inputs and outputs must be separated by commas (,)

- Inputs can be the result of other functions as long only one output is returned. For example, the following are equivalent,

  ```
  >>> y = var(x)
  >>> mean(y)
  ```

  and

  ```
  >>> mean(var(x))
  ```

Required Arguments

Most functions have required arguments. For example, consider the definition of `array` from `help(array)`,

```
array(object, dtype=None, copy=True, order=None, subok=False, ndmin=0)
```

Array has 1 required input, `object`, which is the list or tuple which contains values to use when creating the array. Required arguments can be determined by inspecting the function signature since all of the input follow the patters *keyword=default* except `object` – required arguments will not have a default value provided. The other arguments can be called in order (`array` accepts at most 2 non-keyword arguments).

```
>>> array([[1.0,2.0],[3.0,4.0]])
array([[ 1.,  2.],
       [ 3.,  4.]])

>>> array([[1.0,2.0],[3.0,4.0]], 'int32')
array([[1, 2],
       [3, 4]])
```

Keyword Arguments

All of the arguments to `array` can be called by their keyword, which is listed in the help file definition.

```
array(object=[[1.0,2.0],[3.0,4.0]])
array([[1.0,2.0],[3.0,4.0]], dtype=None, copy=True, order=None, subok=False, ndmin=0)
```

Keyword arguments have two important advantages. First, they do not have to appear in any order (Note: randomly ordering arguments is not good practice, and this is only an example), and second, keyword arguments can be used only when needed since a default value is always given.

```
>>> array(dtype='complex64', object = [[1.0,2.0],[3.0,4.0]], copy=True)
array([[ 1.+0.j,  2.+0.j],
       [ 3.+0.j,  4.+0.j]], dtype=complex64)
```

Default Arguments

Functions have defaults for optional arguments. These are listed in the function definition and appear in the help in the form *keyword=default*. Returning to array, all inputs have default arguments except object – the only required input.

Multiple Outputs

Some functions can have more than 1 output. These functions can be used in a single output mode or in multiple output mode. For example, shape can be used on an array to determine the size of each dimension.

```
>>> x = array([[1.0,2.0],[3.0,4.0]])
>>> s = shape(x)
>>> s
(2L, 2L)
```

Since shape will return as many outputs as there are dimensions, it can be called with 2 outputs when the input is a 2-dimensional array.

```
>>> x = array([[1.0,2.0],[3.0,4.0]])
>>> M,N = shape(x)
>>> M
2L

>>> N
2L
```

Requesting more outputs than are required will produce an error.

```
>>> M,N,P = shape(x) # Error
ValueError: need more than 2 values to unpack
```

Similarly, providing two few output can also produce an error. Consider the case where the argument used with shape is a 3-dimensional array.

```
>>> x = randn(10,10,10)
>>> shape(x)
(10L, 10L, 10L)
>>> M,N = shape(x) # Error
ValueError: too many values to unpack
```

4.11 Exercises

1. Input the following mathematical expressions into Python as both arrays and matrices.

$$u = \begin{bmatrix} 1 & 1 & 2 & 3 & 5 & 8 \end{bmatrix}$$

$$v = \begin{bmatrix} 1 \\ 1 \\ 2 \\ 3 \\ 5 \\ 8 \end{bmatrix}$$

$$x = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$y = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$$z = \begin{bmatrix} 1 & 2 & 1 & 2 \\ 3 & 4 & 3 & 4 \\ 1 & 2 & 1 & 2 \end{bmatrix}$$

$$w = \begin{bmatrix} x & x \\ y & y \end{bmatrix}$$

Note: A column vector must be entered as a 2-dimensional array.

2. What command would pull x out of w? (Hint: `w[?,?]` is the same as x.)

3. What command would pull $\begin{bmatrix} x' & y' \end{bmatrix}'$ out of `w`? Is there more than one? If there are, list all alternatives.

4. What command would pull y out of z? List all alternatives.

5. Explore the options for creating an array using keyword arguments. Create an array containing

$$y = \begin{bmatrix} 1 & -2 \\ -3 & 4 \end{bmatrix}$$

with combination of keyword arguments in:

   (a) `dtype` in `float`, `float64`, `int32` (32-bit integers), `uint32` (32-bit unsigned integers) and `complex128` (double precision complex numbers).

   (b) `copy` either `True` or `False`.

   (c) `ndim` either 3 or 4. Use `shape(y)` to see the effect of this argument.

6. Enter $y = [1.6180\ 2.7182\ 3.1415]$ as an array. Define `x = mat(y)`. How is x different from y?

Chapter 5

Basic Math

Note: Python contains a `math` module providing functions which operate on built-in scalar data types (e.g. `float` and `complex`). This and subsequent chapters assume mathematical functions must operate on arrays and matrices, and so are imported from NumPy.

5.1 Operators

These standard operators are available:

Operator	Meaning	Example	Algebraic
+	Addition	x + y	$x + y$
-	Subtraction	x - y	$x - y$
*	Multiplication	x * y	$x\,y$
/	Division (Left divide)	x/y	$\frac{x}{y}$
**	Exponentiation	x**y	x^y

When x and y are scalars, the behavior of these operators is obvious. The only possible exception occurs when both x and y are integers for division, where x/y returns the smallest integer less than the ratio (e.g. $\lfloor \frac{x}{y} \rfloor$). The simplest method to avoid this problem is use `from __future__ import` division which changes the default behavior. Alternatively, declaring numeric values to be floats using `5.0` rather than `5` will also mitigate this issue as well explicitly casting integers to floats before dividing.

```
>>> x = 9
>>> y = 5
>>> (type(x), type(y))
(int, int)

>>> x/y # Since division imported
1.8

>>> float(x)/y
1.8
```

When x and y are arrays or matrices, the behavior of mathematical operations is more complex. The examples in this chapter refer to arrays, and except where explicit differences are noted, it is safe to assume that the behavior of 2-dimensional arrays and matrices is identical.

I recommend using the import command from `__future__ import division` in all programs and IPython. The "future" division avoids this issue by always casting division to floating point when the result is not an exact integer.

5.2 Broadcasting

Under the normal rules of array mathematics, addition and subtraction are only defined for arrays with the same shape or between an array and a scalar. For example, there is no obvious method to add a 5-element vector and a 5 by 4 matrix. NumPy uses a technique called broadcasting to allow element-by-element mathematical operations on arrays (and matrices) which would not be compatible under the standard rules of array mathematics.

Arrays can be used in element-by-element mathematics if x is broadcastable to y. Suppose x is an m-dimensional array with dimensions $d = [d_1, d_2 \ldots d_m]$, and y is an n-dimensional array with dimensions $f = [f_1, f_2 \ldots f_n]$ where $m \geq n$. Formally, two arrays are broadcastable if the following two conditions hold.

1. If $m > n$, then treat y as a m-dimensional array with size $g = [1, 1, \ldots, 1, f_1, f_2 \ldots f_n]$ where the number of 1s prepended is $m - n$. The dimensions are $g_i = 1$ for $i = 1, \ldots m - n$ and $g_i = f_{i-m+n}$ for $i > m - n$.

2. For $i = 1, \ldots, m$, $\max(d_i, g_i) / \min(d_i, g_i) \in \{1, \max(d_i, g_i)\}$.

The first rule is simply states that if one array has fewer dimensions, it is treated as having the same number of dimensions as the larger array by prepending 1s. The second rule states that arrays will only be broadcastable if either (a) they have the same dimension along axis i or (b) one has dimension 1 along axis i. When 2 arrays are broadcastable, the dimension of the output array is $\max(d_i, g_i)$ for $i = 1, \ldots n$.

Consider the following examples where m, n and p are assumed to have different values.

x	y	Broadcastable	Output Size	x Operation	y Operation
Any	Scalar	✓	Same as x	x	`tile(y,shape(x))`
$m, 1$	$1, n$ or n	✓	m, n	`tile(x,(1,n))`	`tile(y,(m,1))`
$m, 1$	$n, 1$	✗			
m, n	$1, n$ or n	✓	m, n	x	`tile(y,(m,1))`
$m, n, 1$	$1, 1, p$ or $1, p$ or p	✓	m, n, p	`tile(x,(1,1,p))`	`tile(y,(m,n,1))`
m, n, p	$1, 1, p$ or $1, p$ or p	✓	m, n, p	x	`tile(y,(m,n,1))`
$m, n, 1$	$p, 1$	✗			
$m, 1, p$	$1, n, 1, 1, n, p$ or $n, 1$	✓	m, n, p	`tile(x,(1,n,1))`	`tile(y,(m,1,p))`

One simple method to visualize broadcasting is to use an add and subtract operation where the addition causes the smaller array to be broadcast, and then the subtract removes the values in the larger array. This will produce a replicated version of the smaller array which shows the nature of the broadcasting.

```
>>> x = array([[1,2,3.0]])
>>> x
array([[ 1.,  2.,  3.]])

>>> y = array([[0],[0],[0.0]])
>>> y
array([[ 0.],
       [ 0.],
       [ 0.]])

>>> x + y  # Adding 0 produces broadcast
array([[ 1.,  2.,  3.],
       [ 1.,  2.,  3.],
       [ 1.,  2.,  3.]])
```

In the next example, x is 3 by 5, so y must be either scalar or a 5-element array or a 1×5 array to be broadcastable. When y is a 3-element array (and so matches the *leading* dimension), an error occurs.

```
>>> x = reshape(arange(15),(3,5))
>>> x
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> y = 1
>>> x + y - x
array([[5, 5, 5, 5, 5],
       [5, 5, 5, 5, 5],
       [5, 5, 5, 5, 5]])

>>> y = arange(5)
>>> y
array([0, 1, 2, 3, 4])

>>> x + y - x
array([[0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4]])

>>> y = arange(3)
>>> y
array([0, 1, 2])

>>> x + y - x # Error
ValueError: operands could not be broadcast together with shapes (3,5) (3)
```

5.3 Array and Matrix Addition (+) and Subtraction (−)

Subject to broadcasting restrictions, addition and subtraction works element-by-element.

5.4   Array Multiplication (∗)

The standard multiplication operator differs for variables with type `array` and `matrix`. For arrays, ∗ performs *element-by-element* multiplication and so inputs must be broadcastable. For matrices, ∗ is matrix multiplication as defined by linear algebra, and there is *no broadcasting*.

Conformable arrays can be multiplied according to the rules of matrix algebra using the function `dot()`. For simplicity, assume x is N by M and y is K by L. If $M = K$, `dot(x,y)` will produce the array N by L array `z[i,j] = =dot( x[i,:], y[:,j])` where dot on 1-dimensional arrays is the usual vector dot-product. The behavior of `dot()` is described as:

		y	
		Scalar	Array
x	Scalar	Any $z = x\,y$	Any $z_{ij} = x\,y_{ij}$
	Array	Any $z_{ij} = y\,x_{ij}$	Inside Dimensions Match $z_{ij} = \sum_{k=1}^{M} x_{ik}\,y_{kj}$

These rules conform to the standard rules of matrix multiplication. `dot()` can also be used on higher dimensional arrays, and is useful if x is T by M by N and y is N by P to produce an output matrix which is T by M by P, where each of the M by P (T in total) have the form `dot(x[i],y)`.

5.5   Matrix Multiplication (∗)

If x is N by M and y is K by L and both are non-scalar matrices, `x*y` requires $M = K$. Similarly, `y*x` requires $L = N$. If x is scalar and y is a matrix, then `z=x*y` produces `z(i,j)=x*y(i,j)`. Suppose `z=x * y` where both x and y are matrices:

		y	
		Scalar	Matrix
x	Scalar	Any $z = x\,y$	Any $z_{ij} = x\,y_{ij}$
	Matrix	Any $z_{ij} = y\,x_{ij}$	Inside Dimensions Match $z_{ij} = \sum_{k=1}^{M} x_{ik}\,y_{kj}$

Note: These conform to the standard rules of matrix multiplication.

`multiply()` performs element-by-element multiplication of matrices, and will use broadcasting if necessary. Matrices are identical to 2-dimensional arrays when performing element-by-element multiplication.

5.6   Array and Matrix Division (/)

Division is always element-by-element, and the rules of broadcasting are used.

5.7   Array Exponentiation (**)

Array exponentiation operates element-by-element, and the rules of broadcasting are used.

5.8 Matrix Exponentiation (**)

Matrix exponentiation differs from array exponentiation, and can only be used on square matrices. When x is a square matrix and y is a positive integer, x**y produces x*x*...*x (y times). When y is a negative integer, x**y produces `inv`(x**`abs`(y)) where `inv` produces the inverse, and so x must have full rank. Python does not support non-integer values for y, although x^p can be defined (in linear algebra) using eigenvalues and eigenvectors for a subset of all matrices.

5.9 Parentheses

Parentheses can be used in the usual way to control the order in which mathematical expressions are evaluated, and can be nested to create complex expressions. See section 5.11 on Operator Precedence for more information on the order mathematical expressions are evaluated.

5.10 Transpose

Matrix transpose is expressed using either the `transpose` function, or the shortcut `.T`. For instance, if x is an M by N matrix, `transpose`(x), x.`transpose`() and x.T are all its transpose with dimensions N by M. In practice, using the `.T` is the preferred method and will improve readability of code. Consider

```
>>> x = asmatrix(randn(2,2))
>>> xpx1 = x.T * x
>>> xpx2 = x.transpose() * x
>>> xpx3 = transpose(x) * x
```

Transpose has no effect on 1-dimensaional arrays. In 2-dimensions, transpose switches indices so that if z=x.T, z[j,i] is that same as x[i,j]. In higher dimensions, transpose reverses the order or the indices. For example, if x has 3 dimensions and z=x.T, then x[i,j,k] is the same as z[k,j,i]. Transpose takes an optional second argument to specify the axis to use when permuting the array.

5.11 Operator Precedence

Computer math, like standard math, has operator precedence which determined how mathematical expressions such as

$$2**3+3**2/7*13$$

are evaluated. Best practice is to always use parentheses to avoid ambiguity in the order or operations. The order of evaluation is:

Operator	Name	Rank
( ), [ ] , (,)	Parentheses, Lists, Tuples	1
**	Exponentiation	2
~	Bitwise NOT	3
+,-	Unary Plus, Unary Minus	3
*, /, //, %	Multiply, Divide, Modulo	4
+,-	Addition and Subtraction	5
&	Bitwise AND	6
^	Bitwise XOR	7
	Bitwise OR	8
<, <=, >, >=	Comparison operators	9
==, !=	Equality operators	9
in, not in	Identity Operators	9
is, is not	Membership Operators	9
not	Boolean NOT	10
and	Boolean AND	11
or	Boolean OR	12
=,+=,-=,/=,*=,**=	Assignment Operators	13

Note that some rows of the table have the same precedence, and are only separated since they are conceptually different. In the case of a tie, operations are executed left-to-right. For example, x**y**z is interpreted as (x**y)**z. This table has omitted some operators available in Python which are not generally useful in numerical analysis (e.g. shift operators).

Note: Unary operators are + or - operations that apply to a single element. For example, consider the expression (-4). This is an instance of a unary negation since there is only a single operation and so (-4)**2 produces 16. On the other hand, -4**2 produces -16 since ** has higher precedence than unary negation and so is interpreted as -(4**2). -4 * -4 produces 16 since it is interpreted as (-4) * (-4), since unary negation has higher precedence than multiplication.

5.12   Exercises

1. Using the arrays entered in exercise 1 of chapter 4, compute the values of $u + v'$, $v + u'$, $v\,u$, $u\,v$ and $x\,y$ (where the multiplication is as defined as linear algebra).

2. Repeat exercise 1 treating the inputs as matrices.

3. Which of the arrays in exercise 1 are broadcastable with:

$$a = [3\ 2],$$

$$b = \begin{bmatrix} 3 \\ 2 \end{bmatrix},$$

$$c = [3\ 2\ 1\ 0],$$

$$d = \begin{bmatrix} 3 \\ 2 \\ 1 \\ 0 \end{bmatrix}.$$

4. Is x/1 legal? If not, why not. What about 1/x?

5. Compute the values (x+y)**2 and x**2+x*y+y*x+y**2. Are they the same when x and y are arrays? What if they are matrices?

6. Is x**2+2*x*y+y**2 the same as any of the above?

7. When will x**y for matrices be the same as x**y for vectors?

8. For conformable arrays, is a*b+a*c the same as a*b+c? If so, show with an example. If not, how can the second be changed so they are equal.

9. Suppose a command x**y*w+z was entered. What restrictions on the dimensions of w, x, y and z must be true for this to be a valid statement?

10. What is the value of -2**4? What about (-2)**4? What about -2*-2*-2*-2?

Chapter 6

Basic Functions and Numerical Indexing

6.1 Generating Arrays and Matrices

linspace

linspace(l, u, n) generates a set of n points uniformly spaced between l, a lower bound (inclusive) and u, an upper bound (inclusive).

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

logspace

logspace(l, u, n) produces a set of logarithmically spaced points between 10^l and 10^u. It is identical to 10**linspace(l,u,n).

arange

arange(l, u, s) produces a set of points spaced by s between l, a lower bound (inclusive) and u, an upper bound (exclusive). arange can be used with a single parameter, so that arange(n) is equivalent to arange(0,n,1). Note that arange will return integer data type if all inputs are integer.

```
>>> x = arange(11)
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])

>>> x = arange(11.0)
array([  0.,   1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.])

>>> x = arange(4, 10, 1.25)
array([ 4.  ,  5.25,  6.5 ,  7.75,  9.  ])
```

meshgrid

meshgrid broadcasts two vectors to produce two 2-dimensional arrays, and is a useful function when plotting 3-dimensional functions.

```
>>> x = arange(5)
>>> y = arange(3)
>>> X,Y = meshgrid(x,y)
>>> X
array([[0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4]])

>>> Y
array([[0, 0, 0, 0, 0],
       [1, 1, 1, 1, 1],
       [2, 2, 2, 2, 2]])
```

r_

r_ is a convenience function which generates 1-dimensional arrays from slice notation. While r_ is highly flexible, the most common use it r_[*start* : *end* : *stepOrCount*] where *start* and *end* are the start and end points, and *stepOrCount* can be either a step size, if a real value, or a count, if complex.

```
>>> r_[0:10:1] # arange equiv
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> r_[0:10:.5] # arange equiv
array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5,  5. ,
        5.5,  6. ,  6.5,  7. ,  7.5,  8. ,  8.5,  9. ,  9.5])

>>> r_[0:10:5j] # linspace equiv, includes end point
array([ 0. ,  2.5,  5. ,  7.5, 10. ])
```

r_ can also be used to concatenate slices using commas to separate slice notation blocks.

```
>>> r_[0:2, 7:11, 1:4]
array([ 0,  1,  7,  8,  9, 10,  1,  2,  3])
```

Note that r_ is *not* a function and that is used with [].

c_

c_ is virtually identical to r_ except that column arrays are generates, which are 2-dimensional (second dimension has size 1)

```
>>> c_[0:5:2]
array([[0],
       [2],
       [4]])

>>> c_[1:5:4j]
array([[ 1.        ],
       [ 2.33333333],
       [ 3.66666667],
       [ 5.        ]])
```

c_, like r_, is not a function and is used with [].

ix_

ix_(a, b) constructs an n-dimensional open mesh from n 1-dimensional lists or arrays. The output of ix_ is an n-element tuple containing 1-dimensional arrays. The primary use of ix_ is to simplify selecting slabs inside a matrix. Slicing can also be used to select elements from an array as long as the slice pattern is regular. ix_ is particularly useful for selecting elements from an array using indices which are not regularly spaced, as in the final example.

```
>>> x = reshape(arange(25.0),(5,5))
>>> x
array([[  0.,   1.,   2.,   3.,   4.],
       [  5.,   6.,   7.,   8.,   9.],
       [ 10.,  11.,  12.,  13.,  14.],
       [ 15.,  16.,  17.,  18.,  19.],
       [ 20.,  21.,  22.,  23.,  24.]])

>>> x[ix_([2,3],[0,1,2])] # Rows 2 & 3, cols 0, 1 and 2
array([[ 10.,  11.,  12.],
       [ 15.,  16.,  17.]])

>>> x[2:4,:3] # Same, standard slice
array([[ 10.,  11.,  12.],
       [ 15.,  16.,  17.]])

>>> x[ix_([0,3],[0,1,4])] # No slice equiv
```

mgrid

mgrid is very similar to meshgrid but behaves like r_ and c_ in that it takes slices as input, and uses a real valued variable to denote step size and complex to denote number of values. The output is an $n + 1$ dimensional vector where the first index of the output indexes the meshes.

```
>>> mgrid[0:3,0:2:.5]
array([[[ 0. ,  0. ,  0. ,  0. ],
        [ 1. ,  1. ,  1. ,  1. ],
        [ 2. ,  2. ,  2. ,  2. ]],

       [[ 0. ,  0.5,  1. ,  1.5],
        [ 0. ,  0.5,  1. ,  1.5],
        [ 0. ,  0.5,  1. ,  1.5]]])

>>> mgrid[0:3:3j,0:2:5j]
array([[[ 0. ,  0. ,  0. ,  0. ,  0. ],
        [ 1.5,  1.5,  1.5,  1.5,  1.5],
        [ 3. ,  3. ,  3. ,  3. ,  3. ]],

       [[ 0. ,  0.5,  1. ,  1.5,  2. ],
        [ 0. ,  0.5,  1. ,  1.5,  2. ],
        [ 0. ,  0.5,  1. ,  1.5,  2. ]]])
```

ogrid

ogrid is identical to mgrid except that the arrays returned are always 1-dimensional. ogrid output is generally more appropriate for looping code, while mgrid is usually more appropriate for vectorized code. When the size of the arrays is large, then ogrid uses much less memory.

```
>>> ogrid[0:3,0:2:.5]
[array([[ 0.],
       [ 1.],
       [ 2.]]), array([[ 0. ,  0.5,  1. ,  1.5]])]
>>> ogrid[0:3:3j,0:2:5j]
[array([[ 0. ],
       [ 1.5],
       [ 3. ]]),
 array([[ 0. ,  0.5,  1. ,  1.5,  2. ]])]
```

6.2 Rounding

around, round

around rounds to the nearest integer, or to a particular decimal place when called with two arguments.

```
>>> x = randn(3)
array([ 0.60675173, -0.3361189 , -0.56688485])

>>> around(x)
array([ 1.,  0., -1.])

>>> around(x, 2)
array([ 0.61, -0.34, -0.57])
```

around can also be used as a method on an ndarray – except that the method is named round. For example, x.round(2) is identical to around(x, 2). The change of names is needed to avoid conflicting with the Python built-in function round.

floor

floor rounds to the next smallest integer.

```
>>> x = randn(3)
array([ 0.60675173, -0.3361189 , -0.56688485])

>>> floor(x)
array([ 0., -1., -1.])
```

ceil

ceil rounds to the next largest integer.

```
>>> x = randn(3)
array([ 0.60675173, -0.3361189 , -0.56688485])

>>> ceil(x)
array([ 1., -0., -0.])
```

Note that the values returned are still floating points and so -0. is the same as 0..

6.3 Mathematics

sum, cumsum

sum sums elements in an array. By default, it will sum all elements in the array, and so the second argument is normally used to provide the axis to use – 0 to sum down columns, 1 to sum across rows. cumsum produces the cumulative sum of the values in the array, and is also usually used with the second argument to indicate the axis to use.

```
>>> x = randn(3,4)
>>> x
array([[-0.08542071, -2.05598312,  2.1114733 ,  0.7986635 ],
       [-0.17576066,  0.83327885, -0.64064119, -0.25631728],
       [-0.38226593, -1.09519101,  0.29416551,  0.03059909]])

>>> sum(x) # all elements
-0.62339964288008698

>>> sum(x, 0) # Down rows, 4 elements
array([-0.6434473 , -2.31789529,  1.76499762,  0.57294532])

>>> sum(x, 1) # Across columns, 3 elements
array([ 0.76873297, -0.23944028, -1.15269233])

>>> cumsum(x,0) # Down rows
array([[-0.08542071, -2.05598312,  2.1114733 ,  0.7986635 ],
       [-0.26118137, -1.22270427,  1.47083211,  0.54234622],
       [-0.6434473 , -2.31789529,  1.76499762,  0.57294532]])
```

sum and cumsum can both be used as function or as methods. When used as methods, the first input is the axis so that sum(x,0) is the same as x.sum(0).

prod, cumprod

prod and cumprod behave similarly to sum and cumsum except that the product and cumulative product are returned. prod and cumprod can be called as function or methods.

diff

diff computes the finite difference of a vector (also array) and returns n-1 an element vector when used on an n element vector. diff operates on the last axis by default, and so diff(x) operates across columns and returns x[:,1:size(x,1)]-x[:,:size(x,1)-1] for a 2-dimensional array. diff takes an optional keyword

argument `axis` so that `diff(x, axis=0)` will operate across rows. `diff` can also be used to produce higher order differences (e.g. double difference).

```
>>> x= randn(3,4)
>>> x
array([[-0.08542071, -2.05598312,  2.1114733 ,  0.7986635 ],
       [-0.17576066,  0.83327885, -0.64064119, -0.25631728],
       [-0.38226593, -1.09519101,  0.29416551,  0.03059909]])

>>> diff(x) # Same as diff(x,1)
-0.6233964288008698

>>> diff(x, axis=0)
array([[-0.09033996,  2.88926197, -2.75211449, -1.05498078],
       [-0.20650526, -1.92846986,  0.9348067 ,  0.28691637]])

>>> diff(x, 2, axis=0) # Double difference, column-by-column
array([[-0.11616531, -4.81773183,  3.68692119,  1.34189715]])
```

exp

`exp` returns the element-by-element exponential (e^x) for an array.

log

`log` returns the element-by-element natural logarithm ($\ln(x)$) for an array.

log10

`log10` returns the element-by-element base-10 logarithm ($\log_{10}(x)$) for an array.

sqrt

`sqrt` returns the element-by-element square root ($\sqrt{x}$) for an array.

square

`square` returns the element-by-element square (x^2) for an array, and is equivalent to calling `x**2.0` when x is an array (but not a `matrix`)

absolute, abs

`abs` and `absolute` returns the element-by-element absolute value for an array. Complex modulus is returned when the input is complex valued ($|a + bi| = \sqrt{a^2 + b^2}$).

sign

`sign` returns the element-by-element sign function, defined as 0 if $x = 0$, and $x/|x|$ otherwise.

6.4 Complex Values

real

real returns the real elements of a complex array. real can be called either as a function real(x) or as an attribute x.real.

imag

imag returns the complex elements of a complex array. imag can be called either as a function imag(x) or as an attribute x.imag.

conj, conjugate

conj returns the element-by-element complex conjugate for a complex array. conj can be called either as a function conj(x) or as a method x.conj(). conjugate is identical to conj.

6.5 Set Functions

unique

unique returns the unique elements in an array. It only operates on the entire array. An optional second argument can be returned which contains the original indices of the unique elements.

```
>>> x = repeat(randn(3),(2))
array([ 0.11335982,  0.11335982,  0.26617443,  0.26617443,  1.34424621,
        1.34424621])

>>> unique(x)
array([ 0.11335982,  0.26617443,  1.34424621])

>>> y,ind = unique(x, True)
>>> ind
array([0, 2, 4], dtype=int64)

>>> x.flat[ind]
array([ 0.11335982,  0.26617443,  1.34424621])
```

in1d

in1d returns a Boolean array with the same size as the first input array indicating the elements which are also in a second array.

```
>>> x = arange(10.0)
>>> y = arange(5.0,15.0)
>>> in1d(x,y)
array([False, False, False, False, False,  True,  True,  True,  True,  True], dtype=bool)
```

intersect1d

intersect1d is similar to in1d, except that it returns the elements rather than a Boolean array, and only unique elements are returned. It is equivalent to unique(x.flat[in1d(x,y)]).

```
>>> x = arange(10.0)
>>> y = arange(5.0,15.0)
>>> intersect1d(x,y)
array([ 5.,  6.,  7.,  8.,  9.])
```

union1d

union1d returns the unique set of elements in 2 arrays.

```
>>> x = arange(10.0)
>>> y = arange(5.0,15.0)
>>> union1d(x,y)
array([  0.,   1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.,
        11.,  12.,  13.,  14.])
```

setdiff1d

setdiff1d returns the set of the elements which are only in the first array but not in the second array.

```
>>> x = arange(10.0)
>>> y = arange(5.0,15.0)
>>> setdiff1d(x,y)
array([ 0.,  1.,  2.,  3.,  4.])
```

setxor1d

setxor1d returns the set of elements which are in one (and only one) of two arrays.

```
>>> x = arange(10.0)
>>> y = arange(5.0,15.0)
>>> setxor1d(x,y)
array([  0.,   1.,   2.,   3.,   4.,  10.,  11.,  12.,  13.,  14.])
```

6.6 Sorting and Extreme Values

sort

sort sorts the elements of an array. By default, it sorts using the last axis of x. It uses an optional second argument to indicate the axis to use for sorting (i.e. 0 for column-by-column, None for sorting all elements). sort does not alter the input when called as function, unlike the method version of sort.

```
>>> x = randn(4,2)
>>> x
array([[ 1.29185667,  0.28150618],
       [ 0.15985346, -0.93551769],
```

```
        [ 0.12670061,   0.6705467 ],
        [ 2.77186969, -0.85239722]])

>>> sort(x)
array([[ 0.28150618,   1.29185667],
       [-0.93551769,   0.15985346],
       [ 0.12670061,   0.6705467 ],
       [-0.85239722,   2.77186969]])

>>> sort(x, 0)
array([[ 0.12670061, -0.93551769],
       [ 0.15985346, -0.85239722],
       [ 1.29185667,  0.28150618],
       [ 2.77186969,  0.6705467 ]])

>>> sort(x, axis=None)
array([-0.93551769, -0.85239722,  0.12670061,  0.15985346,  0.28150618,
         0.6705467 ,  1.29185667,  2.77186969])
```

`ndarray.sort`, `argsort`

`ndarray.sort` is a method for ndarrays which performs an in-place sort. It economizes on memory use, although x.`sort`() is different from x after the function, unlike a call to `sort`(x). x.`sort`() sorts along the last axis by default, and takes the same optional arguments as `sort`(x). argsort returns the indices necessary to produce a sorted array, but does not actually sort the data. It is otherwise identical to `sort`, and can be used either as a function or a method.

```
>>> x = randn(3)
>>> x
array([ 2.70362768, -0.80380223, -0.10376901])

>>> sort(x)
array([-0.80380223, -0.10376901,  2.70362768])

>>> x
array([ 2.70362768, -0.80380223, -0.10376901])

>>> x.sort() # In-place, changes x
>>> x
array([-0.80380223, -0.10376901,  2.70362768])
```

`max, amax, argmax, min, amin, argmin`

`max` and `min` return the maximum and minimum values from an array. They take an optional second argument which indicates the axis to use.

```
>>> x = randn(3,4)
>>> x
array([[-0.71604847,  0.35276614, -0.95762144,  0.48490885],
```

```
        [-0.47737217,  1.57781686, -0.36853876,  2.42351936],
        [ 0.44921571, -0.03030771,  1.28081091, -0.97422539]])

>>> amax(x)
2.4235193583347918

>>> x.max()
2.4235193583347918

>>> x.max(0)
array([ 0.44921571,  1.57781686,  1.28081091,  2.42351936])

>>> x.max(1)
array([ 0.48490885,  2.42351936,  1.28081091])
```

max and min can only be used on arrays as methods. When used as a function, amax and amin must be used to avoid conflicts with the built-in functions max and min. This behavior is also seen in around and round. argmax and argmin return the index or indices of the maximum or minimum element(s). They are used in an identical manner to max and min, and can be used either as a function or method.

minimum, maximum

maximum and minimum can be used to compute the maximum and minimum of two arrays which are broadcastable.

```
>>> x = randn(4)
>>> x
array([-0.00672734,  0.16735647,  0.00154181, -0.98676201])

>>> y = randn(4)
array([-0.69137963, -2.03640622,  0.71255975, -0.60003157])

>>> maximum(x,y)
array([-0.00672734,  0.16735647,  0.71255975, -0.60003157])
```

6.7   Nan Functions

NaN function are convenience function which act similarly to their non-NaN versions, only ignoring NaN values (rather than propagating) when computing the function.

nansum

nansum is identical sum, except that NaNs are ignored. nansum can be used to easily generate other NaN-functions, such as nanstd (standard deviation, ignoring nans) since variance can be implemented using 2 sums.

```
>>> x = randn(4)
>>> x[1] = nan
>>> x
```

```
array([-0.00672734,        nan,  0.00154181, -0.98676201])

>>> sum(x)
nan

>>> nansum(x)
-0.99194753275859726

>>> nansum(x) / sum(x[logical_not(isnan(x))])
1.0

>>> nansum(x) / sum(1-isnan(x)) # nanmean
-0.33064917999999999
```

nanmax, nanargmax, nanmin, nanargmin

nanmax, nanmin, nanargmax and nanargmin are identical to their non-NaN counterparts, except that NaNs
are ignored.

6.8   Functions and Methods/Properties

Many operations on NumPy arrays and matrices can be performed using a function or as a method of the
array. For example, consider reshape.

```
>>> x = arange(25.0)
>>> y = x.reshape((5,5))
>>> y
array([[  0.,   1.,   2.,   3.,   4.],
       [  5.,   6.,   7.,   8.,   9.],
       [ 10.,  11.,  12.,  13.,  14.],
       [ 15.,  16.,  17.,  18.,  19.],
       [ 20.,  21.,  22.,  23.,  24.]])

>>> z = reshape(x,(5,5))
>>> z
array([[  0.,   1.,   2.,   3.,   4.],
       [  5.,   6.,   7.,   8.,   9.],
       [ 10.,  11.,  12.,  13.,  14.],
       [ 15.,  16.,  17.,  18.,  19.],
       [ 20.,  21.,  22.,  23.,  24.]])
```

Both the function and method produce the same output and the choice of which to use is ultimately a
personal decision. I use both and the choice primarily depends on the context. For example, to get the
shape of an array, my preference is for x.shape over shape(x) since shape appears to be integral to x.[1] On
the other hand, I prefer shape(y+z) over (y+z).shape due to the presence of the mathematical operation.

---

[1]Formally shape is a property of an array, not a method since it does not require a function call.

6.9 Exercises

1. Construct each of the following sequences using `linspace`, `arange` and `r_`:

$$0, 1, \ldots, 10$$

$$4, 5, 6, \ldots, 13$$

$$0, .25, .5, .75, 1$$

$$0, -1, -2, \ldots, -5$$

2. Show that `logspace(0,2,21)` can be constructed using `linspace` and 10 (and `**`). Similarly, show how `linsapce(2,10,51)` can be constructed with `logspace` and `log10`.

3. Determine the differences between the rounding by applying `round` (or `around`), `ceil` and `floor` to

$$y = [0, 0.5, 1.5, 2.5, 1.0, 1.0001, -0.5, -1, -1.5, -2.5]$$

4. Prove the relationship that $\sum_{j=1}^{n} j = n(n+1)/2$ for $0 \le n \le 10$ using `cumsum` and directly using math on an array.

5. `randn(20)` will generate an array containing draws from a standard normal random variable. If `x=randn(20)`, which element of `y=cumsum(x)` is the same as `sum(x)`?

6. `cumsum` computes the cumulative sum while `diff` computes the difference. Is `diff(cumsum(x))` the same as x? If not, how can a small modification be made to the this statement to recover x?

7. Compute the `exp` of
$$y = [\ln 0.5 \ \ln 1 \ \ln e]$$

   Note: You should use `log` and the constant `numpy.e` to construct y.

8. What is `absolute` of `0.0`, `-3.14`, and `3+4j`?

9. Suppose $x = [-4\ 2\ -9\ -8\ 10]$. What is the difference between `y = sort(x)` and `x.sort()`?

10. Using the same x as in the previous problem, find the `max`. Also, using `argmax` and a slice, retrieve the same value.

11. Show that `setdiff1d` could be replaced with `in1d` and `intersect1d` using $x = [1\ 2\ 3\ 4\ 5]$ and $y = [1\ 2\ 4\ 6]$? How could `setxor1d` be replaced with `union1d`, `intersect1d` and `in1d`?

12. Suppose $y = [\text{nan}\ 2.2\ 3.9\ 4.6\ \text{nan}\ 2.4\ 6.1\ 1.8]$. How can `nansum` be used to compute the variance or the data? Note: `sum(1-isnan(y))` will return the count of non-NaN values.

Chapter 7

Special Arrays

Functions are available to construct a number of useful, frequently encountered arrays.

ones

ones generates an array of 1s and is generally called with one argument, a tuple, containing the size of each dimension. ones takes an optional second argument (dtype) to specify the data type. If omitted, the data type is float.

```
>>> M, N = 5, 5
>>> x = ones((M,N)) # M by N array of 1s
>>> x =  ones((M,M,N)) # 3D array
>>> x =  ones((M,N), dtype='int32') # 32-bit integers
```

ones_like creates an array with the same shape and data type as the input. Calling ones_like(x) is equivalent to calling ones(x.shape,x.dtype).

zeros

zeros produces an array of 0s in the same way ones produces an array of 1s, and commonly used to initialize an array to hold values generated by another procedure. zeros takes an optional second argument (dtype) to specify the data type. If omitted, the data type is float.

```
>>> x = zeros((M,N)) # M by N array of 0s
>>> x = zeros((M,M,N)) # 3D array of 0s
>>> x = zeros((M,N),dtype='int64') # 64 bit integers
```

zeros_like creates an array with the same size and shape as the input. Calling zeros_like(x) is equivalent to calling zeros(x.shape,x.dtype).

empty

empty produces an empty (uninitialized) array to hold values generated by another procedure. empty takes an optional second argument (dtype) which specifies the data type. If omitted, the data type is float.

```
>>> x = empty((M,N)) # M by N empty array
>>> x = empty((N,N,N,N)) # 4D empty array
>>> x = empty((M,N),dtype='float32') # 32-bit floats (single precision)
```

Using empty is slightly faster than calling zeros since it does not assign 0 to all elements of the array – the "empty" array created will be populated with (essential random) non-zero values. empty_like creates an array with the same size and shape as the input. Calling empty_like(x) is equivalent to calling empty(x.shape,x.dtype).

eye, identity

eye generates an identity array – an array with ones on the diagonal, zeros everywhere else. Normally, an identity array is square and so usually only 1 input is required. More complex zero-padded arrays containing an identity matrix can be produced using optional inputs.

```
>>> In = eye(N)
```

identity is a virtually identical function with similar use, In = identity(N).

7.1 Exercises

1. Produce two arrays, one containing all zeros and one containing only ones, of size 10×5.

2. Multiply (linear algebra) these two arrays in both possible ways.

3. Produce an identity matrix of size 5. Take the exponential of this matrix, element-by-element.

4. How could ones and zeros be replaced with tile?

5. How could eye be replaced with diag and ones?

6. What is the value of y=empty((1,))? Is it the same as any element in y=empty((10,))?

Chapter 8

Array and Matrix Functions

Many functions operate exclusively on array inputs, including functions which are mathematical in nature, for example computing the eigenvalues and eigenvectors and functions for manipulating the elements of an array.

8.1 Views

Views are computationally efficient methods to produce objects of one type which behave as other objects of another type *without* copying data. For example, an array x can always be converted to a matrix using matrix(x), which will copy the elements in x. View "fakes" the call to matrix and only inserts a thin layer so that x viewed as a matrix behaves like a matrix.

view

view can be used to produce a representation of an array, matrix or recarray as another type without copying the data. Using view is faster than copying data into a new class.

```
>>> x = arange(5)
>>> type(x)
numpy.ndarray

>>> x.view(matrix)
matrix([[0, 1, 2, 3, 4]])

>>> x.view(recarray)
rec.array([0, 1, 2, 3, 4])
```

asmatrix, mat

asmatrix and mat can be used to view an array as a matrix. This view is useful since matrix views will use matrix multiplication by default.

```
>>> x = array([[1,2],[3,4]])
>>> x * x            # Element-by-element
array([[ 1,  4],
       [ 9, 16]])
```

```
>>> mat(x) * mat(x) # Matrix multiplication
matrix([[ 7, 10],
        [15, 22]])
```

Both commands are equivalent to using view(matrix).

asarray

asarray work in a similar matter as asmatrix, only that the view produced is that of ndarray. Calling asarray is equivalent to using view(ndarray)

8.2 Shape Information and Transformation

shape

shape returns the size of all dimensions or an array or matrix as a tuple. shape can be called as a function or an attribute. shape can also be used to reshape an array by entering a tuple of sizes. Additionally, the new shape can contain -1 which indicates to expand along this dimension to satisfy the constraint that the number of elements cannot change.

```
>>> x = randn(4,3)
>>> x.shape
(4L, 3L)

>>> shape(x)
(4L, 3L)

>>> M,N = shape(x)
>>> x.shape = 3,4
>>> x.shape
 (3L, 4L)

>>> x.shape = 6,-1
>>> x.shape
(6L, 2L)
```

reshape

reshape transforms an array with one set of dimensions and to one with a different set, preserving the number of elements. Arrays with dimensions M by N can be reshaped into an array with dimensions K by L as long as $MN = KL$. The most useful call to reshape switches an array into a vector or vice versa.

```
>>> x = array([[1,2],[3,4]])
>>> y = reshape(x,(4,1))
>>> y
array([[1],
       [2],
       [3],
```

```
        [4]])

>>> z=reshape(y,(1,4))
>>> z
array([[1, 2, 3, 4]])

>>> w = reshape(z,(2,2))
array([[1, 2],
       [3, 4]])
```

The crucial implementation detail of reshape is that arrays are stored using row-major notation. Elements in arrays are counted first across rows and then then down columns. reshape will place elements of the old array into the same position in the new array and so after calling reshape, $x(1) = y(1)$, $x(2) = y(2)$, and so on.

size

size returns the total number of elements in an array or matrix. size can be used as a function or an attribute.

```
>>> x = randn(4,3)
>>> size(x)
12

>>> x.size
12
```

ndim

ndim returns the size of all dimensions or an array or matrix as a tuple. ndim can be used as a function or an attribute .

```
>>> x = randn(4,3)
>>> ndim(x)
2

>>> x.ndim
2
```

tile

tile, along with reshape, are two of the most useful non-mathematical functions. tile replicates an array according to a specified size vector. To understand how tile functions, imagine forming an array composed of blocks. The generic form of tile is $\text{tile}(X, (M, N))$ where X is the array to be replicated, M is the number of rows in the new block array, and N is the number of columns in the new block array. For example, suppose X was an array

$$X = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

83

and the block array

$$Y = \begin{bmatrix} X & X & X \\ X & X & X \end{bmatrix}$$

was required. This could be accomplished by manually constructing y using hstack and vstack.

```
>>> x = array([[1,2],[3,4]])
>>> z = hstack((x,x,x))
>>> y = vstack((z,z))
```

However, tile provides a much easier method to construct y

```
>>> w = tile(x,(2,3))
>>> y - w
array([[0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0]])
```

tile has two clear advantages over manual allocation: First, tile can be executed using parameters determined at run-time, such as the number of explanatory variables in a model and second tile can be used for arbitrary dimensions. Manual array construction becomes tedious and error prone with as few as 3 rows and columns. repeat is a related function which copies data is a less useful manner.

ravel

ravel returns a flattened view (1-dimensional) of an array or matrix. ravel does not copy the underlying data (when possible), and so it is very fast.

```
>>> x = array([[1,2],[3,4]])
>>> x
array([[ 1,  2],
       [ 3,  4]])

>>> x.ravel()
array([1, 2, 3, 4])

>>> x.T.ravel()
array([1, 3, 2, 4])
```

flatten

flatten works much like ravel, only that is copies the array when producing the flattened version.

flat

flat produces a numpy.flatiter object (flat iterator) which is an iterator over a flattened view of an array. Because it is an iterator, it is especially fast and memory friendly. flat can be used as an iterator in a for loop or with slicing notation.

```
>>> x = array([[1,2],[3,4]])
>>> x.flat
<numpy.flatiter at 0x6f569d0>

>>> x.flat[2]
3

>>> x.flat[1:4] = -1
>>> x
array([[ 1, -1],
       [-1, -1]])
```

broadcast, broadcast_arrays

broadcast can be used to broadcast two broadcastable arrays without actually copying any data. It returns a broadcast object, which works like an iterator.

```
>>> x = array([[1,2,3,4]])
>>> y = reshape(x,(4,1))
>>> b = broadcast(x,y)
>>> b.shape
(4L, 4L)

>>> for u,v in b:
...     print('x: ', u, ' y: ',v)
x:  1  y:  1
x:  2  y:  1
x:  3  y:  1
x:  4  y:  1
x:  1  y:  2
 ... ... ...
```

broadcast_arrays works similarly to broadcast, except that it copies the broadcast arrays into new arrays.
broadcast_arrays is generally slower than broadcast, and should be avoided if possible.

```
>>> x = array([[1,2,3,4]])
>>> y = reshape(x,(4,1))
>>> b = broadcast_arrays(x,y)
>>> b[0]
array([[1, 2, 3, 4],
       [1, 2, 3, 4],
       [1, 2, 3, 4],
       [1, 2, 3, 4]])

>>> b[1]
array([[1, 1, 1, 1],
       [2, 2, 2, 2],
       [3, 3, 3, 3],
       [4, 4, 4, 4]])
```

vstack, hstack

vstack, and hstack stack compatible arrays and matrices vertically and horizontally, respectively. Arrays are vstack compatible if they have the same number of columns, and are hstack compatible if they have the same number of rows. Any number of arrays can be stacked by placing the input arrays in a list or tuple, e.g. (x,y,z).

```
>>> x = reshape(arange(6),(2,3))
>>> y = x
>>> vstack((x,y))
array([[0, 1, 2],
       [3, 4, 5],
       [0, 1, 2],
       [3, 4, 5]])

>>> hstack((x,y))
array([[0, 1, 2, 0, 1, 2],
       [3, 4, 5, 3, 4, 5]])
```

concatenate

concatenate generalizes vstack and hsplit to allow concatenation along any axis using the keyword argument axis.

split, vsplit, hsplit

vsplit and hsplit split arrays and matrices vertically and horizontally, respectively. Both can be used to split an array into n equal parts or into arbitrary segments, depending on the second argument. If scalar, the array is split into n equal sized parts. If a 1 dimensional array, the array is split using the elements of the array as break points. For example, if the array was [2,5,8], the array would be split into 4 pieces using [:2], [2:5], [5:8] and [8:]. Both vsplit and hsplit are special cases of split, which can split along an arbitrary axis.

```
>>> x = reshape(arange(20),(4,5))
>>> y = vsplit(x,2)
>>> len(y)
2

>>> y[0]
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])

>>> y = hsplit(x,[1,3])
>>> len(y)
3

>>> y[0]
array([[ 0],
       [ 5],
```

```
       [10],
       [15]])

>>> y[1]
array([[ 1,  2],
       [ 6,  7],
       [11, 12],
       [16, 17]])
```

delete

delete removes values from an array, and is similar to splitting an array, and then concatenating the values which are not deleted. The form of delete is delete(x, *rc*, *axis*) where *rc* are the row or column indices to delete, and *axis* is the axis to use (0 or 1 for a 2-dimensional array). If *axis* is omitted, delete operated on the flattened array.

```
>>> x = reshape(arange(20),(4,5))
>>> delete(x,1,0) # Same as x[[0,2,3]]
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])

>>> delete(x,[2,3],1) # Same as x[:,[0,1,4]]
array([[ 0,  1,  4],
       [ 5,  6,  9],
       [10, 11, 14],
       [15, 16, 19]])

>>> delete(x,[2,3]) # Same as hstack((x.flat[:2],x.flat[4:]))
array([ 0,  1,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
       19])
```

squeeze

squeeze removes singleton dimensions from an array, and can be called as a function or a method.

```
>>> x = ones((5,1,5,1))
>>> shape(x)
(5L, 1L, 5L, 1L)

>>> y = x.squeeze()
>>> shape(y)
(5L, 5L)

>>> y = squeeze(x)
```

fliplr, flipud

fliplr and flipud flip arrays in a left-to-right and up-to-down directions, respectively. flipud reverses the elements in a 1-dimensional array, and flipud(x) is identical to x[::-1]. fliplr cannot be used with

1-dimensional arrays.

```
>>> x = reshape(arange(4),(2,2))
>>> x
array([[0, 1],
       [2, 3]])

>>> fliplr(x)
array([[1, 0],
       [3, 2]])

>>> flipud(x)
array([[2, 3],
       [0, 1]])
```

diag

The behavior of `diag` differs depending depending on the form of the input. If the input is a square array, it will return a column vector containing the elements of the diagonal. If the input is an vector, it will return an array containing the elements of the vector along its diagonal. Consider the following example:

```
>>> x = array([[1,2],[3,4]])
>>> x
array([[1, 2],
       [3, 4]])

>>> y = diag(x)
>>> y
array([1, 4])

>>> z = diag(y)
>>> z
array([[1, 0],
       [0, 4]])
```

triu, tril

`triu` and `tril` produce upper and lower triangular arrays, respectively.

```
>>> x = array([[1,2],[3,4]])
>>> triu(x)
array([[1, 2],
       [0, 4]])

>>> tril(x)
array([[1, 0],
       [3, 4]])
```

8.3 Linear Algebra Functions

matrix_power

matrix_power raises a square array or matrix to an integer power, and matrix_power(x,n) is identical to x**n.

svd

svd computes the singular value decomposition of a matrix X, defined as

$$X = U\Sigma V$$

where Σ is diagonal, and U and V are unitary arrays (orthonormal if real valued). SVDs are closely related to eigenvalue decompositions when X is a real, positive definite matrix. The returned value is a tuple containing (U,s,V) where $\Sigma = \text{diag}(s)$.

cond

cond computes the condition number of a matrix, which measures how close to singular a matrix is. Lower numbers indicate that the input is better conditioned (further from singular).

```
>>> x = matrix([[1.0,0.5],[.5,1]])
>>> cond(x)
3
>>> x = matrix([[1.0,2.0],[1.0,2.0]]) # Singular
>>> cond(x)
inf
```

slogdet

slogdet computes the sign and log of the absolute value of the determinant. slogdet is useful for computing determinants which may be very large or small to avoid numerical problems.

solve

solve solves the system $X\beta = y$ when X is square and invertible so that the solution is exact.

```
>>> X = array([[1.0,2.0,3.0],[3.0,3.0,4.0],[1.0,1.0,4.0]])
>>> y = array([[1.0],[2.0],[3.0]])
>>> solve(X,y)
array([[ 0.625],
       [-1.125],
       [ 0.875]])
```

lstsq

lstsq solves the system $X\beta = y$ when X is n by k, $n > k$ by finding the least squares solution. lstsq returns a 4-element tuple where the first element is β and the second element is the sum of squared residuals. The final two outputs are diagnostic – the third is the rank of X and the fourth contains the singular values of X.

```
>>> X = randn(100,2)
>>> y = randn(100)
>>> lstsq(X,y)
(array([ 0.03414346,  0.02881763]),
 array([ 3.59331858]),
 2,
 array([ 3.045516  ,  1.99327863]))array([[ 0.625],
        [-1.125],
        [ 0.875]])
```

cholesky

cholesky computes the Cholesky factor of a positive definite matrix or array. The Cholesky factor is a lower triangular matrix and is defined as C in

$$CC' = \Sigma$$

where Σ is a positive definite matrix.

```
>>> x = matrix([[1,.5],[.5,1]])
>>> C = cholesky(x)
>>> C*C.T - x
matrix([[ 1. ,  0.5],
        [ 0.5,  1. ]])
```

det

det computes the determinant of a square matrix or array.

```
>>> x = matrix([[1,.5],[.5,1]])
>>> det(x)
0.75
```

eig

eig computes the eigenvalues and eigenvectors of a square matrix. When used with one output, the eigenvalues and eigenvectors are returned as a tuple.

```
>>> x = matrix([[1,.5],[.5,1]])
>>> val,vec = eig(x)
>>> vec*diag(val)*vec.T
matrix([[ 1. ,  0.5],
        [ 0.5,  1. ]])
```

eigvals can be used if only eigenvalues are needed.

eigh

eigh computes the eigenvalues and eigenvectors of a symmetric array. When used with one output, the eigenvalues and eigenvectors are returned as a tuple. eigh is faster than eig for symmetrix inputs since it exploits the symmetry of the input. eigvalsh can be used if only eigenvalues are needed from a symmetric array.

inv

inv computes the inverse of an array. inv(R) can alternatively be computed using x**(-1) when x is a matrix.

```
>>> x = array([[1,.5],[.5,1]])
>>> xInv = inv(x)
>>> dot(x,xInv)
array([[ 1.,  0.],
       [ 0.,  1.]])

>>> x = asmatrix(x)
>>> x**(-1)*x
matrix([[ 1.,  0.],
        [ 0.,  1.]])
```

kron

kron computes the Kronecker product of two arrays,

$$z = x \otimes y$$

and is written as z = kron(x,y).

trace

trace computes the trace of a square array (sum of diagonal elements). trace(x) equals sum(diag(x)).

matrix_rank

matrix_rank computes the rank of an array using a SVD.

```
>>> x = array([[1,.5],[1,.5]])
>>> x
array([[ 1. ,  0.5],
       [ 1. ,  0.5]])
>>> matrix_rank(x)
1
```

8.4 Exercises

1. Let x = `arange(12.0)`. Use both `shape` and `reshape` to produce 1×12, 2×6, 3×4, 4×3, 6×2 and $2 \times 2 \times 3$ versions or the array. Finally, return x to its original size.

2. Let x = `reshape(arange(12.0),(4,3))`. Use `ravel`, `flatten` and `flat` to extract elements $1, 3, \ldots, 11$ from the array (using a 0 index).

3. Let x be 2 by 2 array, y be a 1 by 1 array, and z be a 3 by 2 array. Construct

$$
w = \begin{bmatrix}
x & y & y & y \\
  & y & y & y \\
  &   & z' &   \\
z &   &   &   \\
  & y & y & y
\end{bmatrix}
$$

   using `hstack`, `vstack`, and `tile`.

4. Let x = `reshape(arange(12.0),(2,2,3))`. What does `squeeze` do to x?

5. How can a diagonal matrix containing the diagonal elements of

$$
y = \begin{bmatrix}
2 & .5 \\
.5 & 4
\end{bmatrix}
$$

   be constructed using `diag`?

6. Using the y array from the previous problem, verify that `cholesky` work by computing the Cholesky factor, and then multiplying to get y again.

7. Using the y array from the previous problem, verify that the sum of the eigenvalues is the same as the trace, and the product of the eigenvalues is the determinant.

8. Using the y array from the previous problem, verify that the inverse of y is equal to $VD^{-1}V'$ where V is the array containing the eigenvectors, and D is a diagonal array containing the eigenvalues.

9. Simulate some data where x = `randn(100,2)`, e = `randn(100,1)`, B = `array([[1],[0.5]])` and $y = x\beta + \epsilon$. Use `lstsq` to estimate β from x and y.

10. Suppose

$$
y = \begin{bmatrix}
5 & -1.5 & -3.5 \\
-1.5 & 2 & -0.5 \\
-3.5 & -0.5 & 4
\end{bmatrix}
$$

   use `matrix_rank` to determine the rank of this array. Verify the results by inspecting the eigenvalues using `eig` and check that the determinant is 0 using `det`.

11. Let x = `randn(100,2)`. Use `kron` to compute

$$
I_2 \otimes \Sigma_X
$$

92

where Σ_X is the 2 by 2 covariance matrix of x.

Chapter 9

Importing and Exporting Data

9.1 Importing Data using pandas

Pandas is an increasingly important component of the Python scientific stack, and a complete discussion of its main features is included in Chapter 17. All of the data readers in pandas load data into a pandas DataFrame (see Section 17.1.2), and so these examples all make use of the `values` property to extract a NumPy array. In practice, the DataFrame is much more useful since it includes useful information such as column names read from the data source. In addition to the three formats presented here, pandas can also read json, SQL, html tables or from the clipboard, which is particularly useful for interactive work since virtually any source that can be copied to the clipboard can be imported.

9.1.1 CSV and other formatted text files

Comma-separated value (CSV) files can be read using `read_csv`. When the CSV file contains mixed data, the default behavior will read the file into an array with an `object` data type, and so further processing is usually required to extract the individual series.

```
>>> from pandas import read_csv
>>> csv_data = read_csv('FTSE_1984_2012.csv')
>>> csv_data = csv_data.values
>>> csv_data[:4]
array([['2012-02-15', 5899.9, 5923.8, 5880.6, 5892.2, 801550000L, 5892.2],
       ['2012-02-14', 5905.7, 5920.6, 5877.2, 5899.9, 832567200L, 5899.9],
       ['2012-02-13', 5852.4, 5920.1, 5852.4, 5905.7, 643543000L, 5905.7],
       ['2012-02-10', 5895.5, 5895.5, 5839.9, 5852.4, 948790200L, 5852.4]], dtype=object)

>>> open = csv_data[:,1]
```

When the entire file is numeric, the data will be stored as a homogeneous array using one of the numeric data types, typically float64. In this example, the first column contains Excel dates as numbers, which are the number of days past January 1, 1900.

```
>>> csv_data = read_csv('FTSE_1984_2012_numeric.csv')
>>> csv_data = csv_data.values
>>> csv_data[:4,:2]
array([[ 40954. ,   5899.9],
       [ 40953. ,   5905.7],
```

95

```
        [ 40952. ,    5852.4],
        [ 40949. ,    5895.5]])
```

9.1.2 Excel files

Excel files, both 97/2003 (xls) and 2007/10/13 (xlsx), can be imported using `read_excel`. Two inputs are required to use `read_excel`, the filename and the sheet name containing the data. In this example, pandas makes use of the information in the Excel workbook that the first column contains dates and converts these to `datetimes`. Like the mixed CSV data, the array returned has `object` data type.

```
>>> from pandas import read_excel
>>> excel_data = read_excel('FTSE_1984_2012.xls','FTSE_1984_2012')
>>> excel_data = excel_data.values
>>> excel_data[:4,:2]
array([[datetime.datetime(2012, 2, 15, 0, 0), 5899.9],
       [datetime.datetime(2012, 2, 14, 0, 0), 5905.7],
       [datetime.datetime(2012, 2, 13, 0, 0), 5852.4],
       [datetime.datetime(2012, 2, 10, 0, 0), 5895.5]], dtype=object)

>>> open = excel_data[:,1]
```

9.1.3 STATA files

Pandas also contains a method to read STATA files.

```
>>> from pandas import read_stata
>>> stata_data = read_stata('FTSE_1984_2012.dta')
>>> stata_data = stata_data.values
>>> stata_data[:4,:2]
array([[  0.00000000e+00,    4.09540000e+04],
       [  1.00000000e+00,    4.09530000e+04],
       [  2.00000000e+00,    4.09520000e+04],
       [  3.00000000e+00,    4.09490000e+04]])
```

9.2 Importing Data without pandas

Importing data without pandas ranges from easy when files contain only numbers to difficult, depending on the data size and format. A few principles can simplify this task:

- The file imported should contain numbers *only*, with the exception of the first row which may contain the variable names.

- Use another program, such as Microsoft Excel, to manipulate data before importing.

- Each column of the spreadsheet should contain a single variable.

- Dates should be converted to YYYYMMDD, a numeric format, before importing. This can be done in Excel using the formula:
  `=10000*YEAR(A1)+100*MONTH(A1)+DAY(A1)+(A1-FLOOR(A1,1))`

- Store times separately from dates using a numeric format such as seconds past midnight or HH-mmSS.sss.

9.2.1 CSV and other formatted text files

A number of importers are available for regular (e.g. all rows have the same number of columns) comma-separated value (CSV) data. The choice of which importer to use depends on the complexity and size of the file. Purely numeric files are the simplest to import, although most files which have a repeated structure can be directly imported (unless they are very large).

loadtxt

loadtxt is a simple, fast text importer. The basic use is loadtxt(*filename*), which will attempt to load the data in file name as floats. Other useful named arguments include delim, which allow the file delimiter to be specified, and skiprows which allows one or more rows to be skipped.

loadtxt requires the data to be numeric and so is only useful for the simplest files.

```
>>> data = loadtxt('FTSE_1984_2012.csv',delimiter=',') # Error
ValueError: could not convert string to float: Date

# Fails since CSV has a header
>>> data = loadtxt('FTSE_1984_2012_numeric.csv',delimiter=',') # Error
ValueError: could not convert string to float: Date

>>> data = loadtxt('FTSE_1984_2012_numeric.csv',delimiter=',',skiprows=1)
>>> data[0]
array([ 4.09540000e+04, 5.89990000e+03, 5.92380000e+03, 5.88060000e+03, 5.89220000e+03,
8.01550000e+08, 5.89220000e+03])
```

genfromtxt

genfromtxt is a slightly slower, more robust importer. genfromtxt is called using the same syntax as loadtxt, but will not fail if a non-numeric type is encountered. Instead, genfromtxt will return a NaN (not-a-number) for fields in the file it cannot read.

```
>>> data = genfromtxt('FTSE_1984_2012.csv',delimiter=',')
>>> data[0]
array([ nan, nan, nan, nan, nan, nan, nan])
>>> data[1]
array([ nan, 5.89990000e+03, 5.92380000e+03, 5.88060000e+03, 5.89220000e+03, 8.01550000e+08,
5.89220000e+03])
```

Tab delimited data can be read in a similar manner using delimiter='\t'.

```
>>> data = genfromtxt('FTSE_1984_2012_numeric_tab.txt',delimiter='\t')
```

csv2rec

csv2rec is an even more robust – and slower – CSV importer which imports non-numeric data such as dates. It attempts to find the best data type for each column. Note that when pandas is available, `read_csv` is a better option than csv2rec.

```
>>> data = csv2rec('FTSE_1984_2012.csv',delimiter=',')
>>> data[0]
(datetime.date(2012, 2, 15), 5899.9, 5923.8, 5880.6, 5892.2, 801550000L, 5892.2)
```

Unlike `loadtxt` and `genfromtxt`, which both return an array, csv2rec returns a record array (see Chapter 16) which is, in many ways, like a list. csv2rec converted each row of the input file into a `datetime` (see Chapter 14), followed by 4 floats for open, high, low and close, then a long integer for volume, and finally a float for the adjusted close. When the data contain non-numeric values, returned array is not homogeneous, and so it is necessary to create an array to store the numeric content of the imported data.

```
>>> open = data['open']
>>> open
array([ 5899.9,  5905.7,  5852.4, ...,  1095.4,  1095.4,  1108.1])
```

9.2.2 Excel Files

xlrd

Reading Excel files in Python is more involved, and it is simpler to convert the xls to CSV. Excel files can be read using xlrd (which is part of xlutils).

```python
from __future__ import print_function
import xlrd

wb = xlrd.open_workbook('FTSE_1984_2012.xls')
# To read xlsx change the filename
# wb = xlrd.open_workbook('FTSE_1984_2012.xlsx')
sheetNames = wb.sheet_names()
# Assumes 1 sheet name
sheet = wb.sheet_by_name(sheetNames[0])
excelData = [] # List to hold data
for i in xrange(sheet.nrows):
    excelData.append(sheet.row_values(i))

# Subtract 1 since excelData has the header row
open = empty(len(excelData) - 1)
for i in xrange(len(excelData) - 1):
    open[i] = excelData[i+1][1]
```

The listing does a few things. First, it opens the workbook for reading (`open_workbook('FTSE_1984_2012.xls')`), then it gets the sheet names (`wb.sheet_names()`) and opens a sheet (`wb.sheet_by_name(sheetNames[0])`). From the sheet, it gets the number of rows (`sheet.nrows`), and fills a list with the values, row-by-row. Once the data has been read-in, the final block fills an array with the opening prices. This is substantially more complicated than importing from a CSV file, although reading Excel files is useful for automated work (e.g. you have no choice but to import from an Excel file since it is produced by some other software).

openpyxl

openpyxl reads and write the modern Excel file format that is the default in Office 2007 or later. openpyxl also supports a reader and writer which is optimized for large files, a feature not available in xlrd. Unfortunately, openpyxl uses a different syntax from xlrd, and so some modifications are required when using openpyxl.

```python
from __future__ import print_function
import openpyxl

wb = openpyxl.load_workbook('FTSE_1984_2012.xlsx')
sheetNames = wb.get_sheet_names()
# Assumes 1 sheet name
sheet = wb.get_sheet_by_name(sheetNames[0])
rows = sheet.rows

# Subtract 1 since excelData has the header row
open = empty(len(rows) - 1)
for i in xrange(len(excelData) - 1):
    open[i] = rows[i+1][1].value
```

The strategy with 2007/10/13 xlsx files is essentially the same as with 97/2003 files. The main difference is that the command sheet.rows() returns a tuple containing the all of the rows in the selected sheet. Each row is itself a tuple which contains Cells (which are a type created by openpyxl), and each cell has a value (Cells also have other useful attributes such as data_type and methods such as is_date()).

Using the optimized reader is similar. The primary differences are:

- The workbook must be opened using the keyword argument use_iterators = True

- The rows are sequentially accessible using iter_rows().

- value is not available, and so internal_value must be used.

- The number of rows is not known, and so it isn't possible to pre-allocate the storage variable with the correct number of rows.

```python
from __future__ import print_function
import openpyxl

wb = openpyxl.load_workbook('FTSE_1984_2012.xlsx', use_iterators = True)
sheetNames = wb.get_sheet_names()
# Assumes 1 sheet name
sheet = wb.get_sheet_by_name(sheetNames[0])

# Use list to store data
open = []

# Changes since access is via memory efficient iterator
# Note () on iter_rows
for row in sheet.iter_rows():
    # Must use internal_value
```

```
    open.append(row[1].internal_value)

# Burn first row and convert to array
open = array(open[1:])
```

9.2.3 MATLAB Data Files (.mat)

SciPy enables MATLAB data files (mat files) to be read excluding except the latest V7.3 format, which can be read using PyTables or h5py. Data from compatible mat files can be loaded using `loadmat`. The data is loaded into a dictionary, and individual variables are accessed using the keys of the dictionary.

```
>>> import scipy.io as sio
>>> matData = sio.loadmat('FTSE_1984_2012.mat')
>>> type(matData)
dict

>>> matData.keys()
['volume',
 '__header__',
 '__globals__',
 'high',
 'adjclose',
 'low',
 'close',
 '__version__',
 'open']

>>> open = matData['open']
```

MATLAB data files in the newest V7.3 format can be easily read using PyTables.

```
>>> import tables
>>> matfile = tables.openFile('FTSE_1984_2012_v73.mat')
>>> matfile.root
/ (RootGroup) ''
  children := ['volume' (CArray), 'high' (CArray), 'adjclose' (CArray), 'low' (CArray), '
      close' (CArray), 'open' (CArray)]

>>> matfile.root.open
/open (CArray(1, 7042), zlib(3)) ''
  atom := Float64Atom(shape=(), dflt=0.0)
  maindim := 0
  flavor := 'numpy'
  byteorder := 'little'
  chunkshape := (1, 7042)

>>> open = matfile.root.open.read()
open = matfile.root.open.read()

>>> matfile.close() # Close the file
```

9.2.4 Reading Complex Files

Python can be programmed to read any text file format since it contains functions for directly accessing files and parsing strings. Reading poorly formatted data files is an advanced technique and should be avoided if possible. However, some data is only available in formats where reading in data line-by-line is the only option. For example, the standard import methods fail if the raw data is very large (too large for Excel) and is poorly formatted. In this case, the only possibility may be to write a program to read the file line-by-line (or in blocks) and to directly process the raw text.

The file *IBM_TAQ.txt* contains a simple example of data that is difficult to import. This file was downloaded from Wharton Research Data Services and contains all prices for IBM from the TAQ database between January 1, 2001 and January 31, 2001. It is too large to use in Excel and has both numbers, dates and text on each line. The following code block shows one method for importing this data set.

```python
import io
from numpy import array

f = io.open('IBM_TAQ.txt', 'r')
line = f.readline()
# Burn the first list as a header
line = f.readline()

date = []
time = []
price = []
volume = []
while line:
    data = line.split(',')
    date.append(int(data[1]))
    price.append(float(data[3]))
    volume.append(int(data[4]))
    t = data[2]
    time.append(int(t.replace(':','')))
    line = f.readline()

# Convert to arrays, which are more useful than lists
# for numeric data
date = array(date)
price = array(price)
volume = array(volume)
time = array(time)

allData = array([date,price,volume,time])

f.close()
```

This block of code does a few thing:

- Open the file directly using `file`

- Reads the file line by line using `readline`

101

- Initializes lists for all of the data

- Rereads the file parsing each line by the location of the commas using `split(',')` to split the line at each comma into a list

- Uses `replace(':','')` to remove colons from the times

- Uses `int()` and `float()` to convert strings to numbers

- Closes the file directly using `close()`

9.3 Saving or Exporting Data using pandas

Pandas supports writing to CSV, general delimited text files, Excel files, json, html tables, HDF5 and STATA. An understanding of the pandas' DataFrame is required prior to using pandas file writing facilities, and Chapter 17 provides further information.

9.4 Saving or Exporting Data without pandas

Native NumPy Format

A number of options are available for saving data. These include using native npz data files, MATLAB data files, CSV or plain text. Multiple numpy arrays can be saved using `savez_compressed` (`numpy.savez_compressed`).

```
x = arange(10)
y = zeros((100,100))
savez_compressed('test',x,y)
data = load('test.npz')
# If no name is given, arrays are generic names arr_1, arr_2, etc
x = data['arr_1']

savez_compressed('test',x=x,otherData=y)
data = load('test.npz')
# x=x provides the name x for the data in x
x = data['x']
# otherDate = y saves the data in y as otherData
y = data['otherData']
```

A version which does not compress data but is otherwise identical is `savez`. Compression is usually a good idea and is very helpful for storing arrays which have repeated values and are large.

9.4.1 Writing MATLAB Data Files (.mat)

SciPy enables MATLAB data files to be written. Data can be written using `savemat`, which takes two inputs, a file name and a dictionary containing data, in its simplest form.

```
from __future__ import print_function
import scipy.io as sio
```

```
x = array([1.0,2.0,3.0])
y = zeros((10,10))
# Set up the dictionary
saveData = {'x':x, 'y':y}
sio.savemat('test',saveData,do_compression=True)
# Read the data back in
matData = sio.loadmat('test.mat')
```

savemat uses the optional argument do_compression = True, which compresses the data, and is generally a good idea on modern computers and/or for large datasets.

9.4.2 Exporting Data to Text Files

Data can be exported to a tab-delimited text files using savetxt. By default, savetxt produces tab delimited files, although then can be changed using the names argument delimiter.

```
x = randn(10,10)
# Save using tabs
savetxt('tabs.txt',x)
# Save to CSV
savetxt('commas.csv',x,delimiter=',')
# Reread the data
xData = loadtxt('commas.csv',delimiter=',')
```

9.5 Exercises

Note: There are no exercises using pandas in this chapter. For exercises using pandas to read or write data, see Chapter 17.

1. The file *exercise3.xls* contains three columns of data, the date, the return on the S&P 500, and the return on XOM (ExxonMobil). Using Excel, convert the date to YYYYMMDD format and save the file.

2. Save the file as both CSV and tab delimited. Use the three text readers to read the file, and compare the arrays returned.

3. Parse loaded data into three variables, dates, SP500 and XOM.

4. Save NumPy, compressed NumPy and MATLAB data files with all three variables. Which files is the smallest?

5. Construct a new variable, sumreturns as the sum of SP500 and XOM. Create another new variable, outputdata as a horizontal concatenation of dates and sumreturns.

6. Export the variable outputdata to a new CSV file using savetxt.

7. (Difficult) Read in *exercise3.xls* directly using xlrd.

8. (Difficult) Save *exercise3.xls* as *exercise3.xlsx* and read in directly using openpyxl.

Chapter 10

`Inf`, NaN and Numeric Limits

10.1 `inf` and NaN

`inf` represents infinity and `inf` is distinct from `-inf`. `inf` can be constructed in a number for ways, for example or `exp(710)`. `nan` stands for Not a Number, and `nan`s are created whenever a function produces a result that cannot be clearly evaluated to produce a number or infinity. For example, `inf/inf` results in `nan`. `nan`s often cause problems since most mathematical operations involving a `nan` produce a `nan`.

```
>>> x = nan
>>> 1.0 + x
nan

>>> 1.0 * x
nan

>>> 0.0 * x
nan

>>> mean(x)
nan
```

10.2 Floating point precision

All numeric software has limited precision; Python is no different. The easiest to understand the upper and lower limits, which are 1.7976×10^{308} (see `finfo(float).max`) and -1.7976×10^{308} (`finfo(float).min`). Numbers larger (in absolute value) than these are `inf`. The smallest positive number that can be expressed is 2.2250×10^{-308} (see `finfo(float).tiny`). Numbers between -2.2251×10^{-308} and 2.2251×10^{-308} are numerically 0.

However, the hardest concept to understand about numerical accuracy is the limited *relative* precision which is 2.2204×10^{-16} on most x86 and x86_64 systems. This value is returned from the command `finfo(float).eps` and may vary based on the type of CPU and/or the operating system used. Numbers which differ by less than 2.2204×10^{-16} are numerically the same. To explore the role of precision, examine the results of the following:

```
>>> x = 1.0
```

```
>>> eps = finfo(float).eps
>>> x = x+eps/2
>>> x == 1
True

>>> x-1
0.0

>>> x = 1 + 2*eps
>>> x == 1
False

>>> x-1
ans = 4.4408920985006262e-16
```

Moreover, any number y where $y < \left(x \times 2.2204 \times 10^{-16} \right)$ is treated as 0 when added or subtracted. This is referred to as relative range.

```
>>> x=10
>>> x+2*eps
>>> x-10
0

>>> (x-10) == 0
True

>>> (1e120 - 1e103) == 1e120
True

>>> 1e103 / 1e120
1e-17
```

In the first example, eps/2<eps *when compared to 1* so it has no effect while 2*eps>eps and so this value is different from 1. In the second example, 2*eps/10<eps, it has no effect when added. The final example subtracts 10^{103} from 10^{120} and shows that this is numerically the same as 10^{120} – again, this occurs since $10^{103}/10^{120} = 10^{-17}$ <eps. While numeric limits is a tricky concept to understand, failure to understand these limits can results in errors in code that appears to be otherwise correct. The practical usefulness of limited precision is to consider data scaling since many variables have natural scales which are differ by many orders of magnitude.

10.3 Exercises

Let eps = finfo(float).eps in the following exercises.

1. What is the value of log(exp(1000)) both analytically and in Python? Why do these differ?

2. Is eps/10 different from 0? If x = 1 + eps/10 - 1, is x different from 0?

3. Is 1-eps/10-1 difference from 0? What about 1-1-eps/10?

4. Is .1 different from .1+eps/10?

5. Is $x = 10.0**120$ (1×10^{120}) different from $y = 10.0**120 + 10.0**102$? (Hint: Test with $x == y$)

6. Why is $x = 10**120$ (1×10^{120}) different from $y = 10**120 + 10**102$?

7. Suppose $x = 2.0$. How many times (n) can $x = 1.0 + (x-1.0)/2.0$ be run before x==1 shows True? What is the value of $2.0**(-n)$. Is this value surprising?

Chapter 11

Logical Operators and Find

Logical operators are useful when writing batch files or custom functions. Logical operators, when combined with flow control, allow for complex choices to be compactly expressed.

11.1 >, >=, <, <=, ==, !=

The core logical operators are

Symbol	Function	Definition
>	greater	Greater than
>=	greater_equal	Greater than or equal to
<	less	Less than
<=	less_equal	Less than or equal to
==	equal	Equal to
!=	not_equal	Not equal to

Logical operators can be used on scalars, arrays or matrices. All comparisons are done element-by-element and return either `True` or `False`. For example, suppose x and y are arrays which are broadcastable. `z= x < y` will be an array of the same size as `broadcast(x,y).shape` composed of `True` and `False`. Alternatively, if one is scalar, say y, then the elements of z are `z[i,j] = x[i,j] < y`. For instance, suppose $z = xLy$ where L is one of the logical operators above such as `<` or `==`. The following table examines the behavior when x and/or y are scalars or arrays. Suppose `z = x < y`:

		y	
		Scalar	Array
x	Scalar	Any	Any
		$z = x < y$	$z_{ij} = x < y_{ij}$
	Array	Any	Broadcastable
		$z_{ij} = x_{ij} < y$	$z_{ij} = \tilde{x}_{ij} < \tilde{y}_{ij}$

where $\tilde{x}$ and $\tilde{y}$ are the post-broadcasting versions of x and y. Logical operators are frequently used in portions of programs known as flow control (e.g. `if ... else ...` blocks) which are be discussed in Chapter 13. It is important to remember that array logical operations return arrays and that flow control blocks *require scalar* logical expressions.

```
>>> x = array([[1,2],[-3,-4]])
>>> x > 0
array([[ True,  True],
       [False, False]], dtype=bool)

>>> x == -3
array([[False, False],
       [ True, False]], dtype=bool)

>>> y = array([1,-1])
>>> x < y # y broadcast to be (2,2)
array([[False, False],
       [ True,  True]], dtype=bool)

>>> z = array([[1,1],[-1,-1]]) # Same as broadcast y
>>> x < z
array([[False, False],
       [ True,  True]], dtype=bool)
```

11.2   and, or, not and xor

Logical expressions can be combined using four logical devices,

Keyword (Scalar)	Function	Bitwise	True if . . .
and	logical_and	&	Both True
or	logical_or		Either or Both True
not	logical_not	~	Not True
	logical_xor	^	One True and One False

There are three versions of all operators except XOR. The keyword version (e.g. and) can only be used with scalars and so it not useful when working with NumPy. Both the function and bitwise operators can be used with NumPy arrays, although care is requires when using the bitwise operators. Bitwise operators have high priority – higher than logical comparisons – and so parentheses are requires around comparisons. For example, (x>1) & (x<5) is a valid statement, while x>1 & x<5, which is evaluated as (x>(1 & x))<5, produces an error.

```
>>> x = arange(-2.0,4)
>>> y = x >= 0
>>> z = x < 2
>>> logical_and(y, z)
array([False, False,  True,  True, False, False], dtype=bool)

>>> y & z
array([False, False,  True,  True, False, False], dtype=bool)

>>> (x > 0) & (x < 2)
array([False, False,  True,  True, False, False], dtype=bool)
```

```
>>> x > 0 & x < 4 # Error
TypeError: ufunc 'bitwise_and' not supported for the input types, and the inputs could not
    be safely coerced to any supported types according to the casting rule ''safe''

>>> ~(y & z) # Not
array([ True,  True, False, False,  True,  True], dtype=bool)
```

These operators follow the same rules as most mathematical operators on arrays, and so require the broadcastable input arrays.

11.3 Multiple tests

all and any

The commands all and any take logical input and are self-descriptive. all returns True if all logical elements in an array are 1. If all is called without any additional arguments on an array, it returns True if all elements of the array are logical true and 0 otherwise. any returns logical(True) if any element of an array is True. Both all and any can be also be used along a specific dimension using a second argument or the keyword argument axis to indicate the axis of operation (0 is column-wise and 1 is row-wise). When used column- or row-wise, the output is an array with one less dimension than the input, where each element of the output contains the truth value of the operation on a column or row.

```
>>> x = array([[1,2][3,4]])
>>> y = x <= 2
>>> y
array([[ True,  True],
       [False, False]], dtype=bool)

>>> any(y)
True

>>> any(y,0)
array([[ True,  True]], dtype=bool)

>>> any(y,1)
array([[ True],
       [False]], dtype=bool)
```

allclose

allclose can be used to compare two arrays for near equality. This type of function is important when comparing floating point values which may be effectively the same although not identical.

```
>>> eps = np.finfo(np.float64).eps
>>> eps
2.2204460492503131e-16

>>> x = randn(2)
>>> y = x + eps
```

```
>>> x == y
array([False, False], dtype=bool)

>>> allclose(x,y)
True
```

The tolerance for being close can be set using keyword arguments either relatively (`rtol`) or absolutely (`atol`).

array_equal

array_equal tests if two arrays have the same shape and elements. It is safer than comparing arrays directly since comparing arrays which are not broadcastable produces an error.

array_equiv

array_equiv tests if two arrays are equivalent, even if they do not have the exact same shape. Equivalence is defined as one array being broadcastable to produce the other.

```
>>> x = randn(10,1)
>>> y = tile(x,2)
>>> array_equal(x,y)
False

>>> array_equiv(x,y)
True
```

11.4 is*

A number of special purpose logical tests are provided to determine if an array has special characteristics. Some operate element-by-element and produce an array of the same dimension as the input while other produce only scalars. These functions all begin with `is`.

Operator	True if . . .	Method of operation
isnan	1 if nan	element-by-element
isinf	1 if inf	element-by-element
isfinite	1 if not inf and not nan	element-by-element
isposfin,isnegfin	1 for positive or negative inf	element-by-element
isreal	1 if not complex valued	element-by-element
iscomplex	1 if complex valued	element-by-element
isreal	1 if real valued	element-by-element
is_string_like	1 if argument is a string	scalar
is_numlike	1 if is a numeric type	scalar
isscalar	1 if scalar	scalar
isvector	1 if input is a vector	scalar

```
x=array([4,pi,inf,inf/inf])
isnan(x)

array([[False, False, False,  True]], dtype=bool)

isinf(x)

array([[False, False,  True, False]], dtype=bool)

isfinite(x)

array([[ True,  True, False, False]], dtype=bool)
```

isnan(x) *isinf(x)* isfinite(x) always equals True for elements of a numeric array, implying any element falls into one (and only one) of these categories.

11.5 Exercises

1. Using the data file created in Chapter 9, count the number of negative returns in both the S&P 500 and ExxonMobil.

2. For both series, create an indicator variable that takes the value 1 if the return is larger than 2 standard deviations or smaller than -2 standard deviations. What is the average return conditional on falling each range for both returns.

3. Construct an indicator variable that takes the value of 1 when both returns are negative. Compute the correlation of the returns conditional on this indicator variable. How does this compare to the correlation of all returns?

4. What is the correlation when at least 1 of the returns is negative?

5. What is the relationship between all and any. Write down a logical expression that allows one or the other to be avoided (i.e. write def myany(x) and def myall(y)).

Chapter 12

Advanced Selection and Assignment

Elements from NumPy arrays can be selected using four methods: scalar selection, slicing, numerical (or list-of-locations) indexing and logical (or Boolean) indexing. Chapter 4 described scalar selection and slicing, which are the basic methods to access elements in an array. Numerical indexing and logical indexing are closely related and allow for more flexible selection. Numerical indexing uses lists or arrays of locations to select elements while logical indexing uses arrays containing Boolean values to select elements.

12.1 Numerical Indexing

Numerical indexing, also called list-of-location indexing, is an alternative to slice notation. The fundamental idea underlying numerical indexing is to use coordinates to select elements, which is similar to the underlying idea behind slicing. Numerical indexing differs from standard slicing in three important ways:

- Arrays created using numerical indexing are copies of the underlying data, while slices are views (and so do not copy the data). This means that while changing elements in a slice also changes elements in the slice's parent, changing elements in an array constructed using numerical indexing does not. This also can create performance concerns and slicing should generally be used whenever it is capable of selecting the required elements.

- Numerical indices can contain repeated values and are not required to be monotonic, allowing for more flexible selection. The sequences produced using slice notation are always monotonic with unique values.

- The shape of the array selected is determined by the shape of the numerical indices. Slices are similar to 1-dimensional arrays but the shape of the slice is determined by the slice inputs.

Numerical indexing in 1-dimensional arrays uses the numerical index values as locations in the array (0-based indexing) and returns an array with the same dimensions as the numerical index. To understand the core concept behind numerical indexing, consider the case of selecting 4 elements form a 1-dimensional array with locations i_1, ..., i_4. Numerical indexing uses the four indices and arranges them to determine the shape (and order) of the output. For example, if the order was

$$\begin{bmatrix} i_3 & i_2 \\ i_4 & i_1 \end{bmatrix}$$

115

then the array selected would be 2 by 2 with elements

$$\begin{bmatrix} x_{i_3} & x_{i_2} \\ x_{i_4} & x_{i_1} \end{bmatrix}.$$

Numerical indexing allows for arbitrary shapes and repetition, and so the selection matrix

$$\begin{bmatrix} i_3 & i_2 & i_3 & i_2 \\ i_4 & i_1 & i_3 & i_2 \\ i_4 & i_1 & i_4 & i_1 \end{bmatrix}$$

could be used to produce a 4 by 2 array containing the corresponding elements of x. In these examples the indices are not used in any particular order and are repeated to highlight the flexibility of numerical indexing.

Note that the numerical index can be either a list or a NumPy array and *must contain integer data*.

```
>>> x = 10 * arange(5.0)
>>> x[[0]] # List with 1 element
array([ 0.])

>>> x[[0,2,1]] # List
array([ 0.,  20.,  10.])

>>> sel = array([4,2,3,1,4,4]) # Array with repetition
>>> x[sel]
array([ 40.,  20.,  30.,  10.,  40.,  40.])

>>> sel = array([[4,2],[3,1]]) # 2 by 2 array
>>> x[sel] # Selection has same size as sel
array([[ 40.,  20.],
       [ 30.,  10.]])

>>> sel = array([0.0,1]) # Floating point data
>>> x[sel] # Error
IndexError: arrays used as indices must be of integer (or boolean) type

>>> x[sel.astype(int)] # No error
array([ 10.,  20.])

>>> x[0] # Scalar selection, not numerical indexing
1.0
```

These examples show that the numerical indices determine the element location and the shape of the numerical index array determines the shape of the output. The final three examples show slightly different behavior. The first two of these demonstrate that only integer arrays can be used in numerical indexing, while the final example shows that there is a subtle difference between x[[0]] (or x[array([0])]), which is using numerical indexing and x[0] which is using a scalar selector. x[[0]] returns a 1-dimensional array since the list has 1 dimension while x[0] returns a non-array (or scalar or 0-dimensional array) since the input is not a list or array.

Numerical indexing in 2- or higher-dimensional arrays uses numerical index arrays for each dimension. The fundamental idea behind numerical indexing in 2-dimensional arrays is to format coordinate pairs of the form (i_k, j_k) into separate arrays. The size of the arrays will determine the shape of the array selected. For example, if the two selection arrays were

$$[i_1, i_3, i_2, i_4] \text{ and } [j_1, j_3, j_2, j_4]$$

then a 1-dimensional array would be selected containing the elements

$$[x(i_i, j_i), x(i_3, j_3), x(i_2, j_2), x(i_4, j_4)].$$

In practice multidimensional indexing is more flexible that this simple example since the arrays used as selectors can have either the same shape *or* can be broadcastable (see Section 5.2).

Consider the following four examples.

```
>>> x = reshape(arange(10.0), (2,5))
>>> x
array([[ 0.,  1.,  2.,  3.,  4.],
       [ 5.,  6.,  7.,  8.,  9.]])

>>> sel = array([0,1])
>>> x[sel,sel] # 1-dim arrays, no broadcasting
array([ 0.,  6.])

>>> x[sel, sel+1]
array([ 1.,  7.])

>>> sel_row = array([[0,0],[1,1]])
>>> sel_col = array([[0,1],[0,1]])
>>> x[sel_row,sel_col] # 2 by 2, no broadcasting
array([[ 0.,  1.],
       [ 5.,  6.]])

>>> sel_row = array([[0],[1]])
>>> sel_col = array([[0,1]])
>>> x[sel_row,sel_col] # 2 by 1 and 1 by 2 - difference shapes, broadcasted as 2 by 2
array([[ 0.,  1.],
       [ 5.,  6.]])
```

In the first example, sel is a 1-dimensional array containing [0,1], and so the returned value is also a 1-dimensional array containing the $(0, 0)$ and $(1, 1)$ elements of x. Numerical indexing uses the array in the first position to determine row locations and the array in the second position to determine column locations. The first element of the row selection is paired with the first element of column selection (as is the second element). This is why x[sel,sel+1] selects the elements in the $(0, 1)$ and $(1, 2)$ positions (1 and 7, respectively). The third example uses 2-dimensional arrays and selects the elements $(0, 0)$, $(0, 1)$, $(1, 0)$ and $(1, 1)$. The final example also uses 2-dimensional arrays but with different sizes – 2 by 1 and 1 by 2 – which are broadcastable to a common shape of 2 by 2 arrays.

Next, consider what happens when non-broadcastable arrays are used in as numerical indexing.

```
>>> sel_row = array([0,1]) # 1-dimensional with shape (2,)
>>> sel_col = array([1,2,3]) # 1-dimensional with shape (3,)
>>> x[sel_row,sel_col] # Error
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

An error occurs since these two 1-dimensional arrays are not broadcastable. ix_ can be used to easily se-
lect rows and columns using numerical indexing by translating the 1-dimesnional arrays to be the correct
size for broadcasting.

```
>>> x[ix_([0,1],[1,2,3])]
array([[ 2.,  3.,  4.],
       [ 7.,  8.,  9.]])
```

12.1.1   Mixing Numerical Indexing with Scalar Selection

NumPy permits using difference types of indexing in the same expression. Mixing numerical indexing
with scalar selection is trivial since any scalar can be broadcast to any array shape.

```
>>> sel=array([[1],[2]]) # 2 by 1
>>> x[0,sel] # Row 0, elements sel
array([[ 1.],
       [ 2.]])

>>> sel_row = array([[0],[0]])
>>> x[sel_row,sel] # Identical
array([[ 1.],
       [ 2.]])
```

12.1.2   Mixing Numerical Indexing with Slicing

Mixing numerical indexing and slicing allow for entire rows or columns to be selected.

```
>>> x[:,[1]]
array([[ 2.],
       [ 7.]])

>>> x[[1],:]
array([[  6.,   7.,   8.,   9.,  10.]])
```

Note that the mixed numerical indexing and slicing uses a list ([1]) so that it is not a scalar. This is impor-
tant since using a scalar will result in dimension reduction.

```
>>> x[:,1] # 1-dimensional
array([ 2.,  7.])
```

Numerical indexing and slicing can be mixed in more than 2-dimensions, although some care is required.
In the simplest case where only one numerical index is used which is 1-dimensional, then the selection is
equivalent to calling ix_ where the slice $a:b:s$ is replaced with arange(a,b,s).

```
>>> x = reshape(arange(3**3), (3,3,3)) # 3-d array
>>> sel1 = x[::2,[1,0],:1]
>>> sel2 = x[ix_(arange(0,3,2),[1,0],arange(0,1))]
```

```
>>> sel1.shape
(2L, 2L, 1L)

>>> sel2.shape
(2L, 2L, 1L)

>>> amax(abs(sel1-sel2))
0
```

When more than 1 numerical index is used, the selection can be viewed as a 2-step process.

1. Select using only slice notation where the dimensions using numerical indexing use the slice :.

2. Apply the numerical indexing to the array produced in step 1.

```
>>> sel1 = x[[0,0],[1,0],:1]
>>> step1 = x[:,:,:1]
>>> step2 = x[[0,0],[1,0],:]
>>> step2.shape
(2L, 1L)

>>> amax(abs(sel1-step2))
0
```

In the previous example, the shape of the output was (2L, 1L) which may seem surprising since the numerical indices where both 1-dimensional arrays with 2 elements. The "extra" dimension comes from the slice notation which always preserves its dimension. In the next example, the output is 3-dimensional since the numerical indices are 1-dimensional and the 2 slices preserve their dimension.

```
>>> x = reshape(arange(4**4), (4,4,4,4))
>>> sel = x[[0,1],[0,1],:2,:2] # 1-dimensional numerical and 2 slices
>>> sel.shape
(2L, 2L, 2L)
```

It is possible to mix multidimensional numerical indexing with slicing and multidimensional arrays. This type of selection is not explicitly covered since describing the output is complicated and this type of selection is rarely encountered.

12.1.3 Linear Numerical Indexing using `flat`

Like slicing, numerical indexing can be combined with `flat` to select elements from an array using the row-major ordering of the array. The behavior of numerical indexing with `flat` is identical to that of using numerical indexing on a flattened version of the underlying array.

```
>>> x.flat[[3,4,9]]
array([  4.,    5.,   10.])

>>> x.flat[[[3,4,9],[1,5,3]]]
array([[  4.,    5.,   10.],
       [  2.,    6.,    4.]])
```

12.1.4 Mixing Numerical Indexing with Slicing and Scalar Selection

Mixing the three is identical to using numerical indexing and slicing since the scalar selection is always broadcast to be compatible with the numerical indices.

12.2 Logical Indexing

Logical indexing differs from slicing and numeric indexing by using logical indices to select elements, rows or columns. Logical indices act as light switches and are either "on" (True) or "off" (False). Pure logical indexing uses a logical indexing array with the same size as the array being used for selection and always returns a 1-dimensional array.

```
>>> x = arange(-3,3)
>>> x < 0
array([ True,  True,  True, False, False, False], dtype=bool)

>>> x[x < 0]
array([-3, -2, -1])

>>> x[abs(x) >= 2]
array([-3, -2,  2])

>>> x = reshape(arange(-8, 8), (4,4))
>>> x[x < 0]
array([-8, -7, -6, -5, -4, -3, -2, -1])
```

It is tempting to use two 1-dimensional logical arrays to act as row and column masks on a 2-dimensional array. This does not work, and it is necessary to use ix_ if interested in this type of indexing.

```
>>> x = reshape(arange(-8,8),(4,4))
>>> cols = any(x < -6, 0)
>>> rows = any(x < 0, 1)
>>> cols
array([ True,  True, False, False], dtype=bool

>>> rows
array([ True,  True, False, False], dtype=bool)

>>> x[cols,rows] # Not upper 2 by 2
array([-8, -3])

>>> x[ix_(cols,rows)] # Upper 2 by 2
array([[-8, -7],
       [-4, -3]])
```

The difference between the final 2 commands is due to how logical indexing operates when more than logical array is used. When using 2 or more logical indices, they are first transformed to numerical indices using nonzero which returns the locations of the non-zero elements (which correspond to the True elements of a Boolean array).

```
>>> cols.nonzero()
```

```
(array([0, 1], dtype=int64),)

>>> rows.nonzero()
(array([0, 1], dtype=int64),)
```

The corresponding numerical index arrays have compatible sizes – both are 2-element, 1-dimensional arrays – and so numeric selection is possible. Attempting to use two logical index arrays which have non-broadcastable dimensions produces the same error as using two numerical index arrays with non-broadcastable sizes.

```
>>> cols = any(x < -6, 0)
>>> rows = any(x < 4, 1)
>>> rows
array([ True,  True,  True, False], dtype=bool)

>>> x[cols,rows] # Error
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

12.2.1 Mixing Logical Indexing with Scalar Selection

Logical indexing can be combined with scalar selection to select elements from a specific row or column in a 2-dimensional array. Combining these two types of indexing is no different from first applying the scalar selection to the array and then applying the logical indexing.

```
>>> x = reshape(arange(-8,8), (4,4))
>>> x
array([[-8, -7, -6, -5],
       [-4, -3, -2, -1],
       [ 0,  1,  2,  3],
       [ 4,  5,  6,  7]])

>>> sum(x, 0)
array([-8, -4,  0,  4])

>>> sum(x, 0) >= 0
array([False, False,  True,  True], dtype=bool)

>>> x[0,sum(x, 0) >= 0]
array([-6, -5])
```

12.2.2 Mixing Logical Indexing with Slicing

Logical indexing can be freely mixed with slices by using 1-dimensional logical index arrays which act as selectors for columns or rows.

```
>>> sel = sum(x < -1, 0) >= 2
>>> sel
array([ True,  True,  True, False], dtype=bool)

>>> x[:,sel] # All rows, sel selects columns
```

```
array([[-8, -7, -6],
       [-4, -3, -2],
       [ 0,  1,  2],
       [ 4,  5,  6]])

>>> x[1:3,sel] # Rows 1 and 2, sel selects columns
array([[-4, -3, -2],
       [ 0,  1,  2]])

>>> x[sel,2:] # sel selects rows, columns 2 and 3
array([[-6, -5],
       [-2, -1],
       [ 2,  3]])
```

12.2.3 Mixing Logical Indexing with Numerical Indexing

Mixing numerical indexing and logical indexing behaves identically to numerically indexing where the logical index is converted to a numerical index using nonzero. It must be the case that the array returned by nonzero and the numerical index arrays are broadcastable.

```
>>> sel = array([True,True,False,False])
>>> sel.nonzero()
(array([0, 1], dtype=int64),)

>>> x[[2,3],sel] # Elements (2,0) and (3,1)
array([0, 5])

>>> x[[2,3],[0,1]] # Identical
array([0, 5])
```

12.2.4 Logical Indexing Functions

nonzero and `flatnonzero`

nonzero is an useful function for working with multiple data series. nonzero takes logical inputs and returns a tuple containing the indices where the logical statement is true. This tuple is suitable for indexing so that the corresponding elements can be accessed using x[indices].

```
>>> x = array([[1,2],[3,4]])
>>> sel = x <= 3
>>> indices = nonzero(sel)
>>> indices
(array([0, 0, 1], dtype=int64), array([0, 1, 0], dtype=int64))

>>> x[indices]
array([[1, 2, 3]])
```

flatnonzero is similar to nonzero except that the indices returned are for the flattened version of the input.

```
>>> flatnonzero(sel)
array([0, 1, 2], dtype=int64)
```

```
>>> x.flat[flatnonzero(sel)]
array([1, 2, 3])
```

argwhere

argwhere returns an array containing the locations of elements where a logical condition is True. It is the same as transpose(nonzero(x))

```
>>> x = randn(3)
>>> x
array([-0.5910316 ,  0.51475905,  0.68231135])

>>> argwhere(x<0.6)
array([[0],
       [1]], dtype=int64)

>>> argwhere(x<-10.0) # Empty array
array([], shape=(0L, 1L), dtype=int64)

>>> x = randn(3,2)
>>> x
array([[ 0.72945913,  1.2135989 ],
       [ 0.74005449, -1.60231553],
       [ 0.16862077,  1.0589899 ]])

>>> argwhere(x<0)
array([[1, 1]], dtype=int64)

>>> argwhere(x<1)
array([[0, 0],
       [1, 0],
       [1, 1],
       [2, 0]], dtype=int64)
```

extract

extract is similar to argwhere except that it returns the values where the condition is true rather than the indices.

```
>>> x = randn(3)
>>> x
array([-0.5910316 ,  0.51475905,  0.68231135])

>>> extract(x<0, x)
array([-0.5910316])

>>> extract(x<-10.0, x) # Empty array
array([], dtype=float64)
```

```
>>> x = randn(3,2)
>>> x
array([[ 0.72945913,  1.2135989 ],
       [ 0.74005449, -1.60231553],
       [ 0.16862077,  1.0589899 ]])


>>> extract(x>0,x)
array([ 0.72945913,  1.2135989 ,  0.74005449,  0.16862077,  1.0589899 ])
```

12.3   Performance Considerations and Memory Management

Arrays constructed using any numerical indexing and/or logical indexing are *always* copies of the under-
lying array. This is different from the behavior of slicing and scalar selection which returns a view, not a
copy, of an array. This is easily verified by selecting the same elements using different types of selectors.

```
>>> x = reshape(arange(9), (3,3))
>>> s_slice = x[:1,:] # Pure slice
>>> s_scalar = x[0] # Scalar selection
>>> s_numeric = x[[0],:] # Numeric indexing
>>> s_logical = x[array([True,False,False]),:] # Logical indexing
>>> s_logical[0,0] = -40
>>> s_numeric[0,0] = -30
>>> s_numeric # -30
array([[-10,   1,   2]])


>>> s_logical # -40, not -30
array([[-40,   1,   2]])


>>> s_scalar[0] = -10
>>> s_scalar
array([-10,   1,   2])


>>> x # Has a -10
array([[-10,   1,   2],
       [  3,   4,   5],
       [  6,   7,   8]])


>>> s_slice # Has a -10
array([[-10,   1,   2]])
```

Since both numerical and logical indexing produce copies, some care is needed when using these selectors
on large arrays.

12.4   Assignment with Broadcasting

Any of the selection methods can be used for assignment. When the shape of the array to be assigned is
the same as the selection, the assignment simply replaces elements using an element-by-element corre-
spondence.

```
>>> x = arange(-2,2.0)
>>> x
array([-2., -1.,  0.,  1.])

>>> x[0] = 999 # Scalar
>>> x
array([999., -1.,  0.,  1.])
>>> x[:2] = array([99.0,99]) # Slice
>>> x
array([ 99.,  99.,   0.,   1.])

>>> x[[0,1,2]] = array([-3.14,-3.14,-3.14]) # Numerical indexing
>>> x
array([-3.14, -3.14, -3.14,  1.  ])

>>> x[x<0] = zeros(3) # Logical indexing
array([ 0.,  0.,  0.,  1.])
```

Assignment is not limited to arrays with exact shape matches, and any assignment where two conditions are met is allowed:

- Each dimension of the array to be assigned is either 1 or matches the selection.

- The array to be assigned and the selection are broadcastable.

These two conditions ensure that the array to be assigned can be broadcast up to the shape of the selection – it is not sufficient that the selection and the array to be assigned are simply broadcastable. The simplest form of broadcasting assigns a scalar to a selection, and since a scalar can always be broadcast to any shape this is always possible.

```
>>> x = arange(-2,2.0)
>>> x[:2] = 99.0
>>> x
array([ 99.,  99.,   0.,   1.])

>>> x = log(x-2.0)
>>> x
array([ 4.57471098,  4.57471098,         nan,         nan])

>>> x[isnan(x)] = 0 # Logical indexing
>>> x
array([ 4.57471098,  4.57471098,  0.        ,  0.        ])

>>> x.shape = (2,2)
>>> x[:,:] = 3.14 # Could also use x[:]
>>> x
array([[ 3.14,  3.14],
       [ 3.14,  3.14]])
```

While broadcasting a scalar is the most frequently encountered case, there are useful applications of vector (or 1-dimensional array) to 2-dimensional array assignment. For example, it may be necessary to replace all rows in an array where some criteria is met in the row.

```
>>> x = reshape(arange(-10,10.0),(4,5))
array([[-10.,  -9.,  -8.,  -7.,  -6.],
       [ -5.,  -4.,  -3.,  -2.,  -1.],
       [  0.,   1.,   2.,   3.,   4.],
       [  5.,   6.,   7.,   8.,   9.]])

>>> x[sum(x,1)<0,:] = arange(5.0) # Replace rows w/ negative sum
>>> x = reshape(arange(-10,10.0),(4,5))
>>> x[:,sum(x,1)<0] = arange(4.0) # Error
ValueError: array is not broadcastable to correct shape

>>> x[:,sum(x,1)<0] = reshape(arange(4.0),(4,1)) # Correct col replacement
array([[ 0.,   0., -8., -7., -6.],
       [ 1.,   1., -3., -2., -1.],
       [ 2.,   2.,  2.,  3.,  4.],
       [ 3.,   3.,  7.,  8.,  9.]])
```

The error in the previous example occurs because the slice selects a 4 by 2 array, but the array to be assigned is 1-dimensional with 4 elements. The rules of broadcasting always prepend 1s when determining whether two arrays are broadcastable, and so the 1-dimensional array is considered to be a 1 by 4 array, which is not broadcastable to a 4 by 2 array. Using an explicitly 2-dimensional array with shape 4 by 1 allows for broadcasting.

12.5 Exercises

Let x=arange(10.0), y=reshape(arange(25.0),(5,5)) and z=reshape(arange(64.0),(4,4,4)) in all exercises.

1. List all methods to select 4.0 from x.

2. List all methods to select the first 5 elements of x.

3. List all methods to select every second element of x.

4. List all methods to select the row 2 from y.

5. List all methods to select the rows 2 and 4 from y.

6. List all methods to select the rows 2 and 4 and columns 2, 3 and 4 from y.

7. Select all rows of y which have at least one number divisible by 5 and at least one divisible by 7.

8. List all the methods to select panel 1 from z.

9. List all the methods to select rows 2 and 3 from all panels of z.

10. Assign 0 to every second element of z. List the alternative methods.

11. Assign $[-1, -1, -1, -1]$ to all rows of z which have at least one number divisible by 4 and one divisible by 6. For example, the row containing $[16, 17, 18, 19]$ satisfies this criteria.

12. (Difficult) Define `sel = array([[0,1],[1,0]])`, What shape does `y[sel,:]` have? Can this be explained?

Chapter 13

Flow Control, Loops and Exception Handling

The previous chapter explored one use of logical variables, selecting elements from an array. Flow control also utilizes logical variables to allow different code to be executed depending on whether certain conditions are met. Flow control in Python comes in two forms - conditional statement and loops.

13.1   Whitespace and Flow Control

Python uses white space changes to indicate the start and end of flow control blocks, and so indention matters. For example, when using `if . . . elif . . . else` blocks, all of the control blocks must have the same indentation level and all of the statements inside the control blocks should have the same level of indentation. Returning to the previous indentation level instructs Python that the block is complete. Best practice is to only use spaces (and not tabs), and to use 4 spaces when starting a indented level, which is a good balance between readability and wasted space.

13.2   `if . . . elif . . . else`

`if . . . elif . . . else` blocks always begin with an `if` statement immediately followed by a *scalar* logical expression. `elif` and `else` are optional and can always be replicated using nested `if` statements at the expense of more complex logic and deeper nesting. The generic form of an `if . . . elif . . . else` block is

```
if logical_1:
    Code to run if logical_1
elif  logical_2:
    Code to run if logical_2 and not logical_1
elif  logical_3:
    Code to run if logical_3 and not logical_1 or logical_2
...
...
else:
    Code to run if all previous logicals are false
```

However, simpler forms are more common,

```
if logical:
    Code to run if logical true
```

or

```
if logical:
    Code to run if logical true
else:
    Code to run if logical false
```

Important: Remember that all *logical*s should be scalar logical values. While it is possible to use arrays containing a single element, attempting to use an array with more than 1 element results in an error.

A few simple examples

```
>>> x = 5
>>> if x<5:
...     x += 1
... else:
...     x -= 1

>>> x
4
```

and

```
>>> x = 5;
>>> if x<5:
...     x = x + 1
... elif x>5:
...     x = x - 1
... else:
...     x = x * 2

>>> x
 10
```

These examples have all used simple logical expressions. However, any *scalar* logical expressions, such as (y<0 or y>1), (x<0 or x>1) and (y<0 or y>1) or isinf(x) or isnan(x), can be used in if ... elif ... else blocks.

13.3 for

for loops begin with for *item* in *iterable*:, and the generic structure of a for loop is

```
for item in iterable:
    Code to run
```

item is an element from *iterable*, and *iterable* can be anything that is iterable in Python. The most common examples are xrange or range, lists, tuples, arrays or matrices. The for loop will iterate across all items in iterable, beginning with item 0 and continuing until the final item. When using multidimensional arrays, only the outside dimension is directly iterable. For example, if x is a 2-dimensional array, then the iterable elements are x[0], x[1] and so on.

```
count = 0
for i in xrange(100):
```

```
    count += i

count = 0
x = linspace(0,500,50)
for i in x:
    count += i

count = 0
x = list(arange(-20,21))
for i in x:
    count += i
```

The first loop will iterate over $i = 0, 1, 2, \ldots, 99$. The second loops over the values produced by the function `linspace`, which returns an array with 50 uniformly spaced points between 0 and 500, inclusive. The final loops over x, a vector constructed from a call to `list(arange(-20,21))`, which produces a list containing the series $-20, -19, \ldots, 0, \ldots 19, 20$. All three – range, arrays, and lists – are iterable. The key to understanding `for` loop behavior is that `for` always iterates over the elements of the *iterable* in the order they are presented (i.e. *iterable*[0], *iterable[1]*, . . .).

> **Python 2.7 vs. 3**   Note: This chapter exclusively uses `xrange` in loops rather than `range`. `xrange` is the preferred iterator in Python 2.7 since it avoids large memory allocations. `range` has replaced `xrange` in Python 3.

Loops can also be nested

```
count = 0
for i in xrange(10):
    for j in xrange(10):
        count += j
```

or can contain flow control variables

```
returns = randn(100)
count = 0
for ret in returns:
    if ret<0:
        count += 1
```

This `for` expression can be equivalently expressed using `xrange` as the iterator and `len` to get the number of items in the iterable.

```
returns = randn(100)
count = 0
for i in xrange(len(returns)):
    if returns[i]<0:
        count += 1
```

Finally, these ideas can be combined to produce nested loops with flow control.

```
x = zeros((10,10))
for i in xrange(size(x,0)):
```

```
    for j in xrange(size(x,1)):
        if i<j:
            x[i,j]=i+j;
        else:
            x[i,j]=i-j
```

or loops containing nested loops that are executed based on a flow control statement.

```
x = zeros((10,10))
for i in xrange(size(x,0)):
    if (i % 2) == 1:
        for j in xrange(size(x,1)):
            x[i,j] = i+j
    else:
        for j in xrange(int(i/2)):
            x[i,j] = i-j
```

Important: The iterable variable should not be reassigned once inside the loop. Consider, for example,

```
x = range(10)
for i in x:
    print(i)
    print('Length of x:', len(x))
    x = range(5)
```

This produces the output

```
# Output
0
Length of x: 10
1
Length of x: 5
2
Length of x: 5
3
...
8
Length of x: 5
9
Length of x: 5
```

It is not safe to modify the sequence of the *iterable* when looping over it. The means that the *iterable* should not change size, which can occur when using a list and the functions pop(), insert() or append() or the keyword del. The loop below would never terminate (except for the if statement that breaks the loop) since L is being extended each iteration.

```
L = [1, 2]
for i in L:
    print(i)
    L.append(i+2)
    if i>5:
        break
```

Finally, `for` loops can be used with 2 *items* when the *iterable* is wrapped in `enumerate`, which allows the elements of the *iterable* to be directly accessed, as well as their index in the *iterable*.

```python
x = linspace(0,100,11)
for i,y in enumerate(x):
    print('i is :', i)
    print('y is :', y)
```

13.3.1 Whitespace

Like `if ... elif ... else` flow control blocks, `for` loops are whitespace sensitive. The indentation of the line immediately below the `for` statement determines the indentation that all statements in the block must have.

13.3.2 break

A loop can be terminated early using `break`. `break` is usually used after an `if` statement to terminate the loop prematurely if some condition has been met.

```python
x = randn(1000)
for i in x:
    print(i)
    if i > 2:
        break
```

Since for loops iterate over an iterable with a fixed size, `break` is generally more useful in `while` loops.

13.3.3 continue

`continue` can be used to skip an iteration of a loop, immediately returning to the top of the loop using the next item in *iterable*. `continue` is commonly used to avoid a level of nesting, such as in the following two examples.

```python
x = randn(10)
for i in x:
    if i < 0:
        print(i)

for i in x:
    if i >= 0:
        continue
    print(i)
```

Avoiding excessive levels of indentation is essential in Python programming – 4 is usually considered the maximum reasonable level. `continue` is particularly useful since it can be used to in a `for` loop to avoid one level of indentation.

13.4 while

`while` loops are useful when the number of iterations needed depends on the outcome of the loop contents. `while` loops are commonly used when a loop should only stop if a certain condition is met, such as

when the change in some parameter is small. The generic structure of a `while` loop is

```
while logical:
    Code to run
    Update logical
```

Two things are crucial when using a `while` loop: first, the *logical* expression should evaluate to true when the loop begins (or the loop will be ignored) and second, the inputs to the *logical* expression must be updated inside the loop. If they are not, the loop will continue indefinitely (hit CTRL+C to break an interminable loop in IPython). The simplest while loops are (wordy) drop-in replacements for `for` loops:

```
count = 0
i = 1
while i<10:
    count += i
    i += 1
```

which produces the same results as

```
count=0;
for i in xrange(0,10):
    count += i
```

`while` loops should generally be avoided when `for` loops are sufficient. However, there are situations where no `for` loop equivalent exists.

```
# randn generates a standard normal random number
mu = abs(100*randn(1))
index = 1
while abs(mu) > .0001:
    mu = (mu+randn(1))/index
    index=index+1
```

In the block above, the number of iterations required is not known in advance and since `randn` is a standard normal pseudo-random number, it may take many iterations until this criteria is met. Any finite `for` loop cannot be guaranteed to meet the criteria.

13.4.1 break

`break` can be used in a `while` loop to immediately terminate execution. Normally, `break` should not be used in a `while` loop – instead the logical condition should be set to `False` to terminate the loop. However, `break` can be used to avoid running code below the `break` statement even if the logical condition is `False`.

```
condition = True
i = 0
x = randn(1000000)
while condition:
    if x[i] > 3.0:
        break # No printing if x[i] > 3
    print(x[i])
    i += 1
```

It is better to update the logical statement which determines whether the while loop should execute.

```
i = 0
while x[i] <= 3:
    print(x[i])
    i += 1
```

13.4.2 `continue`

`continue` can be used in a `while` loop to skip any remaining code in the loop, immediately returning to the top of the loop, which then checks the while condition, and executes the loop if it still true. Using `continue` when the logical condition in the `while` loop is `False` is the same as using `break`.

13.5 `try ... except`

Exception handling is an advanced programming technique which can be used to make code more resilient (often at the cost of speed). `try ... except` blocks are useful for running code which may fail for reasons outside of the programmer's control. In most numerical applications, code should be deterministic and so dangerous code can usually be avoided. When it can't, for example, if reading data from a data source which isn't always available (e.g. a website), then `try ... except` can be used to attempt to execute the code, and then to do something if the code fails to execute. The generic structure of a `try ... except` block is

```
try:
    Dangerous Code
except ExceptionType1:
    Code to run if ExceptionType1 is raised
except ExceptionType2:
    Code to run if ExceptionType1 is raised
...
...
except:
    Code to run if an unlisted exception type is raised
```

A simple example of exception handling occurs when attempting to convert text to numbers.

```
text = ('a','1','54.1','43.a')
for t in text:
    try:
        temp = float(t)
        print(temp)
    except ValueError:
        print('Not convertable to a float')
```

13.6 List Comprehensions

List comprehensions are an optimized method of building a list which may simplify code when an iterable object is looped across and the results are saved to a list, possibly conditional on some logical test. Simple list can be used to convert a `for` loop which includes an append into a single line statement.

```
>>> x = arange(5.0)
>>> y = []
>>> for i in xrange(len(x)):
...     y.append(exp(x[i]))
>>> y
[1.0,
 2.7182818284590451,
 7.3890560989306504,
 20.085536923187668,
 54.598150033144236]

>>> z = [exp(x[i]) for i in xrange(len(x))]
>>> z
[1.0,
 2.7182818284590451,
 7.3890560989306504,
 20.085536923187668,
 54.598150033144236]
```

This simple list comprehension saves 2 lines of typing. List comprehensions can also be extended to include a logical test.

```
>>> x = arange(5.0)
>>> y = []
>>> for i in xrange(len(x)):
...     if floor(i/2)==i/2:
...         y.append(x[i]**2)
>>> y
[0.0, 4.0, 16.0]

>>> z = [x[i]**2 for i in xrange(len(x)) if floor(i/2)==i/2]
>>> z
[0.0, 4.0, 16.0]
```

List comprehensions can also be used to loop over multiple iterable inputs.

```
>>> x1 = arange(5.0)
>>> x2 = arange(3.0)
>>> y = []
>>> for i in xrange(len(x1)):
...     for j in xrange(len(x2)):
...         y.append(x1[i]*x2[j])
>>> y
[0.0, 0.0, 0.0, 0.0, 1.0, 2.0, 0.0, 2.0, 4.0, 0.0, 3.0, 6.0, 0.0, 4.0, 8.0]

>>> z = [x1[i]*x2[j] for i in xrange(len(x1)) for j in xrange(len(x2))]
[0.0, 0.0, 0.0, 0.0, 1.0, 2.0, 0.0, 2.0, 4.0, 0.0, 3.0, 6.0, 0.0, 4.0, 8.0]

>>> # Only when i==j
>>> z = [x1[i]*x2[j] for i in xrange(len(x1)) for j in xrange(len(x2)) if i==j]
[0.0, 1.0, 4.0]
```

While list comprehensions are powerful methods to compactly express complex operations, they are never essential to Python programming.

13.7 Tuple, Dictionary and Set Comprehensions

The other mutable Python structures, the dictionary and the set, support construction using comprehension, as does the immutable type tuple. Set and dictionary comprehensions use {} while tuple comprehensions require an explicit call to `tuple` since () has another meaning.

```
>>> x = arange(-5.0,5.0)
>>> z_set = {x[i]**2.0 for i in xrange(len(x))}
>>> z_set
{0.0, 1.0, 4.0, 9.0, 16.0, 25.0}

>>> z_dict = {i:exp(i) for i in x}
{-5.0: 0.006737946999085467,
 -4.0: 0.018315638888734179,
 -3.0: 0.049787068367863944,
 -2.0: 0.1353352832366127,
 -1.0: 0.36787944117144233,
 ...

>>> z_tuple = tuple(i**3 for i in x)
(-125.0, -64.0, -27.0, -8.0, -1.0, 0.0, 1.0, 8.0, 27.0, 64.0)
```

13.8 Exercises

1. Write a code block that would take a different path depending on whether the returns on two series are simultaneously positive, both are negative, or they have different signs using an `if ... elif ... else` block.

2. Simulate 1000 observations from an ARMA(2,2) where ϵ_t are independent standard normal innovations. The process of an ARMA(2,2) is given by

$$y_t = \phi_1 y_{t-1} + \phi_2 y_{t-2} + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \epsilon_t$$

Use the values $\phi_1 = 1.4$, $\phi_2 = -.8$, $\theta_1 = .4$ and $\theta_2 = .8$. **Note**: A T vector containing standard normal random variables can be simulated using `e = randn(T)`. When simulating a process, always simulate more data than needed and throw away the first block of observations to avoid start-up biases. This process is fairly persistent, at least 100 extra observations should be computed.

3. Simulate a GARCH(1,1) process where ϵ_t are independent standard normal innovations. A GARCH(1,1) process is given by

$$y_t = \sigma_t \epsilon_t$$

$$\sigma_t^2 = \omega + \alpha y_{t-1}^2 + \beta \sigma_{t-1}^2$$

Use the values $\omega = 0.05$, $\alpha = 0.05$ and $\beta = 0.9$, and set $h_0 = \omega/(1 - \alpha - \beta)$.

4. Simulate a GJR-GARCH(1,1,1) process where ϵ_t are independent standard normal innovations. A GJR-GARCH(1,1) process is given by

$$y_t = \sigma_t \epsilon_t$$

$$\sigma_t^2 = \omega + \alpha y_{t-1}^2 + \gamma y_{t-1}^2 I_{[y_{t-1}<0]} + \beta \sigma_{t-1}^2$$

Use the values $\omega = 0.05$, $\alpha = 0.02$ $\gamma = 0.07$ and $\beta = 0.9$ and set $h_0 = \omega/\left(1 - \alpha - \frac{1}{2}\gamma - \beta\right)$. Note that some form of logical expression is needed in the loop. $I_{[\epsilon_{t-1}<0]}$ is an indicator variable that takes the value 1 if the expression inside the [ ] is true.

5. Simulate a ARMA(1,1)-GJR-GARCH(1,1)-in-mean process,

$$y_t = \phi_1 y_{t-1} + \theta_1 \sigma_{t-1} \epsilon_{t-1} + \lambda \sigma_t^2 + \sigma_t \epsilon_t$$

$$\sigma_t^2 = \omega + \alpha \sigma_{t-1}^2 \epsilon_{t-1}^2 + \gamma \sigma_{t-1}^2 \epsilon_{t-1}^2 I_{[\epsilon_{t-1}<0]} + \beta \sigma_{t-1}^2$$

Use the values from Exercise 4 for the GJR-GARCH model and use the $\phi_1 = -0.1$, $\theta_1 = 0.4$ and $\lambda = 0.03$.

6. Find two different methods to use a `for` loop to fill a 5×5 array with $i \times j$ where i is the row index, and j is the column index. One will use `xrange` as the *iterable*, and the other should directly iterate on the rows, and then the columns of the matrix.

7. Using a `while` loop, write a bit of code that will do a bisection search to invert a normal CDF. A bisection search cuts the interval in half repeatedly, only keeping the sub interval with the target in it. Hint: keep track of the upper and lower bounds of the random variable value and use flow control. This problem requires `stats.norm.cdf`.

8. Test out the loop using by finding the inverse CDF of 0.01, 0.5 and 0.975. Verify it is working by taking the absolute value of the difference between the final value and the value produced by `stats.norm.ppf`.

9. Write a list comprehension that will iterate over a 1-dimensional array and extract the negative elements to a list. How can this be done using *only* logical functions (no explicit loop), without the list comprehension (and returning an array)?

Chapter 14

Dates and Times

Date and time manipulation is provided by a built-in Python module `datetime`. This chapter assumes that `datetime` has been imported using `import datetime as dt`.

14.1   Creating Dates and Times

Dates are created using `date` by providing integer values for year, month and day and times are created using `time` using hours, minutes, seconds and microseconds.

```
>>> import datetime as dt
>>> yr, mo, dd = 2012, 12, 21
>>> dt.date(yr, mo, dd)
datetime.date(2012, 12, 21)

>>> hr, mm, ss, ms= 12, 21, 12, 21
>>> dt.time(hr, mm, ss, ms)
dt.time(12,21,12,21)
```

Dates created using `date` do not allow times, and dates which require a time stamp can be created using `datetime`, which combine the inputs from `date` and `time`, in the same order.

```
>>> dt.datetime(yr, mo, dd, hr, mm, ss, ms)
datetime.datetime(2012, 12, 21, 12, 21, 12, 21)
```

14.2   Dates Mathematics

Date-times and dates (but not times, and only within the same type) can be subtracted to produce a `timedelta`, which consists of three values, days, seconds and microseconds. Time deltas can also be added to dates and times compute different dates – although `date` types will ignore any information in the time delta hour or millisecond fields.

```
>>> d1 = dt.datetime(yr, mo, dd, hr, mm, ss, ms)
>>> d2 = dt.datetime(yr + 1, mo, dd, hr, mm, ss, ms)
>>> d2-d1
datetime.timedelta(365)
```

Date Unit	Common Name	Range	Time Unit	Common Name	Range
Y	Year	$\pm 9.2 \times 10^{18}$ years	h	Hour	$\pm 1.0 \times 10^{15}$ years
M	Month	$\pm 7.6 \times 10^{17}$ years	m	Minute	$\pm 1.7 \times 10^{13}$ years
W	Week	$\pm 1.7 \times 10^{17}$ years	s	Second	$\pm 2.9 \times 10^{12}$ years
D	Day	$\pm 2.5 \times 10^{16}$ years	ms	Millisecond	$\pm 2.9 \times 10^{9}$ years
			us	Microsecond	$\pm 2.9 \times 10^{6}$ years
			ns	Nanosecond	± 292 years
			ps	Picosecond	± 106 days
			fs	Femtosecond	± 2.6 hours
			as	Attosecond	± 9.2 seconds

Table 14.1: NumPy `datetime64` range. The absolute range is January 1, 1970 plus the range.

```
>>> d2 + dt.timedelta(30,0,0)
datetime.datetime(2014, 1, 20, 12, 21, 12, 20)

>>> dt.date(2012,12,21) + dt.timedelta(30,12,0)
datetime.date(2013, 1, 20)
```

If times stamps are important, `date` types can be promoted to `datetime` using `combine` and a `time`.

```
>>> d3 = dt.date(2012,12,21)
>>> dt.datetime.combine(d3, dt.time(0))
datetime.datetime(2012, 12, 21, 0, 0)
```

Values in `dates`, `times` and `datetimes` can be modified using `replace` through keyword arguments.

```
>>> d3 = dt.datetime(2012,12,21,12,21,12,21)
>>> d3.replace(month=11,day=10,hour=9,minute=8,second=7,microsecond=6)
datetime.datetime(2012, 11, 10, 9, 8, 7, 6)
```

14.3 Numpy `datetime64`

Version 1.7.0 of NumPy introduces a NumPy native datetime type known as `datetime64` (to distinguish it from the usual datetime type). The NumPy datetime type is considered experimental and is not fully supported in the scientific python stack at the time of writing these notes. This said, it is already widely used and should see complete support in the near future. Additionally, the native NumPy data type is generally better suited to data storage and analysis and extends the Python datetime with additional features such as business day functionality.

NumPy contains both datetime (`datetime64`) and timedelta (`timedelta64`) objects. These differ from the standard Python datetime since they always store the datetime or timedelta using a 64-bit integer *plus* a date or time unit. The choice of the date/time unit affects both the resolution of the datetime as well as the permissible range. The unit directly determines the resolution - using a date unit of a day (`'D'`) limits to resolution to days. Using a date unit of a week (`'W'`) will allow a minimum of 1 week difference. Similarly, using a time unit of a second (`'s'`) will allow resolution up to the second (but not millisecond). The set of date and time units, and their range are presented in Table 14.1.

NumPy datetimes can be initialized using either human readable strings or using numeric values. The string initialization is simple and datetimes can be initialized using year only, year and month, the complete date or the complete date including a time (and optional timezone). The default time resolution is nanoseconds (10^{-9}) and `T` is used to separate the time from the date.

```
>>> datetime64('2013')
numpy.datetime64('2013')

>>> datetime64('2013-09')
numpy.datetime64('2013-09')

>>> datetime64('2013-09-01')
numpy.datetime64('2013-09-01')

>>> datetime64('2013-09-01T12:00') # Time
numpy.datetime64('2013-09-01T12:00+0100')

>>> datetime64('2013-09-01T12:00:01') # Seconds
numpy.datetime64('2013-09-01T12:00:01+0100')

>>> datetime64('2013-09-01T12:00:01.123456789') # Nanoseconds
numpy.datetime64('2013-09-01T12:00:01.123456789+0100')
```

Date or time units can be explicitly included as the second input. The final example shows that rounding can occur if the date input is not exactly representable using the date unit chosen.

```
>>> datetime64('2013-01-01T00','h')
numpy.datetime64('2013-01-01T00:00+0000','h')

>>> datetime64('2013-01-01T00','s')
numpy.datetime64('2013-01-01T00:00:00+0000')

>>> datetime64('2013-01-01T00','ms')
numpy.datetime64('2013-01-01T00:00:00.000+0000')

>>> datetime64('2013-01-01','W')
numpy.datetime64('2012-12-27')
```

NumPy datetimes can also be initialized from arrays.

```
>>> dates = array(['2013-09-01','2013-09-02'],dtype='datetime64')
>>> dates
array(['2013-09-01', '2013-09-02'], dtype='datetime64[D]')

>>> dates[0]
numpy.datetime64('2013-09-01')
```

The NumPy datetime type also supports including timezone information, and when no timezone is provided the local timezone is used (currently BST on this computer, which is GMT+0100). These two commands show a time in US/Central (using -0600) and in GMT (using z for Zulu). Note that the returned time is always displayed in the local time zone and so the time stamp is changed. *Warning*: datetime64 that have times *always* include a timezone – this may be problematic in some situations.

```
>>> datetime64('2013-09-01T12:00:00-0600')
numpy.datetime64('2013-09-01T19:00:00+0100')

>>> datetime64('2013-09-01T19:00:00Z')
```

```
numpy.datetime64('2013-09-01T20:00:00+0100')
```

Dates which are initialized using one of the shorter forms are initialized at the earliest date (and time) in the period.

```
>>> datetime64('2013')==datetime64('2013-01-01')
True

>>> datetime64('2013-09')==datetime64('2013-09-01')
True
```

However, dates which contain time information are not always equal to dates which have no time information. This occurs since time information forces a timezone onto the datetime while the pure date has no timezone information.

```
>>> datetime64('2013-09-01')==datetime64('2013-09-01T00:00:00')
False

>>> datetime64('2013-09-01')==datetime64('2013-09-01T00:00:00Z')
True

>>> datetime64('2013-09-01T00:00:00') # Time is 00:00:00+0100
numpy.datetime64('2013-09-01T00:00:00+0100')

>>> datetime64('2013-09-01T00:00:00Z') # Time is 01:00:00+0100
numpy.datetime64('2013-09-01T01:00:00+0100')
```

A corresponding timedelta class, similarly named `timedelta64`, is created when dates are differenced. The second example shows why the previous equality test returned `False` – the dates differ by 1 hour due to the timezone difference.

```
>>> datetime64('2013-09-02') - datetime64('2013-09-01')
numpy.timedelta64(1,'D')

>>> datetime64('2013-09-01') - datetime64('2013-09-01T00:00:00')
numpy.timedelta64(3600,'s')
```

`timedelta64` types contain two pieces of information, a number indicating the number of steps between the two dates and the size of the step.

Chapter 15

Graphics

Matplotlib is a complete plotting library capable of high-quality graphics. Matplotlib contains both high level functions which produce specific types of figures, for example a simple line plot or a bar chart, as well as a low level API for creating highly customized charts. This chapter covers the basics of producing plots and only scratches the surface of the capabilities of matplotlib. Further information is available on the matplotlib website or in books dedicated to producing print quality graphics using matplotlib.

15.1 2D Plotting

Throughout this chapter, the following modules have been imported.

```
>>> import matplotlib.pyplot as plt
>>> import scipy.stats as stats
```

Other modules will be included only when needed for a specific graphic.

15.1.1 Line Plots

The most basic, and often most useful 2D graphic is a line plot. Basic line plots are produced using `plot` using a single input containing a 1-dimensional array.

```
>>> y = randn(100)
>>> plot(y)
```

The output of this command is presented in panel (a) of figure 15.1. A more flexible form adds a format string which has 1 to 3 elements: a color, represented using a letter (e.g. g for green), a marker symbol which is either a letter of a symbol (e.g. s for square, ^ for triangle up), and a line style, which is always a symbol or series of symbols. In the next example, `'g--'` indicates green (g) and dashed line (–).

```
>>> plot(y,'g--')
```

Format strings may contain any of the elements in the next table.

Color		Marker		Line Style	
Blue	b	Point	.	Solid	–
Green	g	Pixel	,	Dashed	--
Red	r	Circle	o	Dash-dot	-.
Cyan	c	Square	s	Dotted	:
Magenta	m	Diamond	D		
Yellow	y	Thin diamond	d		
Black	k	Cross	x		
White	w	Plus	+		
		Star	*		
		Hexagon	H		
		Alt. Hexagon	h		
		Pentagon	p		
		Triangles	^, v, <, >		
		Vertical Line			
		Horizontal Line	_		

The default behavior is to use a blue solid line with no marker (unless there is more than one line, in which case the colors will alter, in order, through those in the Colors column, skipping white). The format string contains 1 or more or the three categories of formatting information. For example, kx-- would produce a black dashed line with crosses marking the points, *: would produce a dotted line with the default color using stars to mark points and yH would produce a solid yellow line with a hexagon marker.

When plot is called with one array, the default x-axis values 1,2, ... are used. plot(x,y) can be used to plot specific x values against y values. Panel (c) shows the results of running the following code.

```
>>> x = cumsum(rand(100))
>>> plot(x,y,'r-')
```

While format strings are useful for quickly adding meaningful colors or line styles to a plot, they only expose a limited range of the available customizations. The next example shows how keyword arguments are used to add customizations to a plot. Panel (d) contains the plot produced by the following code.

```
>>> plot(x,y,alpha = 0.5, color = '#FF7F00', \
...     label = 'Line Label', linestyle = '-.', \
...     linewidth = 3, marker = 'o', markeredgecolor = '#000000', \
...     markeredgewidth = 2, markerfacecolor = '#FF7F00', \
...     markersize=30)
```

Note that in the previous example, \ is used to indicate to the Python interpreter that a statement is spanning multiple lines. Some of the more useful keyword arguments are listed in the table below.

Keyword	Description
alpha	Alpha (transparency) of the plot – default is 1 (no transparency)
color	Color description for the line.[1]
label	Label for the line – used when creating legends
linestyle	A line style symbol
linewidth	A positive integer indicating the width of the line
marker	A marker shape symbol or character
markeredgecolor	Color of the edge (a line) around the marker
markeredgewidth	Width of the edge (a line) around the marker
markerfacecolor	Face color of the marker
markersize	A positive integer indicating the size of the marker

Many more keyword arguments are available for a plot. The full list can be found in the docstring or by running the following code. The functions getp and setp can be used to get the list of properties for a line (or any matplotlib object), and setp can also be used to set a particular property.

```
>>> h = plot(randn(10))
>>> getp(h)
    agg_filter = None
    alpha = None
    animated = False
    ...

>>> setp(h, 'alpha')
alpha: float (0.0 transparent through 1.0 opaque)

>>> setp(h, 'color')
color: any matplotlib color

>>> setp(h, 'linestyle')
linestyle: [ '''-''' | '''--''' | '''-.''' | ''':''' | '''None''' | ''' ''' | ''''''' ]
and any drawstyle in combination with a linestyle, e.g. '''steps--'''.

>>> setp(h, 'linestyle', '--') # Change the line style
```

Note that setp(h, *prop*) returns a description of the property and setp(h, *prop*, *value*) sets *prop* to *value*.

15.1.2 Scatter Plots

scatter produces a scatter plot between 2 1-dimensional arrays. All examples use a set of simulated normal data with unit variance and correlation of 50%. The output of the basic scatter command is presented in figure 15.2, panel (a).

```
>>> z = randn(100,2)
>>> z[:,1] = 0.5*z[:,0] + sqrt(0.5)*z[:,1]
>>> x=z[:,0]
>>> y=z[:,1]
>>> scatter(x,y)
```
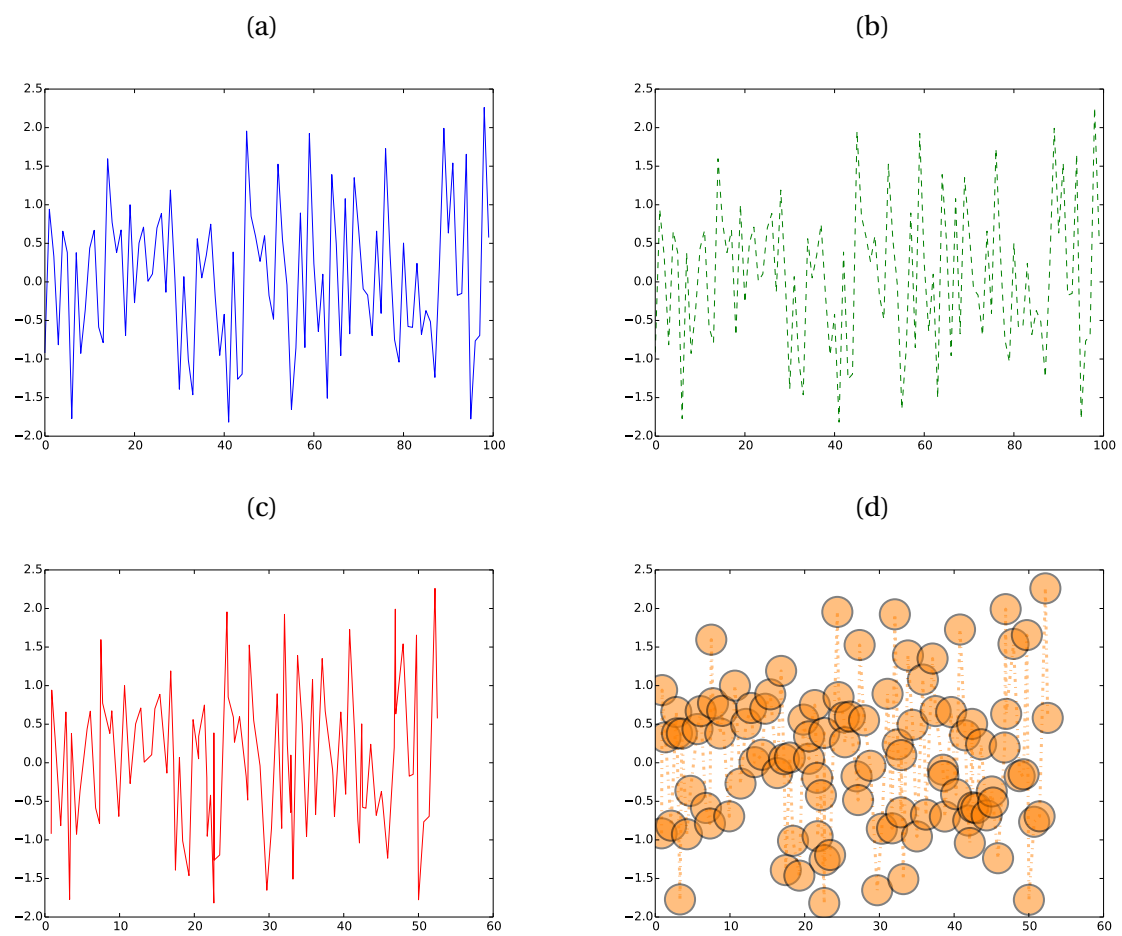
(a)

(b)

(c)
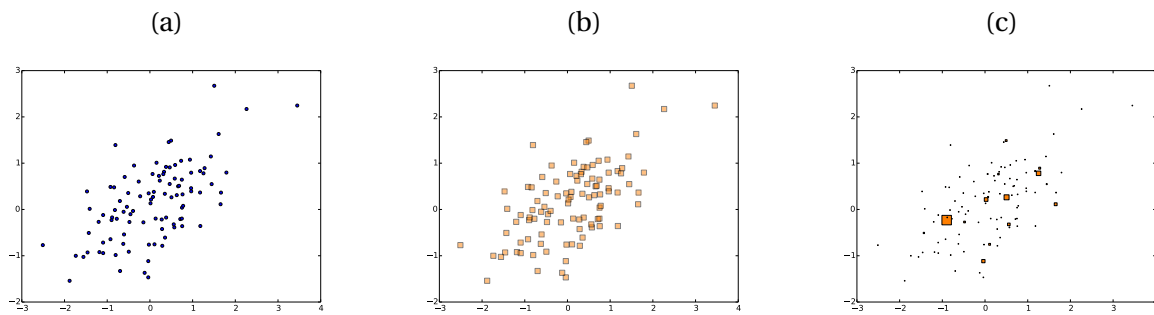
(d)

Figure 15.1: Line plots produced using `plot`.

Figure 15.2: Scatter plots produced using `scatter`.

Scatter plots can also be modified using keyword arguments. The most important are included in the next example, and have identical meaning to those used in the line plot examples. The effect of these keyword arguments is shown in panel (b).

```
>>> scatter(x,y, s = 60,  c = '#FF7F00', marker='s', \
...     alpha = .5, label = 'Scatter Data')
```

One interesting use of `scatter` is to add a 3rd dimension to the plot by including an array of size data which uses the shapes to convey an extra dimension of data. The use of variable size data is illustrated in the code below, which produced the scatter plot in panel (c).

```
>>> size_data = exp(exp(exp(rand(100))))
>>> size_data = 200 * s/amax(size_data)
>>> size_data[size_data<1]=1.0
>>> scatter(x,y, s = size_data, c = '#FF7F00', marker='s', \
...     label = 'Scatter Data')
```

15.1.3 Bar Charts

`bar` produces bar charts using two 1-dimensional arrays . The first specifies the left ledge of the bars and the second the bar heights. The next code segment produced the bar chart in panel (a) of figure 15.3.

```
>>> y = rand(5)
>>> x = arange(5)
>>> bar(x,y)
```

Bar charts take keyword arguments to alter colors and bar width. Panel (b) contains the output of the following code.

```
>>> bar(x,y, width = 0.5, color = '#FF7F00', \
...     edgecolor = '#000000', linewidth = 5)
```

Finally, `barh` can be used instead of `bar` to produce a horizontal bar chart. The next code snippet produces the horizontal bar chart in panel (c), and demonstrates the use of a list of colors to alter the appearance of the chart.

```
>>> colors = ['#FF0000','#FFFF00','#00FF00','#00FFFF','#0000FF']
>>> barh(x, y, height = 0.5, color = colors, \
...     edgecolor = '#000000', linewidth = 5)
```
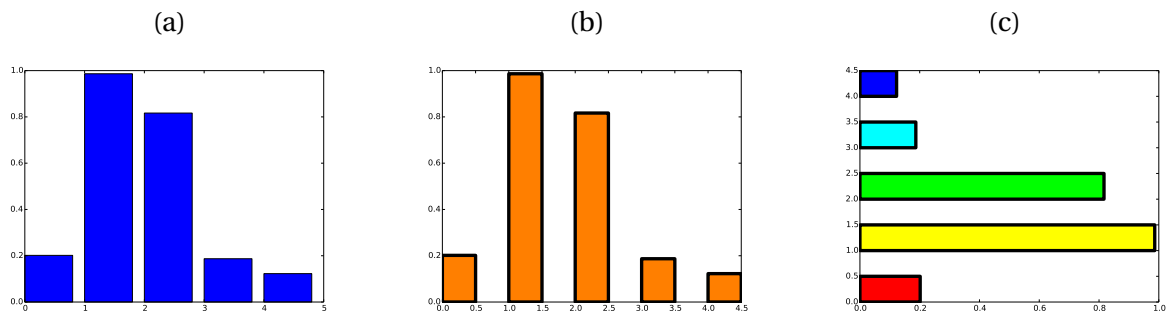
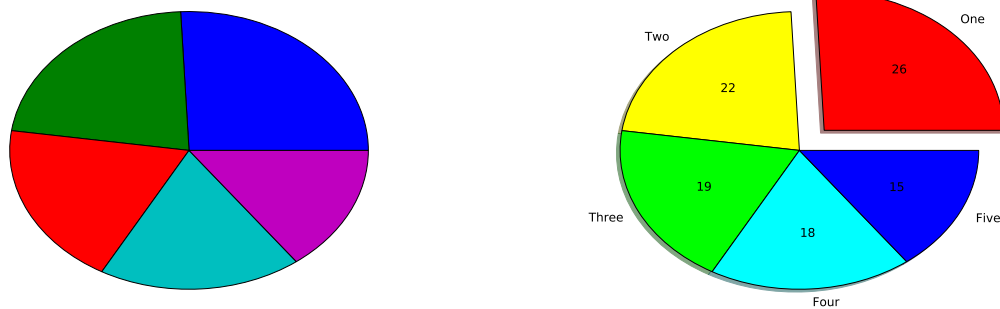Figure 15.3: Bar charts produced using `bar` and `barh`.

Figure 15.4: Pie charts produced using `pie`.

15.1.4 Pie Charts

`pie` produces pie charts using a 1-dimensional array of data (the data can have any values, and does not need to sum to 1). The basic use of `pie` is illustrated below, and the figure produced appears in panel (a) of figure 15.4.

```
>>> y = rand(5)
>>> y = y/sum(y)
>>> y[y<.05] = .05
>>> pie(y)
```

Pie charts can be modified using a large number of keyword arguments, including labels and custom colors. Exploded views of a pie chart can be produced by providing a vector of distances to the keyword argument explode. Note that autopct = `'%2.0f'` is using an old style format string to format the numeric labels. The results of running this code is shown in panel (b).

```
>>> explode = array([.2,0,0,0,0])
>>> colors = ['#FF0000','#FFFF00','#00FF00','#00FFFF','#0000FF']
>>> labels = ['One','Two','Three','Four','Five']
>>> pie(y, explode = explode, colors = colors, \
...     labels = labels, autopct = '%2.0f', shadow = True)
```
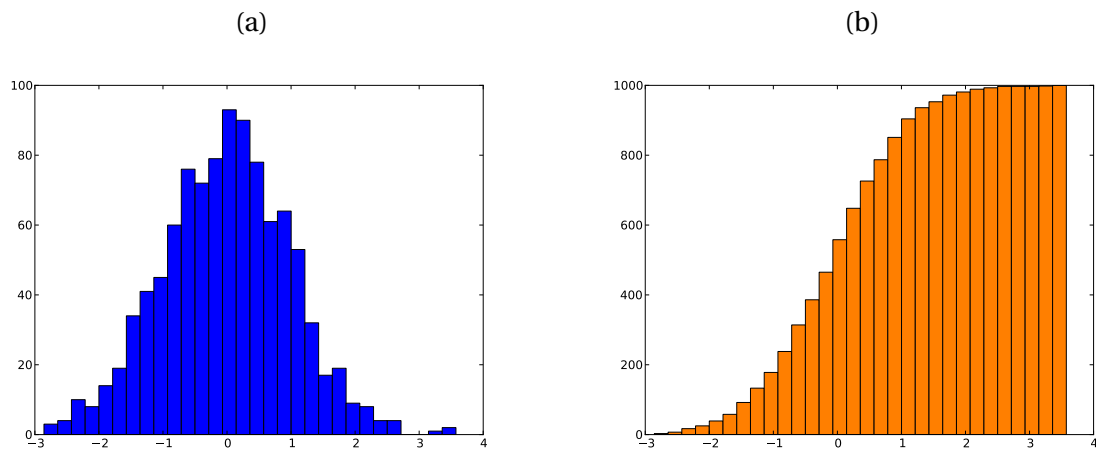
(a)

(b)

Figure 15.5: Histograms produced using `hist`.

15.1.5 Histograms

Histograms can be produced using `hist`. A basic histogram produced using the code below is presented in Figure 15.5, panel (a). This example sets the number of bins used in producing the histogram using the keyword argument `bins`.

```
>>> x = randn(1000)
>>> hist(x, bins = 30)
```

Histograms can be further modified using keyword arguments. In the next example, `cumulative=True` produces the cumulative histogram. The output of this code is presented in figure (b).

```
>>> hist(x, bins = 30, cumulative=True, color='#FF7F00')
```

15.2 Advanced 2D Plotting

15.2.1 Multiple Plots

In some scenarios it is advantageous to have multiple plots or charts in a single figure. Implementing this is simple using `figure` to initialize the figure window and then using `add_subplot`. Subplots are added to the figure using a grid notation with m rows and n columns where 1 is the upper left, 2 is the right of 1, and so on until the end of a row, where the next element is below 1. For example, the plots in a 3 by 2 subplot have indices

$$
\begin{bmatrix}
1 & 2 \\
3 & 4 \\
5 & 6
\end{bmatrix}.
$$

`add_subplot` is called using the notation `add_subplot(mni)` or `add_subplot(m,n,i)` where m is the number of rows, n is the number of columns and i is the index of the subplot.

Note that `add_subplot` must be called as a method from figure. Note that the next code block is sufficient long that it isn't practical to run interactively, and so `draw()` is used to force an update to the window to ensure that all plots and charts are visible. Figure 15.6 contains the result running the code below.

```python
from matplotlib.pyplot import figure, plot, bar, pie, draw, scatter
from numpy.random import randn, rand
from numpy import sqrt, arange

fig = figure()
# Add the subplot to the figure
# Panel 1
ax = fig.add_subplot(2, 2, 1)
y = randn(100)
plot(y)
ax.set_title('1')

# Panel  2
y = rand(5)
x = arange(5)
ax = fig.add_subplot(2, 2, 2)
bar(x, y)
ax.set_title('2')

# Panel 3
y = rand(5)
y = y / sum(y)
y[y < .05] = .05
ax = fig.add_subplot(2, 2, 3)
pie(y)
ax.set_title('3')

# Panel 4
z = randn(100, 2)
z[:, 1] = 0.5 * z[:, 0] + sqrt(0.5) * z[:, 1]
x = z[:, 0]
y = z[:, 1]
ax = fig.add_subplot(2, 2, 4)
scatter(x, y)
ax.set_title('4')
draw()
```

15.2.2 Multiple Plots on the Same Axes

Occasionally two different types of plots are needed in the same axes, for example, plotting a histogram
and a PDF. Multiple plots can be added to the same axes by plotting the first one (e.g. a histogram), calling
`hold(True)` to "hold" the contents of the axes (rather than overdrawing), and then plotting any remaining
data. In general it is a good idea to call `hold(False)` when finished.

   The code in the next example begins by initializing a figure window and then adding axes. A histogram
is then added to the axes, `hold` is called, and then a Normal PDF is plotted. `legend()` is called to produce
a legend using the labels provided in the potting commands. `get_xlim` and `get_ylim` are used to get the
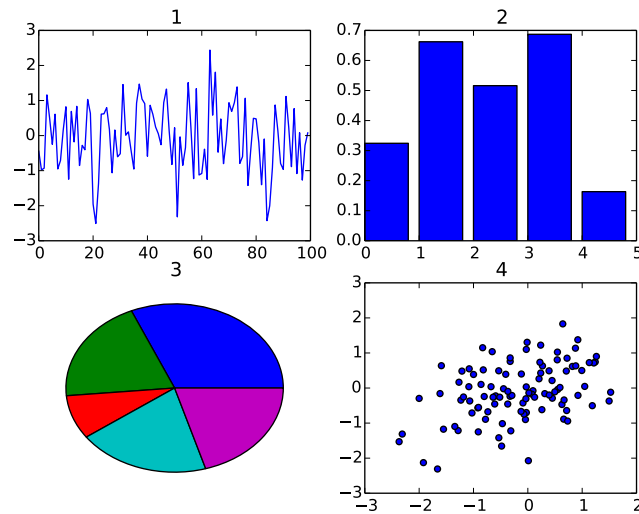
Figure 15.6: A figure containing a 2 by 2 subplot produced using `add_subplot`.

limits of the axis after adding the histogram. These points are used when computing the PDF, and finally `set_ylim` is called to increase the axis height so that the PDF is against the top of the chart. Figure 15.7 contains the output of these commands.

```python
from matplotlib.pyplot import figure, hold, plot, legend, draw
from numpy import linspace
import scipy.stats as stats
from numpy.random import randn

x = randn(100)
fig = figure()
ax = fig.add_subplot(111)
ax.hist(x, bins=30, label='Empirical')
xlim = ax.get_xlim()
ylim = ax.get_ylim()
pdfx = linspace(xlim[0], xlim[1], 200)
pdfy = stats.norm.pdf(pdfx)
pdfy = pdfy / pdfy.max() * ylim[1]
hold(True)
plot(pdfx, pdfy, 'r-', label='PDF')
ax.set_ylim((ylim[0], 1.2 * ylim[1]))
legend()
hold(False)
draw()
```

15.2.3 Adding a Title and Legend

Titles are added with `title` and legends are added with `legend`. `legend` requires that lines have labels, which is why 3 calls are made to `plot` – each series has its own label. Executing the next code block produces a the image in figure 15.8, panel (a).
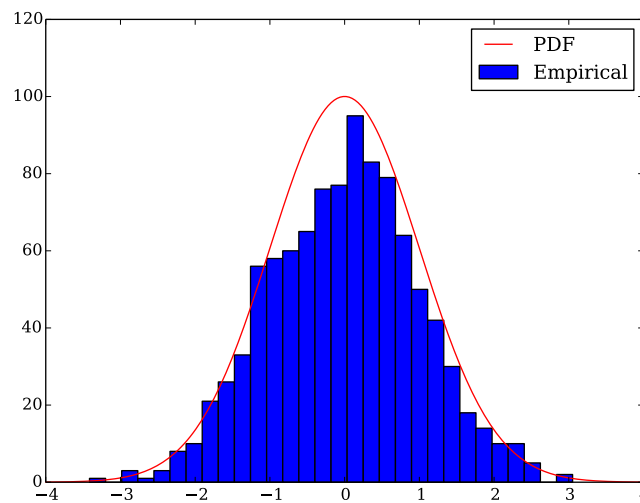
Figure 15.7: A figure containing a histogram and a line plot on the same axes using `hold`.

```
>>> x = cumsum(randn(100,3), axis = 0)
>>> plot(x[:,0],'b-',label = 'Series 1')
>>> hold(True)
>>> plot(x[:,1],'g-.',label = 'Series 2')
>>> plot(x[:,2],'r:',label = 'Series 3')
>>> legend()
>>> title('Basic Legend')
```

`legend` takes keyword arguments which can be used to change its location (`loc` and an integer, see the docstring), remove the frame (`frameon`) and add a title to the legend box (`title`). The output of a simple example using these options is presented in panel (b).

```
>>> plot(x[:,0],'b-',label = 'Series 1')
>>> hold(True)
>>> plot(x[:,1],'g-.',label = 'Series 2')
>>> plot(x[:,2],'r:',label = 'Series 3')
>>> legend(loc = 0, frameon = False, title = 'The Legend')
>>> title('Improved Legend')
```

15.2.4 Dates on Plots

Plots with date x-values on the x-axis are important when using time series data. Producing basic plots with dates is as simple as `plot(x,y)` where x is a list or array of dates. This first block of code simulates a random walk and constructs 2000 datetime values beginning with March 1, 2012 in a list.

```
from numpy import cumsum
from numpy.random import randn
from matplotlib.pyplot import figure, draw
import matplotlib.dates as mdates
import datetime as dt
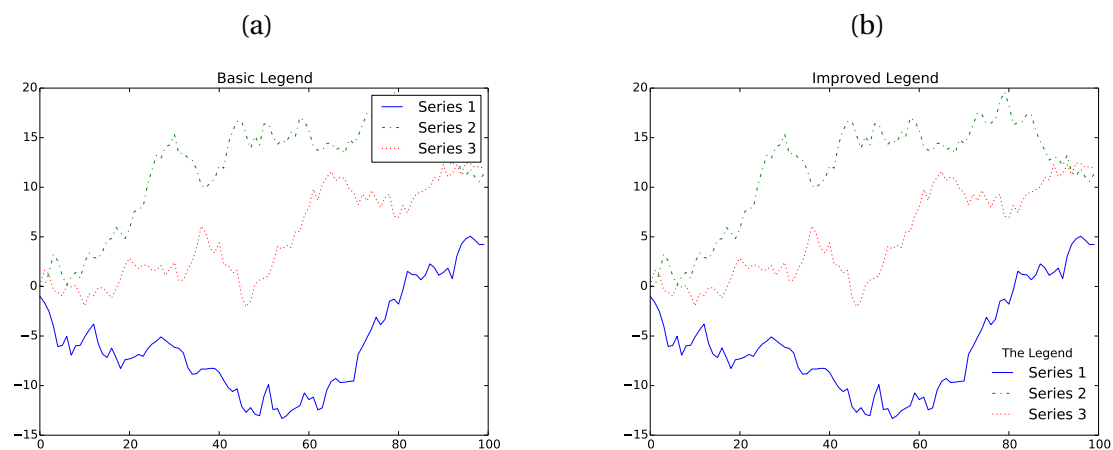```

152

(a)                          (b)

Figure 15.8: Figures with titles and legend produced using `title` and `legend`.

```
# Simulate data
T = 2000
x = []
for i in xrange(T):
    x.append(dt.datetime(2012,3,1)+dt.timedelta(i,0,0))
y = cumsum(rnd.randn(T))
```

A basic plot with dates only requires calling `plot(x,y)` on the x and y data. The output of this code is in panel (a) of figure 15.9.

```
fig = figure()
ax = fig.add_subplot(111)
ax.plot(x,y)
draw()
```

Once the plot has been produced `autofmt_xdate()` is usually called to rotate and format the labels on the x-axis. The figure produced by running this command on the existing figure is in panel (b).

```
fig.autofmt_xdate()
draw()
```

Sometime, depending on the length of the sample plotted, automatic labels will not be adequate. To show a case where this issue arises, a shorted sample with only 100 values is simulated.

```
T = 100
x = []
for i in xrange(1,T+1):
    x.append(dt.datetime(2012,3,1)+dt.timedelta(i,0,0))
y = cumsum(rnd.randn(T))
```

A basic plot is produced in the same manner, and is depicted in panel (c). Note the labels overlap and so this figure is not acceptable.

```
fig = figure()
ax = fig.add_subplot(111)
ax.plot(x,y)
draw()
```

A call to `autofmt_xdate()` can be used to address the issue of overlapping labels. This is shown in panel (d).

```
fig.autofmt_xdate()
draw()
```

While the formatted x-axis dates are an improvement, they are still unsatisfactory in that the date labels have too much information (month, day and year) and are not at the start of the month. The next piece of code shows how markers can be placed at the start of the month using `MonthLocator` which is in the `matplotlib.dates` module. This idea is to construct a `MonthLocator` instance (which is a class), and then to pass this axes using `xaxis.set_major_locator` which determines the location of major tick marks (minor tick marks can be set using `xaxis.set_mijor_locator`). This will automatically place ticks on the 1st of every month. Other locators are available, including `YearLocator` and `WeekdayLocator`, which place ticks on the first day of the year and on week days, respectively. The second change is to format the labels on the x-axis to have the short month name and year. This is done using `DateFormatter` which takes a custom format string containing the desired format. Options for formatting include:

- `%Y` - 4 digit numeric year

- `%m` - Numeric month

- `%d` - Numeric day

- `%b` - Short month name

- `%H` - Hour

- `%M` - Minute

- `%D` - Named day

These can be combined along with other characters to produce format strings. For example, `%b %d, %Y` would produce a string with the format Mar 1, 2012. Finally `autofmt_xdate` is used to rotate the labels. The result of running this code is in panel (e).

```
months = mdates.MonthLocator()
ax.xaxis.set_major_locator(months)
fmt = mdates.DateFormatter('%b %Y')
ax.xaxis.set_major_formatter(fmt)
fig.autofmt_xdate()
draw()
```

Note that March 1 is not present in the figure in panel (e). This is because the plot doesn't actually include the date March 1 12:00:00 AM, but starts slightly later. To address this, simply change the axis limits using first calling `get_xlim` to get the 2-element tuple containing the limits, change the it to include March 1 12:00:00 AM using `set_xlim`. The line between these call is actually constructing the correctly formatted date. Internally, matplotlib uses serial dates which are simply the number of days past some initial date. For example March 1, 2012 12:00:00 AM is 734563.0, March 2, 2012 12:00:00 AM is 734564.0 and March 2, 2012 12:00:00 PM is 734563.5. The function `date2num` can be used to convert datetimes to serial dates. The output of running this final price of code on the existing figure is presented in panel (f)

```
xlim = list(ax.get_xlim())
xlim[0] = mdates.date2num(dt.datetime(2012,3,1))
ax.set_xlim(xlim)
draw()
```

15.2.5 Shading Areas

For a simple demonstration of the range of matplotlib, consider the problem of producing a plot of Macroeconomic time series with shaded regions to indicate business conditions. Capacity utilization data from FRED has been used to illustrate the steps needed to produce a plot with the time series, dates and shaded regions indicate periods classified as recessions by the National Bureau of Economic Research.

The code has been split into two parts. The first is the code needed to read the data, find the common dates, and finally format the data so that only the common sample is retained.

```
# Reading the data
from matplotlib.pyplot import figure, plot_date, axis, draw
import matplotlib.mlab as mlab

# csv2rec for simplicity
recessionDates = mlab.csv2rec('USREC.csv',skiprows=0)
capacityUtilization = mlab.csv2rec('TCU.csv')
d1 = set(recessionDates['date'])
d2 = set(capacityUtilization['date'])

# Find the common dates
commonDates = d1.intersection(d2)
commonDates = list(commonDates)
commonDates.sort()

# And the first date
firstDate = min(commonDates)

# Find the data after the first date
plotData = capacityUtilization[capacityUtilization['date']>firstDate]
shadeData = recessionDates[recessionDates['date']>firstDate]
```

The second part of the code produces the plot. Most of the code is very simple. It begins by constructing a figure, then add_subplot to the figure, and the plotting the data using plot. fill_between is only one of many useful functions in matplotlib – it fills an area whenever a variable is 1, which is the structure of the recession indicator. The final part of the code adds a title with a custom font (set using a dictionary), and then changes the font and rotation of the axis labels. The output of this code is figure 15.10.

```
# The shaded plot
x = plotData['date']
y = plotData['value']

# z is the shading values, 1 or 0
z = shadeData['value']!=0
```
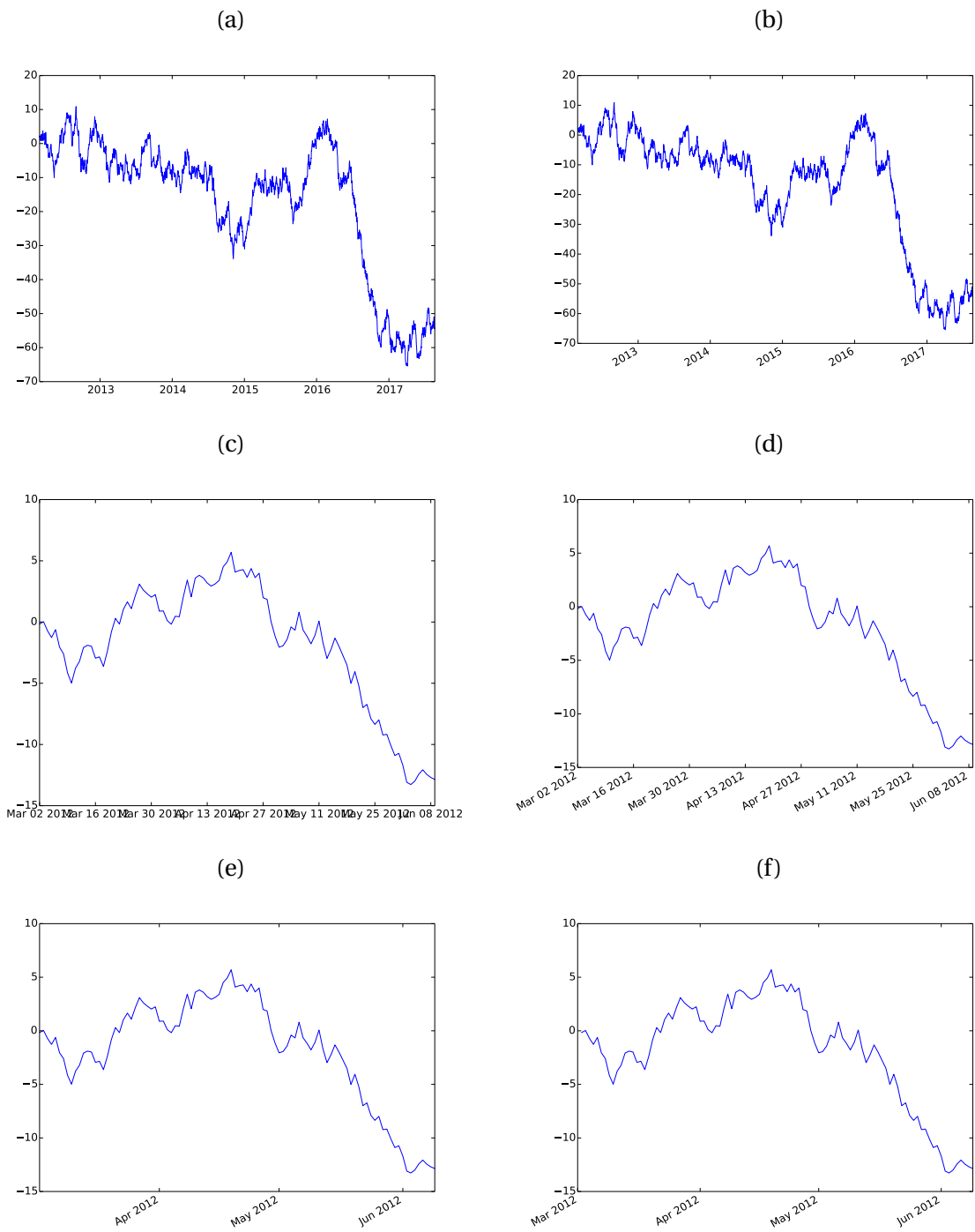
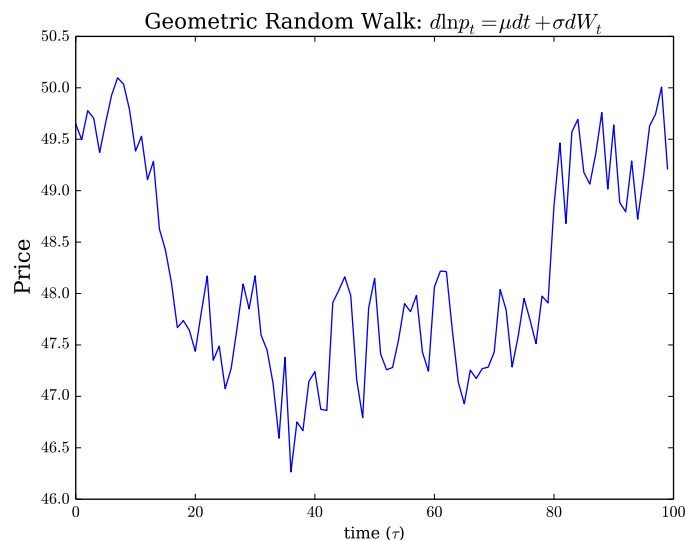Figure 15.9: Figures with dates and additional formatting.

Figure 15.10: A plot of capacity utilization (US data) with shaded regions indicating NBER recession dates.

```
# Figure
fig = figure()
ax = fig.add_subplot(111)
plot_date(x,y,'r-')
limits = axis()
font = { 'fontname':'Times New Roman', 'fontsize':14 }
ax.fill_between(x, limits[2], limits[3], where=z, edgecolor='#BBBBBB', \
    facecolor='#BBBBBB', alpha=0.5)
axis(ymin=limits[2])
ax.set_title('Capacity Utilization',font)
xl = ax.get_xticklabels()
for label in xl:
    label.set_fontname('Times New Roman')
    label.set_fontsize(14)
    label.set_rotation(45)
yl = ax.get_yticklabels()
for label in yl:
    label.set_fontname('Times New Roman')
    label.set_fontsize(14)
draw()
```

15.2.6 TeX in plots

Matplotlib supports using TeX in plots. The only steps needed are the first three lines in the code below, which configure some settings. the labels use raw mode (`r'...'`) to avoid needing to escape the \ in the TeX string. The final plot with TeX in the labels is presented in figure 15.11.

```
>>> from matplotlib import rc
>>> rc('text', usetex=True)
>>> rc('font', family='serif')
```
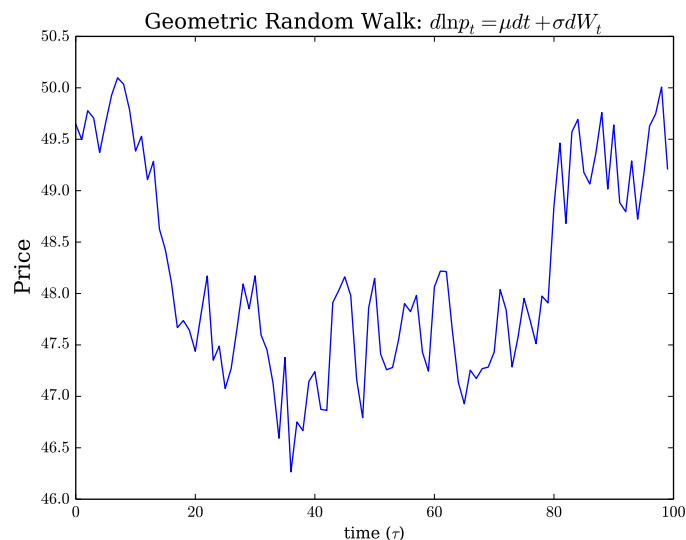
Figure 15.11: A plot that uses TeX in the labels.

```
>>> y = 50*exp(.0004 + cumsum(.01*randn(100)))
>>> plot(y)
>>> xlabel(r'time ($\tau$)')
>>> ylabel(r'Price',fontsize=16)
>>> title(r'Geometric Random Walk: $d\ln p_t = \mu dt + \sigma dW_t$',fontsize=16)
>>> rc('text', usetex=False)
```

15.3 3D Plotting

The 3D plotting capabilities of matplotlib are decidedly weaker than the 2D plotting facilities, and yet the 3D capabilities are typically adequate for most application (especially since 3D graphics are rarely necessary, and often distracting).

15.3.1 Line Plots

Line plot in 3D are virtually identical to plotting in 2D, except that 3 1-dimensional vectors are needed: x, y and z (height). This simple example demonstrates how `plot` can be used with the keyword argument zs to construct a 3D line plot. The line that sets up the axis using `Axed3D(fig)` is essential when producing 3D graphics. The other new command, `view_init`, is used to rotate the view using code (the view can be interactive rotated in the figure window). The result of running the code below is presented in figure 15.12.

```
>>> from mpl_toolkits.mplot3d import Axes3D
>>> x = linspace(0,6*pi,600)
>>> z = x.copy()
>>> y = sin(x)
>>> x=  cos(x)
>>> fig = plt.figure()
>>> ax = Axes3D(fig) # Different usage
>>> ax.plot(x, y, zs=z, label='Spiral')
```
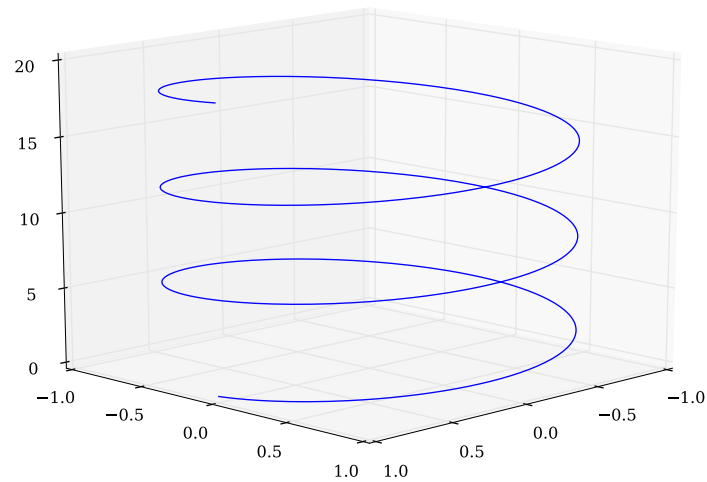
Figure 15.12: A 3D line plot constructed using `plot`.

```
>>> ax.view_init(15,45)
>>> plt.draw()
```

15.3.2 Surface and Mesh (Wireframe) Plots

Surface and mesh or wireframe plots are occasionally useful for visualizing functions with 2 inputs, such as a bivariate probability density. This example produces both types of plots for a bivariate normal PDF with mean 0, unit variances and correlation of 50%. The first block of code generates the points to use in the plot with `meshgrid` and evaluates the PDF for all combinations of x and y.

```
from numpy import linspace, meshgrid, mat, zeros, shape, sqrt
import numpy.linalg as linalg

x = linspace(-3,3,100)
y = linspace(-3,3,100)
x,y = meshgrid(x,y)
z = mat(zeros(2))
p = zeros(shape(x))
R = matrix([[1,.5],[.5,1]])
Rinv = linalg.inv(R)
for i in xrange(len(x)):
    for j in xrange(len(y)):
        z[0,0] = x[i,j]
        z[0,1] = y[i,j]
        p[i,j] = 1.0/(2*pi)*sqrt(linalg.det(R))*exp(-(z*Rinv*z.T)/2)
```

The next code segment produces a mesh (wireframe) plot using `plot_wireframe`. The setup of the case is identical to that of the 3D line, and the call `ax = Axes3D(fig)` is again essential. The figure is drawn using the 2-dimensional arrays x, y and p. The output of this code is presented in panel (a) of 15.13.
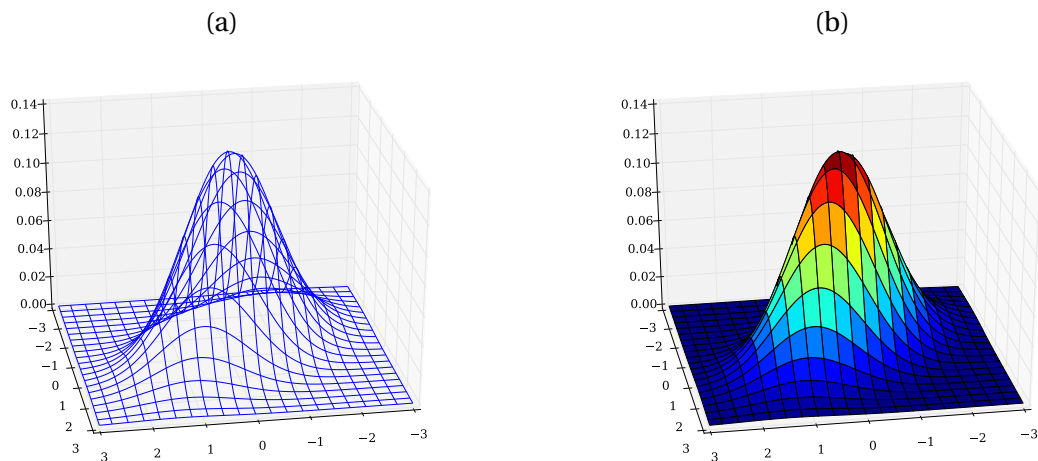
159

Figure 15.13: 3D figures produced using `plot_wireframe` and `plot_surface`.

```
>>> from mpl_toolkits.mplot3d import Axes3D
>>> fig = plt.figure()
>>> ax = Axes3D(fig)
>>> ax.plot_wireframe(x, y, p, rstride=5, cstride=5)
>>> ax.view_init(29,80)
>>> plt.draw()
```

Producing a surface plot is identical, only that a color map is needed from the module `matplotlib.cm` to provide different colors across the range of values. The output of this code is presented in panel (b).

```
>>> import matplotlib.cm as cm
>>> fig = plt.figure()
>>> ax = Axes3D(fig)
>>> ax.plot_surface(x, y, p, rstride=5, cstride=5, cmap=cm.jet)
>>> ax.view_init(29,80)
>>> plt.draw()
```

15.3.3 Contour Plots

Contour plots are not technically 3D, although they are used as a 2D representation of 3D data. Since they are ultimately 2D, little setup is needed, aside from a call to `contour` using the same inputs as `plot_surface` and `plot_wireframe`. The output of the code below is in figure 15.14.

```
>>> fig = plt.figure()
>>> ax = fig.gca()
>>> ax.contour(x,y,p)
>>> plt.draw()
```
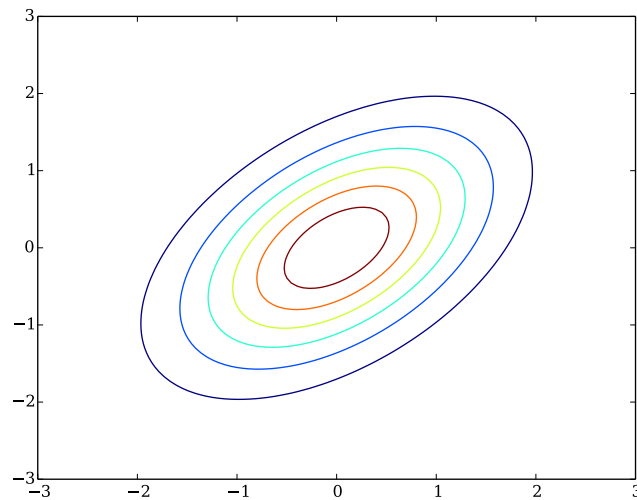
160

Figure 15.14: Contour plot produced using `contour`.

15.4 General Plotting Functions

figure

`figure` is used to open a figure window, and can be used to generate axes. `fig = figure(n)` produces a figure object with id *n*, and assigns the object to `fig`.

add_subplot

`add_subplot` is used to add axes to a figure. `ax = fig.add_subplot(111)` can be used to add a basic axes to a figure. `ax = fig.add_subplot(m,n,i)` can be used to add an axes to a non-trivial figure with a *m* by *n* grid of plots.

close

`close` closes figures. `close(n)` closes the figure with id *n*, and `close('all')` closes all figure windows.

show

`show` is used to force an update to a figure, and pauses execution if not used in an interactive console (close the figure window to resume execution). `show` should not be used in standalone Python programs – `draw` should be used instead.

draw

`draw` forces an update to a figure.

15.5 Exporting Plots

Exporting plots is simple using `savefig('`*filename.ext*`')` where *ext* determines the type of exported file to produce. *ext* can be one of png, pdf, ps, eps or svg.

```
>>> plot(randn(10,2))
>>> savefig('figure.pdf') # PDF export
>>> savefig('figure.png') # PNG export
>>> savefig('figure.svg') # Scalable Vector Graphics export
```

savefig has a number of useful keyword arguments. In particular, dpi is useful when exporting png files. The default dpi is 100.

```
>>> plot(randn(10,2))
>>> savefig('figure.png', dpi = 600) # High resolution PNG export
```

15.6 Exercises

1. Download data for the past 20 years for the S&P 500 from Yahoo!. Plot the price against dates, and ensure the date display is reasonable.

2. Compute Friday-to-Friday returns using the log difference of closing prices and produce a histogram. Experiment with the number of bins.

3. Compute the percentage of weekly returns and produce a pie chart containing the percentage of weekly returns in each of:

    (a) $r \leq -2\%$

    (b) $-2\% < r \leq 0\%$

    (c) $0 < r \leq 2\%$

    (d) $r > 2\%$

4. Download 20 years of FTSE data, and compute Friday-to-Friday returns. Produce a scatter plot of the FTSE returns against the S&P 500 returns. Be sure to label the axes and provide a title.

5. Repeat exercise 4, but add in the fit line showing is the OLS fit of regressing FTSE on the S&P plus a constant.

6. Compute EWMA variance for both the S&P 500 and FTSE and plot against dates. An EWMA variance has $\sigma_{t'}^2 = (1 - \lambda) r_{t-1}^2 + \sigma_{t-1}^2$ where $r_0^2 = \sigma_0^2$ is the full sample variance and $\lambda = 0.97$.

7. Explore the chart gallery on the matplotlib website.

Chapter 16

Structured Arrays

pandas, the topic of Chapter 17, has substantially augmented the structured arrays provided by NumPy. The pandas Series and DataFrame types are the preferred method to handle heterogeneous data and/or data sets which have useful metadata. This chapter has been retained since the NumPy data structures may be encountered when using some functions, or in legacy code produced by others.

The standard, homogeneous NumPy array is a highly optimized data structure where all elements have the same data type (e.g. float) and can be accessed using slicing in many dimensions. These data structures are essential for high-performance numerical computing – especially for linear algebra. Unfortunately, actual data is often heterogeneous (e.g. mixtures of dates, strings and numbers) and it is useful to track series by meaningful names, not just "column 0". These features are not available in a homogeneous NumPy array. However, NumPy also supports mixed arrays which solve both of these issues and so are a useful data structures for managing data prior to statistical analysis. Conceptually, a mixed array with named columns is similar to a spreadsheet where each column can have its own name and data type.

16.1 Mixed Arrays with Column Names

A mixed NumPy array can be initialized using array, zeros or other functions which create arrays and allow the data type to be directly specified. Mixed arrays are in many ways similar to standard NumPy arrays, except that the dtype input to the function is specified either using tuples of the form (*name*, *type*), or using a dictionary of the form {'names':*name*s,'formats':*formats*) where *names* is a tuple of column names and *formats* is a tuple of NumPy data types.

```
>>> x = zeros(4,[('date','int'),('ret','float')])
>>> x = zeros(4,{'names': ('date','ret'), 'formats': ('int', 'float')})
>>> x
array([(0, 0.0), (0, 0.0), (0, 0.0), (0, 0.0)],
      dtype=[('date', '<i4'), ('ret', '<f8')])
```

These two command are identical, and illustrate the two methods to create an array which contain a named column "date", for integer data, and a named column "ret" for floats. Named columns allows for access using dictionary-type syntax.

```
>>> x['date']
array([0, 0, 0, 0])
```

```
>>> x['ret']
array([0.0, 0.0, 0.0, 0.0])
```

Standard multidimensional slice notation is not available since heterogeneous arrays behave like nested lists and not homogeneous NumPy arrays.

```
>>> x[0] # Data tuple 0
(0, 0.0)

>>> x[:3] # Data tuples 0, 1 and 2
array([(0, 0.0), (0, 0.0), (0, 0.0)],
      dtype=[('date', '<i4'), ('ret', '<f8')])

>>> x[:,1] # Error
IndexError: too many indices
```

The first two commands show that the array is composed of tuples and so differs from standard homogeneous NumPy arrays. The error in the third command occurs since columns are accessed using names and not multidimensional slices.

16.1.1 Data Types

A large number of primitive data types are available in NumPy.

Type	Syntax	Description
Boolean	b	True/False
Integers	i1,i2,i4,i8	1 to 8 byte signed integers $(-2^{B-1}, \ldots 2^{B-1} - 1)$
Unsigned Integers	u1,u2,u4,u8	1 to 8 byte signed integers $(0, \ldots 2^{B})$
Floating Point	f4,f8	Single (4) and double (8) precision float
Complex	c8,c16	Single (8) and double (16) precision complex
Object	On	Generic n-byte object
String	Sn, an	n-letter string
Unicode String	Un	n-letter unicode string

The majority of data types are for numeric data, and are simple to understand. The n in the string data type indicates the maximum length of a string. Attempting to insert a string with more than n characters will truncate the string. The object data type is somewhat abstract, but allows for storing Python objects such as datetimes.

Custom data types can be built using dtype. The constructed data type can then be used in the construction of a mixed array.

```
>>> type = dtype([('var1','f8'), ('var2','i8'), ('var3','u8')])
>>> type
dtype([('var1', '<f8'), ('var2', '<i8'), ('var3', '<u8')])
```

Data types can even be nested to create a structured environment where one of the "variables" has multiple values. Consider this example which uses a nested data type to contain the bid and ask price of a stock, along with the time of the transaction.

```
>>> ba_type = dtype([('bid','f8'), ('ask','f8')])
```

```
>>> t = dtype([('date', 'O8'), ('prices', ba_type)])
>>> data = zeros(2,t)
>>> data
array([(0, (0.0, 0.0)), (0, (0.0, 0.0))],
      dtype=[('date', 'O'), ('prices', [('bid', '<f8'), ('ask', '<f8')])])

>>> data['prices']
array([(0.0, 0.0), (0.0, 0.0)],
      dtype=[('bid', '<f8'), ('ask', '<f8')])

>>> data['prices']['bid']
array([ 0.,  0.])
```

In this example, data is an array where each item has 2 elements, the date and the price. Price is also an array with 2 elements. Names can also be used to access values in nested arrays (e.g. data['prices']['bid'] returns an array containing all bid prices). In practice nested arrays can almost always be expressed as a non-nested array without loss of fidelity.

Determining the size of object   NumPy arrays can store objects which are anything which fall outside of the usual data types. One example of a useful, but abstract, data type is datetime. One method to determine the size of an object is to create a plain array containing the object – which will automatically determine the data type – and then to query the size from the array.

```
>>> import datetime as dt
>>> x = array([dt.datetime.now()])
>>> x.dtype.itemsize # The size in bytes
>>> x.dtype.descr # The name and description
```

16.1.2 Example: TAQ Data

TAQ is the NYSE Trade and Quote database which contains all trades and quotes of US listed equities which trade on major US markets (not just the NYSE). A record from a trade contains a number of fields:

- Date - The Date in YYYYMMDD format stored as a 4-byte unsigned integer

- Time - Time in HHMMSS format, stored as a 4-byte unsigned integer

- Size - Number of shares trades, stores as a 4 byte unsigned integer

- G127 rule indicator - Numeric value, stored as a 2 byte unsigned integer

- Correction - Numeric indicator of a correction, stored as a 2 byte unsigned integer

- Condition - Market condition, a 2 character string

- Exchange - The exchange where the trade occurred, a 1-character string

First consider a data type which stores the data in an identical format.

```
>>> t = dtype([('date', 'u4'), ('time', 'u4'),
          ('size', 'u4'), ('price', 'f8'),
          ('g127', 'u2'), ('corr', 'u2'),
          ('cond', 'S2'), ('ex', 'S2')])
>>> taqData = zeros(10, dtype=t)
>>> taqData[0] = (20120201,120139,1,53.21,0,0,'','N')
```

An alternative is to store the date and time as a `datetime`, which is an 8-byte object.

```
>>> import datetime as dt
>>> t = dtype([('datetime', 'O8'), ('size', 'u4'), ('price', 'f8'), \
          ('g127', 'u2'), ('corr', 'u2'), ('cond', 'S2'), ('ex', 'S2')])
>>> taqData = zeros(10, dtype=t)
>>> taqData[0] = (dt.datetime(2012,2,1,12,01,39),1,53.21,0,0,'','N')
```

16.2 Record Arrays

The main feature of record arrays, that the series can be accessed by series name as a property of a variable, is also available in a pandas' DataFrame.

Record arrays are closely related to mixed arrays with names. The primary difference is that elements record arrays can be accessed using *variable.name* format.

```
>>> x = zeros((4,1),[('date','int'),('ret','float')])
>>> y = rec.array(x)
>>> y.date
array([[0],
       [0],
       [0],
       [0]])

>>> y.date[0]
array([0])
```

In practice record arrays may be slower than standard arrays, and unless the *variable.name* is really important, record arrays are not compelling.

Chapter 17

pandas

pandas is a high-performance module that primarily provides a comprehensive set of structures for working with data. pandas excels at handling structured data, such as data sets containing many variables, working with missing values and merging across multiple data sets. While extremely useful, pandas is not an essential component of the Python scientific stack unlike NumPy, SciPy or matplotlib, and so while pandas doesn't make data analysis possible in Python, it makes it much easier. pandas also provides high-performance, robust methods for importing from and exporting to to a wide range of formats.

17.1 Data Structures

pandas provides a set of data structures which include Series, DataFrames and Panels. Series are 1-dimensional arrays. DataFrames are collections of Series and so are 2-dimensional, and Panels are collections of DataFrames, and so are 3-dimensional.

Note that the Panel type is relatively immature and is not covered in this chapter.

17.1.1 Series

Series are the primary building block of the data structures in pandas, and in many ways a Series behaves similarly to a NumPy array. A Series is initialized using a list or tupel, or directly from a NumPy array.

```
>>> a = array([0.1, 1.2, 2.3, 3.4, 4.5])
>>> a
array([ 0.1,  1.2,  2.3,  3.4,  4.5])

>>> from pandas import Series
>>> s = Series([0.1, 1.2, 2.3, 3.4, 4.5])
>>> s
0    0.1
1    1.2
2    2.3
3    3.4
4    4.5
dtype: float64

>>> s = Series(a) # Direct from NumPy array
```

Series, like arrays, are sliceable. However, unlike a 1-dimensional array, a Series has an additional column, an index – that is, a set of values which are associated to the rows of the Series. In this example, pandas has automatically generated an index using the sequence $0, 1, \ldots$ since none was provided. It is also possible to use other values as the index when initializing the Series using a keyword argument.

```
>>> s = Series([0.1, 1.2, 2.3, 3.4, 4.5], index = ['a','b','c','d','e'])
>>> s
a    0.1
b    1.2
c    2.3
d    3.4
e    4.5
dtype: float64
```

The index is part of the "magic" of the pandas's data structures (Series and DataFraame) and allows for dictionary-like access to elements in the index (in addition to both numeric slicing and logical indices).[1]

```
>>> s['a']
0.10000000000000001

>>> s[0]
0.10000000000000001

>>> s[['a','c']]
a    0.1
c    2.3
dtype: float64

>>> s[[0,2]]
a    0.1
c    2.3
dtype: float64

>>> s[:2]
a    0.1
b    1.2
dtype: float64

>>> s[s>2]
c    2.3
d    3.4
e    4.5
dtype: float64
```

In this examples, 'a' and 'c' behave in the same manner as 0 and 2 would in a standard NumPy array. The elements of an index do not have to be unique which another way in which a Series generalizes a NumPy array.

```
>>> s = Series([0.1, 1.2, 2.3, 3.4, 4.5], index = ['a','b','c','a','b'])
```

---

[1]Using numeric index values other than the default sequence will break scalar selection since there is ambiguity between numerical slicing and index access. For this reason, custom numerical indices should be used with care.

```
>>> s
a    0.1
b    1.2
c    2.3
a    3.4
b    4.5
dtype: float64

>>> s['a']
a    0.1
a    3.4
dtype: float64
```

Series can also be initialized directly from dictionaries.

```
>>> s = Series({'a': 0.1, 'b': 1.2, 'c': 2.3})
>>> s
a    0.1
b    1.2
c    2.3
dtype: float64
```

Series are like NumPy arrays in that they support most numerical operations.

```
>>> s = Series({'a': 0.1, 'b': 1.2, 'c': 2.3})
>>> s * 2.0
a    0.2
b    2.4
c    4.6
dtype: float64

>>> s - 1.0
a    0.2
b    2.4
c    4.6
dtype: float64
```

However, Series are different from arrays when math operations are performed across two Series. In particular, math operations involving two series will perform operations by *aligning indices*. The mathematical operation is performed in two steps. First, the union of all indices is created, and then the mathematical operation is performed on matching indices. Indices that do not match are given the value NaN (not a number), and values are computed for all unique pairs of repeated indices.

```
>>> s1 = Series({'a': 0.1, 'b': 1.2, 'c': 2.3})
>>> s2 = Series({'a': 1.0, 'b': 2.0, 'c': 3.0})
>>> s3 = Series({'c': 0.1, 'd': 1.2, 'e': 2.3})
>>> s1 + s2
a    1.1
b    3.2
c    5.3
dtype: float64
```

```
>>> s1 * s2
a    0.1
b    2.4
c    6.9
dtype: float64

>>> s1 + s3
a    NaN
b    NaN
c    2.4
d    NaN
e    NaN
dtype: float64
```

Mathematical operations performed on series which have non-unique indices will broadcast the operation to all indices which are common. For example, when one array has 2 elements with the same index, and another has 3, adding the two will produce 6 outputs.

```
>>> s1 = Series([1.0,2,3],index=['a']*3)
>>> s2 = Series([4.0,5],index=['a']*2)
>>> s1 + s2
a    5
a    6
a    6
a    7
a    7
a    8
dtype: float64
```

The underlying NumPy array is accessible through the `values` property, and the index is accessible the `index` property, which returns an `Index` type. The NumPy array underlying the index can be retrieved using `values` on the `Index` object returned.

```
>>> s1 = Series([1.0,2,3])
>>> s1.values
array([ 1.,  2.,  3.])

>>> s1.index
Index([u'a', u'a', u'a'], dtype=object)

>>> s1.index.values
array([0, 1, 2], dtype=int64)

>>> s1.index = ['cat','dog','elephant']
>>> s1.index
Index([u'cat', u'dog', u'elephant'], dtype=object)
```

Notable Methods and Properties

Series provide a large number of methods to manipulate data. These can broadly be categorized into mathematical and non-mathematical functions. The mathematical functions are generally very similar

to those in NumPy due to the underlying structure of a Series, and generally do not warrant a separate discussion. In contrast, the non-mathematical methods are unique to pandas.

head and tail

`head()` shows the first 5 rows of a series, and `tail()` shows the last 5 rows.

isnull and notnull

`isnull()` returns a Series with the same indices containing Boolean values indicating `True` for null values which include `NaN` and `None`, among others. `notnull()` returns the negation of `isnull()` – that is, `True` for non-null values, and `False` otherwise.

ix

`ix` is the indexing function and `s.ix[0:2]` is the same as `s[0:2]`. `ix` is more useful for DataFrames.

describe

`describe()` returns a simple set of summary statistics about a Series. The value returned is a series where the index values are name of the statistic computed.

```
>>> s1 = Series(arange(10.0,20.0))
>>> s1.describe()
count    10.00000
mean     14.50000
std       3.02765
min      10.00000
25%      12.25000
50%      14.50000
75%      16.75000
max      19.00000
dtype: float64

>>> summ = s1.describe()
>>> summ['mean']
```

nunique

`nunique()` returns the number of unique values in a Series.

drop and dropna

drop(*labels*) drop elements with the selected labels form a Series.

```
>>> s1 = Series(arange(1.0,6),index=['a','a','b','c','d'])
>>> s1
a    1
a    2
b    3
```

```
c    4
d    5
dtype: float64

>>> s1.drop('a')
b    3
c    4
d    5
dtype: float64
```

dropna() is similar to drop() except that it only drops null values – NaN or similar.

```
>>> s1 = Series(arange(1.0,4.0),index=['a','b','c'])
>>> s2 = Series(arange(1.0,4.0),index=['c','d','e'])
>>> s3 = s1 + s2
>>> s3
a    NaN
b    NaN
c      4
d    NaN
e    NaN
dtype: float64

>>> s3.dropna()
c      4
dtype: float64
```

fillna

fillna(*value*) fills all null values in a series with a specific value.

```
>>> s1 = Series(arange(1.0,4.0),index=['a','b','c'])
>>> s2 = Series(arange(1.0,4.0),index=['c','d','e'])
>>> s3 = s1 + s2
>>> s3.fillna(-1.0)
a    -1
b    -1
c     4
d    -1
e    -1
dtype: float64
```

append

append(*series*) appends one series to another, and is similar to list.append.

replace

replace(*list*, *values*) replaces a set of values in a Series with a new value. replace is similar to fillna except that replace also replaces non-null values.

update

update(*series*) replaces values in a series with those in another series, matching on the index.

```
>>> s1 = Series(arange(1.0,4.0),index=['a','b','c'])
>>> s1
a    1
b    2
c    3
dtype: float64

>>> s2 = Series(-1.0 * arange(1.0,4.0),index=['c','d','e'])
>>> s1.update(s2)
>>> s1
a    1
b    2
c    -1
dtype: float64
```

17.1.2 DataFrame

While the Series class is the building block of data structures in pandas, the DataFrame is the work-horse. DataFrames collect multiple series in the same way that a spreadsheet collects multiple columns of data. In a simple sense, a DataFrame is like a 2-dimensions NumPy array – and when all data is numeric and of the same type (e.g. float64), it is virtually indistinguishable. However, a DataFrame is composed of Series and each Series has its own data type, and so not all DataFrames are representable as homogeneous NumPy arrays.

   A number of methods are available to initialize a DataFrame. The simplest is from a homogeneous NumPy array.

```
>>> from pandas import DataFrame
>>> df = DataFrame(array([[1,2],[3,4]]))
>>> df
   0  1
0  1  2
1  3  4
```

Like a Series, a DataFrame contains the input data as well as row labels. However, since a DataFrame is a collection of columns, it also contains column labels (located along the top edge). When none are provided, the numeric sequence $0, 1, \ldots$ is used.

   Column names are entered using a keyword argument or later by assigning to columns.

```
>>> df = DataFrame(array([[1,2],[3,4]]),columns=['a','b'])
>>> df
   a  b
0  1  2
```

```
1  3  4

>>> df = DataFrame(array([[1,2],[3,4]]))
>>> df.columns = ['dogs','cats']
>>> df
   dog  cat
0    1    2
1    3    4
```

Index values are similarly assigned using either the keyword argument index or by setting the index property.

```
>>> df = DataFrame(array([[1,2],[3,4]]), columns=['dogs','cats'], index=['Alice','Bob'])
>>> df
        dogs  cats
Alice      1     2
Bob        3     4
```

DataFrames can also be created form NumPy arrays with structured data.

```
>>> import datetime
>>> t = dtype([('datetime', 'O8'), ('value', 'f4')])
>>> x = zeros(1,dtype=t)
>>> x[0][0] = datetime.datetime(2013,01,01)
>>> x[0][1] = -99.99
>>> x
array([[(datetime.datetime(2013, 1, 1, 0, 0), -99.98999786376953)],
       dtype=[('datetime', 'O'), ('value', '<f4')])
>>> df = DataFrame(x)
>>> df
            datetime       value
0 2013-01-01 00:00:00 -99.989998
```

In the previous example, the DataFrame has automatically pulled the column names and column types from the NumPy structured data.

The final method to create a DataFrame uses a dictionary containing Series, where the keys contain the column names. The DataFrame will automatically align the data using the common indices.

```
>>> s1 = Series(arange(0.0,5))
>>> s2 = Series(arange(1.0,3))
>>> DataFrame({'one': s1, 'two': s2})
   one  two
0    0    1
1    1    2
2    2    3
3    3    4
4    4    5

>>> s3 = Series(arange(0.0,3))
>>> DataFrame({'one': s1, 'two': s2, 'three': s3})
   one  three  two
0    0      0    1
```

```
1    1    1    2
2    2    2    3
3    3   NaN   4
4    4   NaN   5
```

In the final example, the third series (s3) has fewer values and the DataFrame automatically fills missing values as NaN. Note that is possible to create DataFrames from Series which do not have unique index values, although in these cases the index values of the two series must match *exactly* – that is, have the same index values in the same order.

Manipulating DataFrames

The use of DataFrames will be demonstrated using a data set containing a mix of data types using state-level GDP data from the US. The data set contains both the GDP level between 2009 and 2012 (constant 2005 US$) and the growth rates for the same years as well as a variable containing the region of the state. The data is loaded directly into a DataFrame using read_excel, which is described in Section 17.4.

```
>>> from pandas import read_excel
>>> state_gdp = read_excel('US_state_GDP.xls','Sheet1')
>>> state_gdp.head()
  state_code        state  gdp_2009  gdp_2010  gdp_2011  gdp_2012
0         AK       Alaska     44215     43472     44232     44732
1         AL      Alabama    149843    153839    155390    157272
2         AR     Arkansas     89776     92075     92684     93892
3         AZ      Arizona    221405    221016    224787    230641
4         CA   California   1667152   1672473   1692301   1751002

   gdp_growth_2009  gdp_growth_2010  gdp_growth_2011  gdp_growth_2012 region
0              7.7             -1.7              1.7              1.1     FW
1             -3.9              2.7              1.0              1.2     SE
2             -2.0              2.6              0.7              1.3     SE
3             -8.2             -0.2              1.7              2.6     SW
4             -5.1              0.3              1.2              3.5     FW
```

Selecting Columns

Single columns are selectable using the column name, as in state_gdp['state'], and the value returned in a Series. Multiple columns are similarly selected using a list of column names as in state_gdp[['state_code', 'state']], or equivalently using an Index object.

```
>>> state_gdp[['state_code','state']].head()
  state_code        state
0         AL      Alabama
1         AK       Alaska
2         AZ      Arizona
3         AR     Arkansas
4         CA   California

>>> index = state_gdp.index
>>> state_gdp[index[1:3]].head() # Elements 1 and 2 (0-based counting)
```

```
        state   gdp_2009
0      Alabama    149843
1       Alaska     44215
2      Arizona    221405
3     Arkansas     89776
4   California   1667152
```

Finally, single columns can also be selected using dot-notation and the column name.[2]

```
>>> state_gdp.state_code.head()
0    AL
1    AK
2    AZ
3    AR
4    CA
Name: state_code, dtype: object
```

Selecting Rows

Row selection uses standard numerical slices.

```
>>> state_gdp[1:3]
  state_code     state  gdp_2009  gdp_2010  gdp_2011  gdp_2012
1         AL   Alabama    149843    153839    155390    157272
2         AR  Arkansas     89776     92075     92684     93892

   gdp_growth_2009  gdp_growth_2010  gdp_growth_2011  gdp_growth_2012 region
1             -3.9              2.7              1.0              1.2     SE
2             -2.0              2.6              0.7              1.3     SE
```

A function version is also available using iloc[*rows*] which is identical to the standard slicing syntax. Labeled rows can also be selected using the method loc(*label*).

Finally, rows can also be selected using logical selection using a Boolean array with the same number of elements as the number of rows as the DataFrame.

```
>>> state_long_recession = state_gdp['gdp_growth_2010']<0
>>> state_gdp[state_long_recession].head()
   state_code    state  gdp_2009  gdp_2010  gdp_2011  gdp_2012
1          AK   Alaska     44215     43472     44232     44732
2          AZ  Arizona    221405    221016    224787    230641
28         NV   Nevada    110001    109610    111574    113197
50         WY  Wyoming     32439     32004     31231     31302

    gdp_growth_2009  gdp_growth_2010  gdp_growth_2011  gdp_growth_2012
1               7.7             -1.7              1.7              1.1
2              -8.2             -0.2              1.7              2.6
28             -8.2             -0.4              1.8              1.5
50              3.4             -1.3             -2.4              0.2
```

---

[2]The column name must be a legal Python variable name, and so cannot contain spaces or reserved notation.

Selecting Rows and Columns

Since the behavior of slicing depends on whether the input is text (selects columns) or numeric/Boolean (selects rows), it isn't possible to use standard slicing to select both rows and columns. Instead, the selector method ix[*rowselector*, *colselector*] allows joint selection where *rowselector* is either a scalar selector, a slice selector, a Boolean array, a numeric selector or a row label or list of row labels and *colselector* is a scalar selector, a slice selector, a Boolean array, a numeric selector or a column name or list of column names.

```
>>> state_gdp.ix[state_long_recession,'state']
1       Alaska
2       Arizona
28       Nevada
50      Wyoming
Name: state, dtype: object

>>> state_gdp.ix[state_long_recession,['state','gdp_growth_2009','gdp_growth_2010']]
      state  gdp_growth_2009  gdp_growth_2010
1    Alaska              7.7             -1.7
2   Arizona             -8.2             -0.2
28   Nevada             -8.2             -0.4
50  Wyoming              3.4             -1.3

>>> state_gdp.ix[10:15,0] # Slice and scalar
10    GA
11    HI
12    IA
13    ID
14    IL
15    IN
```

Adding Columns

Columns are added using one of three methods. The most obvious is to add a Series merging along the index using a dictionary-like syntax.

```
>>> state_gdp_2012 = state_gdp[['state','gdp_2012']]
>>> state_gdp_2012.head()
        state  gdp_2012
0     Alabama    157272
1      Alaska     44732
2     Arizona    230641
3    Arkansas     93892
4  California   1751002

>>> state_gdp_2012['gdp_growth_2012'] = state_gdp['gdp_growth_2012']
>>> state_gdp_2012.head()
        state  gdp_2012  gdp_growth_2012
0     Alabama    157272              1.2
1      Alaska     44732              1.1
```

177

```
2     Arizona    230641         2.6
3     Arkansas    93892         1.3
```

This syntax always adds the column at the end. `insert(`*location*, *column_name*, *series*`)` inserts a Series at an specified location, where *location* uses 0-based indexing (i.e. 0 places the column first, 1 places it second, etc.), *column_name* is the name of the column to be added and *series* is the series data. *series* is either a Series or another object that is readily convertible into a Series such as a NumPy array.

```
>>> state_gdp_2012 = state_gdp[['state','gdp_2012']]
>>> state_gdp_2012.insert(1,'gdp_growth_2012',state_gdp['gdp_growth_2012'])
>>> state_gdp_2012.head()
       state  gdp_growth_2012  gdp_2012
0    Alabama              1.2    157272
1     Alaska              1.1     44732
2    Arizona              2.6    230641
3   Arkansas              1.3     93892
4 California              3.5   1751002
```

Inserting columns with different indices or fewer items than the DataFrame results in a DataFrame with the *original indices* with `NaN`-filled missing values in the new Series.

```
>>> state_gdp_2012 = state_gdp.ix[0:2,['state','gdp_2012']]
>>> state_gdp_2012
     state  gdp_2012
0  Alabama    157272
1   Alaska     44732
2  Arizona    230641

>>> gdp_2011 = state_gdp.ix[1:4,'gdp_2011']
>>> state_gdp_2012['gdp_2011'] = gdp_2011
     state  gdp_2012  gdp_2011
0  Alabama    157272       NaN
1   Alaska     44732     44232
2  Arizona    230641    224787
```

Deleting Columns

Columns are deleted using either the `del` keyword or by using `pop(`*column*`)` on the DataFrame. The behavior of these two differs slightly: `del` will simply delete the Series from the DataFrame while `pop()` will delete the Series and return the Series as an output.

```
>>> state_gdp_copy = state_gdp.copy()
>>> state_gdp_copy = state_gdp_copy[['state_code','gdp_growth_2011','gdp_growth_2012']]
>>> state_gdp_copy.index = state_gdp['state_code']
>>> state_gdp_copy.head()
            gdp_growth_2011  gdp_growth_2012
state_code
AK                      1.7              1.1
AL                      1.0              1.2
AR                      0.7              1.3
AZ                      1.7              2.6
```

```
CA                      1.2             3.5

>>> gdp_growth_2012 = state_gdp_copy.pop('gdp_growth_2012')
>>> gdp_growth_2012.head()
state_code
AK         1.1
AL         1.2
AR         1.3
AZ         2.6
CA         3.5
Name: gdp_growth_2012, dtype: float64

>>> state_gdp_copy.head()
          gdp_growth_2011
state_code
AK                    1.7
AL                    1.0
AR                    0.7
AZ                    1.7
CA                    1.2

>>> del state_gdp_copy['gdp_growth_2011']
>>> state_gdp_copy.head()
Empty DataFrame
Columns: []
Index: [AK, AL, AR, AZ, CA]
```

Notable Properties and Methods

`drop`, `dropna` and `drop_duplicates`

drop(), dropna() and drop_duplicates() can all be used to drop rows or columns from a DataFrame. drop(*labels*) drops rows based on the row labels in a label or list *labels*. drop(*column_name*, axis=1) drops columns based on a column name or list *column names*.

dropna() drops rows with any NaN (or null) values. It can also be used with the keyword argument dropna(how='all') to only drop rows which have missing values for all variables. Finally, drop_duplicates() removes rows which are duplicates or other rows, and is used with the keyword argument drop_duplicates(cols=*col_lis* to only consider a subset of all columns when checking for duplicates.

`values` and `index`

values retrieves a the NumPy array (structured if the data columns are heterogeneous) underlying the DataFrame, and index returns the index of the DataFrame or can be assigned to to set the index.

`fillna`

fillna() fills NaN or other null values with other values. The simplest use fill all NaNs with a single value and is called fillna(value=*value*). Using a dictionary allows for more sophisticated na-filling with col-

umn names as the keys and the replacements as the values.

```
>>> df = DataFrame(array([[1, nan],[nan, 2]]))
>>> df.columns = ['one','two']
>>> replacements = {'one':-1, 'two':-2}
>>> df.fillna(value=replacements)
   one  two
0    1   -2
1   -1    2
```

T and transpose

T and transpose are identical – both swap rows and columns of a DataFrame. T operates like a property, while transpose is used as a method.

sort and sort_index

sort and sort_index are identical in their outcome and only differ in the inputs. The default behavior of sort is to sort by the row labels. Using a keyword argument axis=1 sorts the DataFrame by the column names. Both can also be used to sort by the data in the DataFrame. sort does this using the keyword argument columns, which is either a single column name or a list of column names. Using a list produces a nested sort. sort_index uses the keyword argument by to do the same. Another keyword argument determines the direction of the sort (ascending by default). sort(ascending=False) will produce a descending sort, and when using a nested sort, the sort direction is specified using a list sort(columns=['one','two'], ascending=[True,False]) where each entry corresponds to the columns used to sort.

```
>>> df = DataFrame(array([[1, 3],[1, 2],[3, 2],[2,1]]), columns=['one','two'])
>>> df.sort(columns='one')
   one  two
0    1    3
1    1    2
3    2    1
2    3    2

>>> df.sort(columns=['one','two'])
   one  two
1    1    2
0    1    3
3    2    1
2    3    2

>>> df.sort(columns=['one','two'], ascending=[0,1])
   one  two
2    3    2
3    2    1
1    1    2
0    1    3
```

The default behavior is to *not* sort in-place and so it is necessary to assign the output of a sort. Using the keyword argument inplace=True will change the default behavior.

pivot

pivot reshapes a table using column values when reshaping. pivot takes three inputs. The first, index, defines the column to use as the index of the pivoted table. The second, columns, defines the column to use to form the column names, and values defines the columns to for the data in the constructed DataFrame. The following example show how a flat DataFrame with repeated values is transformed into a more meaningful representation.

```
>>> prices = [101.0,102.0,103.0]
>>> tickers = ['GOOG','AAPL']
>>> import itertools
>>> data = [v for v in itertools.product(tickers,prices)]
>>> dates = pandas.date_range('2013-01-03',periods=3)
>>> df = DataFrame(data, columns=['ticker','price'])
>>> df['dates'] = dates.append(dates)
>>> df
  ticker  price              dates
0   GOOG    101 2013-01-03 00:00:00
1   GOOG    102 2013-01-04 00:00:00
2   GOOG    103 2013-01-05 00:00:00
3   AAPL    101 2013-01-03 00:00:00
4   AAPL    102 2013-01-04 00:00:00
5   AAPL    103 2013-01-05 00:00:00

>>> df.pivot(index='dates',columns='ticker',values='price')
ticker      AAPL  GOOG
dates
2013-01-03   101   101
2013-01-04   102   102
2013-01-05   103   103
```

stack and unstack

stack and unstack transform a DataFrame to a Series (stack) and back to a DataFrame (unstack).

concat and append

append appends rows of another DataFrame to the end of an existing DataFrame. If the data appended has a different set of columns, missing values are NaN-filled. The keyword argument ignore_index=True instructs append to ignore the existing index in the appended DataFrame. This is useful when index values are not meaningful, such as when they are simple numeric values.

FIXME: Concat!!!

reindex, reindex_like and reindex_axis

reindex changes the labels while null-filling any missing values, which is useful for selecting subsets of a DataFrame or re-ordering rows. reindex_like behaves similarly, but uses the index from another DataFrame. The keyword argument axis directs reindex_axis to alter either rows or columns.

```
>>> original = DataFrame([[1,1],[2,2],[3.0,3]],index=['a','b','c'], columns=['one','two'])
>>> original.reindex(index=['b','c','d'])
   one  two
b    2    2
c    3    3
d  NaN  NaN

>>> different = DataFrame([[1,1],[2,2],[3.0,3]],index=['c','d','e'], columns=['one','two'])
>>> original.reindex_like(different)
   one  two
c    3    3
d  NaN  NaN
e  NaN  NaN

>>> original.reindex_axis(['two','one'], axis = 1)
```

merge and join

merge and join provide SQL-like operations for merging the DataFrames using row labels or the contents of columns. The primary difference between the two is that merge defaults to using column contents while join defaults to using index labels. Both commands take a large number of optional inputs. The important keyword arguments are:

- how, which must be one of 'left', 'right', 'outer', 'inner' describes which set of indices to use when performing the join. 'left' uses the indices of the DataFrame that is used to call the method and 'right' uses the DataFrame input into merge or join. 'outer' uses a union of all indices from both DataFrames and 'inner' uses an intersection from the two DataFrames.

- on is a single column name of list of column names to use in the merge. on assumes the names are common. If no value is given for on or left_on/right_on, then the common column names are used.

- left_on and right_on allow for a merge using columns with different names. When left_on and right_on contains the same column names, the behavior is the same as on.

- left_index and right_index indicate that the index labels are the join key for the left and right DataFrames.

```
>>> left = DataFrame([[1,2],[3,4],[5,6]],columns=['one','two'])
>>> right = DataFrame([[1,2],[3,4],[7,8]],columns=['one','three'])
>>> left.merge(right,on='one') # Same as how='inner'
   one  two  three
0    1    2      2
1    3    4      4

>>> left.merge(right,on='one', how='left')
   one  two  three
0    1    2      2
1    3    4      4
2    5    6    NaN
```

```
>>> left.merge(right,on='one', how='right')
   one  two  three
0   1    2      2
1   3    4      4
2   7   NaN     8

>>> left.merge(right,on='one', how='outer')
   one  two  three
0   1    2      2
1   3    4      4
2   5    6     NaN
3   7   NaN     8
```

update

update updates the values in one DataFrame using the non-null values from another DataFrame, using
the index labels to determine which records to update.

```
>>> left = DataFrame([[1,2],[3,4],[5,6]],columns=['one','two'])
>>> left
   one  two
0   1    2
1   3    4
2   5    6

>>> right = DataFrame([[nan,12],[13,nan],[nan,8]],columns=['one','two'],index=[1,2,3])
>>> right
   one  two
1  NaN   12
2   13  NaN
3  NaN    8

>>> left.update(right) # Updates values in left
>>> left
   one  two
0   1    2
1   3   12
2   13   6
```

groupby

groupby produces a DataFrameGroupBy object which is a grouped DataFrame, and is useful when a DataFrame
has columns containing group data (e.g. sex or race in cross-sectional data). By itself, groupby does not
produce any output, and so it is necessary to use other functions on the output DataFrameGroupBy.

```
>>> subset = state_gdp[['gdp_growth_2009','gdp_growth_2010','region']]
>>> subset.head()
   gdp_growth_2009  gdp_growth_2010 region
0            7.7            -1.7     FW
```

```
1                -3.9                   2.7        SE
2                -2.0                   2.6        SE
3                -8.2                  -0.2        SW
4                -5.1                   0.3        FW

>>> grouped_data = subset.groupby(by='region')
>>> grouped_data.groups # Lists group names and index labels for group membership
{u'FW': [0L, 4L, 11L, 33L, 37L, 47L],
 u'GL': [14L, 15L, 22L, 35L, 48L],
 u'MW': [7L, 8L, 20L, 31L, 34L, 38L],
 u'NE': [6L, 19L, 21L, 30L, 39L, 46L],
 u'PL': [12L, 16L, 23L, 24L, 28L, 29L, 41L],
 u'RM': [5L, 13L, 26L, 44L, 50L],
 u'SE': [1L, 2L, 9L, 10L, 17L, 18L, 25L, 27L, 40L, 42L, 45L, 49L],
 u'SW': [3L, 32L, 36L, 43L]}

>>> grouped_data.mean()   # Same as a pivot table
        gdp_growth_2009   gdp_growth_2010
region
FW             -2.483333          1.550000
GL             -5.400000          3.660000
MW             -1.250000          2.433333
NE             -2.350000          2.783333
PL             -1.357143          2.900000
RM             -0.940000          1.380000
SE             -2.633333          2.850000
SW             -2.175000          1.325000

>>> grouped_data.std()   # Can use other methods
        gdp_growth_2009   gdp_growth_2010
region
FW              5.389403          2.687564
GL              2.494995          1.952690
MW              2.529624          1.358921
NE              0.779102          1.782601
PL              2.572196          2.236068
RM              2.511573          1.522170
SE              2.653071          1.489051
SW              4.256270          1.899781
```

apply

apply executes a function along the columns or rows of a DataFrame. The following example applies the mean function both down columns and across rows, which is a trivial since `mean` could be executed on the DataFrame directly. `apply` is more general since it allows custom functions to be applied to a DataFrame.

```
>>> subset = state_gdp[['gdp_growth_2009','gdp_growth_2010','gdp_growth_2011','gdp_growth_2012']]
>>> subset.index = state_gdp['state_code'].values
>>> subset.head()
    gdp_growth_2009  gdp_growth_2010  gdp_growth_2011  gdp_growth_2012
```

```
AK              7.7             -1.7            1.7             1.1
AL             -3.9              2.7            1.0             1.2
AR             -2.0              2.6            0.7             1.3
AZ             -8.2             -0.2            1.7             2.6
CA             -5.1              0.3            1.2             3.5

>>> subset.apply(mean) # Same as subset.mean()
gdp_growth_2009   -2.313725
gdp_growth_2010    2.462745
gdp_growth_2011    1.590196
gdp_growth_2012    2.103922
dtype: float64


>>> subset.apply(mean, axis=1).head() # Same as subset.mean(axis=1)
AK     2.200
AL     0.250
AR     0.650
AZ    -1.025
CA    -0.025
dtype: float64
```

applymap

applymap is similar to apply, only that it applies element-by-element rather than column- or row-wise.

pivot_table

Pivot tables provide a method to summarize data by groups. A pivot table first forms groups based on an keyword argument rows and then returns an aggregate of all values within the group (using mean by default). The keyword argument aggfun allows for other aggregation function.

```
>>> subset = state_gdp[['gdp_growth_2009','gdp_growth_2010','region']]
>>> subset.head()
   gdp_growth_2009  gdp_growth_2010 region
0             7.7             -1.7     FW
1            -3.9              2.7     SE
2            -2.0              2.6     SE
3            -8.2             -0.2     SW
4            -5.1              0.3     FW


>>> subset.pivot_table(rows='region')
        gdp_growth_2009  gdp_growth_2010
region
FW            -2.483333         1.550000
GL            -5.400000         3.660000
MW            -1.250000         2.433333
NE            -2.350000         2.783333
PL            -1.357143         2.900000
RM            -0.940000         1.380000
SE            -2.633333         2.850000
```

SW	−2.175000	1.325000

17.2 Statistical Function

pandas Series and DataFrame are derived from NumPy arrays and so the vast majority of simple statistical functions are available. This list includes `sum`, `mean`, `std`, `var`, skew, kurt, `prod`, median, quantile, `abs`, `cumsum`, and `cumprod`. DataFrame also supports `cov` and corr – the keyword argument `axis` determines the direction of the operation (0 for down columns, 1 for across rows). Novel statistical routines are described below.

count

`count` returns number of non-null values – that is, those which are not `NaN` or another null value such as `None` or `NaT` (not a time, for datetimes).

describe

`describe` provides a summary of the Series or DataFrame.

```
>>> state_gdp.describe()
          gdp_2009        gdp_2010        gdp_2011        gdp_2012
count    51.000000       51.000000       51.000000       51.000000
mean    246866.980392   252840.666667   256995.647059   263327.313725
std     299134.165365   304446.797050   309689.475995   319842.518074
min      22108.000000    23341.000000    23639.000000    23912.000000
25%      64070.500000    65229.000000    65714.000000    66288.000000
50%     149843.000000   153839.000000   155390.000000   157272.000000
75%     307522.500000   318748.500000   327488.500000   337016.000000
max    1667152.000000  1672473.000000  1692301.000000  1751002.000000

        gdp_growth_2009  gdp_growth_2010  gdp_growth_2011  gdp_growth_2012
count        51.000000        51.000000        51.000000        51.000000
mean         -2.313725         2.462745         1.590196         2.103922
std           3.077663         1.886474         1.610497         1.948944
min          -9.100000        -1.700000        -2.600000        -0.100000
25%          -3.900000         1.450000         0.900000         1.250000
50%          -2.400000         2.300000         1.700000         1.900000
75%          -1.050000         3.300000         2.200000         2.500000
max           7.700000         7.200000         7.800000        13.400000
```

value_counts

`value_counts` performs histogramming or the Series or DataFrame.

```
>>> state_gdp.region.value_counts()
SE    12
PL     7
```

```
NE      6
FW      6
MW      6
GL      5
RM      5
SW      4
dtype: int64
```

17.3 Time-series Data

The pandas TimeSeries object is currently limited to a span of about 585 years centered at 1970. While this is unlikely to create problems, it may not be appropriate for some applications to historical data.

pandas includes a substantial number of routines which are primarily designed to work with time-series data. A TimeSeries is basically a series where the index contains datetimes index values (more formally the class TimeSeries inherits from Series), and Series constructor will automatically promote a Series with datetime index values to a TimeSeries. The TimeSeries examples all make use of US real GDP data from the Federal Reserve Economic Database (FRED).

```
>>> GDP_data = read_excel('GDP.xls','GDP',skiprows=19)
>>> GDP_data.head()
                DATE  VALUE
0 1947-01-01 00:00:00  243.1
1 1947-04-01 00:00:00  246.3
2 1947-07-01 00:00:00  250.1
3 1947-10-01 00:00:00  260.3
4 1948-01-01 00:00:00  266.2

>>> type(GDP_data.VALUE)
pandas.core.series.Series

>>> gdp = GDP_data.VALUE
>>> gdp.index = GDP_data.DATE
>>> gdp.head()
DATE
1947-01-01    243.1
1947-04-01    246.3
1947-07-01    250.1
1947-10-01    260.3
1948-01-01    266.2
Name: VALUE, dtype: float64

>>> type(gdp)
pandas.core.series.TimeSeries
```

TimeSeries have some useful indexing tricks. For example, all of the data for a particular year can retrieved using gdp['*yyyy*'] syntax where *yyyy* is a year.

```
>>> gdp['2009']
DATE
```

```
2009-01-01    14381.2
2009-04-01    14342.1
2009-07-01    14384.4
2009-10-01    14564.1
Name: VALUE, dtype: float64


>>> gdp['2009-04'] # All for a particular month
DATE
2009-04-01    14342.1
Name: VALUE, dtype: float6
```

Dates can also be used for slicing using the notation gdp['*d1*:*d2*:'] where *d1* and *d2* are both valid date formats (e.g '2009' or '2009-01-01')

```
>>> gdp['2009':'2010']
DATE
2009-01-01    14381.2
2009-04-01    14342.1
2009-07-01    14384.4
2009-10-01    14564.1
2010-01-01    14672.5
2010-04-01    14879.2
2010-07-01    15049.8
2010-10-01    15231.7
Name: VALUE, dtype: float64


>>> gdp['2009-06-01':'2010-06-01']
DATE
2009-07-01    14384.4
2009-10-01    14564.1
2010-01-01    14672.5
2010-04-01    14879.2
Name: VALUE, dtype: float64
```

Slicing indexing can also be accomplished using datetime, for example gdp[datetime(2009,01,01):datetime(2011,12 where datetime has been imported using from pandas import datetime.

date_range

date_range is a very useful function provided by pandas to generate ranges of dates (from pandas import date_range). The basic use is either date_range(*beginning_date*, *ending_date*) which will produce a daily series between the two dates (inclusive) or date_range(*beginning_date*, periods=*periods*) which will produce a daily series starting at *beginning_date* with *periods* periods.

```
>>> from pandas import date_range
>>> date_range('2013-01-03','2013-01-05')
<class 'pandas.tseries.index.DatetimeIndex'>
[2013-01-03 00:00:00, ..., 2013-01-05 00:00:00]
Length: 3, Freq: D, Timezone: None


>>> date_range('2013-01-03', periods = 3)
```

```
<class 'pandas.tseries.index.DatetimeIndex'>
[2013-01-03 00:00:00, ..., 2013-01-05 00:00:00]
Length: 3, Freq: D, Timezone: None
```

The keyword argument `freq` changes the frequency, and common choices include

S	seconds	U	micro-second
T	minutes	L	millisecond
H	hourly	BD	daily (business)
D	daily	BM	month (end, business)
W	weekly	BMS	month (start, business)
M	monthly (end)	MS	monthly (start)
Q	quarterly (end)	QS	quarterly (start)
A	annual (end)	AS	annual (start)

Scaling the frequency produces skips that are a multiple of the default, such as in 2D which uses every other day. Combining multiple frequencies produces less regular skips, e.g. 2H10T.

```
>>> from pandas import date_range
>>> date_range('2013-01-03',periods=4, freq='Q').values
array(['2013-03-31T00:00:00.000000000+0000',
       '2013-06-30T01:00:00.000000000+0100',
       '2013-09-30T01:00:00.000000000+0100',
       '2013-12-31T00:00:00.000000000+0000'], dtype='datetime64[ns]')

>>> date_range('2013-01-03',periods=4, freq='7D4H').values
array(['2013-01-03T00:00:00.000000000+0000',
       '2013-01-10T04:00:00.000000000+0000',
       '2013-01-17T08:00:00.000000000+0000',
       '2013-01-24T12:00:00.000000000+0000'], dtype='datetime64[ns]')
```

Note that the underlying array uses NumPy's `datetime64` as the data type (with nano-second resolution, indicated by [ns]).

resample

pandas supports sophisticated resampling which is useful for aggregating form a higher frequency to a lower one using `resample`. This example uses annual ('A') and alternative aggregation functions.

```
>>> gdp.resample('A',how=mean).tail() # Annual average
DATE
2009-12-31    14417.950
2010-12-31    14958.300
2011-12-31    15533.825
2012-12-31    16244.575
2013-12-31    16601.600
Freq: A-DEC, dtype: float64

>>> gdp.resample('A',how=max).tail() # Maximum
```

```
DATE
2009-12-31    14564.1
2010-12-31    15231.7
2011-12-31    15818.7
2012-12-31    16420.3
2013-12-31    16667.9
Freq: A-DEC, dtype: float64
```

`pct_change`

Growth rates are computed using `pct_change`. The keyword argument `periods` constructs overlapping growth rates which are useful when using seasonal data.

```
>>> gdp.pct_change().tail()
DATE
2012-04-01    0.007406
2012-07-01    0.012104
2012-10-01    0.003931
2013-01-01    0.007004
2013-04-01    0.008019
Name: VALUE, dtype: float64


>>> gdp.pct_change(periods=4).tail() # Quarterly data, annual difference
DATE
2012-04-01    0.045176
2012-07-01    0.047669
2012-10-01    0.038031
2013-01-01    0.030776
2013-04-01    0.031404
Name: VALUE, dtype: float64
```

17.4   Importing and Exporting Data

In addition to providing data management tools, pandas also excels at importing and exporting data. pandas supports reading and Excel, csv and other delimited files, Stata files, fixed-width text, html, json, HDF5 and from SQL databases. The functions to read follow the common naming convention read_*type* where type is the file type, e.g. `excel` or `csv`. The writers are all methods of Series or DataFrame and follow the naming convention to_*type*.

Reading Data

`read_excel`

`read_excel` supports reading data from both xls (Excel 2003) and xlsx (Excel 2007/10/13) formats. Reading these formats depends on the Python package xlrd. The basic usage required two inputs, the file name and the sheet name. Other notable keyword arguments include:

- `header`, an integer indicating which row to use for the column labels. The default is 0 (top) row, and if `skiprows` is used, this value is relative.

- `skiprows`, typically an integer indicating the number of rows at the top of the sheet to skip before reading the file. The default is 0.

- `skip_footer`, typically an integer indicating the number of rows at the bottom of the sheet to skip when reading the file. The default is 0.

- `index_col`, an integer indicating the column to use as the index. If not provided, a basic numeric index is generated.

- `parse_cols`, `None`, an integer, a list of integers or a string, tells pandas whether to attempt to parse a column. The default is `None` which will parse all columns. Alternatively, if an integer is provided then the value is interpreted as the last column to parse. Finally, if a list of integers is provided, the values are interpreted as the columns to parse (0-based, e.g. `[0,2,5]`). The string version takes one of the forms `'A'`, `'A,C,D'`, `'A:D'` or a mix of the latter two (`'A,C:D,G,W:Z'`).

read_csv

read_csv reads comma separated value files. The basic use only requires one input, a file name. `read_csv` also accepts valid URLs (http, ftp, or s3 (Amazon) if the boto package is available) or any object that provides a `read` method in places of the file name. A huge range of options are available, and so only the most relevant are presented in the list below.

- `delimiter`, the delimiter used to separate values. The default is `','`. Complicated delimiters are matched using a regular expression.

- `delim_whitespace`, Boolean indicating that the delimiter is white space (a space or tab). This is preferred to using a regular expression to detect white space.

- `header`, an integer indicating the row number to use for the column names. The default is 0.

- `skiprows`, similar to `skiprows` in `read_excel`.

- `skip_footer`, similar to `skip_footer` in `read_excel`.

- `index_col`, similar to `index_col` in `read_excel`.

- `names`, a list of column names to use in-place of any found in the file Must use `header=0` (the default value).

- `parse_dates`, a Boolean indicating whether to parse dates encountered. Supports more complicated options to combine columns (see `read_csv`).

- `date_parser`, a function to use when parsing dates. The default parser is `dateutil.parser`.

- `dayfirst`, a Boolean indicating whether to use European date format (DD/MM, `True`) or American date format (MM/DD `False`) when encountering dates. The default is `False`.

- `error_bad_lines`, when `True` stops processing on a bad line. If `False`, continues skipping any bad lines encountered.

- `encoding`, a string containing the file encoding (e.g. `'utf-8'` or `'latin-1'`).

- converters, a dictionary of functions for converting values in certain columns, where keys can either integers (column-number) or column labels.

- nrows, an integer, indicates the maximum number of rows to read. This is useful for reading a subset of a file.

- usecols, a list of integers or column names indicating which column to retain.

- dtype A data type to use for the read data or a dictionary of data types using the column names as keys. If not provided, the type is inferred.

read_table

read_table is similar to read_csv and both are wrappers around the private read function provided by pandas.

read_hdf

read_hdf is primarily for reading pandas DataTables which were written using DataTable.to_hdf

Writing Data

Writing data from a Series or DataFrame is much simpler since the starting point (the Series or the DataFrame) is well understood by pandas. While the file writing methods all have a number of options, most can safely be ignored.

```
>>> state_gdp.to_excel('state_gdp_from_dataframe.xls')
>>> state_gdp.to_excel('state_gdp_from_dataframe_sheetname.xls', sheet_name='State GDP')
>>> state_gdp.to_excel('state_gdp_from_dataframe.xlsx')
>>> state_gdp.to_csv('state_gdp_from_dataframe.csv')
>>> import StringIO
>>> sio = StringIO.StringIO()
>>> state_gdp.to_json(sio)
>>> sio.seek(0)
>>> sio.buf[:50]
'{"state_code":{"0":"AK","1":"AL","2":"AR","3":"AZ"'

>>> state_gdp.to_string()[:50]
u'   state_code                  state  gdp_2009  gdp'
```

One writer, to_hdf is worth special mention. to_hdf writes pandas DataFrames to HDF5 files which are binary files which support compression. HDF5 files can achieve fantastic compression ratios when data are regular, and so are often much more useful than csv or xlsx (which is also compressed). The usage of to_hdf is not meaningfully different from the other writers except that:

- In addition to the filename, an argument is required containing the key, which is usually the variable name.

- Two additional arguments must be passed for the output file to be compressed. These two keyword arguments are complib and complevel, which I recommend to setting to 'zlib' and 6, respectively.

```
>>> df = DataFrame(zeros((1000,1000)))
>>> df.to_csv('size_test.csv')
>>> df.to_hdf('size_test.h5','df') # h5 is the usual extension for HDF5
# h5 is the usual extension for HDF5
>>> df.to_hdf('size_test_compressed.h5','df',complib='zlib',complevel=6)
>>> ls size_* # Ignore
09/19/2013  04:16 PM           4,008,782 size_test.csv
09/19/2013  04:16 PM           8,029,368 size_test.h5
09/19/2013  04:16 PM              33,812 size_test_compressed.h5

>>> import gzip
>>> f = gzip.open('size_test.csvz','w')
>>> df.to_csv(f)
>>> f.close()
>>> ls size_test.csvz # Ignore
09/19/2013  04:18 PM              10,533 size_test.csvz
>>> from pandas import read_csv
>>> df_from_csvz = read_csv('size_test.csvz',compression='gzip')
```

The final block of lines shows how a csv with gzip compression is written and directly read using pandas. This method also achieves a very high level of compression.

Any NumPy array is easily written to a file using a single, simple line using pandas.

```
>>> x = randn(100,100)
>>> DataFrame(x).to_csv('numpy_array.csv',header=False,index=False)
```

17.5 Graphics

pandas provides a set of useful plotting routines based on matplotlib which makes use of the structure of a DataFrame. Everything in pandas plot library is reproducible using matplotlib, although often at the cost of additional typing and code complexity (for example, axis labeling).

plot

plot is the main plotting method, and by default will produce a line graph of the data in a DataFrame. Calling plot on a DataFrame will plot all series using different colors and generate a legend. A number of keyword argument are available to affect the contents and appearance of the plot.

- style, a list of matplotlib styles, one for each series plotted. A dictionary using column names as keys and the line styles as values allows for further customization.

- title, a string containing the figure title.

- subplots, a Boolean indicating whether to plot using one subplot per series (True). The default it False.

- legend, a Boolean indicating whether to show a legend

- secondary_y, a Boolean indicating whether to plot a series on a secondary set of axis values. See the example below.

- ax, a matplotlib axis object to use for the plot. If no axis is provided, then a new axis is created.

- kind, a string, one of:

  - 'line', the default
  - 'bar' to produce a bar chart. Can also use the keyword argument stacked=True to produce a stacked bar chart.
  - 'barh' to produce a horizontal bar chart. Also support stacked=True.
  - 'kde' or 'density' to produce a kernel density plot.

hist

hist produces a histogram plot, and is similar to producing a bar plot using the output of value_count.

boxplot

boxplot produces box plots of the series in a DataFrame.

scatter_plot

scatter_plot produce a scatter plot from two series in a DataFrame. Three inputs are required: the DataFrame, the column name for the x-axis data and the column name for the y-axis data. scatter_plot is located in pandas.tools.plotting.

scatter_matrix

scatter_matrix produces a *n* by *n* set of subplots where each subplot contains the bivariate scatter of two series. One input is required, the DataFrame. scatter_matrix is located in pandas.tools.plotting. By default, the diagonal elements are histograms, and the keyword argument diagonal='kde' produces a kernel density plot.

lag_plot

lag_plot produces a scatter plot of a series against its lagged value. The keyword argument lag chooses the lag used in the plot (default is 1).

17.6 Examples

17.6.1 FRED Data

The Federal Reserve Economics Database is a comprehensive database of US, and increasingly global, macroeconomics data. This example will directly download a small macroeconomics data set from FRED and merge it into a single DataFrame. The data in FRED is available in csv using the url pattern http://research.stlouisfed.org/fred2/data/CODE.csv where CODE is the series code. This example will make use of Real GDP, Industrial Production, Core CPI the Unemployment Rate, the Treasury yield slope (10 year yield minus 1 year yield) and the default premium, based on the difference between BAA and AAA rated bonds. The list of series is in table 17.1.

Series	Code	Frequency
Real GDP	GDPC1	Quarterly
Industrial Production	INDPRO	Quarterly
Core CPI	CPILFESL	Monthly
Unemployment Rate	UNRATE	Monthly
10 Year Yield	GS10	Monthly
1 Year Yield	GS1	Monthly
Baa Yield	BAA	Monthly
Aaa Yield	AAA	Monthly

Table 17.1: The series, codes and their frequencies used in the FRED example.

The initial block of code imports the future functions, `read_csv`, `DataFrame` and `scatter_matrix`, the only pandas functions directly used in this example. It also sets up lists containing the codes and nice names for the series. Finally, the url root to use to fetch the data is included.

```python
from __future__ import print_function, division
from pandas import read_csv
from pandas.tools.plotting import scatter_matrix

codes = ['GDPC1','INDPRO','CPILFESL','UNRATE','GS10','GS1','BAA','AAA']
names = ['Real GDP','Industrial Production','Core CPI','Unemployment Rate',\
    '10 Year Yield','1 Year Yield','Baa Yield','Aaa Yield']

# r to disable escape
base_url = r'http://research.stlouisfed.org/fred2/data/'
```

The next piece of code starts with an empty list to hold the DataFrames produced by `read_csv`. The codes are then looped over and directly used in the csv reader.

```python
data = []
for code in codes:
    print(code)
    url = base_url + code + '.csv'
    data.append(read_csv(url))
```

Next, the data is merged into a single DataFrame by building a dictionary where the keys are the codes and the values are the Series from each downloaded DataFrame. This block makes use of `zip` to quickly concatenate two lists into a single iterable.

```python
time_series = {}
for code, d in zip(codes,data):
    d.index = d.DATE
    time_series[code] = d.VALUE

merged_data = DataFrame(time_series)
# Unequal length series
print(merged_data)
```

The next step is to construct the Term and Default premia series using basic math on the series. The resulting Series are given a name, which is requires to the join operation. Finally, the non-required columns are dropped.

```
term_premium = merged_data['GS10'] - merged_data['GS1']
term_premium.name = 'Term'
merged_data = merged_data.join(term_premium,how='outer')
default_premium = merged_data['BAA'] - merged_data['AAA']
default_premium.name = 'Default'
merged_data = merged_data.join(default_premium,how='outer')
merged_data = merged_data.drop(['AAA','BAA','GS10','GS1'],axis=1)
print(merged_data.tail())
```

The next block forms a quarterly data set by dropping the rows with any null values.

```
quarterly = merged_data.dropna()
print(quarterly.tail())
```

Finally, it is necessary to transform some of the series to be growth rates since the data contains both $I(0)$ and $I(1)$ series. This is done using `pct_change` on a subset of the quarterly data.

```
growth_rates_selector = ['GDPC1','INDPRO','CPILFESL']
growth_rates = quarterly[growth_rates_selector].pct_change()
final = quarterly.drop(growth_rates_selector, axis=1).join(growth_rates)
```

The last step is to rename some of the columns using `rename` with the keyword argument `columns`. The names are changed using a dictionary where the key is the old name and the value is the new name. The last two lines save the final version of the data to HDF5 and to an excel file.

```
new_names = {'GDPC1':'GDP_growth','INDPRO':'IP_growth','CPILFESL':'Inflation','UNRATE':'Unemp_rate'}
final = final.rename(columns = new_names ).dropna()
final.to_hdf('FRED_data.h5','FRED',complevel=6,complib='zlib')
final.to_excel('FRED_data.xlsx')
```

The plots provide a simple method to begin exploring the data. Both plots are shown in Figure 17.1.

```
ax = final[['GDP_growth','IP_growth','Unemp_rate']].plot(subplots=True)
fig = ax[0].get_figure()
fig.savefig('FRED_data_line_plot.pdf')

ax = scatter_matrix(final[['GDP_growth','IP_growth','Unemp_rate']], diagonal='kde')
fig = ax[0,0].get_figure()
fig.savefig('FRED_data_scatter_matrix.pdf')
```

17.6.2 NSW Data

The National Supported Work Demonstration was a program to determine whether giving disadvantaged workers useful job skills would translate into increased earnings. The data set used here is a subset of the complete data set and contains the variables in table 17.2.

The first block contains the standard imports as well as the functions which are used in this example. Both `sqrt` and `stats` are used to perform a t-test.

```
from __future__ import print_function, division
from pandas import read_excel
from numpy import sqrt
import scipy.stats as stats
```
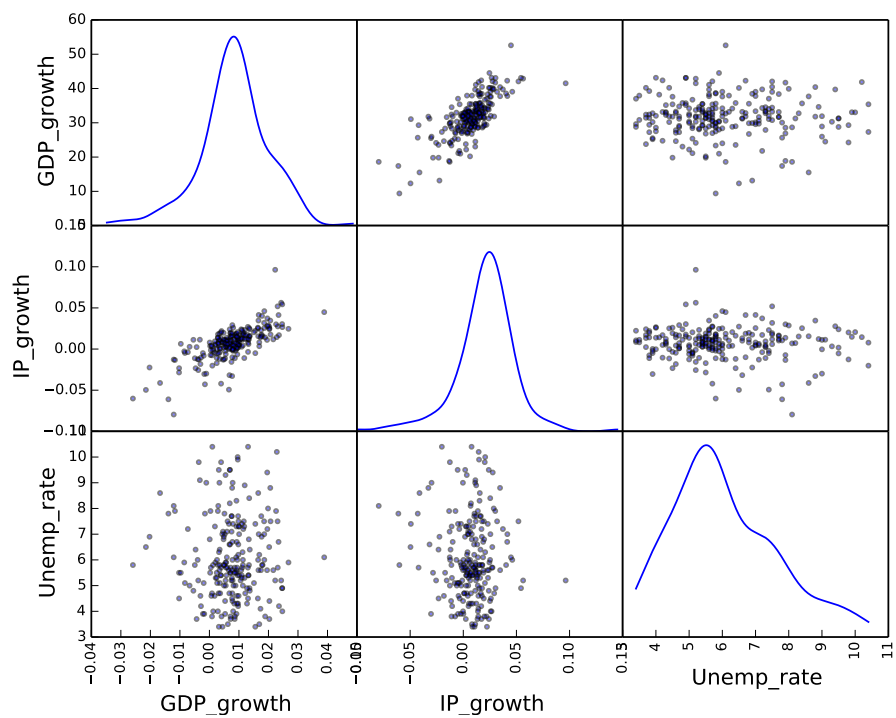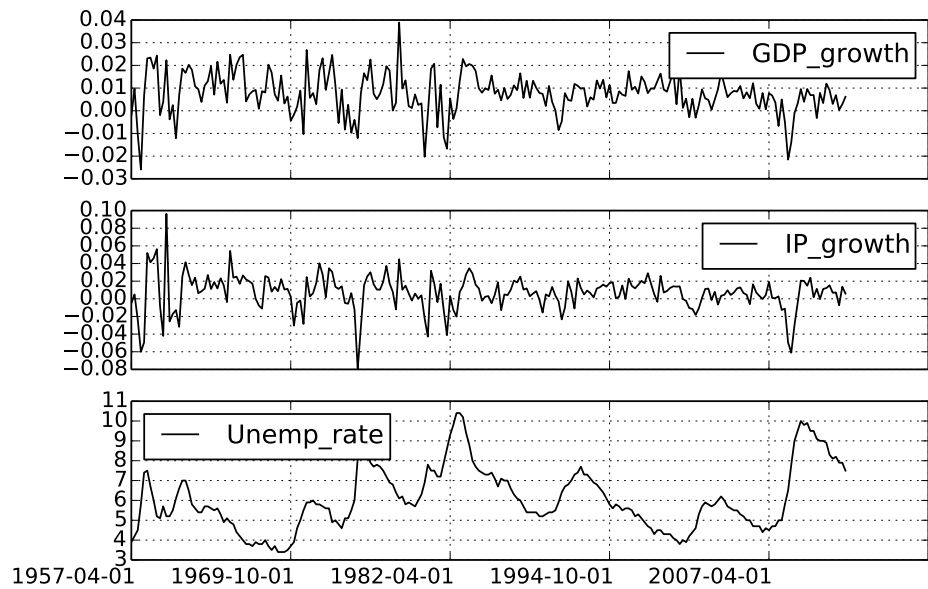
Figure 17.1: The top panel contains line plots of the FRED data. The bottom panel shows the output of `scatter_matrix` using kernel density plots along the diagonal.

Series	Description
Treated	Dummy indicating whether the candidate received the treatment
Age	Age in years
Education (years)	Years of Education
Black	Dummy indicating African-American
Hispanic	Dummy indicating Hispanic
Married	Dummy indicating married
Real income Before ($)	Income before program
Real income After ($)	Income after program

Table 17.2: The series, codes and their frequencies used in the FRED example.

The data is contained in a well-formatted Excel file, and so importing the data using `read_excel` is straightforward. The second line in this block prints the standard descriptive statistics.

```
NSW = read_excel('NSW.xls','NSW')
print(NSW.describe())
```

`rename` is then used to give the columns some more useful names – names with spaces cannot be directly accessed using dot notation (i.e. `NSW.Income_after` works, but there is not method to do the same using `NSW.'Real income After ($)'`.

```
NSW = NSW.rename(columns={'Real income After ($)':'Income_after',
                          'Real income Before ($)':'Income_before',
                          'Education (years)':'Education'})
NSW['Minority'] = NSW['Black']+NSW['Hispanic']
```

Next, `pivot_table` is used to look at the variable means using some of the groups. The third use does a double sort.

```
print(NSW.pivot_table(rows='Treated'))


print(NSW.pivot_table(rows='Minority'))


print(NSW.pivot_table(rows=['Minority','Married']))
```

Next, density plots of the income before and after are plotted. Figure 17.2 shows the plots.

```
ax = NSW[['Income_before','Income_after']].plot(kind='kde',subplots=True)
fig = ax[0].get_figure()
fig.savefig('NSW_density.pdf')
```

Finally a t-test of equal incomes using the before and after earnings for the treated and non-treated is computed. The t-stat has a one-sided p-val of .9%, indicating rejection of the null of no impact at most significance levels.

```
income_diff = NSW['Income_after']-NSW['Income_before']
t = income_diff[NSW['Treated']==1]
nt = income_diff[NSW['Treated']==0]
tstat = (t.mean() - nt.mean())/sqrt(t.var()/t.count() - nt.var()/nt.count())
pval = 1 - stats.norm.cdf(tstat)
print('T-stat: {0:.2f}, P-val: {1:.3f}'.format(tstat,pval))
```
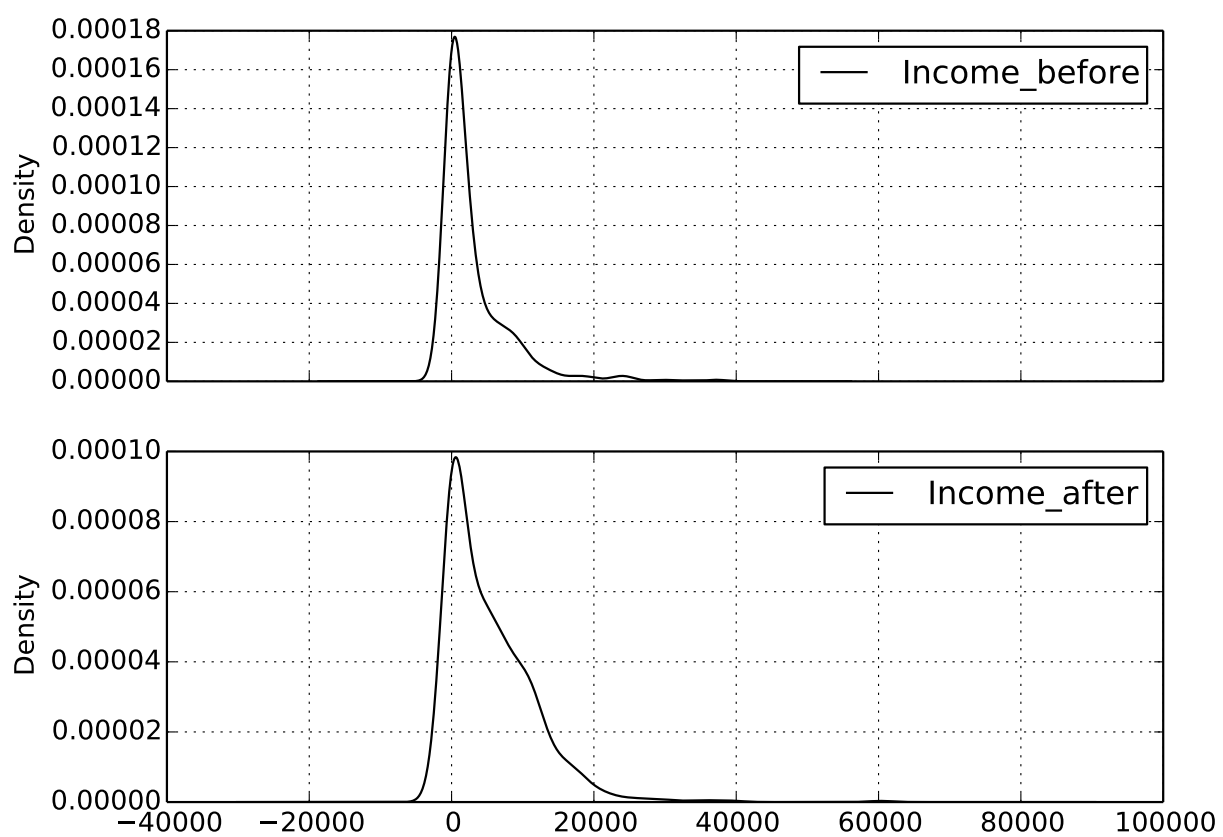
Figure 17.2: Density plot of the before and after income.

Chapter 18

Custom Function and Modules

Python supports a wide range of programming styles including procedural (imperative), object oriented and functional. While object oriented programming and functional programming are powerful programming paradigms, especially in large, complex software, procedural is often both easier to understand and a direct representation of a mathematical formula. The basic idea of procedural programming is to produce a function or set of function (generically) of the form

$$y = f(x).$$

That is, the functions take one or more inputs and produce one or more outputs.

18.1 Functions

Python functions are very simple to declare and can occur in the same file as the main program or a standalone file. Functions are declared using the `def` keyword, and the value produced is returned using the `return` keyword. Consider a simple function which returns the square of the input, $y = x^2$.

```python
from __future__ import print_function, division

def square(x):
    return x**2

# Call the function
x = 2
y = square(x)
print(x,y)
```

In this example, the same Python file contains the main program – the final 3 lines – as well as the function. More complex function can be crafted with multiple inputs.

```python
from __future__ import print_function, division

def l2distance(x,y):
    return (x-y)**2

# Call the function
```

```
x = 3
y = 10
z = l2distance(x,y)
print(x,y,z)
```

Function can also be defined using NumPy arrays and matrices.

```
from __future__ import print_function, division

import numpy as np

def l2_norm(x,y):
    d = x - y
    return np.sqrt(np.dot(d,d))

# Call the function
x = np.random.randn(10)
y = np.random.randn(10)
z = l2_norm(x,y)
print(x-y)
print("The L2 distance is ",z)
```

When multiple outputs are returned but only a single variable is available for assignment, all outputs are returned in a tuple. Alternatively, the outputs can be directly assigned when the function is called with the same number of variables as outputs.

```
from __future__ import print_function, division

import numpy as np

def l1_l2_norm(x,y):
    d = x - y
    return sum(np.abs(d)),np.sqrt(np.dot(d,d))

# Call the function
x = np.random.randn(10)
y = np.random.randn(10)
# Using 1 output returns a tuple
z = l1_l2_norm(x,y)
print(x-y)
print("The L1 distance is ",z[0])
print("The L2 distance is ",z[1])

# Using 2 output returns the values
l1,l2 = l1_l2_norm(x,y)
print("The L1 distance is ",l1)
print("The L2 distance is ",l2)
```

All of these functions have been placed in the same file as the main program. Placing functions in modules allows for reuse in multiple programs, and will be discussed later in this chapter.

18.1.1 Keyword Arguments

All input variables in functions are automatically keyword arguments, so that the function can be accessed either by placing the inputs in the order they appear in the function (positional arguments), or by calling the input by their name using *keyword=value*.

```python
from __future__ import print_function, division
import numpy as np

def lp_norm(x,y,p):
    d = x - y
    return sum(abs(d)**p)**(1/p)

# Call the function
x = np.random.randn(10)
y = np.random.randn(10)
z1 = lp_norm(x,y,2)
z2 = lp_norm(p=2,x=x,y=y)
print("The Lp distances are ",z1,z2)
```

Because variable names are automatically keywords, it is important to use meaningful variable names when possible, rather than generic variables such as a, b, c or x, y and z. In some cases, x may be a reasonable default, but in the previous example which computed the L_p norm, calling the third input z would be bad idea.

18.1.2 Default Values

Default values are set in the function declaration using the syntax *input=default*.

```python
from __future__ import print_function, division
import numpy as np

def lp_norm(x,y,p = 2):
    d = x - y
    return sum(abs(d)**p)**(1/p)

# Call the function
x = np.random.randn(10)
y = np.random.randn(10)
# Inputs with default values can be ignored
l2 = lp_norm(x,y)
l1 = lp_norm(x,y,1)
print("The l1 and l2 distances are ",l1,l2)
print("Is the default value overridden?", sum(abs(x-y))==l1)
```

Default values should not normally be mutable (e.g. lists or arrays) since they are only initialized the first time the function is called. Subsequent calls will use the same value, which means that the default value could change every time the function is called.

```python
from __future__ import print_function, division
import numpy as np
```

```python
def bad_function(x = zeros(1)):
    print(x)
    x[0] = np.random.randn(1)

# Call the function
bad_function()
bad_function()
bad_function()
```

Each call to `bad_function` shows that x has a different value – despite the default being 0. The solution to this problem is to initialize mutable objects to None, and then the use an `if` to check and initialize only if the value is None. Note that tests for None use the `is` keyword rather the testing for equality using ==.

```python
from __future__ import print_function, division
import numpy as np

def good_function(x = None):
    if x is None:
        x = zeros(1)
    print(x)
    x[0] = np.random.randn(1)


# Call the function
good_function()
good_function()
```

Repeated calls to `good_function()` all show x as 0.

18.1.3 Variable Number of Inputs

Most function written as an "end user" have an known (*ex ante*) number of inputs. However, functions which evaluate other functions often must accept variable numbers of input. Variable inputs can be handled using the *args (arguments) or **kwargs (keyword arguments) syntax. The *args syntax will generate tuple a containing all inputs past the required input list. For example, consider extending the L_p function so that it can accept a set of p values as extra inputs (Note: in practice it would make more sense to accept an array for p).

```python
from __future__ import print_function, division

import numpy as np

def lp_norm(x,y,p = 2, *args):
    d = x - y
    print('The L' + str(p) + ' distance is :', sum(abs(d)**p)**(1/p))
    out = [sum(abs(d)**p)**(1/p)]

    print('Number of *args:', len(args))
    for p in args:
        print('The L' + str(p) + ' distance is :', sum(abs(d)**p)**(1/p))
        out.append(sum(abs(d)**p)**(1/p))
```

```
    return tuple(out)


# Call the function
x = np.random.randn(10)
y = np.random.randn(10)
# x & y are required inputs and so are not in *args
lp = lp_norm(x,y)
# Function takes 3 inputs, so no *args
lp = lp_norm(x,y,1)
# Inputs with default values can be ignored
lp = lp_norm(x,y,1,2,3,4,1.5,2.5,0.5)
```

The alternative syntax, **kwargs, generates a dictionary with all keyword inputs which are not in the function signature. One reason for using **kwargs is to allow a long list of optional inputs without having to have an excessively long function definition. This is how this input mechanism operates in many matplotlib functions such as plot.

```
from __future__ import print_function, division
import numpy as np

def lp_norm(x,y,p = 2, **kwargs):
    d = x - y
    print('Number of *kwargs:', len(kwargs))
    for key in kwargs:
        print('Key :', key, ' Value:', kwargs[key])

    return sum(abs(d)**p)

# Call the function
x = np.random.randn(10)
y = np.random.randn(10)
# Inputs with default values can be ignored
lp = lp_norm(x,y,kword1=1,kword2=3.2)
# The p keyword is in the function def, so not in **kwargs
lp = lp_norm(x,y,kword1=1,kword2=3.2,p=0)
```

It is possible to use both *args and **kwargs in a function definition and their role does not change – *args appears in the function as a tuple that contains all extraneous non-keyword inputs, and **kwargs appears inside the function as a dictionary that contains all keyword arguments not appearing in the function definition. Functions with both often have the simple signature y = f(*args, **kwargs) which allows for any set of inputs.

18.1.4   The Docstring

The docstring is one of the most important elements of any function – especially a function written for use by others. The docstring is a special string, enclosed with triple-quotation marks, either ''' or """, which is available using help(). When help(fun) is called (or fun?/?fun in IPython), Python looks for the docstring which is placed immediately below the function definition.

```python
from __future__ import print_function, division
import numpy as np

def lp_norm(x,y,p = 2):
    """ The docstring contains any available help for
        the function.  A good docstring should explain the
        inputs and the outputs, provide an example and a list
        of any other related function.
    """
    d = x - y
    return sum(abs(d)**p)
```

Calling `help(lp_norm)` produces

```
>>> help(lp_norm)
Help on function lp_norm in module __main__:

lp_norm(x, y, p=2)
    The docstring contains any available help for
    the function.  A good docstring should explain the
    inputs and the outputs, provide an example and a list
    of any other related function.
```

This docstring is not a good example. I suggest following the NumPy guidelines, currently available at the
NumPy source repository (or search for *numpy docstring*). Also see NumPy example.py These differ from
and are more specialized than the standard Python docstring guidelines, and are more appropriate for
numerical code. A better docstring for `lp_norm` would be

```python
from __future__ import print_function, division
import numpy as np

def lp_norm(x,y,p = 2):
    r""" Compute the distance between vectors.

    The Lp normed distance is sum(abs(x-y)**p)**(1/p)

    Parameters
    ----------
    x : ndarray
        First argument
    y : ndarray
        Second argument
    p : float, optional
        Power used in distance calculation, >=0

    Returns
    -------
    output : scalar
        Returns the Lp normed distance between x and y

    Notes
```

```
    -----

    For p>=1, returns the Lp norm described above.  For 0<=p<1,
    returns sum(abs(x-y)**p).  If p<0, p is set to 0.

    Examples
    --------
    >>> x=[0,1,2]
    >>> y=[1,2,3]

    L2 norm is the default

    >>> lp_norm(x,y)

    Lp can be computed using the optional third input

    >>> lp_norm(x,y,1)

    """

    if p<0: p=0
    d = x - y

    if p == 0:
        return sum(d != 0)
    elif p < 1:
        return sum(abs(d)**p)
    else:
        return sum(abs(d)**p)**(1/p)
```

Convention is to use triple double-quotes in docstrings, with r""" used to indicate "raw" strings, which will ignore backslashes rather than treating them like an escape character (use u""" if the docstring contains Unicode text, which is not usually necessary). A complete docstring may contain, in order:

- Parameters - a description of key inputs

- Returns - a description of outputs

- Other Parameters - a description of seldom used inputs

- Raises - an explanation of any exceptions raised. See Section 13.5.

- See also - a list of related functions

- Notes - details of the algorithms or assumptions used

- References - any bibliographic information

- Examples - demonstrates use form console

207

18.2 Variable Scope

Variable scope determines which functions can access, and possibly modify a variable. Python determines variable scope using two principles: where the variable appears in the file, and whether the variable is inside a function or in the main program. Variables declared inside a function are local variables and are only available to that function. Variables declared outside a function are global variables, and can be accessed but not modified (unless using `global`). Consider the example which shows that variables at the root of the program that have been declared before a function definition can be printed by that function.

```python
from __future__ import print_function, division
import numpy as np

a, b, c = 1, 3.1415, 'Python'

def scope():
    print(a)
    print(b)
    print(c)
    # print(d) #Error, d has not be declared yet

scope()
d = np.array(1)

def scope2():
    print(a)
    print(b)
    print(c)
    print(d) # Ok now

scope2()

def scope3():
    a = 'Not a number' # Local variable
    print('Inside scope3, a is ', a)

print('a is ',a)
scope3()
print('a is now ',a)
```

Using the name of a global variable inside a function does produce any changes outside of the function. In scope3, a is given a different value. That value is specific to the function scope3 and outside of the function, a will have its global value. Generally, global variables can be accessed, but not modified inside a function. The only exception is when a variable is declared inside the function using the keyword `global`.

```python
from __future__ import print_function, division
import numpy as np

a = 1

def scope_local():
```

```
    a = -1
    print('Inside scope_local, a is ',a)

def scope_global():
    global a
    a = -10
    print('Inside scope_global, a is ',a)

print('a is ',a)
scope_local()
print('a is now ',a)
scope_global()
print('a is now ',a)
```

One word of caution: a variable name cannot be used as a local and global variable in the same function. Attempting to access the variable as a global (e.g. for printing) and then locally assign the variable produces an error.

18.3   Example: Least Squares with Newey-West Covariance

Estimating cross-section regressions using time-series data is common practice. When regressors are persistent, and errors are not white noise, standard inference, including White standard errors, is no longer consistent. The most common solution is to use a long-run covariance estimator, and the most common long-run covariance estimator is known as the Newey-West covariance estimator which uses a Bartlett kernel applied to the autocovariances of the scores. This example produces a function which returns parameter estimates, the estimated asymptotic covariance matrix of the parameters, the variance of the regression error, the R^2, and adjusted R^2 and the fit values (or errors, since actual is equal to fit plus errors). These are computed using a T-vector for the regressand (dependent variable), a T by k matrix for the regressors, an indicator for whether to include a constant in the model (default True), and the number of lags to include in the long-run covariance (default behavior is to automatically determine based on sample size). The steps required to produce the function are:

1. Determine the size of the variables

2. Append a constant, if needed

3. Compute the regression coefficients

4. Compute the errors

5. Compute the covariance or the errors

6. Compute the covariance of the parameters

7. Compute the R^2 and $\bar{R}^2$

The function definition is simple and allows for up to 4 inputs, where 2 have default values: `def olsnw(y, X, constant=True, lags=None):`. The size of the variables is then determined using `size` and the constant is prepended to the regressors, if needed, using `hstack`. The regression coefficients are computed using

209

lstsq, and then the Newey-West covariance is computed for both the errors and scores. The covariance of the parameters is then computed using the NW covariance of the scores. Finally the R^2 and $\bar{R}^2$ are computed. A complete code listing is presented in the appendix to this chapter.

18.4   Anonymous Functions

Python support anonymous functions using the keyword `lambda`. Anonymous functions are usually encountered when another function expects a function as an input *and* a simple function will suffice. Anonymous function take the generic form `lambda` $a, b, c, \dots : code\ using\ a,b,c$. The key elements are the keyword `lambda`, a list of comma separated inputs, a colon between the inputs and the actual function code. For example `lambda x,y:x+y` would return the sum of the variables x and y.

Anonymous functions are simple but useful. For example, when lists containing other lists it isn't directly possible to sort on an arbitrary element of the nested list. Anonymous functions allow sorting through the keyword argument `key` by returning the element Python should use to sort. In this example, a direct call to `sort()` will sort on the first element (first name). Using the anonymous function `lambda x:x[1]` to return the second element of the tuple allows for sorting on the last name. `lambda x:x[2]` would allow for sorting on the University.

```
>>> nested = [('John','Doe','Oxford'),\
...           ('Jane','Dearing','Cambridge'),\
...           ('Jerry','Dawn','Harvard')]

>>> nested.sort()
>>> nested
[('Jane', 'Dearing', 'Cambridge'),
 ('Jerry', 'Dawn', 'Harvard'),
 ('John', 'Doe', 'Oxford')]

>>> nested.sort(key=lambda x:x[1])
>>> nested
[('Jerry', 'Dawn', 'Harvard'),
 ('Jane', 'Dearing', 'Cambridge'),
 ('John', 'Doe', 'Oxford')]
```

18.5   Modules

The previous examples all included the function in inside the same Python file that contained the main program. While this is convenient, especially when coding the function, it hinders use in other code. Modules allow multiple functions to be combined in a single Python file and accessed using `import` *module* and then *module.function* syntax. Suppose a file named core.py contains the following code:

```
r"""Demonstration module.
This is the module docstring.
"""


def square(x):
    r"""Returns the square of a scalar input
```

```
    """
    return x*x

def cube(x):
    r"""Returns the cube of a scalar input
    """
    return x*x*x
```

The functions square and cube can be accessed by other files in the same directory using

```
from __future__ import print_function, division
import core

y = -3
print(core.square(y))
print(core.cube(y))
```

The functions in core.py can be imported using any of the standard import methods: import core as c, from core import square or from core import * in which case both functions could be directly accessed.

18.5.1 __main__

Normally modules should only have code required for the module to run, and other code should reside in a different function. However, it is possible that a module could be both directly importable and also directly runnable. If this is the case, it is important that the directly runnable code should not be executed when the module is imported by other code. This can be accomplished using a special construct, if __name__=="__main__": before any code that should execute when the module is run as a standalone program. Consider the following simple example in a module namedtest.py.

```
from __future__ import print_function, division

def square(x):
    return x**2

if __name__=="__main__":
    print('Program called directly.')
else:
    print('Program called indirectly using name: ', __name__)
```

Running and importing test cause the different paths to be executed.

```
>>> %run test.py
Program called directly.

>>> import test
Program called indirectly using name:  test
```

18.6 Packages

As a modules grows, organizing large amounts of code a single file – especially code that serve very different purposes – becomes difficult. Packages solve this problem by allowing multiple files to exist in the

same namespace, as well as sub-namespaces. Python packages are constructed using directory structures using a special file name: __init__.py. A Python package begins with a file folder using the name of the package. For example, consider developing a package called metrics which will contain common econometrics routines. The minimal package structure would have

```
metrics/
    __init__.py
```

The __init__.py file instructs Python to treat this directory as part of a package. __init__.py is a standard Python file, although it is not necessary to include any code in this file. However, code included in __init__.py will appear in the root of the package namespace. Suppose __init__.py contained a function with the name reg. Assuming `import` core was used to import the module, this function would be accessible as `core.reg`. Next, suppose other Python files are included in the directory under core, so that the directory structure looks like

```
core/
    __init__.py
    crosssection.py
    timeseries.py
```

This would allow functions to be directly included the core namespace by including the function in __init__.py. Functions that resided in crosssection.py would be accessible using `import core.crosssection as` cs and then `cs.reg`.

Finally, suppose that crosssection.py was replaced with a directory where the directory contained other Python files, including __init__.py.

```
core/
    __init__.py
    crosssection/
        __init__.py
        regression.py
        limdep.py
    timeseries/
        __init__.py
        arma.py
        var.py
```

This structure allows functions to be accessible directly from core using the __init__.py file, accessible from `core.crosssection` using the __init__.py located in the directory crosssection or accessible using `core.crosssection.reg` for functions inside the file regression.py.

__init__.py is useful in Python packages beyond simply instructing Python that a directory is part of a package. It can be used to initialize any common information required by functions in the module or to "fake" the location of a deeply nested functions. __init__.py is executed whenever a package is imported, and since it can contain standard Python code, it is possible define variables in the package namespace or execute code which will be commonly used by functions in the module (e.g. reading a config file). Suppose that the __init__.py located in the directory core contains

```
from core.crosssection.regression import *
```

This single import will make all functions in the file regression.py available directly after running `import` core. For example, suppose regression.py contains the function `leastsquares`. Without the import statement

in __init__.py, `leastsquares` would only be available through `core.crosssection.regression`. However, after including the `import` statement in __init__.py, `leastsquares` is directly accessible from `core`. Using __init__.py allows for a flexible file and directory structure that reflects the code's function while avoiding complex import statements.

18.7   PYTHONPATH

While it is simple to reference files in the same current working directory, this behavior is undesirable for code shared between multiple projects. The PYTHONPATH allows directories to be added so that they are automatically searched if a matching module cannot be found in the current directory. The current path can be checked by running

```
>>> import sys
>>> sys.path
```

Additional directories can be added at runtime using

```
import sys

# New directory is first to be searched
sys.path.insert(0, 'c:\\path\\to\add')
# New directory is last to be searched
sys.path.append('c:\\path\\to\add')
```

Directories can also be added permanently by adding or modifying the environment variable PYTHON-PATH. On Windows, the System environment variables can be found in My Computer > Properties > Advanced System Settings > Environment Variables. PYTHONPATH should be a System Variable. If it is present, it can be edited, and if not, added. The format of PYTHONPATH is

```
c:\dir1;c:\dir2;c:\dir2\dir3;
```

which will add 3 directories to the path. On Linux, PYTHONPATH is stored in either ~/.bash_rc or ~/.bash_profile, and it should resemble

```
PYTHONPATH="${PYTHONPATH}:/dir1/:/dir2/:/dir2/dir3/"
export PYTHONPATH
```

after three directories have been added, using : as a separator between directories. On OSX the PYTHON-PATH is stored in ~/.profile.

18.8   Python Coding Conventions

There are a number of common practices which can be adopted to produce Python code which looks more like code found in other modules:

1. Use 4 spaces to indent blocks – avoid using tab, except when an editor automatically converts tabs to 4 spaces

2. Avoid more than 4 levels of nesting, if possible

3. Limit lines to 79 characters. The \ symbol can be used to break long lines

4. Use two blank lines to separate functions, and one to separate logical sections in a function.

5. Use ASCII mode in text editors, not UTF-8

6. One module per import line

7. Avoid `from` *module* `import` `*` (for any module). Use either `from` *module* `import` *func1, func2* or `import` *module* `as` *shortname*.

8. Follow the NumPy guidelines for documenting functions

More suggestions can be found in PEP8.

18.9 Exercises

1. Write a function which takes an array with T elements contains categorical data (e.g. 1,2,3), and returns a T by C array of indicator variables where C is the number of unique values of the categorical variable, and each column of the output is an indicator variable (0 or 1) for whether the input data belonged to that category. For example, if $x = [1\ 2\ 1\ 1\ 2]$, then the output is

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

The function should provide a second output containing the categories (e.g. [1 2] in the example).

2. Write a function which takes a T by K array X, a T by 1 array y, and a T by T array Ω are computes the GLS parameter estimates. The function definition should be
   `def gls(X, y, Omega = None)`
   and if Ω is not provided, an identity matrix should be used.

3. Write a function which will compute the partial correlation. Lower partial correlation is defined as

$$\frac{\sum_{\mathcal{S}} \left(r_{i,1} - \bar{r}_{1,\mathcal{S}} \right) \left(r_{i,2} - \bar{r}_{2,\mathcal{S}} \right)}{\sum_{\mathcal{S}} \left(r_{j,1} - \bar{r}_{1,\mathcal{S}} \right)^2 \sum_{\mathcal{S}} \left(r_{k,2} - \bar{r}_{2,\mathcal{S}} \right)^2}$$

where $\mathcal{S}$ is the set where $r_{1,i}$ and $r_{2,i}$ are both less than their (own) quantile q. Upper partial correlation uses returns greater than quantile q. The function definition should have definition
   `def partial_corr(x, y=None, quantile = 0.5, tail = 'Lower')`
   and should take either a T by K array for x, or T by 1 arrays for x and y. If x is T by K, then y is ignored and the partial correlation should be computed pairwise. `quantile` determines the quantile to use for the cut off. Note: if $\mathcal{S}$ is empty or has 1 element, `nan` should be returned. `tail` is either `'Lower'` or `'Upper'`, and determined whether the lower or upper tail is used. The function should return both the partial correlation matrix (K by K), and the number of observations used in computing the partial correlation.

18.A Listing of econometrics.py

The complete code listing of econometrics, which contains the function olsnw, is presented below.

```python
from __future__ import print_function, division

from numpy import dot, mat, asarray, mean, size, shape, hstack, ones, ceil, \
    zeros, arange
from numpy.linalg import inv, lstsq

def olsnw(y, X, constant=True, lags=None):
    r""" Estimation of a linear regression with Newey-West covariance

    Parameters
    ----------
    y : array_like
        The dependent variable (regressand).  1-dimensional with T elements.
    X : array_like
        The independent variables (regressors). 2-dimensional with sizes T
        and K.  Should not include a constant.
    constant: bool, optional
        If true (default) includes model includes a constant.
    lags: int or None, optional
        If None, the number of lags is set to 1.2*T**(1/3), otherwise the
        number of lags used in the covariance estimation is set to the value
        provided.

    Returns
    -------
    b : ndarray, shape (K,) or (K+1,)
        Parameter estimates.  If constant=True, the first value is the
        intercept.
    vcv : ndarray, shape (K,K) or (K+1,K+1)
        Asymptotic covariance matrix of estimated parameters
    s2 : float
        Asymptotic variance of residuals, computed using Newey-West variance
        estimator.
    R2 : float
        Model R-square
    R2bar : float
        Adjusted R-square
    e : ndarray, shape (T,)
        Array containing the model errors

    Notes
    -----
    The Newey-West covariance estimator applies a Bartlett kernel to estimate
    the long-run covariance of the scores.  Setting lags=0 produces White's
    Heteroskedasticity Robust covariance matrix.
```

```python
    See also
    --------
    np.linalg.lstsq

    Example
    -------
    >>> X = randn(1000,3)
    >>> y = randn(1000,1)
    >>> b,vcv,s2,R2,R2bar = olsnw(y, X)

    Exclude constant:

    >>> b,vcv,s2,R2,R2bar = olsnw(y, X, False)

    Specify number of lags to use:

    >>> b,vcv,s2,R2,R2bar = olsnw(y, X, lags = 4)

    """


    T = y.size
    if size(X, 0) != T:
        X = X.T

    T,K = shape(X)
    if constant:
        X = copy(X)
        X = hstack((ones((T,1)),X))
        K = size(X,1)

    if lags is None:
        lags = int(ceil(1.2 * float(T)**(1.0/3)))

    # Parameter estimates and errors

    out = lstsq(X,y)
    b = out[0]
    e = y - dot(X,b)

    # Covariance of errors
    gamma = zeros((lags+1))
    for lag in xrange(lags+1):
        gamma[lag] = dot(e[:T-lag],e[lag:]) / T

    w = 1 - arange(0,lags+1)/(lags+1)
    w[0] = 0.5
    s2 = dot(gamma,2*w)
```

```python
# Covariance of parameters
Xe = mat(zeros(shape(X)))
for i in xrange(T):
    Xe[i] = X[i] * float(e[i])


Gamma = zeros((lags+1,K,K))
for lag in xrange(lags+1):
    Gamma[lag] = Xe[lag:].T*Xe[:T-lag]


Gamma = Gamma/T


S = Gamma[0].copy()
for i in xrange(1,lags+1):
    S = S + w[i]*(Gamma[i]+Gamma[i].T)


XpX = dot(X.T,X)/T
XpXi = inv(XpX)
vcv = mat(XpXi)*S*mat(XpXi)/T
vcv = asarray(vcv)


# R2, centered or uncentered
if constant:
    R2 = dot(e,e)/dot(y-mean(y),y-mean(y))
else:
    R2 = dot(e,e)/dot(y,y)


R2bar = 1-R2*(T-1)/(T-K)
R2 = 1 - R2


return b,vcv,s2,R2,R2bar,e
```

Chapter 19

Probability and Statistics Functions

This chapter is divided into two main parts, one for NumPy and one for SciPy. Both packages contain important functions for simulation, probability distributions and statistics.

NumPy

19.1 Simulating Random Variables

19.1.1 Core Random Number Generators

NumPy random number generators are all stored in the module `numpy.random`. These can be imported with using `import numpy as np` and then calling `np.random.rand`, for example, or by importing `import numpy.random as rnd` and using `rnd.rand`.[1]

rand, random_sample

`rand` and `random_sample` are uniform random number generators which are identical except that `rand` takes a variable number of integer inputs – one for each dimension – while `random_sample` takes a n-element tuple. `random_sample` is the preferred NumPy function, and `rand` is a convenience function primarily for MATLAB users.

```
>>> x = rand(3,4,5)
>>> y = random_sample((3,4,5))
```

randn, standard_normal

`randn` and `standard_normal` are standard normal random number generators. `randn`, like `rand`, takes a variable number of integer inputs, and `standard_normal` takes an n-element tuple. Both can be called with no arguments to generate a single standard normal (e.g. `randn()`). `standard_normal` is the preferred NumPy function, and `randn` is a convenience function primarily for MATLAB users .

```
>>> x = randn(3,4,5)
>>> y = standard_normal((3,4,5))
```

---

[1] Other import methods can also be used, such as `from numpy.random import rand` and then calling `rand`.

randint, random_integers

randint and random_integers are uniform integer random number generators which take 3 inputs, low, high and size. Low is the lower bound of the integers generated, high is the upper and size is a n-element tuple. randint and random_integers differ in that randint generates integers exclusive of the value in high (as do most Python functions), while random_integers includes the value in high in its range.

```
>>> x = randint(0,10,(100))
>>> x.max() # Is 9 since range is [0,10)
9

>>> y = random_integers(0,10,(100))
>>> y.max() # Is 10 since range is [0,10]
10
```

19.1.2 Random Array Functions

shuffle

shuffle randomly reorders the elements of an array *in place*.

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

permutation

permutation returns randomly reordered elements of an array as a copy while not directly changing the input.

```
>>> x = arange(10)
>>> permutation(x)
array([2, 5, 3, 0, 6, 1, 9, 8, 4, 7])

>>> x
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

19.1.3 Select Random Number Generators

NumPy provides a large selection of random number generators for specific distribution. All take between 0 and 2 required inputs which are parameters of the distribution, plus a tuple containing the size of the output. All random number generators are in the module numpy.random.

Bernoulli

There is no Bernoulli generator. Instead use binomial(1,p) to generate a single draw or binomial(1,p,(10,10)) to generate an array where p is the probability of success.

beta

`beta(a,b)` generates a draw from the Beta(a, b) distribution. `beta(a,b,(10,10))` generates a 10 by 10 array of draws from a Beta(a, b) distribution.

binomial

`binomial(n,p)` generates a draw from the Binomial(n, p) distribution. `binomial(n,p,(10,10))` generates a 10 by 10 array of draws from the Binomial(n, p) distribution.

chisquare

`chisquare(nu)` generates a draw from the χ^2_ν distribution, where ν is the degree of freedom. `chisquare(nu,(10,10))` generates a 10 by 10 array of draws from the χ^2_ν distribution.

exponential

`exponential()` generates a draw from the Exponential distribution with scale parameter $\lambda = 1$. `exponential(lambda, (10,10))` generates a 10 by 10 array of draws from the Exponential distribution with scale parameter λ.

f

`f(v1,v2)` generates a draw from the distribution F_{ν_1,ν_2} distribution where ν_1 is the numerator degree of freedom and ν_2 is the denominator degree of freedom. `f(v1,v2,(10,10))` generates a 10 by 10 array of draws from the F_{ν_1,ν_2} distribution.

gamma

`gamma(a)` generates a draw from the Gamma($\alpha, 1$) distribution, where α is the shape parameter. `gamma(a, theta, (10,10))` generates a 10 by 10 array of draws from the Gamma(α, θ) distribution where θ is the scale parameter.

laplace

`laplace()` generates a draw from the Laplace (Double Exponential) distribution with centered at 0 and unit scale. `laplace(loc, scale, (10,10))` generates a 10 by 10 array of Laplace distributed data with location `loc` and scale `scale`. Using `laplace(loc, scale)` is equivalent to calling `loc + scale*laplace()`.

lognormal

`lognormal()` generates a draw from a Log-Normal distribution with $\mu = 0$ and $\sigma = 1$. `lognormal(mu, sigma, (10,10))` generates a 10 by 10 array or Log-Normally distributed data where the underlying Normal distribution has mean parameter μ and scale parameter σ.

multinomial

multinomial(n, p) generates a draw from a multinomial distribution using n trials and where each outcome has probability p, a k-element array where $\Sigma_{i=1}^{k} p = 1$. Note that p must be an array or other iterable value. The output is a k-element array containing the number of successes in each category. multinomial(n, p, (10,10)) generates a 10 by 10 by k array of multinomially distributed data with n trials and probabilities p.

multivariate_normal

multivariate_normal(mu, Sigma) generates a draw from a multivariate Normal distribution with mean μ (k-element array) and covariance Σ (k by k array). multivariate_normal(mu, Sigma, (10,10)) generates a 10 by 10 by k array of draws from a multivariate Normal distribution with mean μ and covariance Σ.

negative_binomial

negative_binomial(n, p) generates a draw from the Negative Binomial distribution where n is the number of failures before stopping and p is the success rate. negative_binomial(n, p, (10, 10)) generates a 10 by 10 array of draws from the Negative Binomial distribution where n is the number of failures before stopping and p is the success rate.

normal

normal() generates draws from a standard Normal (Gaussian). normal(mu, sigma) generates draws from a Normal with mean μ and standard deviation σ. normal(mu, sigma, (10,10)) generates a 10 by 10 array of draws from a Normal with mean μ and standard deviation σ. normal(mu, sigma) is equivalent to mu + sigma * standard_normal().

poisson

poisson() generates a draw from a Poisson distribution with $\lambda = 1$. poisson(lambda) generates a draw from a Poisson distribution with expectation λ. poisson(lambda, (10,10)) generates a 10 by 10 array of draws from a Poisson distribution with expectation λ.

standard_t

standard_t(nu) generates a draw from a Student's t with shape parameter ν. standard_t(nu, (10,10)) generates a 10 by 10 array of draws from a Student's t with shape parameter ν.

uniform

uniform() generates a uniform random variable on $(0, 1)$. uniform(low, high) generates a uniform on (l, h). uniform(low, high, (10,10)) generates a 10 by 10 array of uniforms on (l, h).

19.2   Simulation and Random Number Generation

Computer simulated random numbers are usually constructed from very complex but ultimately deterministic functions. Because they are not actually random, simulated random numbers are generally described to as pseudo-random. All pseudo-random numbers in NumPy use one core random number generator based on the Mersenne Twister, a generator which can produce a very long series of pseudo-random data before repeating (up to $2^{19937} - 1$ non-repeating values).

RandomState

RandomState is the class used to control the random number generators. Multiple generators can be initialized by RandomState.

```
>>> gen1 = np.random.RandomState()
>>> gen2 = np.random.RandomState()
>>> gen1.uniform() # Generate a uniform
0.6767614077579269

>>> state1 = gen1.get_state()
>>> gen1.uniform()
0.6046087317893271

>>> gen2.uniform() # Different, since gen2 has different seed
0.04519705909244154

>>> gen2.set_state(state1)
>>> gen2.uniform() # Same uniform as gen1 after assigning state
0.6046087317893271
```

19.2.1   State

Pseudo-random number generators track a set of values known as the state. The state is usually a vector which has the property that if two instances of the same pseudo-random number generator have the same state, the sequence of pseudo-random numbers generated will be identical. The state of the default random number generator can be read using numpy.random.get_state and can be restored using numpy.random.set_state.

```
>>> st = get_state()
>>> randn(4)
array([ 0.37283499,  0.63661908, -1.51588209, -1.36540624])

>>> set_state(st)
>>> randn(4)
array([ 0.37283499,  0.63661908, -1.51588209, -1.36540624])
```

The two sequences are identical since they the state is the same when randn is called. The state is a 5-element tuple where the second element is a 625 by 1 vector of unsigned 32-bit integers. In practice the state should only be stored using get_state and restored using set_state.

get_state

get_state() gets the current state of the random number generator, which is a 5-element tuple. It can be called as a function, in which case it gets the state of the default random number generator, or as a method on a particular instance of RandomState.

set_state

set_state(state) sets the state of the random number generator. It can be called as a function, in which case it sets the state of the default random number generator, or as a method on a particular instance of RandomState. set_state should generally only be called using a state tuple returned by get_state.

19.2.2 Seed

numpy.random.seed is a more useful function for initializing the random number generator, and can be used in one of two ways. seed() will initialize (or reinitialize) the random number generator using some actual random data provided by the operating system.[2] seed(*s*) takes a vector of values (can be scalar) to initialize the random number generator at particular state. seed(*s*) is particularly useful for producing simulation studies which are reproducible. In the following example, calls to seed() produce different random numbers, since these reinitialize using random data from the computer, while calls to seed(0) produce the same (sequence) of random numbers.

```
>>> seed()
>>> randn()
array([ 0.62968838])

>>> seed()
>>> randn()
array([ 2.230155])

>>> seed(0)
>>> randn()
array([ 1.76405235])

>>> seed(0)
>>> randn()
array([ 1.76405235])
```

NumPy always calls seed() when the first random number is generated. As a result. calling standard_normal() across two "fresh" sessions will not produce the same random number.

seed

seed(value) uses *value* to seed the random number generator. seed() takes actual random data from the operating system when initializing the random number generator (e.g. /dev/random on Linux, or CryptGen-Random in Windows).

---

[2]All modern operating systems collect data that is effectively random by collecting noise from device drivers and other system monitors.

19.2.3 Replicating Simulation Data

It is important to have reproducible results when conducting a simulation study. There are two methods to accomplish this:

1. Call `seed()` and then `state = get_state()`, and save `state` to a file which can then be loaded in the future when running the simulation study.

2. Call `seed(s)` at the start of the program (where s is a constant).

Either of these will allow the same sequence of random numbers to be used.
Warning: Do not *over-initialize* the pseudo-random number generators. The generators should be initialized once per session and then allowed to produce the pseudo-random sequence. Repeatedly re-initializing the pseudo-random number generators will produce a sequence that is decidedly less random than the generator was designed to provide.

Considerations when Running Simulations on Multiple Computers

Simulation studies are ideally suited to parallelization, although parallel code makes reproducibility more difficult. There are 2 methods which can ensure that a parallel study is reproducible.

1. Have a single process produce all of the random numbers, where this process has been initialized using one of the two methods discussed in the previous section. Formally this can be accomplished by pre-generating all random numbers, and then passing these into the simulation code as a parameter, or equivalently by pre-generating the data and passing the state into the function. Inside the simulation function, the random number generator will be set to the state which was passed as a parameter. The latter is a better option if the amount of data per simulation is large.

2. Seed each parallel worker independently, and then return the state from the simulation function along with the simulation results. Since the state is saved for each simulation, it is possible to use the same state if repeating the simulation using, for example, a different estimator.

19.3 Statistics Functions

mean

`mean` computes the average of an array. An optional second argument provides the axis to use (default is to use entire array). `mean` can be used either as a function or as a method on an array.

```
>>> x = arange(10.0)
>>> x.mean()
4.5

>>> mean(x)
4.5

>>> x= reshape(arange(20.0),(4,5))
>>> mean(x,0)
```

```
array([ 7.5,   8.5,   9.5,  10.5,  11.5])

>>> x.mean(1)
array([  2.,   7.,  12.,  17.])
```

median

median computed the median value in an array. An optional second argument provides the axis to use (default is to use entire array).

```
>>> x= randn(4,5)
>>> x
array([[-0.74448693, -0.63673031, -0.40608815,  0.40529852, -0.93803737],
       [ 0.77746525,  0.33487689,  0.78147524, -0.5050722 ,  0.58048329],
       [-0.51451403, -0.79600763,  0.92590814, -0.53996231, -0.24834136],
       [-0.83610656,  0.29678017, -0.66112691,  0.10792584, -1.23180865]])

>>> median(x)
-0.45558017286810903

>>> median(x, 0)
 array([-0.62950048, -0.16997507,  0.18769355, -0.19857318, -0.59318936])
```

Note that when an array or axis dimension contains an even number of elements (n), median returns the average of the 2 inner elements.

std

std computes the standard deviation of an array. An optional second argument provides the axis to use (default is to use entire array). std can be used either as a function or as a method on an array.

var

var computes the variance of an array. An optional second argument provides the axis to use (default is to use entire array). var can be used either as a function or as a method on an array.

corrcoef

corrcoef(x) computes the correlation between the rows of a 2-dimensional array x. corrcoef(x, y) computes the correlation between two 1- dimensional vectors. An optional keyword argument rowvar can be used to compute the correlation between the columns of the input – this is corrcoef(x, rowvar=False) and corrcoef(x.T) are identical.

```
>>> x= randn(3,4)
>>> corrcoef(x)
array([[ 1.        ,  0.36780596,  0.08159501],
       [ 0.36780596,  1.        ,  0.66841624],
       [ 0.08159501,  0.66841624,  1.        ]])

>>> corrcoef(x[0],x[1])
```

```
array([[ 1.        ,  0.36780596],
       [ 0.36780596,  1.        ]])

>>> corrcoef(x, rowvar=False)
array([[ 1.        , -0.98221501, -0.19209871, -0.81622298],
       [-0.98221501,  1.        ,  0.37294497,  0.91018215],
       [-0.19209871,  0.37294497,  1.        ,  0.72377239],
       [-0.81622298,  0.91018215,  0.72377239,  1.        ]])

>>> corrcoef(x.T)
array([[ 1.        , -0.98221501, -0.19209871, -0.81622298],
       [-0.98221501,  1.        ,  0.37294497,  0.91018215],
       [-0.19209871,  0.37294497,  1.        ,  0.72377239],
       [-0.81622298,  0.91018215,  0.72377239,  1.        ]])
```

cov

cov(x) computes the covariance of an array x. cov(x,y) computes the covariance between two 1-dimensional vectors. An optional keyword argument rowvar can be used to compute the covariance between the columns of the input – this is cov(x, rowvar=False) and cov(x.T) are identical.

histogram

histogram can be used to compute the histogram (empirical frequency, using k bins) of a set of data. An optional second argument provides the number of bins. If omitted, $k = 10$ bins are used. histogram returns two outputs, the first with a k-element vector containing the number of observations in each bin, and the second with the $k + 1$ endpoints of the k bins.

```
>>> x = randn(1000)
>>> count, binends = histogram(x)
>>> count
array([  7,  27,  68, 158, 237, 218, 163,  79,  36,   7])

>>> binends
array([-3.06828057, -2.46725067, -1.86622077, -1.26519086, -0.66416096,
       -0.06313105,  0.53789885,  1.13892875,  1.73995866,  2.34098856,
        2.94201846])

>>> count, binends = histogram(x, 25)
```

histogram2d

histogram2d(x,y) computes a 2-dimensional histogram for 1-dimensional vectors. An optional keyword argument bins provides the number of bins to use. bins can contain either a single scalar integer or a 2-element list or array containing the number of bins to use in each dimension.

SciPy

SciPy provides an extended range of random number generators, probability distributions and statistical tests.

```
import scipy
import scipy.stats as stats
```

19.4 Continuous Random Variables

SciPy contains a large number of functions for working with continuous random variables. Each function resides in its own class (e.g. `norm` for Normal or `gamma` for Gamma), and classes expose methods for random number generation, computing the PDF, CDF and inverse CDF, fitting parameters using MLE, and computing various moments. The methods are listed below, where *dist* is a generic placeholder for the distribution name in SciPy. While the functions available for continuous random variables vary in their inputs, all take 3 generic arguments:

1. `*args` a set of distribution specific non-keyword arguments. These must be entered in the order listed in the class docstring. For example, when using a F-distribution, two arguments are needed, one for the numerator degree of freedom, and one for the denominator degree of freedom.

2. `loc` a location parameter, which determines the center of the distribution.

3. `scale` a scale parameter, which determine the scaling of the distribution. For example, if z is a standard normal, then $s \times z$ is a scaled standard normal.

dist.`rvs`

Pseudo-random number generation. Generically, `rvs` is called using *dist*`.rvs(*args, loc=0, scale=1, size=size)` where `size` is an n-element tuple containing the size of the array to be generated.

dist.`pdf`

Probability density function evaluation for an array of data (element-by-element). Generically, `pdf` is called using *dist*`.pdf(x, *args, loc=0, scale=1)` where x is an array that contains the values to use when evaluating PDF.

dist.`logpdf`

Log probability density function evaluation for an array of data (element-by-element). Generically, `logpdf` is called using *dist*`.logpdf(x, *args, loc=0, scale=1)` where x is an array that contains the values to use when evaluating log PDF.

dist.`cdf`

Cumulative distribution function evaluation for an array of data (element-by-element). Generically, `cdf` is called using *dist*`.cdf(x, *args, loc=0, scale=1)` where x is an array that contains the values to use when evaluating CDF.

dist.ppf

Inverse CDF evaluation (also known as percent point function) for an array of values between 0 and 1. Generically, `ppf` is called using *dist*.`ppf(p, *args, loc=0, scale=1)` where `p` is an array with all elements between 0 and 1 that contains the values to use when evaluating inverse CDF.

dist.fit

Estimate shape, location, and scale parameters from data by maximum likelihood using an array of data. Generically, `fit` is called using *dist*.`fit(data, *args, floc=0, fscale=1)` where `data` is a data array used to estimate the parameters. `floc` forces the location to a particular value (e.g. `floc=0`). `fscale` similarly forces the scale to a particular value (e.g. `fscale=1`). It is necessary to use `floc` and/or `fscale` when computing MLEs if the distribution does not have a location and/or scale. For example, the gamma distribution is defined using 2 parameters, often referred to as shape and scale. In order to use ML to estimate parameters from a gamma, `floc=0` must be used.

dist.median

Returns the median of the distribution. Generically, `median` is called using *dist*.`median(*args, loc=0, scale=1)`.

dist.mean

Returns the mean of the distribution. Generically, `mean` is called using *dist*.`mean(*args, loc=0, scale=1)`.

dist.moment

n^{th} non-central moment evaluation of the distribution. Generically, `moment` is called using *dist*.`moment(r, *args, loc=0, scale=1)` where `r` is the order of the moment to compute.

dist.varr

Returns the variance of the distribution. Generically, `var` is called using *dist*.`var(*args, loc=0, scale=1)`.

dist.std

Returns the standard deviation of the distribution. Generically, `std` is called using *dist*.`std(*args, loc=0, scale=1)`.

19.4.1 Example: `gamma`

The gamma distribution is used as an example. The gamma distribution takes 1 shape parameter *a* (*a* is the only element of `*args`), which is set to 2 in all examples.

```
>>> import scipy.stats as stats
>>> gamma = stats.gamma
>>> gamma.mean(2), gamma.median(2), gamma.std(2), gamma.var(2)
 (2.0, 1.6783469900166608, 1.4142135623730951, 2.0)

>>> gamma.moment(2,2) - gamma.moment(1,2)**2 # Variance
2
```

```
>>> gamma.cdf(5, 2), gamma.pdf(5, 2)
(0.95957231800548726, 0.033689734995427337)

>>> gamma.ppf(.95957231800548726, 2)
5.0000000000000018

>>> log(gamma.pdf(5, 2)) - gamma.logpdf(5, 2)
0.0

>>> gamma.rvs(2, size=(2,2))
array([[ 1.83072394,  2.61422551],
       [ 1.31966169,  2.34600179]])

>>> gamma.fit(gamma.rvs(2, size=(1000)), floc = 0) # a, 0, shape
(2.209958533078413, 0, 0.89187262845460313)
```

19.4.2 Important Distributions

SciPy provides classes for a large number of distribution. The most important are listed in the table below, along with any required arguments (shape parameters). All classes can be used with the keyword arguments `loc` and `scale` to set the location and scale, respectively. The default location is 0 and the default scale is 1. Setting `loc` to something other than 0 is equivalent to adding `loc` to the random variable. Similarly setting `scale` to something other than 0 is equivalent to multiplying the variable by `scale`.

Distribution Name	SciPy Name	Required Arguments	Notes
Normal	norm		Use `loc` to set mean (μ), `scale` to set std. dev. (σ)
Beta(a, b)	beta	a: a, b: b	
Cauchy	cauchy		
χ^2_ν	chi2	ν: df	
Exponential(λ)	expon		Use `scale` to set shape parameter (λ)
Exponential Power	exponpow	shape: b	Nests normal when b=2, Laplace when b=1
F(ν_1, ν_2)	f	ν_1: dfn, ν_2: dfd	
Gamma(a, b)	gamma	a: a	Use `scale` to set scale parameter (b)
Laplace, Double Exponential	laplace		Use `loc` to set mean (μ), `scale` to set std. dev. (σ)
Log Normal(μ, σ^2)	lognorm	σ: s	Use `scale` to set μ where scale=exp(mu)
Student's-t_ν	t	ν: df	

19.4.3 Frozen Random Variable Object

Random variable objects can be used in one of two ways:

1. Calling the class along with any shape, location and scale parameters, simultaneously with the method. For example `gamma(1, scale=2).cdf(1)`.

2. Initializing the class with any shape, location and scale arguments and assigning a variable name. Using the assigned variable name with the method. For example:

```
>>> g = scipy.stats.gamma(1, scale=2)
>>> g.cdf(1)
```

```
0.39346934028736652
```

The second method is known as using a frozen random variable object. If the same distribution (with fixed parameters) is repeatedly used, frozen objects can be used to save typing potentially improve performance since frozen objects avoid re-initializing the class.

19.5   Select Statistics Functions

mode

mode computes the mode of an array. An optional second argument provides the axis to use (default is to use entire array). Returns two outputs: the first contains the values of the mode, the second contains the number of occurrences.

```
>>> x=randint(1,11,1000)
>>> stats.mode(x)
(array([ 4.]), array([ 112.]))
```

moment

moment computed the rth central moment for an array. An optional second argument provides the axis to use (default is to use entire array).

```
>>> x = randn(1000)
>>> moment = stats.moment
>>> moment(x,2) - moment(x,1)**2
0.94668836546169166

>>> var(x)
0.94668836546169166

>>> x = randn(1000,2)
>>> moment(x,2,0) # axis 0
array([ 0.97029259,  1.03384203])
```

skew

skew computes the skewness of an array. An optional second argument provides the axis to use (default is to use entire array).

```
>>> x = randn(1000)
>>> skew = stats.skew
>>> skew(x)
0.027187705042705772

>>> x = randn(1000,2)
>>> skew(x,0)
array([ 0.05790773, -0.00482564])
```

kurtosis

kurtosis computes the *excess* kurtosis (actual kurtosis minus 3) of an array. An optional second argument provides the axis to use (default is to use entire array). Setting the keyword argument `fisher=False` will compute the actual kurtosis.

```
>>> x = randn(1000)
>>> kurtosis = stats.kurtosis
>>> kurtosis(x)
 -0.2112381820194531

>>> kurtosis(x, fisher=False)
2.788761817980547

>>> kurtosis(x, fisher=False) - kurtosis(x) # Must be 3
3.0

>>> x = randn(1000,2)
>>> kurtosis(x,0)
array([-0.13813704, -0.08395426])
```

pearsonr

pearsonr computes the Pearson correlation between two 1-dimensional vectors. It also returns the 2-tailed p-value for the null hypothesis that the correlation is 0.

```
>>> x = randn(10)
>>> y = x + randn(10)
>>> pearsonr = stats.pearsonr
>>> corr, pval = pearsonr(x, y)
>>> corr
0.40806165708698366

>>> pval
0.24174029858660467
```

spearmanr

spearmanr computes the Spearman correlation (rank correlation). It can be used with a single 2-dimensional array input, or 2 1-dimensional arrays. Takes an optional keyword argument `axis` indicating whether to treat columns (0) or rows (1) as variables. If the input array has more than 2 variables, returns the correlation matrix. If the input array as 2 variables, returns only the correlation between the variables.

```
>>> x = randn(10,3)
>>> spearmanr = stats.spearmanr
>>> rho, pval = spearmanr(x)
>>> rho
array([[ 1.        , -0.02087009, -0.05867387],
       [-0.02087009,  1.        ,  0.21258926],
       [-0.05867387,  0.21258926,  1.        ]])
```

```
>>> pval
array([[ 0.        ,  0.83671325,  0.56200781],
       [ 0.83671325,  0.        ,  0.03371181],
       [ 0.56200781,  0.03371181,  0.        ]])

>>> rho, pval = spearmanr(x[:,1],x[:,2])
>>> corr
-0.020870087008700869

>>> pval
0.83671325461864643
```

kendalltau

kendalltau computed Kendall's τ between 2 1-dimensonal arrays.

```
>>> x = randn(10)
>>> y = x + randn(10)
>>> kendalltau = stats.kendalltau
>>> tau, pval = kendalltau(x,y)
>>> tau
0.46666666666666673

>>> pval
0.06034053974834707
```

linregress

linregress estimates a linear regression between 2 1-dimensional arrays. It takes two inputs, the independent variables (regressors) and the dependent variable (regressand). Models always include a constant.

```
>>> x = randn(10)
>>> y = x + randn(10)
>>> linregress = stats.linregress
>>> slope, intercept, rvalue, pvalue, stderr = linregress(x,y)
>>> slope
1.6976690163576993

>>> rsquare = rvalue**2
>>> rsquare
0.59144988449163494

>>> x.shape = 10,1
>>> y.shape = 10,1
>>> z = hstack((x,y))
>>> linregress(z) # Alternative form, [x y]
(1.6976690163576993,
 -0.79983724584931648,
 0.76905779008578734,
```

```
0.0093169560056056751,
0.4988520051409559)
```

19.6 Select Statistical Tests

normaltest

`normaltest` tests for normality in an array of data. An optional second argument provides the axis to use (default is to use entire array). Returns the test statistic and the p-value of the test. This test is a small sample modified version of the Jarque-Bera test statistic.

kstest

`kstest` implements the Kolmogorov-Smirnov test. Requires two inputs, the data to use in the test and the distribution, which can be a string or a frozen random variable object. If the distribution is provided as a string, then any required shape parameters are passed in the third argument using a tuple containing these parameters, in order.

```
>>> x = randn(100)
>>> kstest = stats.kstest
>>> stat, pval = kstest(x, 'norm')
>>> stat
0.11526423481470172

>>> pval
0.12963296757465059

>>> ncdf = stats.norm().cdf # No () on cdf to get the function
>>> kstest(x, ncdf)
 (0.11526423481470172, 0.12963296757465059)

>>> x = gamma.rvs(2, size = 100)
>>> kstest(x, 'gamma', (2,)) # (2,) contains the shape parameter
(0.079237623453142447, 0.54096739528138205)

>>> gcdf = gamma(2).cdf
>>> kstest(x, gcdf)
(0.079237623453142447, 0.54096739528138205)
```

ks_2samp

`ks_2samp` implements a 2-sample version of the Kolmogorov-Smirnov test. It is called `ks_2samp(x,y)` where both inputs are 1-dimensonal arrays, and returns the test statistic and p-value for the null that the distribution of x is the same as that of y.

shapiro

shapiro implements the Shapiro-Wilk test for normality on a 1-dimensional array of data. It returns the test statistic and p-value for the null of normality.

19.7   Exercises

1. For each of the following distributions, simulate 1000 pseudo-random numbers:

    (a) $N(0, 1^2)$
    (b) $N\left(3, 3^2\right)$
    (c) $Unif\left(0, 1\right)$
    (d) $Unif\left(-1, 1\right)$
    (e) $Gamma\left(1, 2\right)$
    (f) $LogN\left(.08, .2^2\right)$

2. Use kstest to compute the p-value for each set of simulated data.

3. Use seed to re-initialize the random number generator.

4. Use get_state and set_state to produce the same set of pseudo-random numbers.

5. Write a custom function that will take a T vector of data and returns the mean, standard deviation, skewness and kurtosis (not excess) as a 4-element array.

6. Generate a 100 by 2 array of normal data with covariance matrix

$$
\begin{array}{cc}
1 & -.5 \\
-.5 & 1
\end{array}
$$

    and compute the Pearson and Spearman correlation and Kendall's τ.

7. Compare the analytical median of a Gamma$(1, 2)$ with that of 10,000 simulated data points. (You will need a hist , which is discussed in the graphics chapter to finish this problem.)

8. For each of the sets of simulated data in exercise 1, plot the sorted CDF values to verify that these lie on a 45° line. (You will need plot , which is discussed in the graphics chapter to finish this problem.)

Chapter 20

Optimization

SciPy contains a number of routines to the find extremum of a user-supplied objective function located in `scipy.optimize`. Most of these implement a version of the Newton-Raphson algorithm which uses the gradient to find the *minimum* of a function. However, this is not a limitation since if f is a function to be maximized, $-f$ is a function with the minimum at located the same point as the maximum of f.

A custom function that returns the function value at a set of parameters – for example a log-likelihood or a GMM quadratic form – is required to use one an optimizer. All optimization targets must have the parameters as the first argument. For example, consider finding the minimum of x^2. A function which allows the optimizer to work correctly has the form

```python
def optim_target1(x):
    return x**2
```

When multiple parameters (a parameter vector) are used, the objective function must take the form

```python
def optim_target2(params):
    x, y = params

    return x**2-3*x+3+y*x-3*y+y**2
```

Optimization targets can also have additional inputs that are not parameters of interest such as data or hyper-parameters.

```python
def optim_target3(params,hyperparams):
    x, y = params
    c1, c2, c3=hyperparams

    return x**2+c1*x+c2+y*x+c3*y+y**2
```

This form is especially useful when optimization targets require both parameters and data. Once an optimization target has been specified, the next step is to use one of the optimizers find the minimum. The remainder of this chapter assumes that the following import is used to import the SciPy optimizers.

```python
import scipy.optimize as opt
```

20.1 Unconstrained Optimization

A number of functions are available for unconstrained optimization using derivative information. Each uses a different algorithm to determine the best direction to move and the best step size to take in the direction. The basic structure of all of the unconstrained optimizers is

```
optimizer(f, x0)
```

where *optimizer* is one of `fmin_bfgs`, `fmin_cg`, `fmin_ncg` or `fmin_powell`, `f` is a callable function and `x0` is an initial value used to start the algorithm. All of the unconstrained optimizers take the following keyword arguments, except where noted:

Keyword	Description	Note
fprime	Function returning derivative of `f`. Must take same inputs as `f`	(1)
args	Tuple containing extra parameters to pass to `f`	
gtol	Gradient norm for terminating optimization	(1)
norm	Order of norm (e.g. inf or 2)	(1)
epsilon	Step size to use when approximating f'	(1)
maxiter	Integer containing the maximum number of iterations	
disp	Boolean indicating whether to print convergence message	
full_output	Boolean indicating whether to return additional output	
retall	Boolean indicating whether to return results for each iteration.	
callback	User supplied function to call after each iteration.	

(1) Except `fmin`, `fmin_powell`.

fmin_bfgs

`fmin_bfgs` is a classic optimizer which uses information in the 1st derivative to estimate the second derivative, an algorithm known as BFGS (after the initials of the creators). This should usually be the first option explored when optimizing a function without constraints. A function which returns the first derivative of the problem can also be provided, and if not provided, the first derivative is numerically approximated. The basic use of `fmin_bfgs` for finding the minimum of `optim_target1` is shown below.

```
>>> opt.fmin_bfgs(optim_target1, 2)
Optimization terminated successfully.
        Current function value: 0.000000
        Iterations: 2
        Function evaluations: 12
        Gradient evaluations: 4
array([ -7.45132576e-09])
```

This is a very simple function to minimize and the solution is accurate to 8 decimal places. `fmin_bfgs` can also use first derivative information, which is provided using a function which *must have the same inputs are the optimization target*. In this simple example, $f'(x) = 2x$.

```
def optim_target1_grad(x):
    return 2*x
```

The derivative information is used through the keyword argument `fprime`. Using analytic derivatives typically improves both the accuracy of the solution and the time required to find the optimum.

```
>>> opt.fmin_bfgs(optim_target1, 2, fprime = optim_target1_grad)
Optimization terminated successfully.
        Current function value: 0.000000
        Iterations: 2
        Function evaluations: 4
        Gradient evaluations: 4
array([  2.71050543e-20])
```

Multivariate optimization problems are defined using an array for the starting values, but are otherwise identical.

```
>>> opt.fmin_bfgs(optim_target2, array([1.0,2.0]))
Optimization terminated successfully.
        Current function value: 0.000000
        Iterations: 3
        Function evaluations: 20
        Gradient evaluations: 5
array([ 1.        ,  0.99999999])
```

Additional inputs are can be passed to the optimization target using the keyword argument `args` and a tuple containing the input arguments in the correct order. Note that since there is a single additional input, the comma is necessary in (`hyperp,`) to let Python know that this is a tuple.

```
>>> hyperp = array([1.0,2.0,3.0])
>>> opt.fmin_bfgs(optim_target3, array([1.0,2.0]), args=(hyperp,))
Optimization terminated successfully.
        Current function value: -0.333333
        Iterations: 3
        Function evaluations: 20
        Gradient evaluations: 5
array([ 0.33333332, -1.66666667])
```

Derivative functions can be produced in a similar manner, although the derivative of a scalar function with respect to an n-element vector is an n-element vector. It is important that the derivative (or gradient) returned has the same order as the input parameters. Note that the inputs must both be present, even when not needed, and in the same order.

```
def optim_target3_grad(params,hyperparams):
    x, y = params
    c1, c2, c3=hyperparams

    return array([2*x+c1+y, x+c3+2*y])
```

Using the analytical derivative reduces the number of function evaluations and produces the same solution.

```
>>> optimum = opt.fmin_bfgs(optim_target3, array([1.0,2.0]), fprime=optim_target3_grad, args=(hyperp ,))
Optimization terminated successfully.
        Current function value: -0.333333
        Iterations: 3
```

```
        Function evaluations: 5
        Gradient evaluations: 5
>>> optimum
array([ 0.33333333, -1.66666667])
>>> optim_target3_grad(optimum, hyperp) # Numerical zero
array([ -2.22044605e-16,   0.00000000e+00])
```

fmin_cg

fmin_cg uses a nonlinear conjugate gradient method to minimize a function. A function which returns the first derivative of the problem can be provided, and when not provided, the gradient is numerically approximated.

```
>>> opt.fmin_cg(optim_target3, array([1.0,2.0]), args=(hyperp ,))
Optimization terminated successfully.
        Current function value: -0.333333
        Iterations: 7
        Function evaluations: 59
        Gradient evaluations: 12
array([ 0.33333334, -1.66666666])
```

fmin_ncg

fmin_ncg use a Newton conjugate gradient method. fmin_ncg *requires* a function which can compute the first derivative of the optimization target, and can also take a function which returns the second derivative (Hessian) of the optimization target. It not provided, the Hessian will be numerically approximated.

```
>>> opt.fmin_ncg(optim_target3, array([1.0,2.0]), optim_target3_grad, args=(hyperp,))
Optimization terminated successfully.
        Current function value: -0.333333
        Iterations: 5
        Function evaluations: 6
        Gradient evaluations: 21
        Hessian evaluations: 0
array([ 0.33333333, -1.66666666])
```

The hessian can optionally be provided to fmin_ncg using the keyword argument fhess. The hessian returns $\partial^2 f / \partial x \partial x'$, which is an n by n array of derivatives. In this simple problem, the hessian does not depend on the hyper-parameters, although the Hessian function *must* take the same inputs are the optimization target.

```
def optim_target3_hess(params,hyperparams):
    x, y = params
    c1, c2, c3=hyperparams

    return(array([[2, 1],[1, 2]]))
```

Using an analytical Hessian can reduce the number of function evaluations. While in theory an analytical Hessian should produce better results, it may not improve convergence, especially if the Hessian is nearly singular for some parameter values (for example, near a saddle point which is not a minimum).

```
>>> opt.fmin_ncg(optim_target3, array([1.0,2.0]), optim_target3_grad, \
... fhess = optim_target3_hess, args=(hyperp ,))
Optimization terminated successfully.
        Current function value: -0.333333
        Iterations: 5
        Function evaluations: 6
        Gradient evaluations: 5
        Hessian evaluations: 5
array([ 0.33333333, -1.66666667])
```

In addition to the keyword argument outlined in the main table, `fmin_ncg` can take the following additional arguments.

Keyword	Description	Note
fhess_p	Function returning second derivative of f times a vector p. Must take same inputs as f	Only `fmin_ncg`
fhess	Function returning second derivative of f. Must take same inputs as f	Only `fmin_ncg`
avestol	Average relative error to terminate optimizer.	Only `fmin_ncg`

20.2 Derivative-free Optimization

Derivative free optimizers do not use gradients and so can be used in a wider variety of problems such as functions which are not continuously differentiable. They can also be used for functions which are continuously differentiable, although they are likely to be slower than derivative-based optimizers. Derivative free optimizers take some alternative keyword arguments.

Keyword	Description	Note
xtol	Change in x to terminate optimization	
ftol	Change in function to terminate optimization	
maxfun	Maximum number of function evaluations	
direc	Initial direction set, same size as x_0 by m	Only `fmin_powell`

fmin

`fmin` uses a simplex algorithm to minimize a function. The optimization in a simplex algorithm is often described as an amoeba which crawls around on the function surface expanding and contracting while looking for lower points. The method is derivative free, and so optimization target need not be continuously differentiable(e.g. the "tick" loss function used in estimation of quantile regression).

```python
def tick_loss(quantile, data, alpha):
    e = data - quantile

    return dot((alpha - (e<0)),e)
```

The tick loss function is used to estimate the median by using $\alpha = 0.5$. This loss function is not continuously differential and so standard derivative-based optimizers cannot be used.

```
>>> data = randn(1000)
>>> opt.fmin(tick_loss, 0, args=(data, 0.5))
Optimization terminated successfully.
        Current function value: -0.333333
        Iterations: 48
        Function evaluations: 91
array([-0.00475])
>>> median(data)
-0.0047118168472319406
```

The estimate is close to the sample median, as expected.

fmin_powell

fmin_powell used Powell's method, which is derivative free, to minimize a function. It is an alternative to fmin which uses a different algorithm.

```
>>> opt.fmin_powell(tick_loss, 0, args=(data, 0.5))
Optimization terminated successfully.
        Current function value: 396.760642
        Iterations: 1
        Function evaluations: 17
array(-0.004673123552046776)
```

fmin_powell converged quickly and requires far fewer function calls.

20.3   Constrained Optimization

Constrained optimization is frequently encountered in economic problems where parameters are only meaningful in some particular range – for example, a variance which must be weakly positive. The relevant class constrained optimization problems can be formulated

$$
\begin{aligned}
\min_\theta f(\theta) \quad & \text{subject to} \\
g(\theta) = 0 \quad & \text{(equality)} \\
h(\theta) \geq 0 \quad & \text{(inequality)} \\
\theta_L \leq \theta \leq \theta_H \quad & \text{(bounds)}
\end{aligned}
$$

where the bounds constraints are redundant if the optimizer allows for general inequality constraints since when a scalar x satisfies $x_L \leq x \leq x_H$, then $x - x_L \geq 0$ and $x_H - x \geq 0$. The optimizers in SciPy allow for different subsets of these constraints.

fmin_slsqp

fmin_slsqp is the most general constrained optimizer and allows for equality, inequality and bounds constraints. While bound constraints are redundant, constraints which take the form of bounds should be implemented using bounds since this provides more information directly to the optimizer. Constraints

are provided either as list of callable functions or as a single function which returns an array. The latter is simpler if there are multiple constraints, especially if the constraints can be easily calculated using linear algebra. Functions which compute the derivative of the optimization target, the derivative of the equality constraints, and the derivative of the inequality constraints can be optionally provided. If not provided, these are numerically approximated.

As an example, consider the problem of optimizing a CRS Cobb-Douglas utility function of the form $U(x_1, x_2) = x_1^\lambda x_2^{1-\lambda}$ subject to a budget constraint $p_1 x_1 + p_2 x_2 \leq 1$. This is a nonlinear function subject to a linear constraint (note that is must also be that case that $x_1 \geq 0$ and $x_2 \geq 0$). First, specify the optimization target

```
def utility(x, p, alpha):
    # Minimization, not maximization so -1 needed
    return -1.0 * (x[0]**alpha)*(x[1]**(1-alpha))
```

There are three constraints, $x_1 \geq 0$, $x_2 \geq 0$ and the budget line. All constraints must take the form of ≥ 0 constraint, so that the budget line can be reformulated as $1 - p_1 x_1 - p_2 x_2 \geq 0$. Note that the arguments in the constraint must be identical to those of the optimization target, which is why the utility function takes prices as an input, even though the prices are not required to compute the utility. Similarly the constraint function takes α as an unnecessary input.

```
def utility_constraints(x, p, alpha):
    return array([x[0], x[1], 1 - p[0]*x[0] - p[1]*x[1]])
```

The optimal combination of goods can be computed using `fmin_slsqp` once the starting values and other inputs for the utility function and budget constraint are constructed.

```
>>> p = array([1.0,1.0])
>>> alpha = 1.0/3
>>> x0 = array([.4,.4])
>>> opt.fmin_slsqp(utility, x0, f_ieqcons=utility_constraints, args=(p, alpha))
Optimization terminated successfully.    (Exit mode 0)
            Current function value: -0.529133683989
            Iterations: 2
            Function evaluations: 8
            Gradient evaluations: 2
array([ 0.33333333,  0.66666667])
```

`fmin_slsqp` can also take functions which compute the gradient of the optimization target, as well as the gradients of the constraint functions (both inequality and equality). The gradient of the optimization function should return a n-element vector, one for each parameter of the problem.

```
def utility_grad(x, p, alpha):
    grad = zeros(2)
    grad[0] = -1.0 * alpha * (x[0]**(alpha-1))*(x[1]**(1-alpha))
    grad[1] = -1.0 * (1-alpha) * (x[0]**(alpha))*(x[1]**(-alpha))
    return grad
```

The gradient of the constraint function returns a m by n array where m is the number of constraints. When both equality and inequality constraints are used, the number of constraints will be m_{eq} and m_{in} which will generally not be the same.

```
def utility_constraint_grad(x, p, alpha):
```

```
    grad = zeros((3,2)) # 3 constraints, 2 variables
    grad[0,0] = 1.0
    grad[0,1] = 0.0
    grad[1,0] = 0.0
    grad[1,1] = 1.0
    grad[2,0] = -p[0]
    grad[2,1] = -p[1]
    return grad
```

The two gradient functions can be passed using keyword arguments.

```
>>> opt.fmin_slsqp(utility, x0, f_ieqcons=utility_constraints, args=(p, alpha), \
... fprime = utility_grad, fprime_ieqcons = utility_constraint_grad)
Optimization terminated successfully.    (Exit mode 0)
          Current function value: -0.529133683989
          Iterations: 2
          Function evaluations: 2
          Gradient evaluations: 2
array([ 0.33333333,  0.66666667])
```

Like in other problems, gradient information reduces the number of iterations and/or function evaluations needed to find the optimum.

fmin_slsqp also accepts bounds constraints. Since two of the three constraints are $x_1 \geq 0$ and $x_2 \geq 0$, these can be easily specified as a bound. Bounds are given as a list of tuples, where there is a tuple for each variable with an upper and lower bound. It is not always possible to use np.inf as the upper bound, even if there is no implicit upper bound since this may produce a nan. In this example, 2 was used as the upper bound since it was outside of the possible range given the constraint. Using bounds also requires reformulating the budget constraint to only include the budget line.

```
def utility_constraints_alt(x, p, alpha):
    return array([1 - p[0]*x[0] - p[1]*x[1]])
```

Bounds are used with the keyword argument bounds.

```
>>> opt.fmin_slsqp(utility, x0, f_ieqcons=utility_constraints_alt, args=(p, alpha), \
... bounds = [(0.0,2.0),(0.0,2.0)])
Optimization terminated successfully.    (Exit mode 0)
          Current function value: -0.529133683989
          Iterations: 2
          Function evaluations: 8
          Gradient evaluations: 2
array([ 0.33333333,  0.66666667])
```

The use of non-linear constraints can be demonstrated by formulating the dual problem of cost minimization subject to achieving a minimal amount of utility. In this alternative formulation, the optimization problems becomes

$$\min_{x_1, x_2} p_1 x_1 + p_2 x_2 \text{ subject to } U(x_1, x_2) \geq \bar{U}$$

```
def total_expenditure(x,p,alpha,Ubar):
    return dot(x,p)
```

```
def min_utility_constraint(x,p,alpha,Ubar):
    x1,x2 = x
    u=x1**(alpha)*x2**(1-alpha)
    return array([u - Ubar]) # >= constraint, must be array, even if scalar
```

The objective and the constraint are used along with a bounds constraint to solve the constrained optimization problem.

```
>>> x0 = array([1.0,1.0])
>>> p = array([1.0,1.0])
>>> alpha = 1.0/3
>>> Ubar = 0.529133683989
>>> opt.fmin_slsqp(total_expenditure, x0, f_ieqcons=min_utility_constraint, \
...     args=(p, alpha, Ubar), bounds =[(0.0,2.0),(0.0,2.0)])
Optimization terminated successfully.    (Exit mode 0)
            Current function value: 0.999999999981
            Iterations: 6
            Function evaluations: 26
            Gradient evaluations: 6
Out[84]: array([ 0.33333333,  0.66666667])
```

As expected, the solution is the same.

fmin_tnc

fmin_tnc supports only bounds constraints.

fmin_l_bfgs_b

fmin_l_bfgs_b supports only bounds constraints.

fmin_cobyla

fmin_cobyla supports only inequality constraints, which must be provided as a list of functions. Since it supports general inequality constraints, bounds constraints are included as a special case, although these must be included in the list of constraint functions.

```
def utility_constraints1(x, p, alpha):
    return x[0]

def utility_constraints2(x, p, alpha):
    return x[1]

def utility_constraints3(x, p, alpha):
    return (1 - p[0]*x[0] - p[1]*x[1])
```

Note that fmin_cobyla takes a list rather than an array for the starting values. Using an array produces a warning, but otherwise works.

```
>>> p = array([1.0,1.0])
>>> alpha = 1.0/3
>>> x0 = array([.4,.4])
```

```
>>> cons = [utility_constraints1, utility_constraints2, utility_constraints3]
>>> opt.fmin_cobyla(utility, x0, cons, args=(p, alpha), rhoend=1e-7)
array([ 0.33333326,  0.66666674])
```

20.3.1 Reparameterization

Many constrained optimization problems can be converted into an unconstrained program by reparameterizing from the space of unconstrained variables into the space where the parameters must reside. For example, the constraints in the utility function optimization problem require $0 \leq x_1 \leq 1/p_1$ and $0 \leq x_2 \leq 1/p_2$. Additionally the budget constraint must be satisfied so that if $x_1 \in [0, 1/p_1]$, $x_2 \in [0, (1 - p_1 x_1)/p_2]$. These constraints can be implemented using a "squasher" function which maps x_1 into its domain, and x_2 into its domain and is one-to-one and onto (i.e. a bijective relationship). For example,

$$
x_1 = \frac{1}{p_1} \frac{e^{z_1}}{1 + e^{z_1}}, \quad x_2 = \frac{1 - p_1 x_1}{p_2} \frac{e^{z_2}}{1 + e^{z_2}}
$$

will always satisfy the constraints, and so the constrained utility function can be mapped to an unconstrained problem, which can then be optimized using an unconstrained optimizer.

```
def reparam_utility(z,p,alpha,printX = False):
    x = exp(z)/(1+exp(z))
    x[0] = (1.0/p[0]) * x[0]
    x[1] = (1-p[0]*x[0])/p[1] * x[1]
    if printX:
        print(x)
    return -1.0 * (x[0]**alpha)*(x[1]**(1-alpha))
```

The unconstrained utility function can be minimized using `fmin_bfgs`. Note that the solution returned is in the transformed space, and so a special call to `reparam_utility` is used to print the actual values of x at the solution (which are virtually identical to those found using the constrained optimizer).

```
>>> x0 = array([.4,.4])
>>> optX = opt.fmin_bfgs(reparam_utility, x0, args=(p,alpha))
Optimization terminated successfully.
         Current function value: -0.529134
         Iterations: 24
         Function evaluations: 104
         Gradient evaluations: 26
>>> reparam_utility(optX, p, alpha, printX=True)
[ 0.33334741  0.66665244]
```

20.4 Scalar Function Minimization

SciPy provides a number of scalar function minimizers. These are very fast since additional techniques are available for solving scalar problems which are not applicable when the parameter vector has more than 1 element. A simple quadratic function will be used to illustrate the scalar solvers. Scalar function minimizers do not require starting values, but may require bounds for the search.

```
def optim_target5(x, hyperparams):
```

```
    c1,c2,c3 = hyperparams

    return c1*x**2 + c2*x + c3
```

fminbound

fminbound finds the minimum of a scalar function between two bounds.

```
>>> hyperp = array([1.0, -2.0, 3])
>>> opt.fminbound(optim_target5, -10, 10, args=(hyperp,))
1.0000000000000002
>>> opt.fminbound(optim_target5, -10, 0, args=(hyperp,))
-5.3634455116374429e-06
```

golden

golden uses a golden section search algorithm to find the minimum of a scalar function. It can optionally be provided with bracketing information which can speed up the solution.

```
>>> hyperp = array([1.0, -2.0, 3])
>>> opt.golden(optim_target5, args=(hyperp,))
0.999999992928981
>>> opt.golden(optim_target5, args=(hyperp,), brack=[-10.0,10.0])
0.9999999942734483
```

brent

brent uses Brent's method to find the minimum of a scalar function.

```
>>> opt.brent(optim_target5, args=(hyperp,))
0.99999998519
```

20.5 Nonlinear Least Squares

Non-linear least squares (NLLS) is similar to general function minimization. In fact, a generic function minimizer can (attempt to) minimize a NLLS problem. The main difference is that the optimization target returns a vector of errors rather than the sum of squared errors.

```
def nlls_objective(beta, y, X):
    b0 = beta[0]
    b1 = beta[1]
    b2 = beta[2]

    return y - b0 - b1 * (X**b2)
```

A simple non-linear model is used to demonstrate leastsq, the NLLS optimizer in SciPy.

$$y_i = \beta_1 + 2\beta_2 x^{\beta_3} + e_i$$

247

where x and e are i.i.d. standard normal random variables. The true parameters are $\beta_1 = 10$, $\beta_2 = 2$ and $\beta_3 = 1.5$.

```
>>> X = 10 *rand(1000)
>>> e = randn(1000)
>>> y = 10 + 2 * X**(1.5) + e
>>> beta0 = array([10.0,2.0,1.5])
>>> opt.leastsq(nlls_objective, beta0, args = (y, X))
(array([ 10.08885711,   1.9874906 ,   1.50231838]), 1)
```

leastsq returns a tuple containing the solution, which is very close to the true values, as well as a flag indicating that convergence was achieved. leastsq takes many of the same additional keyword arguments as other optimizers, including full_output, ftol, xtol, gtol, maxfev (same as maxfun). It has the additional keyword argument:

Keyword	Description	Note
Ddun	Function to compute the Jacobian of the problem. Element i, j should be $\partial e_i / \partial \beta_j$	
col_deriv	Direction to use when computing Jacobian numerically	
epsfcn	Step to use in numerical Jacobian calculation.	
diag	Scalar factors for the parameters. Used to rescale if scale is very different.	
factor	used to determine the initial step size.	Only fmin_powell

20.6 Exercises

1. The MLE for μ in a normal random variable is the sample mean. Write a function which takes a scalar parameter μ (1st argument) and a T vector of data and computes the negative of the log-likelihood, assuming the data is random and the variance is 1. Minimize the function (starting from something other than the same mean) using fmin_bfgs and fmin.

2. Extend to previous example where the first input is a 2-element vector containing μ and σ^2, and compute the negative log-likelihood. Use fmin_slsqp along with a lower bound of 0 for σ^2.

3. Repeat the exercise in problem 2, except using reparameterization so that σ is input (and then squared).

4. Verify that the OLS β is the MLE by writing a function which takes 3 inputs: K vector β,T by K array X and T by 1 array y, and computes the negative log-likelihood for these values. Minimize the function using fmin_bfgs starting at the OLS estimates of β.

Chapter 21

String Manipulation

Strings are usually less interesting than numerical values in econometrics and statistics. There are, however, some important uses for strings:

- Reading complex data formats

- Outputting formatted results to screen or file

Recall that strings are sliceable, but unlike arrays, are immutable, and so it is not possible to replace part of a string.

21.1 String Building

21.1.1 Adding Strings (+)

Strings are concatenated using +.

```
>>> a = 'Python is'
>>> b = 'a rewarding language.'
>>> a + ' ' + b
'Python is a rewarding language.'
```

While + is a simple method to joint strings, the modern method is to use `join`. `join` is a string method which joins a list of strings (the input) using the object calling the string as the separator.

```
>>> a = 'Python is'
>>> b = 'a rewarding language.'
>>> ' '.join([a,b])
'Python is a rewarding language.'
```

Alternatively, the same output may be constructed using an empty string `''`.

```
>>> a = 'Python is'
>>> b = 'a rewarding language.'
>>> ''.join([a,' ',b])
'Python is a rewarding language.'
```

`join` is also useful for producing comma separated lists.

```
>>> words = ['Python','is','a','rewarding','language']
>>> ','.join(words)
'Python,is,a,rewarding,language'
```

21.1.2 Multiplying Strings (∗)

Strings, like lists, can be repeated using ∗.

```
>>> a = 'Python is '
>>> 2*a
'Python is Python is '
```

21.1.3 Using cStringIO

While adding strings using + or join is extremely simple, concatenation is slow for large strings. The module cStringIO provides an optimized class for performing string operations, including buffering strings for fast string building. This example shows how write(*string*) fills a StringIO buffer. Before reading the contents seek(0) is called to return to cursor to the beginning of the buffer, and then read() returns the entire string from the buffer.

```
>>> import cStringIO
>>> sio = cStringIO.StringIO()
>>> for i in xrange(10000):
...     sio.write('cStringIO is faster than +! ')

>>> sio.seek(0)
>>> sio.read()
```

Note that this example is trivial since ∗ could have been used instead.

21.2 String Functions

21.2.1 `split` and `rsplit`

split splits a string into a list based on a character, for example a comma. An optional third argument maxsplit can be used to limit the number of outputs in the list. rsplit works identically to split, only scanning from the end of the string – split and rsplit only differ when maxsplit is used.

```
>>> s = 'Python is a rewarding language.'
>>> s.split(' ')
['Python', 'is', 'a', 'rewarding', 'language.']

>>> s.split(' ',3)
['Python', 'is', 'a', 'rewarding language.']

>>> s.rsplit(' ',3)
['Python is', 'a', 'rewarding', 'language.']
```

21.2.2 `join`

`join` concatenates a list or tuple of strings, using an optional argument `sep` which specified a separator (default is space).

```
>>> import string
>>> a = 'Python is'
>>> b = 'a rewarding language.'
>>> string.join((a,b))
'Python is a rewarding language.'

>>> string.join((a,b),':')
'Python is:a rewarding language.'

>>> ' '.joint((a,b)) # Method version
'Python is a rewarding language.'
```

21.2.3 `strip`, `lstrip`, and `rstrip`

`strip` removes leading and trailing whitespace from a string. An optional input `char` removes leading and trailing occurrences of the input value (instead of space). `lstrip` and `rstrip` work identically, only stripping from the left and right, respectively.

```
>>> s = '    Python is a rewarding language.    '
>>> s=s.strip()
'Python is a rewarding language.'

>>> s.strip('P')
'ython is a rewarding language.'
```

21.2.4 `find` and `rfind`

`find` locates the lowest index of a substring in a string and returns -1 if not found. Optional arguments limit the range of the search, and `s.find('i',10,20)` is identical to `s[10:20].find('i')`. `rfind` works identically, only returning the highest index of the substring.

```
>>> s = 'Python is a rewarding language.'
>>> s.find('i')
7

>>> s.find('i',10,20)
18

>>> s.rfind('i')
18
```

`find` and `rfind` are commonly used in flow control.

```
>>> words = ['apple','banana','cherry','date']
>>> words_with_a = []
>>> for word in words:
```

```
...    if word.find('a')>=0:
...        words_with_a.append(word)

>>> words_with_a
['apple', 'banana', 'date']
```

21.2.5 index and rindex

index returns the lowest index of a substring, and is identical to find except that an error is raised if the substring does not exist. As a result, index is only safe to use in a try ... except block.

```
>>> s = 'Python is a rewarding language.'
>>> s.index('i')
7

>>> s.index('q') # Error
ValueError: substring not found
```

21.2.6 count

count counts the number of occurrences of a substring, and takes optional arguments to limit the search range.

```
>>> s = 'Python is a rewarding language.'
>>> s.count('i')
2

>>> s.count('i', 10, 20)
1
```

21.2.7 lower and upper

lower and upper convert strings to lower and upper case, respectively. They are useful to remove case when comparing strings.

```
>>> s = 'Python is a rewarding language.'
>>> s.upper()
'PYTHON IS A REWARDING LANGUAGE.'

>>> s.lower()
'python is a rewarding language.'
```

21.2.8 ljust, rjust and center

ljust, rjust and center left justify, right justify and center, respectively, a string while expanding its size to a given length. If the desired length is smaller than the string, the unchanged string is returned.

```
>>> s = 'Python is a rewarding language.'
>>> s.ljust(40)
'Python is a rewarding language.         '
```

```
>>> s.rjust(40)
'         Python is a rewarding language.'

>>> s.center(40)
'    Python is a rewarding language.     '
```

21.2.9 `replace`

`replace` replaces a substring with an alternative string, which can have different size. An optional argument limits the number of replacement.

```
>>> s = 'Python is a rewarding language.'
>>> s.replace('g','Q')
'Python is a rewardinQ lanQuaQe.'

>>> s.replace('is','Q')
'Python Q a rewarding language.'

>>> s.replace('g','Q',2)
'Python is a rewardinQ lanQuage.'
```

21.2.10 `textwrap.wrap`

The module `textwrap` contains a function `wrap` which reformats a long string into a fixed width paragraph stored line-by-line in a list. An optional argument changes the width of the output paragraph form the default of 70 characters.

```
>>> import textwrap
>>> s = 'Python is a rewarding language. '
>>> s = 10*s
>>> textwrap.wrap(s)
['Python is a rewarding language. Python is a rewarding language. Python',
'is a rewarding language. Python is a rewarding language. Python is a',
'rewarding language. Python is a rewarding language. Python is a',
'rewarding language. Python is a rewarding language. Python is a',
'rewarding language. Python is a rewarding language.']

>>> textwrap.wrap(s,50)
['Python is a rewarding language. Python is a',
'rewarding language. Python is a rewarding',
'language. Python is a rewarding language. Python',
'is a rewarding language. Python is a rewarding',
'language. Python is a rewarding language. Python',
'is a rewarding language. Python is a rewarding',
'language. Python is a rewarding language.']
```

21.3 Formatting Numbers

Formatting numbers when converting to a string allows for automatic generation of tables and well formatted screen output. Numbers are formatted using the `format` function, which is used in conjunction with a format specifier. For example, consider these examples which format π.

```
>>> pi
3.141592653589793

>>> '{:12.5f}'.format(pi)
'     3.14159'

>>> '{:12.5g}'.format(pi)
'      3.1416'

>>> '{:12.5e}'.format(pi)
' 3.14159e+00'
```

These all provide alternative formats and the difference is determined by the letter in the format string. The generic form of a format string is $\{n : f\,a\,s\,w\,c.p\,t\}$ or $\{n : f\,a\,s\,w\,c\,m\,t\}$. To understand the the various choices, consider the output produced by the basic output string `'{0:}'`

```
>>> '{0:}'.format(pi)
'3.14159265359'
```

- n is a number 0,1,... indicating which value to take from the format function

  ```
  >>> '{0:}, {1:} and {2:} are all related to pi'.format(pi,pi+1,2*pi)
  '3.14159265359, 4.14159265359 and 6.28318530718 are all related to pi'

  >>> '{2:}, {0:} and {1:} reorder the output.'.format(pi,pi+1,2*pi)
  '6.28318530718, 3.14159265359 and 4.14159265359 reorder the output.
  ```

- $f\,a$ are fill and alignment characters, typically a 2 character string. Fill may be any character except }, although space is the most common choice. Alignment can < (left) ,> (right), ^ (center) or = (pad to the right of the sign). Simple left 0-fills can omit the alignment character so that $f\,a = 0$.

  ```
  >>> '{0:0<20}'.format(pi) # Left, 0 padding, precion 20
  '3.141592653590000000'

  >>> '{0:0>20}'.format(pi) # Right, 0 padding, precion 20
  '00000003.14159265359'

  >>> '{0:0^20}'.format(pi) # Center, 0 padding, precion 20
  '0003.141592653590000'

  >>> '{0: >20}'.format(pi) # Right, space padding, precion 20
  '         3.14159265359'

  >>> '{0:$^20}'.format(pi) # Center, dollar sign padding, precion 20
  '$$$3.14159265359$$$$'
  ```

- *s* indicates whether a sign should be included. + indicates always include sign, – indicates only include if needed, and a blank space indicates to use a blank space for positive numbers, and a − sign for negative numbers – this format is useful for producing aligned tables.

```
>>> '{0:+}'.format(pi)
'+3.14159265359'

>>> '{0:+}'.format(-1.0 * pi)
'-3.14159265359'

>>> '{0:-}'.format(pi)
'3.14159265359'

>>> '{0: }'.format(pi)
' 3.14159265359'

>>> '{0: }'.format(-1.0 * pi)
'-3.14159265359'
```

- *m* is the minimum total size of the formatted string

```
>>> '{0:10}'.format(pi)
'3.14159265359'

>>> '{0:20}'.format(pi)
'       3.14159265359'

>>> '{0:30}'.format(pi)
'                 3.14159265359'
```

- *c* may be , or omitted. , produces numbers with 1000s separated using a ,. In order to use *c* it is necessary to include the . before the precision.

```
>>> '{0:.10}'.format(1000000 * pi)
'3141592.654'

>>> '{0:,.10}'.format(1000000 * pi)
'3,141,592.654'
```

- *p* is the precision. The interpretation of precision depends on *t*. In order to use *p*, it is necessary to include a . (dot). If not included, *p* will be interpreted as *m*.

```
>>> '{0:.1}'.format(pi)
'3e+00'

>>> '{0:.2}'.format(pi)
'3.1'
```

```
>>> '{0:.5}'.format(pi)
'3.1416'
```

- *t* is the type. Options include:

Type	Description
e, E	Exponent notation, e produces e+ and E produces E+ notation
f, F	Display number using a fixed number of digits
g, G	General format, which uses f for smaller numbers, and e for larger. G is equivalent to switching between F and E. g is the default format if no presentation format is given
n	Similar to g, except that it uses locale specific information.
%	Multiplies numbers by 100, and inserts a % sign

```
>>> '{0:.5e}'.format(pi)
'3.14159e+00'

>>> '{0:.5g}'.format(pi)
'3.1416'

>>> '{0:.5f}'.format(pi)
'3.14159'

>>> '{0:.5%}'.format(pi)
'314.15927%'

>>> '{0:.5e}'.format(100000 * pi)
'3.14159e+05'

>>> '{0:.5g}'.format(100000 * pi)
'3.1416e+05'

>>> '{0:.5f}'.format(100000 * pi)
'314159.26536'
```

Combining all of these features in a single format string produces complexly presented data.

```
>>> '{0: > 20.4f}, {1: > 20.4f}'.format(pi,-pi)
'              3.1416,              -3.1416'

>>> '{0: >+20,.2f}, {1: >+20,.2f}'.format(100000 * pi,-100000 * pi)
'         +314,159.27,         -314,159.27'
```

In the first example, reading from left to right after the colon, the format string consists of:

1. Space fill (the blank space after the colon)

2. Right align (>)

3. Use no sign for positive numbers, $-$ sign for negative numbers (the blank space after >)

256

4. Minimum 20 digits

5. Precision of 4 fixed digits

The second is virtually identical to the first, except that it includes a , to show the 1000s separator and a +
to force the sign to be shown.

21.3.1 Formatting Strings

`format` outputs formatted strings using a similar syntax to number formatting, although some options
such as precision, sign, comma and type are not relevant.

```
>>> s = 'Python'
>>> '{0:}'.format(s)
'Python'

>>> '{0: >20}'.format(s)
'              Python'

>>> '{0:!>20}'.format(s)
'!!!!!!!!!!!!!!Python'

>>> 'The formatted string is: {0:!<20}'.format(s)
'The formatted string is: Python!!!!!!!!!!!!!!'
```

21.3.2 Formatting Multiple Objects

`format` also formats multiple objects in the same string output. There are three methods to do this:

- No position arguments, in which case the objects are matched to format strings in order

- Numeric positional arguments, in which case the first object is mapped to `'{0:}'`, the second to
  `'{1:}'`, and so on.

- Named arguments such as `'{price:}'` and volume `'{volume:}'`, which match keyword arguments
  inside format.

```
>>> price = 100.32
>>> volume = 132000
>>> 'The price yesterday was {:} with volume {:}'.format(price,volume)
'The price yesterday was 100.32 with volume  132000'

>>> 'The price yesterday was {0:} and the volume was {1:}'.format(price,volume)
'The price yesterday was 100.32 with volume 132000'

>>> 'The price yesterday was {1:} and the volume was {0:}'.format(volume,price)
'The price yesterday was 100.32 with volume 132000'

>>> 'The price yesterday was {price:} and the volume was {volume:}'.format(price=price,volume=volume)
'The price yesterday was 100.32 with volume 132000'
```

21.3.3 Old style format strings

Some Python code still uses an older style format string. Old style format strings have %(*map*)*f l m.p t*, where:

- (*map*) is a mapping string containing a name, for example (price)

- *f l* is a flag which may be one or more of:

  - 0: Zero pad
  - (blank space)
  - – Left adjust output
  - + Include sign character

- *m*, *p* and *t* are identical to those of the new format strings.

In general, the old format strings should only be used when required by other code (e.g. matplotlib). Below are some examples of their use in strings.

```
>>> price = 100.32
>>> volume = 132000
>>> 'The price yesterday was %0.2f with volume %d' % (price, volume)
'The price yesterday was 100.32 with volume  132000'

>>> 'The price yesterday was %(price)0.2f with volume %(volume)d' \
...     % {'price': price, 'volume': volume}
'The price yesterday was 100.32 with volume 132000'

>>> 'The price yesterday was %+0.3f and the volume was %010d' % (price, volume)
'The price yesterday was +100.320 and the volume was 0000132000'
```

21.4 Regular Expressions

Regular expressions are powerful tools for matching patterns in strings. While reasonable coverage of regular expressions is beyond the scope of these notes – there are 500 page books dedicated to constructing regular expressions – they are sufficiently useful to warrant an introduction. There are many online regular expression generators which can assist in finding the pattern to use, and so they are accessible to even casual users working with unformatted text.

Using regular expression requires the re module. The most useful functions for regular expression matching are findall, finditer and sub. findall and finditer work in similar manners, except that findall returns a list while finditer returns an iterable. finditer is preferred if a large number of matches is possible. Both search through a string and find all non-overlapping matches of a regular expression.

```
>>> import re
>>> s = 'Find all numbers in this string: 32.43, 1234.98, and 123.8.'
>>> re.findall('[\s][0-9]+\.\d*',s)
[' 32.43', ' 1234.98', ' 123.8']
```

```
>>> matches = re.finditer('[\s][0-9]+\.\d*',s)
>>> for m in matches:
...     print(s[m.span()[0]:m.span()[1]])
 32.43
 1234.98
 123.8
```

`finditer` returns `MatchObjects` which contain the method `span`. `span` returns a 2 element tuple which contains the start and end position of the match.

sub replaces all matched text with another text string (or a function which takes a `MatchObject`).

```
>>> s = 'Find all numbers in this string: 32.43, 1234.98, and 123.8.'
>>> re.sub('[\s][0-9]+\.\d*',' NUMBER',s)
'Find all numbers in this string: NUMBER, NUMBER, and NUMBER.'

>>> def reverse(m):
...     """Reverse the string in the MatchObject group"""
...     s = m.group()
...     s = s.rstrip()
...     return ' ' + s[::-1]

>>> re.sub('[\s][0-9]+\.\d*',reverse,s)
'Find all numbers in this string: 34.23, 89.4321, and 8.321.'
```

21.4.1 Compiling Regular Expressions

When repeatedly using a regular expression, for example running it on all lines in a file, it is better to compile the regular expression, and then to use the resulting `RegexObject`.

```
>>> import re
>>> s = 'Find all numbers in this string: 32.43, 1234.98, and 123.8.'
>>> numbers = re.compile('[\s][0-9]+\.\d*')
>>> numbers.findall(s)
[' 32.43', ' 1234.98', ' 123.8']
```

Parsing the regular expression text is relatively expensive, and compiling the expression avoids this cost.

21.5 Safe Conversion of Strings

When reading data into Python using a mixed format, blindly converting text to integers or floats is dangerous. For example, `float('a')` returns a `ValueError` since Python doesn't know how to convert `'a'` to a string. The simplest method to safely convert potentially non-numeric data is to use a `try ... except` block.

```
from __future__ import print_function
from __future__ import division

S = ['1234','1234.567','a','1234.a34','1.0','a123']
for s in S:
    try:
```

```python
        # If integer, use int
        int(s)
        print(s, 'is an integer.')
except:
    try:
        # If not integer, may be float
        float(s)
        print(s, 'is a float.')
    except:
        print('Unable to convert', s)
```

Chapter 22

File System Operations

Manipulating files and directories is surprising useful when undertaking complex projects. The most important file system commands are located in the modules os and shutil. This chapter assumes that

```python
import os
import shutil
```

have been included.

22.1 Changing the Working Directory

The working directory is where files can be created and accessed without any path information. os.getcwd() can be used to determine the current working directory, and os.chdir(*path*) can be used to change the working directory, where *path* is a directory, such as /temp or c:\\temp.[1] Alternatively, *path* can can be .. to more up the directory tree.

```python
pwd = os.getcwd()
os.chdir('c:\\temp')
os.chdir(r'c:\temp') # Raw string, no need to escape \
os.chdir('c:/temp')  # Identical
os.chdir('..')       # Walk up the directory tree
os.getcwd()          # Now in 'c:\\'
```

22.2 Creating and Deleting Directories

Directories can be created using os.mkdir(*dirname*), although it must be the case that the higher level directories exist (e.g. to create /home/username/Python/temp, it /home/username/Python already exists). os.makedirs(*dirnam*) works similar to os.mkdir(*dirname*), except that is will create any higher level directories needed to create the target directory.

Empty directories can be deleted using os.rmdir(*dirname*) – if the directory is not empty, an error occurs. shutil.rmtree(*dirname*) works similarly to os.rmdir(*dirname*), except that it will delete the directory, and any files or other directories contained in the directory.

---

[1]On Windows, directories use the backslash, which is used to escape characters in Python, and so an escaped backslash – \\ – is needed when writing Windows' paths. Alternatively, the forward slash can be substituted, so that c:\\temp and c:/temp are equivalent.

```
os.mkdir('c:\\temp\\test')
os.makedirs('c:/temp/test/level2/level3')  # mkdir will fail
os.rmdir('c:\\temp\\test\\level2\\level3')
shutil.rmtree('c:\\temp\\test')  # rmdir fails, since not empty
```

22.3   Listing the Contents of a Directory

The contents of a directory can be retrieved in a list using os.listdir(*dirname*), or simply os.listdir('.')
to list the current working directory. The list returned contains all files and directories. os.path.isdir(
name) can be used to determine whether a value in the list is a directory, and os.path.isfile(*name*)
can be used to determine if it is a file. os.path contains other useful functions for working with directory
listings and file attributes.

```
os.chdir('c:\\temp')
files = os.listdir('.')
for f in files:
    if os.path.isdir(f):
        print(f, ' is a directory.')
    elif os.path.isfile(f):
        print(f, ' is a file.')
    else:
        print(f, ' is a something else.')
```

A more sophisticated listing which accepts wildcards and is similar to dir (Windows) and ls (Linux)
can be constructed using the glob module.

```
import glob
files = glob.glob('c:\\temp\\*.txt')

for file in files:
    print(file)
```

22.4   Copying, Moving and Deleting Files

File contents can be copied using shutil.copy(*src* , *dest*), shutil.copy2(*src* , *dest*) or shutil.copyfile(
src , *dest*). These functions are all similar, and the differences are:

- shutil.copy will accept either a filename or a directory as *dest*. If a directory is given, the a file is
  created in the directory with the same name as the original file

- shutil.copyfile requires a filename for *dest*.

- shutil.copy2 is identical to shutil.copy except that metadata, such as last access times, is also
  copied.

Finally, shutil.copytree(*src* , *dest*) will copy an entire directory tree, starting from the directory *src* to
the directory *dest,* which must *not* exist. shutil.move(*src* , *dest*) is similar to shutil.copytree, except that
it moves a file or directory tree to a new location. If preserving file metadata (such as permissions or file

streams) is important, it is better use system commands (copy or move on Windows, cp or mv on Linux) as an external program.

```
os.chdir('c:\\temp\\python')
# Make an empty file
f = file('file.ext','w')
f.close()
# Copies file.ext to 'c:\temp\'
shutil.copy('file.ext','c:\\temp\\')
# Copies file.ext to 'c:\temp\\python\file2.ext'
shutil.copy('file.ext','file2.ext')
# Copies file.ext to 'c:\\temp\\file3.ext', plus metadata
shutil.copy2('file.ext','file3.ext')
shutil.copytree('c:\\temp\\python\\','c:\\temp\\newdir\\')
shutil.move('c:\\temp\\newdir\\','c:\\temp\\newdir2\\')
```

22.5 Executing Other Programs

Occasionally it is necessary to call other programs, for example to decompress a file compressed in an unusual format or to call system copy commands to preserve metadata and file ownership. Both os.system and subprocess.call (which requires import subprocess) can be used to execute commands as if they were executed directly in the shell.

```
import subprocess

# Copy using xcopy
os.system('xcopy /S /I c:\\temp c:\\temp4')
subprocess.call('xcopy /S /I c:\\temp c:\\temp5',shell=True)
# Extract using 7-zip
subprocess.call('"C:\\Program Files\\7-Zip\\7z.exe" e -y c:\\temp\\zip.7z')
```

22.6 Creating and Opening Archives

Creating and extracting files from archives often allows for further automation in data processing. Python has native support for zip, tar, gzip and bz2 file formats using shutil.make_archive(*archivename* , *format*, *root*) where *archivename* is the name of the archive to create, without the extension, *format* is one of the supported formats (e..g 'zip' for a zip archive or 'gztar', for a gzipped tar file) and *root* is the root directory which can be '.' for the current working directory.

```
# Creates files.zip
shutil.make_archive('files','zip','c:\\temp\\folder_to_archive')
# Creates files.tar.gz
shutil.make_archive('files','gztar','c:\\temp\\folder_to_archive')
```

Creating a standard gzip from an existing file is slightly more complicated, and requires using the gzip module.[2]

---

[2]A gzip can only contain 1 file, and is usually used with a tar file to compress a directory or set of files.

```python
import gzip

# Create file.csv.gz from file.csv
csvin = file('file.csv','rb')
gz = gzip.GzipFile('file.csv.gz','wb')
gz.writelines(csvin.read())
gz.close()
csvin.close()
```

Zip files can be extracted using the module `zipfile`, gzip files can be extracted using `gzip`, and gzipped tar files can be extracted using `tarfile`.

```python
import zipfile
import gzip
import tarfile

# Extract zip
zip = zipfile.ZipFile('files.zip')
zip.extractall('c:\\temp\\zip\\')
zip.close()

# Extract gzip tar 'r:gz' indicates read gzipped
gztar = tarfile.open('file.tar.gz', 'r:gz')
gztar.extractall('c:\\temp\\gztar\\')
gztar.close()

# Extract csv from gzipped csv
gz = gzip.GzipFile('file.csv.gz','rb')
csvout = file('file.csv','wb')
csvout.writelines(gz.read())
csvout.close()
gz.close()
```

22.7 Reading and Writing Files

Occasionally it may be necessary to directly read or write a file, for example to output a formatted LaTeX table. Python contains low level file access tools which can be used to to generate files with any structure. Writing text files begins by using `file` to create a new file or to open an existing file. Files can be opened in different modes: 'r' for reading, 'w' for writing, and 'a' for appending ('w' will overwrite an existing file). An additional modifier 'b' can be be used if the file is binary (not text), so that 'rb', 'wb' and 'ab' allow reading, writing and appending binary files.

Reading text files is usually implemented using `readline()` to read a single line, `readlines(`n`)` to reads approximately n bytes or `readlines()` to read all lines in a file. `readline` and `readlines(`n`)` are usually used inside a while loop which terminates if the value returned is an empty string (`''`, `readline`) or an empty list (`[]`, `readlines`). Note that both `''` and `[]` are false, and so can be directly used in a while statement.

```python
# Read all lines using readlines()
f = file('file.csv','r')
lines = f.readlines()
```

```python
for line in lines:
    print(line)
f.close()

# Using blocking via readline()
f = file('file.csv','r')
line = f.readline()
while line:
    print(line)
    line = f.readline()

f.close()

# Using larger blocks via readlines(n)
f = file('file.csv','r')
lines = f.readlines(2)
while lines:
    for line in lines:
        print(line)
    lines = f.readline(2)

f.close()
```

Writing text files is similar, and begins by using `file` to create a file and then `fwrite` to output information. `fwrite` is conceptually similar to using print, except that the output will be written to a file rather than printed on screen. The next example show how to create a LATEX table from an array.

```python
import numpy as np
import scipy.stats as stats

x = np.random.randn(100,4)
mu = np.mean(x,0)
sig = np.std(x,0)
sk = stats.skew(x,0)
ku = stats.kurtosis(x,0)

summaryStats = np.vstack((mu,sig,sk,ku))
rowHeadings = ['Var 1','Var 2','Var 3','Var 4']
colHeadings = ['Mean','Std Dev','Skewness','Kurtosis']

# Build table, then print
latex = []
latex.append('\\begin{tabular}{r|rrrr}')
line = ' '
for i in xrange(len(colHeadings)):
    line += ' & ' + rowHeadings[i]

line += ' \\ \hline'
latex.append(line)
```

```
for i in xrange(size(summaryStats,0)):
    line = rowHeadings[i]
    for j in xrange(size(summaryStats,1)):
        line += ' & ' + str(summaryStats[i,j])

    latex.append(line)

latex.append('\\end{tabular}')

# Output using write()
f = file('latex_table.tex','w')
for line in latex:
    f.write(line + '\n')

f.close()
```

22.8 Exercises

1. Create a new directory, *chapter22*.

2. Change into this directory.

3. Create a new file names *tobedeleted.py* a text editor in this new directory (It can be empty).

4. Create a zip file *tobedeleted.zip* containing *tobedeleted.py*.

5. Get and print the directory listing.

6. Delete the newly created file, and then delete this directory.

Chapter 23

Performance and Code Optimization

> *We should forget about small efficiencies, say about 97% of the time: premature optimiza-*
> *tion is the root of all evil.*

Donald Knuth

23.1 Getting Started

Occasionally the performance of a direct implementation of a statistical algorithm will not execute quickly enough be applied to interesting data sets. When this occurs, there are a number of alternatives ranging from improvements possible using only NumPy and Python to using native code through a Python module.

Note that before any code optimization, it is *essential* that a clean, working implementation is available. This allows for both measuring performance improvements and to ensure that optimizations have not introduced any bugs. The famous quote of Donald Knuth should also be heeded, and in practice code optimization is only needed for a very small amount of code – code that is frequently executed.

23.2 Timing Code

Timing code is an important step in measuring performance. IPython contains the magic keywords `%timeit` and `%time` which can be used to measure the execution time of a block of code. `%time` simply runs the code and reports the time needed. `%timeit` is smarter in that it will vary the number of iterations to increase the accuracy of the timing. Both are used with the same syntax, `%timeit` *code to time*.[1]

```
>>> x = randn(1000,1000)
>>> %timeit inv(dot(x.T,x))
1 loops, best of 3: 387 ms per loop

>>> %time inv(dot(x.T,x))
Wall time: 0.52 s

>>> x = randn(100,100)
```

---

[1] All timings were performed on an Intel i3 550 using Anaconda 1.7.0.

```
>>> %timeit inv(dot(x.T,x))
1000 loops, best of 3: 797 us per loop
```

23.3   Vectorize to Avoid Unnecessary Loops

Vectorization is the key to writing high performance code in Python. Code that is vectorized run *insides* NumPy and so executes as quickly as possible (with some small technical caveats, see NumExpr). Consider the difference between manually multiplying two matrices and using dot.

```
def pydot(a, b):
    M,N = shape(a)
    P,Q = shape(b)
    c = zeros((M,Q))
    for i in xrange(M):
        for j in xrange(Q):
            for k in xrange(N):
                c[i,j] = c[i,j] + a[i,k] * b[k,j]
    return c
```

Timing the difference shows that NumPy is about 10000x faster than looping Python.

```
>>> a = randn(100,100)
>>> b = randn(100,100)
>>> %timeit pydot(a,b)
1 loops, best of 3: 1.02 s per loop

>>> %timeit dot(a,b)
10000 loops, best of 3: 95.2 us per loop

>>> 1.02/0.0000952
10714.285
```

A less absurd example is to consider computing a weighted moving average across m consecutive values of a vector.

```
def naive_weighted_avg(x, w):
    T = x.shape[0]
    m = len(w)
    m12 = int(ceil(m/2))
    y = zeros(T)
    for i in xrange(len(x)-m+1):
        y[i+m12] = dot(x[i:i+m].T,w)

    return y
```

```
>>> w = array(r_[1:11,9:0:-1],dtype=float64)
>>> w = w/sum(w)
>>> x = randn(10000)
>>> %timeit naive_weighted_avg(x,w)
10 loops, best of 3: 22.1 ms per loop
```

An alternative method which completely avoids loops can be constructed by carefully constructing an array containing the data. This array allows `dot` to be used with the weights.

```python
def clever_weighted_avg(x,w):
    T = x.shape[0]
    m = len(w)
    wc = copy(w)
    wc.shape = m,1
    T = x.size
    xc = copy(x)
    xc.shape=T,1
    y = vstack((xc,zeros((m,1))))
    y = tile(y,(m,1))

    y = reshape(y[:len(y)-m],(m,T+m-1))
    y = y.T
    y = y[m-1:T,:]

    return dot(y,flipud(wc))
```

```
>>> %timeit clever_weighted_avg(x,w)
100 loops, best of 3: 1.59 ms per loop
```

The loop-free method which uses copying and slicing is about 12 times faster than the simple looping specification.

23.4 Alter the loop dimensions

In many applications, it may be natural to loop over the long dimension in a time series. This is especially common if the mathematical formula underlying the program has a sum from $t = 1$ to T. In some cases, it is possible to replace a loop over time, which is assumed to be the larger dimension, with an alternative loop across another iterable. For example, in the moving average, it is possible to loop over the weights rather than the data, and if the moving windows length is much smaller than the length of the data, the code should run much faster.

```python
def sideways_weighted_avg(x, w):
    T = x.shape[0]
    m = len(w)
    y = zeros(T)
    m12 = int(ceil(m/2))
    for i in xrange(m):
        y[m12:T-m+m12] = x[i:T+i-m] * w[i]

    return y
```

```
>>> %timeit sideways_weighted_avg(x,w)
1000 loops, best of 3: 498 us per loop
```

In this example, the "sideways" loop is much faster than fully vectorized version since it avoids allocating a large amount of memory.

23.5 Utilize Broadcasting

NumPy uses broadcasting for virtually all primitive mathematical operations (and for some more complicated functions). Broadcasting avoids unnecessary matrix replication and memory allocation, and so improves performance.

```
>>> x = randn(1000,1)
>>> y = randn(1,1000)
>>> %timeit x*y
100 loops, best of 3: 8.77 ms per loop

>>> %timeit dot(x,ones((1,1000))) * dot(ones((1000,1)),y)
10 loops, best of 3: 36.7 ms per loop
```

Broadcasting is about 4 times as fast as manually expanding the arrays.

23.6 Use In-place Assignment

In-place assignment uses the save variable and avoids unnecessary memory allocation. The in-place operators use a syntax similar to x += 0.0 or x *= 1.0 instead of x = x + 0.0.

```
>>> x = zeros(1000000)
>>> %timeit global x; x += 0.0
1000 loops, best of 3: 1.89 ms per loop

>>> %timeit global x; x = x + 0.0
100 loops, best of 3: 6.89 ms per loop
```

The gains to in-place allocation are larger as the dimension of x increases.

23.7 Avoid Allocating Memory

Memory allocation is relatively expensive, especially if it occurs inside a for loop. It is often better to pre-allocate storage space for computed values, and also to reuse existing space. Similarly, prefer slices and views to operations which create copies of arrays.

23.8 Inline Frequent Function Calls

Function calls are fast but not completely free. Simple functions, especially inside loops, should be inlined to avoid the cost of calling functions.

23.9 Consider Data Locality in Arrays

Arrays are stored using row major format, and so data is stored across a row first, and then down columns second. This means that in an m by n array, element i, j is stored next to elements $i, j + 1$ and $i, j - 1$ (except when j is the first (previous is $i - 1, n$) or last element in a row (next is $i + 1, 1$)). Spatial location matters for performance, and it is faster to access data which is stored physically adjacent. The simplest method to understand array storage is to use:

```
>>> x = arange(16.0)
>>> x.shape = 4,4
>>> x
array([[  0.,   1.,   2.,   3.],
       [  4.,   5.,   6.,   7.],
       [  8.,   9.,  10.,  11.],
       [ 12.,  13.,  14.,  15.]])
```

23.10 Profile Long Running Functions

Profiling provides detailed information about the number of times a line is executed as well as the execution time spent on each line. The default Python profiling tools are not adequate to address all performance measurement issues in NumPy code, and so a third party library known as line_profiler is needed. line_profiler is not currently available in Anaconda and so it must be installed before use. On Linux, this module can be installed using

```
source ANACONDA/bin/activate econometrics
pip install line_profiler
```

where ANACONDA is the full path to the Anaconda installation (e.g. ~/anaconda). These two lines assume that the Anaconda environment is being used. If not using anaconda, simply activate the virtualenv-created virtual environment and then run the `pip` command above.

Installation on Windows/Anaconda is somewhat more complicated since line_profiler uses compiled code. `pip` cannot be used without first setting up a compiler environment, which is a challenging task. These alternative instructions make use of binary installer made available by Christoph Gohlke.

1. Download line_profiler-1.0b3.win-amd64-py2.7.exe from Christoph Gohlke's website. Note that there may be a newer version available on the site.

2. Copy register_python.py from Section 1.A to

   ```
   ANACONDA\envs\econometrics
   ```

   where ANACONDA is the full path to the Anaconda installation (e.g. c:\Anaconda). This file is also available for download with the solutions to these notes.

3. Open an elevated command prompt using Run as Administrator.

4. Activate the environment by running

   ```
   ANACONDA\Scripts\activate.bat econometrics
   ```

5. Run the register_python file using

   ```
   cd ANACONDA\envs\econometrics
   python register_python.py
   ```

6. Run the line_profiler installer.

7. [OPTIONAL] If the default Python should not be the same as the environment used in the notes (and it probably should not), repeat steps 3 – 5 using the default Python. For example, if using a standard Python installation in C:\Python27, first copy register_python.py to C:\Python27 and then run

```
cd c:\Python27
python register_python.py
```

Alternatively, if using Anaconda as the default Python, copy register_python.py to ANACONDA and then run

```
cd ANACONDA
python register_python.py
```

If using a non-Anaconda Python install on windows, the instructions are identical to those in Chapter 1 – run register_python.py and the line_profiler installer. If required, the virtual environment can be unregistered following the instructions in step 7.

IPython Magic Keyword for Line Profiling

The simplest method to profile function is to use IPython. This requires a small amount of setup to define a new magic word, `%lprun`.

```
>>> import IPython
>>> ip = IPython.get_ipython()
>>> import line_profiler
>>> ip.define_magic('lprun', line_profiler.magic_lprun)
```

Note that the final two of these fours lines can also be incorporated into startup.py (see Chapter 1) so that the magic word `%lprun` is available in all IPython sessions.

To demonstrate the use of line_profiler, the three moving average functions where combined into a single python file moving_avgs.py. line_profiler is used with the syntax `%lprun -f` *function command* where *function* is the function to profile and *command* is a command which will cause the function to run. *command* can be either a simple call to the function or a call to some other code that will run the function.

```
>>> from moving_avgs import naive_weighted_avg
>>> w = array(r_[1:11,9:0:-1],dtype=float64)
>>> w = w/sum(w)
>>> x = randn(100000)
>>> %lprun -f naive_weighted_avg naive_weighted_avg(x,w)
Timer unit: 3.94742e-07 s

File: moving_avgs.py
Function: naive_weighted_avg at line 16
Total time: 1.04589 s

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    16                                           def naive_weighted_avg(x, w):
    17         1           27     27.0      0.0       T = x.shape[0]
```

```
    18          1          13     13.0       0.0        m = len(w)
    19          1         120    120.0       0.0        m12 = int(ceil(m/2))
    20          1         755    755.0       0.0        y = zeros(T)
    21      99983      505649      5.1      19.1        for i in xrange(len(x)-m+1):
    22      99982     2142994     21.4      80.9            y[i+m12] = dot(x[i:i+m].T,w)
    23
    24          1           6      6.0       0.0        return y
```

The first attempt at a weighted average, `naive_weighted_average`, spent all of the time in the loop and most of this on the `dot` product.

```
>>> from moving_avgs import clever_weighted_avg
>>> %lprun -f clever_weighted_avg clever_weighted_avg(x,w)
Timer unit: 3.94742e-07 s

File: moving_avgs.py
Function: clever_weighted_avg at line 27
Total time: 0.0302076 s

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    27                                           def clever_weighted_avg(x,w):
    28          1          33     33.0       0.0      T = x.shape[0]
    29          1          11     11.0       0.0      m = len(w)
    30          1          98     98.0       0.1      wc = copy(w)
    31          1          33     33.0       0.0      wc.shape = m,1
    32          1           9      9.0       0.0      T = x.size
    33          1         738    738.0       1.0      xc = copy(x)
    34          1          42     42.0       0.1      xc.shape=T,1
    35          1        1605   1605.0       2.1      y = vstack((xc,zeros((m,1))))
    36          1       25286  25286.0      33.0      y = tile(y,(m,1))
    37
    38          1          98     98.0       0.1      y = reshape(y[:len(y)-m],(m,T+m-1))
    39          1          12     12.0       0.0      y = y.T
    40          1          38     38.0       0.0      y = y[m-1:T,:]
    41
    42          1       48522  48522.0      63.4      return dot(y,flipud(wc))
```

The second attempt, `clever_weighted_avg`, spends 1/3 of the time in the tile `tile` command and the remainder in the `dot`.

```
>>> from moving_avgs import sideways_weighted_avg
>>> %lprun -f sideways_weighted_avg sideways_weighted_avg(x,w)
Timer unit: 3.94742e-07 s

File: moving_avgs.py
Function: sideways_weighted_avg at line 45
Total time: 0.00962302 s

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    45                                           def sideways_weighted_avg(x, w):
```

```
46              1          25      25.0     0.1       T = x.shape[0]
47              1          10      10.0     0.0       m = len(w)
48              1         417     417.0     1.7       y = zeros(T)
49              1         182     182.0     0.7       m12 = int(ceil(m/2))
50             20         230      11.5     0.9       for i in xrange(m):
51             19       23508    1237.3    96.4           y[m12:T-m+m12] = x[i:T+i-m] * w[i]
52
53              1           6       6.0     0.0       return y
```

The final version spends most of its time in the dot product and the only other line with meaningful time is the call to zeros. Note the actual time was .0096 vs 1.06 for the naive version and .030 for the loop-free version. Comparing the naive and the sideways version really highlights the cost of repeated calls to simple functions inside loops dot as well as the loop overhead.

Directly Using the Line Profiler

Directly using line_profiler requires adding the *decorator* @profile to a function. Consider profiling the three weighted average functions.

```python
from __future__ import print_function, division
from numpy import ceil, zeros, dot, copy, vstack, flipud, reshape, tile, array, float64, r_
from numpy.random import randn

# Useful block but not necessary
import __builtin__

try:
    __builtin__.profile
except AttributeError:
    # No line profiler, provide a pass-through version
    def profile(func): return func
    __builtin__.profile = profile
# Useful block but not necessary

@profile
def naive_weighted_avg(x, w):
    T = x.shape[0]
    m = len(w)
    m12 = int(ceil(m/2))
    y = zeros(T)
    for i in xrange(len(x)-m+1):
        y[i+m12] = dot(x[i:i+m].T,w)

    return y


@profile
def clever_weighted_avg(x,w):
    T = x.shape[0]
    m = len(w)
    wc = copy(w)
```

```
    wc.shape = m,1
    T = x.size
    xc = copy(x)
    xc.shape=T,1
    y = vstack((xc,zeros((m,1))))
    y = tile(y,(m,1))

    y = reshape(y[:len(y)-m],(m,T+m-1))
    y = y.T
    y = y[m-1:T,:]

    return dot(y,flipud(wc))

@profile
def sideways_weighted_avg(x, w):
    T = x.shape[0]
    m = len(w)
    y = zeros(T)
    m12 = int(ceil(m/2))
    y = zeros(x.shape)
    for i in xrange(m):
        y[m12:T-m+m12] = x[i:T+i-m] * w[i]

    return y

w = array(r_[1:11,9:0:-1],dtype=float64)
w = w/sum(w)
x = randn(100000)

naive_weighted_avg(x,w)
clever_weighted_avg(x,w)
sideways_weighted_avg(x,w)
```

The decorator @profile specifies which functions should be profiled by line_profiler, and should only be used on functions where profiling is needed. The final lines in this file call the functions, which is necessary for the profiling.

To profile the on Windows code (saved in moving_avgs_direct.py), run the following commands from a command prompt (not inside IPython)

```
ANACONDA\Scripts\activate.bat econometrics
cd PATHTOFILE
python ANACONDA\envs\econometrics\Scripts\kernprof.py -l moving_avgs_direct.py
python -m line_profiler moving_avgs_direct.py.lprof > moving_avgs_direct.prof.txt
```

where PATHTOFILE is the location of moving_avgs_direct.py. The first command activates the environment. The second changes to the directory where moving_avgs_direct.py is located. The Third actually executes the file with profiling, and the final produces a report in moving_avgs_direct.prof.txt, which can then be viewed in any text editor.[2]

---

[2]The Windows command is more complex than the Linux command to ensure that the correct Python interpreter and environment is used to execute kernprof.py.

On Linux or OSX, run

```
source ANACONDA\bin\activate econometrics
cd PATHTOFILE
kernprof -l moving_avgs.py
python -m line_profiler moving_avgs.py.lprof > moving_avgs.prof.txt
```

The file moving_avg.prof.txt will contain a line-by-line listing of the three function which includes the number to times the line was hit as well as the time spent on each line.

Modification of Code

In the direct method, the file moving_avgs_direct.py has a strange block reproduced below.

```python
# Useful block but not necessary
import __builtin__

try:
    __builtin__.profile
except AttributeError:
    # No line profiler, provide a pass-through version
    def profile(func): return func
    __builtin__.profile = profile
# Useful block but not necessary
```

I like to use this block since the decorator `@profile` is only defined when running in a profile session. Attempting the run a file without this block in a standard python session will produce an `AttributeError` since `profile` is not defined. This block allows the code to be run both with and without profiling by first checking if `profile` is defined, and if not, providing a trivial definition that does nothing.

23.11   Numba

If pure Python/NumPy is slow due to the presence of loops, Numba may be useful for transforming standard Python to a faster form of code that can run in a Low Level Virtual Machine (LLVM). Numba is particularly attractive since it usually only requires adding a *decorator* immediately before the def *function*(): line. Consider a generic recursion from a GARCH(P,Q) model that computes the conditional variance given parameters, data and a backcast value. In pure Python/NumPy this function is

```python
def garch_recursion(parameters, data, sigma2, p, q, backcast):
    T = size(data,0)
    for i in xrange(T):
        sigma2[i] = parameters[0]
        for j in xrange(p):
            if (i-j)<0:
                sigma2[i] = parameters[1+j] * backcast
            else:
                sigma2[i] = parameters[1+j] * (data[i-j]*data[i-j])
        for j in xrange(q):
            if (i-j)<0:
                sigma2[i] = parameters[1+p+j] * backcast
            else:
```

```
                    sigma2[i] = parameters[1+p+j] * sigma2[i-j]

    return sigma2
```

This example is fairly straight forward and only involves the (slow) recursive calculation of the conditional variance, not the other portions of the log-likelihood (which can be vectorized using NumPy). The pure Python version can be tested using `timeit`.

```
>>> parameters = array([.1,.1,.8])
>>> data = randn(10000)
>>> sigma2 = zeros(shape(data))
>>> p,q = 1,1
>>> backcast = 1.0
>>> %timeit garch_recursion(parameters, data, sigma2, p, q, backcast)
10 loops, best of 3: 21.7 ms per loop
```

Using Numba starts with `from numba import autojit`, and then specifying a function with the decorator `@autojit`.

```
@autojit
def garch_recursion_numba_auto(parameters, data, sigma2, p, q, backcast):
    T = size(data,0)
    for i in xrange(T):
        sigma2[i] = parameters[0]
        for j in xrange(p):
            if (i-j)<0:
                sigma2[i] = parameters[1+j] * backcast
            else:
                sigma2[i] = parameters[1+j] * (data[i-j]*data[i-j])
        for j in xrange(q):
            if (i-j)<0:
                sigma2[i] = parameters[1+p+j] * backcast
            else:
                sigma2[i] = parameters[1+p+j] * sigma2[i-j]

    return sigma2
```

The Numba version can be tested by changing the function name.

```
>>> %timeit garch_recursion_numba_auto(parameters, data, sigma2, p, q, backcast)
10000 loops, best of 3: 66.5 us per loop

>>> 'The speed-up is {0:.1f} times'.format(0.0217/0.0000665 - 1.0)
'The speed-up is 325.3 times'
```

Two lines of code – an import and a decorator – produce code that runs over 300 times faster. Alternatively, `autojit` can be used as a function to produce a just-in-time compiled function. This version is an alternative to use `autojit` but is otherwise identical.

```
>>> garch_recursion_numba_auto_command = autojit(garch_recursion)
```

In some cases, it may be desirable to give more information to Numba. This can be done using the function `jit`. The key input to `jit` is the description of the inputs and outputs. In the code below, `double[:]`

means 1-dimensional `float64` (`float` in Python, which corresponds to double precision in C), `double` indicates a scalar float and `int32` indicates a 32-bit integer. The string tells Numba to expect a 1-dimensional float, and that the inputs, in-order, are 3 1-dimensional floats followed by 2 32-bit integers and finally a scalar float.

```
>>> from numba import jit
>>> garch_recursion_numba_jit = jit('double[:](double[:],double[:],double[:],int32,int32,double)')(garch
```

Running the timing code, there is a small gain over the autojit version. The gain is possible since the autojit version allows virtually any compatible input while the jit versions will fail if the input types are correct.

```
>>> %timeit garch_recursion_numba_jit(parameters, data, sigma2, p, q, backcast)
10000 loops, best of 3: 64.6 us per loop

>>> 'The speed-up is {0:.1f} times'.format(0.0217/0.0000646 - 1.0)
'The speed-up is 334.9 times'
```

The pure Python dot product can also be easily converted to Numba using only the `@autojit` decorator.

```
@autojit
def pydot_autojit(a, b):
    M,N = shape(a)
    P,Q = shape(b)
    c = zeros((M,Q))
    for i in xrange(M):
        for j in xrange(Q):
            for k in xrange(N):
                c[i,j] = c[i,j] + a[i,k] * b[k,j]
    return c
```

Timing both the `autojit` and the `jit` versions produces large gains, although the performance of the two just-in-time versions is similar. The input declaration in `jit` uses the notation `double[:,::1]` which tells Numba to expect a 2-dimensional array using row-major ordering, which is the default in NumPy.

```
>>> %timeit -r 10 pydot_autojit(a,b)  # -r 10 uses 10 instead of 3
1000 loops, best of 10: 1.47 ms per loop

>>> 1.02/.00147 - 1
692.87755

>>> pydot_jit = pydot_jit=jit('double[:,::1](double[:,::1],double[:,::1])')(pydot)
>>> %timeit -r 10 pydot_jit(a,b)
1000 loops, best of 10: 1.42 ms per loop
```

23.12 Cython

Cython is a powerful, but somewhat complex, solution for situations where pure NumPy or Numba cannot achieve performance targets. Unless you are familiar with C, Cython should be considered a last resort. Cython translates Python code into C code, which can then be compiled into a Python extension. Cython code has three distinct advantages over Numba to just-in-time compilation of Python code:

- Cython modules are statically compiled and so using a Cython module does not incur a "warm-up" penalty due to just-in-time compilation.

- A Python extension produced by Cython can be distributed to other users and does not require Cython to be installed. In contrast, Numba must be installed and performance gains will typically vary across Numba or LLVM versions.

- Numba is a relatively new, rapidly evolving project – this may produce breaks in otherwise working code.

Using Cython on Linux is relatively painless, and only requires that the system compiler is installed in addition to Cython. To use Cython in Python x64, it is necessary to have the x64 version of Cython installed along with both the Windows 7 SDK and the .NET 3.5 SDK – it must be this SDK and not a newer SDK – which ships with the Microsoft Optimizing Compiler version 15 (the same compiler used to build Python 2.7.5).

The main idea behind Cython is to write standard Python and then to add some special syntax and hints about the type of data used. This first example will use the same GARCH(P,Q) code as in the Numba example. Applying Cython to an existing Python function requires a number of steps (for standard numeric code):

- Save the file with the extension pyx – for Python Extension.

- Use `cimport`, which is a special version of `import` for Cython, to import both `cython` and `numpy` `as` np.

- Declare types for *every* variable:

  – Scalars have standard C-types, and in almost all cases should be `double` (same as `float64` in NumPy, and `float` in Python), `int` (signed integer) `uint` (unsigned integer) or `size_t` (system unsigned integer type). `size_t` would typically only be used to counter variables in loops.

  – NumPy arrays `np.ndarray[` *type* `,ndim=` *numdims* `]` where *type* is a Cython NumPy type, and should almost always be `np.float64_t` for numeric data, and *numdims* is the number of dimensions of the NumPy array, likely to be 1 or 2.

- Declare all arrays as `not None`.

- Ensure that all array access uses only single item access and not more complex slicing. For example is `x` is a 2-dimensional array, `x[i,j]` must be used and not `x[i,:]` or `x[:,j]`.

The "Cythonized" version of the GARCH(P,QP recursion is presented below. All arrays are declared using `np.ndarray[np.float64_t, ndim=1]` and so the inputs must all have 1 dimension (and 1 dimension only). The inputs `p` and `q` are declared to be integers, and `backcast` is declared to be a `double`. The three local variables `T`, `i` and `j` are all declared to be `int`s. Note that is crucial that the variables used as iterators are declared as `int` (or other integer type, such as `uint` or `size_t`). The remainder of the function is *unchanged*.

```
import numpy as np
cimport numpy as np
cimport cython


@cython.boundscheck(False)
```

```python
@cython.wraparound(False)
def garch_recursion(np.ndarray[np.float64_t, ndim=1] parameters not None,
                    np.ndarray[np.float64_t, ndim=1] data not None,
                    np.ndarray[np.float64_t, ndim=1] sigma2 not None,
                    int p,
                    int q,
                    double backcast):

    cdef int T = np.size(data,0)
    cdef int i, j

    for i in xrange(T):
        sigma2[i] = parameters[0]
        for j in xrange(p):
            if (i-j)<0:
                sigma2[i] = parameters[1+j] * backcast
            else:
                sigma2[i] = parameters[1+j] * (data[i-j]*data[i-j])
        for j in xrange(q):
            if (i-j)<0:
                sigma2[i] = parameters[1+p+j] * backcast
            else:
                sigma2[i] = parameters[1+p+j] * sigma2[i-j]

    return sigma2
```

Two additional decorators were included in the Cython version of the function, `@cython.boundscheck(False)` and `@cython.wraparound(False)`. The first disables bounds checking which speeds up the final code, but is dangerous if the data used in the loop has fewer elements than expected. The second rules out the use of negative indices, which is simple to verify and enforce.

The next step is to code up a `setup.py` file which is used to build the actual extension. The code is located in a file named `garch_ext.pyx`, which will be the name of the extension. The setup code is standard, and is unlikely to require altering (aside from the extension and file name).

```python
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext
import numpy

setup(
    cmdclass = {'build_ext': build_ext},
    ext_modules = [Extension("garch_ext", ["garch_ext.pyx"])],
    include_dirs = [numpy.get_include()]
)
```

The final step is to build the extension by running `python setup.py build_ext --inplace` from the terminal. This will produce `garch_ext.pyd` which contains the compiled code.

```python
>>> parameters = array([.1,.1,.8])
>>> data = randn(10000)
>>> sigma2 = zeros(shape(data))
```

```
>>> p,q = 1,1
>>> backcast = 1.0
>>> %timeit garch_recursion(parameters, data, sigma2, p, q, backcast)
10 loops, best of 3: 21.7 ms per loop

>>> import garch_ext
>>> %timeit garch_ext.garch_recursion(parameters, data, sigma2, p, q, backcast)
10000 loops, best of 3: 111 us per loop

>>> 'The speed-up is {0:.1f} times'.format(0.0217/0.000111 - 1.00)
'The speed-up is 194.5 times'
```

The Cythonized version is about 200 times faster than the standard Python version, and only required about 3 minutes to write (after the main Python function has been written). However, it is slower than the Numba version of the same function.

The function pydot was similarly Cythonized. This Cython program demonstrates how arrays should be allocated within the function. Note that the Cython type for an array is np.float64_t which corresponds to the usual NumPy data type of np.float64 (other _t types are available for different NumPy data types). Again, it only required a couple of minutes to Cythonize the original Python function.

```
import numpy as np
cimport numpy as np
cimport cython

@cython.boundscheck(False)
@cython.wraparound(False)
def pydot(np.ndarray[np.float64_t, ndim=2] a not None,
          np.ndarray[np.float64_t, ndim=2] b not None):
    cdef int M, N, P, Q
    M,N = np.shape(a)
    P,Q = np.shape(b)
    assert N==P
    cdef np.ndarray[np.float64_t, ndim=2] c = np.zeros((M,N), dtype=np.float64)
    for i in xrange(M):
        for j in xrange(Q):
            for k in xrange(N):
                c[i,j] = c[i,j] + a[i,k] * b[k,j]
    return c
```

The Cythonized function is about 350 times faster than straight Python, although it is still much slower than the native NumPy routine dot.

```
>>> a = randn(100,100)
>>> b = randn(100,100)
>>> %timeit pydot(a,b)
3 loops, best of 3: 1.02 s per loop

>>> import pydot as p
>>> %timeit -r 10 p.pydot(a,b)
100 loops, best of 10: 2.75 ms per loop
```

```
>>> 1.02/.00275 - 1.0
369.909090

>>> %timeit -r 10 dot(a,b)
10000 loops, best of 3: 160 us per loop

>>> .00275/0.0000956
28.76569
```

The final example will produce a Cython version of the weighted average. Since the original Python code used slicing, this is removed and replaced with a second loop.

```python
def super_slow_weighted_avg(x, w):
    T = x.shape[0]
    m = len(w)
    m12 = int(ceil(m/2))
    y = zeros(T)
    for i in xrange(len(x)-m+1):
        for j in xrange(m):
            y[i+m12] += x[i+j] * w[j]

    return y
```

This makes writing the Cython version simple.

```python
import numpy as np
cimport numpy as np
cimport cython

@cython.boundscheck(False)
@cython.wraparound(False)
def cython_weighted_avg(np.ndarray[np.float64_t, ndim=1] x,
                        np.ndarray[np.float64_t, ndim=1] w):
    cdef int T, m, m12, i, j
    T = x.shape[0]
    m = len(w)
    m12 = int(np.ceil(float(m)/2))
    cdef np.ndarray[np.float64_t, ndim=1] y = np.zeros(T, dtype=np.float64)
    for i in xrange(T-m+1):
        for j in xrange(m):
            y[i+m12] += x[i+j] * w[j]

    return y
```

The Cython version can be compiled using a setup function in the same way that the GARCH recursion was compiled.

```
>>> w = array(r_[1:11,9:0:-1],dtype=float64)
>>> w = w/sum(w)
>>> x = randn(10000)
>>> %timeit super_slow_weighted_avg(x,w)
1 loops, best of 3: 1.31 s per loop
```

```
>>> import cython_weighted_avg as c
>>> %timeit c.cython_weighted_avg(x,w)
1000 loops, best of 3:  567 us per loop
```

The gains are unsurprisingly large (around 300×) – however, the Cython code is no faster than the pure NumPy sideways version. This demonstrates that Cython is not a magic bullet and that good vectorized code, even with a small amount of looping, can be very fast.

23.13 Exercises

1. Write a Python function which will accept a $p + q + 1$ vector of parameters, a T vector of data, and p and q (integers, AR and MA order, respectively) and recursively computes the ARMA error beginning with observation $p + 1$. If an MA index is negative it should be backcast to 0.

2. Use line_profiler to measure the performance of the ARMA written in exercise 1.

3. Use `autojit` and `jit` to accelerate the ARMA function written in the exercise 1. Compare the speed to the pure Python implementation.

4. [Only for the brave] Convert the ARMA function to Cython, compile it, and compare the performance against both the pure Python and the Numba versions.

Chapter 24

Parallel

24.1 map and related functions

`map` is a built-in function which is used to apply a function to a generic iterable. It is used as `map(` *function* `,` *iterable* `)`, and returns a list containing the results of applying the function to each item of *iterable*. Note that the list returned can be either a simple list if the function returns a single item, or a list of tuples if the function returns more than 1 value.

```
def powers(x):
    return x**2, x**3, x**4
```

This function can be called on any iterable, for example a list.

```
>>> y = [1.0, 2.0, 3.0, 4.0]
>>> map(powers, y)
[(1.0, 1.0, 1.0), (4.0, 8.0, 16.0), (9.0, 27.0, 81.0), (16.0, 64.0, 256.0)]
```

The output is a list of tuples where each tuple contains the result of calling the function on a single input. Note that the same result could be achieved using a list comprehension. In general usage, list comprehensions are preferable to using map.

```
>>> [powers(i) for i in y]
[(1.0, 1.0, 1.0), (4.0, 8.0, 16.0), (9.0, 27.0, 81.0), (16.0, 64.0, 256.0)]
```

map can be used with more than 1 iterable, in which case it iterates along the longest iterable. If one of the iterable is shorter than the other(s), then it is extended with `None`. It is usually best practice to ensure that all iterables have the same length before using `map`.

```
def powers(x,y):
    if x is None or y is None:
        return None
    else:
        return x**2, x*y, y**2
```

```
>>> x = [10.0, 20.0, 30.0]
>>> y = [1.0, 2.0, 3.0, 4.0]
>>> map(powers, x, y)
[(100.0, 10.0, 1.0), (400.0, 40.0, 4.0), (900.0, 90.0, 9.0), None]
```

A related function is `zip`. While `zip` does not apply a function to data, it can be used to combine two or more lists into a single list of tuples. It is similar to calling `map` except that it will stop at the end of the shortest iterable, rather than extending using None.

```
>>> x = [10.0, 20.0, 30.0]
>>> y = [1.0, 2.0, 3.0, 4.0]
>>> zip(x, y)
[(10.0, 1.0), (20.0, 2.0), (30.0, 3.0)]
```

24.2 Multiprocess module

The real advantage of `map` over list comprehensions is that it can be combined with the `multiprocess` module to run code on more than 1 (local) processor. `multiprocess` *module does not work correctly in IPython, and so it is necessary to use stand-alone Python programs.* `multiprocess` includes a `map` function which is similar to that in the standard Python distribution except that it executes using a `Pool` rather than on a single processor. The gains to using a `Pool` may be large, and should be close to the number of pool processes if completely independent (which should be less than or equal to the number of physical processors on a system).

This example uses `multiprocess` to compute eigenvalues for some random matrices and is illustrative of a Monte Carlo-like setup. The program has the standard set of imports including the `multiprocess` module.

```
from __future__ import print_function
import multiprocessing as mp
import numpy as np
import matplotlib.pyplot as plt
```

Next, a simple function is defined to compute eigenvalues. While `map` requires both a function and an iterable, the function can be any function located in any module and so does not need to reside in the same file as the main code.

```
def compute_eig(arg):
    n = arg[0]
    state = arg[1]
    print(arg[2])
    np.random.set_state(state)
    x = np.random.standard_normal((n))
    m = int(np.round(np.sqrt(n)))
    x.shape=m,m
    w = np.linalg.eigvalsh(np.dot(x.T,x)/m)
    return w
```

Using `multiprocess` requires a `__name__=='main'` block in the function. The main block does three things:

1. Compute states to use in the simulation. This is done so that the state can be given to the function executed in parallel.

2. Initialize the pool using `mp.Pool(processes=2)`

3. Call `map` from the `multiprocess` module

286

4. Plot the results.

```python
if __name__ == '__main__':
    states = []
    np.random.seed()
    for i in xrange(1000):
        n = 1000000
        states.append((n,np.random.get_state(),i))
        temp = np.random.standard_normal((n))

    # Non parallel map
    # res = map(compute_eig,states)

    # Parallem map
    po = mp.Pool(processes=2)
    res = po.map(compute_eig,states)
    print(len(res))
    po.close()

    ax = plt.hist(maxEig)
    ax = ax[2]
    fig = ax[0].get_figure()
    fig.savefig('multiprocess.pdf')
```

24.3   IPython Parallel

IPython contains a sophisticated parallel framework which allows for interactive parallel execution both locally and across a network (e.g. a supercomputer or using a cloud provider such as Amazon Web Services). IPython's parallelization framework provides both mechanisms similar to map in the previous example as well as more sophisticated schedulers that perform load balancing, which is useful is some processing may complete faster. Coverage of IPython's parallelization framework is beyond the score of these notes, although users who are interested in large scale problems in Python should be aware of this feature.

Chapter 25

Examples

These examples are all actual econometric problems chosen to demonstrate the use of Python in an end-to-end manner, from importing data to presenting estimates. A reasonable familiarity with the underlying econometric models and methods is assumed and this chapter focuses on translating the mathematics to Python.

25.1 Estimating the Parameters of a GARCH Model

This example will highlight the steps needed to estimate the parameters of a GJR-GARCH(1,1,1) model with a constant mean. The volatility dynamics in a GJR-GARCH model are given by

$$\sigma_t^2 = \omega + \sum_{i=1}^{p} \alpha_i \epsilon_{t-i}^2 + \sum_{j=1}^{o} \gamma_j \epsilon_{t-j}^2 I_{[\epsilon_{t-j}<0]} + \sum_{k=1}^{q} \beta_k \sigma_{t-k}^2.$$

Returns are assumed to be conditionally normal, $r_t|\mathcal{F}_{t-1} \sim N\left(\mu, \sigma_t^2\right)$, $\epsilon_t = r_t - \mu$ and parameters are estimated by maximum likelihood. To estimate the parameters, it is necessary to:

1. Produce some starting values

2. Estimate the parameters using (quasi-) maximum likelihood

3. Compute standard errors using a "sandwich" covariance estimator (also known as the Bollerslev & Wooldridge (1992) covariance estimator)

The first task is to write the log-likelihood which can be used in an optimizer. The log-likelihood function will compute the volatility recursion and the log-likelihood. It will also, optionally, return the T by 1 vector of individual log-likelihoods which are useful when approximating the scores.

The first step is to import the required modules. I prefer to use the import form `from` *module* `import` *func1*, *func2* for commonly used functions. This both saves typing and increases code readability. I use `import` *module* `as` *shortname* for functions which are used less. This is a personal choice, and any combination is acceptable, although `from` *module* `import` `*` should be avoided.

```
from __future__ import print_function, division
import numpy as np
import matplotlib.pyplot as plt
```

```python
from numpy import size, log, pi, sum, diff, array, zeros, diag, dot, mat, asarray, sqrt,
    copy
from numpy.linalg import inv
from pandas import read_csv
from scipy.optimize import fmin_slsqp
```

The conditional log-likelihood of a normal random variable is

$$\ln f\left(r_t | \mu, \sigma_t^2\right) = -\frac{1}{2}\left(\ln 2\pi + \ln \sigma_t^2 + \frac{(r_t - \mu)^2}{\sigma_t^2}\right),$$

which is negated in the code since the optimizers all minimize.

```python
def gjr_garch_likelihood(parameters, data, sigma2, out=None):
    ''' Returns negative log-likelihood for GJR-GARCH(1,1,1) model.'''
    mu = parameters[0]
    omega = parameters[1]
    alpha = parameters[2]
    gamma = parameters[3]
    beta = parameters[4]

    T = size(data,0)
    eps = data - mu
    # Data and sigma2 are T by 1 vectors
    for t in xrange(1,T):
        sigma2[t] = (omega + alpha * eps[t-1]**2
                     + gamma * eps[t-1]**2 * (eps[t-1]<0) + beta * sigma2[t-1])

    logliks = 0.5*(log(2*pi) + log(sigma2) + eps**2/sigma2)
    loglik = sum(logliks)

    if out is None:
        return loglik
    else:
        return loglik, logliks, copy(sigma2)
```

The keyword argument `out` has a default value of `None`, and is used to determine whether to return 1 output or 3. This is common practice since the optimizer requires a single output – the log-likelihood function value, but it is also useful to be able to output other useful quantities, such as $\left\{\sigma_t^2\right\}$.

The optimization is constrained so that $\alpha + \gamma/2 + \beta \leq 1$, and the constraint is provided in a separate function.

```python
def gjr_constraint(parameters, data, sigma2, out=None):
    ''' Constraint that alpha+gamma/2+beta<=1'''

    alpha = parameters[2]
    gamma = parameters[3]
    beta = parameters[4]

    return array([1-alpha-gamma/2-beta])
```

Note that the constraint function takes the same inputs as the negative of the log-likelihood function, even though only `parameters` is required to compute the constraint.

It is necessary to discuss one other function before proceeding with the main block of code. The asymptotic variance is estimated using the "sandwich" form which is commonly expressed as

$$\mathcal{J}^{-1}\mathcal{I}\mathcal{J}^{-1}$$

where $\mathcal{J}$ is the expected Hessian and $\mathcal{I}$ is the covariance of the scores. Both are numerically approximated, and the strategy for computing the Hessian is to use the definition that

$$\mathcal{J}_{ij} \approx \frac{f\left(\theta + e_i h_i + e_j h_j\right) - f\left(\theta + e_i h_i\right) - f\left(\theta + e_j h_j\right) + f\left(\theta\right)}{h_i h_j}$$

where h_i is a scalar "step size" and e_i is a vector of 0s except for element i, which is 1. A 2-sided version of this approximation, which takes both forward and backward steps and then averages, is below. For more on numerical derivatives, see Flannery et al. (1992).

```python
def hessian_2sided(fun, theta, args):
    f = fun(theta, *args)
    h = 1e-5*np.abs(theta)
    thetah = theta + h
    h = thetah - theta
    K = size(theta,0)
    h = np.diag(h)

    fp = zeros(K)
    fm = zeros(K)
    for i in xrange(K):
        fp[i] = fun(theta+h[i], *args)
        fm[i] = fun(theta-h[i], *args)

    fpp = zeros((K,K))
    fmm = zeros((K,K))
    for i in xrange(K):
        for j in xrange(i,K):
            fpp[i,j] = fun(theta + h[i] + h[j],  *args)
            fpp[j,i] = fpp[i,j]
            fmm[i,j] = fun(theta - h[i] - h[j],  *args)
            fmm[j,i] = fmm[i,j]

    hh = (diag(h))
    hh = hh.reshape((K,1))
    hh = dot(hh,hh.T)

    H = zeros((K,K))
    for i in xrange(K):
        for j in xrange(i,K):
            H[i,j] = (fpp[i,j] - fp[i] - fp[j] + f
                    + f - fm[i] - fm[j] + fmm[i,j])/hh[i,j]/2
```

```
            H[j,i] = H[i,j]

    return H
```

Finally, the code that does the actual work can be written. The first block imports the data, flips it using a slicing operator, and computes 100 times returns. Scaling data can be useful to improve optimizer performance, and ideally estimated parameters should have similar magnitudes (i.e. $\omega \approx .01$ and $\alpha \approx .05$).

```
# Import data
FTSEdata = read_csv('FTSE_1984_2012.csv', parse_dates=[0])
# Flip upside down
FTSEdata = FTSEdata[::-1]
# Compute returns
FTSEprice = FTSEdata['Adj Close']
FTSEreturn = 100*diff(log(FTSEprice).values)
```

Good starting values are important. These are my guesses based on experience fitting these types of models models. An alternative is to attempt a crude grid search and use the best (smallest) log-likelihood value from the grid search.

```
# Starting values
startingVals = array([FTSEreturn.mean(),
                      FTSEreturn.var() * .01,
                      .03, .09, .90])
```

Bounds are used in estimation to ensure that all parameters in the conditional variance are ≥ 0 and to set sensible upper bounds on the mean and ω. The vector `sigma2` is then initialized, and the arguments are placed in a tuple.

```
# Estimate parameters
finfo = np.finfo(np.float64)
bounds = [(-10*FTSEreturn.mean(), 10*FTSEreturn.mean()),
          (finfo.eps, 2*FTSEreturn.var() ),
          (0.0,1.0), (0.0,1.0), (0.0,1.0)]

T = size(FTSEreturn,0)
sigma2 = np.repeat(FTSEreturn.var(),T)
args = (FTSEreturn, sigma2)
estimates = fmin_slsqp(gjr_garch_likelihood, startingVals, \
            f_ieqcons=gjr_constraint, bounds = bounds, \
            args = args)
```

The optimized log-likelihood and the time series of variances are computed by calling the objective using the keyword argument `out=True`.

```
loglik, logliks, sigma2final = gjr_garch_likelihood(estimates, \
                               FTSEreturn, sigma2, out=True)
```

Next, the numerical scores and the covariance of the scores are computed. These exploit the definition of a derivative, so that for a scalar function,

$$\frac{\partial f(\theta)}{\partial \theta_i} \approx \frac{f(\theta + e_i h_i) - f(\theta)}{h_i}.$$

292

The covariance is computed as the outer product of the scores since the scores should have mean 0 when evaluated at the solution to the optimization problem.

```
step = 1e-5 * estimates
scores = np.zeros((T,5))
for i in xrange(5):
    h = step[i]
    delta = np.zeros(5)
    delta[i] = h

    loglik, logliksplus, sigma2 = gjr_garch_likelihood(estimates + delta, \
                              FTSEreturn, sigma2, out=True)
    loglik, logliksminus, sigma2 = gjr_garch_likelihood(estimates - delta, \
                              FTSEreturn, sigma2, out=True)
    scores[:,i] = (logliksplus - logliksminus)/(2*h)

I = np.dot(scores.T,scores)/T
```

The next block calls `hessian_2sided` to estimate the Hessian, and then computes the asymptotic covariance.

```
J = hessian_2sided(gjr_garch_likelihood, estimates, args)
J = J/T
Jinv = mat(inv(J))
vcv = Jinv*mat(I)*Jinv/T
vcv = asarray(vcv)
```

The penultimate step is to pretty print the results and to produce a plot of the conditional variances.

```
output = np.vstack((estimates,sqrt(diag(vcv)),estimates/sqrt(diag(vcv)))).T
print('Parameter   Estimate       Std. Err.      T-stat')
param = ['mu','omega','alpha','gamma','beta']
for i in xrange(len(param)):
    print('{0:<11} {1:>0.6f}        {2:0.6f}     {3: 0.5f}'.format(param[i],output[i,0],output[i,1],outpu
```

This final block produces a plot of the annualized conditional standard deviations.

```
# Produce a plot
dates = FTSEdata.Date[1:]
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(dates,np.sqrt(252*sigma2))
fig.autofmt_xdate()
ax.set_ylabel('Volatility')
ax.set_title('Annualized FTSE Volatility (GJR GARCH(1,1,1))')
plt.show()
```

25.2 Estimating the Risk Premia using Fama-MacBeth Regressions

This example highlights how to implement a Fama-MacBeth 2-stage regression to estimate factor risk premia, make inference on the risk premia, and test whether a linear factor model can explain a cross-

section of portfolio returns. This example closely follows Cochrane (2001) (See also Jagannathan et al. (2010)). As in the previous example, the first segment contains the imports.

```
from __future__ import print_function, division
from numpy import mat, cov, mean, hstack, multiply,sqrt,diag, genfromtxt, \
    squeeze, ones, array, vstack, kron, zeros, eye, savez_compressed
from numpy.linalg import lstsq, inv
from scipy.stats import chi2
from pandas import read_csv
```

Next, the data are imported. I formatted the data downloaded from Ken French's website into an easy-to-import CSV which can be read by `pandas.read_csv`. The data is split using named columns for the small sets of variables and `ix` for the portfolios. The code uses pure NumPy arrays, and so `values` is used to retrieve the array from the DataFrame. The dimensions are determined using `shape`. Finally the risk free rate is forced to have 2 dimensions so that it will be broadcastable with the portfolio returns in the construction of the excess returns to the Size and Value-weighted portfolios. `mat` is used to return matrix views of all of the arrays. This code is linear algebra-heavy and so matrices are easier to use than arrays.

```
data = read_csv('FamaFrench.csv')

# Split using both named colums and ix for larger blocks
dates = data['date'].values
factors = data[['VWMe', 'SMB', 'HML']].values
riskfree = data['RF'].values
portfolios = data.ix[:, 5:].values

# Use mat for easier linear algebra
factors = mat(factors)
riskfree = mat(riskfree)
portfolios = mat(portfolios)

# Shape information
T,K = factors.shape
T,N = portfolios.shape
# Reshape rf and compute excess returns
riskfree.shape = T,1
excessReturns = portfolios - riskfree
```

The next block does 2 things:

1. Compute the time-series βs. This is done be regressing the full array of excess returns on the factors (augmented with a constant) using `lstsq`.

2. Compute the risk premia using a cross-sectional regression of average excess returns on the estimates βs. This is a standard regression where the step 1 β estimates are used as regressors, and the dependent variable is the average excess return.

```
# Time series regressions
X = hstack((ones((T, 1)), factors))
out = lstsq(X, excessReturns)
alpha = out[0][0]
```

```
beta = out[0][1:]
avgExcessReturns = mean(excessReturns, 0)
# Cross-section regression
out = lstsq(beta.T, avgExcessReturns.T)
riskPremia = out[0]
```

The asymptotic variance requires computing the covariance of the demeaned returns and the weighted pricing errors. The problem is formulated using 2-step GMM where the moment conditions are

$$
g_t(\theta) = \begin{bmatrix}
\epsilon_{1t} \\
\epsilon_{1t} f_t \\
\epsilon_{2t} \\
\epsilon_{2t} f_t \\
\vdots \\
\epsilon_{Nt} \\
\epsilon_{Nt} f_t \\
\beta u_t
\end{bmatrix}
$$

where $\epsilon_{it} = r_{it}^e - \alpha_i - \beta_i' f_t$, β_i is a K by 1 vector of factor loadings, f_t is a K by 1 set of factors, $\beta = [\beta_1 \beta_2 \ldots \beta_N]$ is a K by N matrix of all factor loadings, $u_t = r_t^e - \beta' \lambda$ are the N by 1 vector of pricing errors and λ is a K by 1 vector of risk premia. The collection of parameters is then $\theta = [\alpha_1 \ \beta_1' \ \alpha_2 \ \beta_2' \ \ldots \ \alpha_N \ \beta_N' \ \lambda']'$. To make inference in this problem, the derivative of the moments with respect to the parameters, $\partial g_t(\theta)/\partial \theta'$ is needed. With some work, the estimator of this matrix can be seen to be

$$
G = E\left[\frac{\partial g_t(\theta)}{\partial \theta'}\right] = \begin{bmatrix}
-I_n \otimes \Sigma_X & 0 \\
G_{21} & -\beta\beta'
\end{bmatrix}.
$$

where $X_t = [1 \ f_t']'$ and $\Sigma_X = E[X_t X_t']$. G_{21} is a matrix with the structure

$$
G_{21} = [G_{21,1} \ G_{21,2} \ \ldots G_{21,N}]
$$

where

$$
G_{21,i} = \begin{bmatrix} 0_{K,1} & \mathrm{diag}(E[u_i] - \beta_i \odot \lambda) \end{bmatrix}
$$

and where $E[u_i]$ is the expected pricing error. In estimation, all expectations are replaced with their sample analogues.

```
# Moment conditions
X = hstack((ones((T, 1)), factors))
p = vstack((alpha, beta))
epsilon = excessReturns - X * p
moments1 = kron(epsilon, ones((1, K + 1)))
moments1 = multiply(moments1, kron(ones((1, N)), X))
u = excessReturns - riskPremia.T * beta
moments2 = u * beta.T
# Score covariance
S = mat(cov(hstack((moments1, moments2)).T))
# Jacobian
```

```
G = mat(zeros((N * K + N + K, N * K + N + K)))
SigmaX = X.T * X / T
G[:N * K + N, :N * K + N] = kron(eye(N), SigmaX)
G[N * K + N:, N * K + N:] = -beta * beta.T
for i in xrange(N):
    temp = zeros((K, K + 1))
    values = mean(u[:, i]) - multiply(beta[:, i], riskPremia)
    temp[:, 1:] = diag(values.A1)
    G[N * K + N:, i * (K + 1):(i + 1) * (K + 1)] = temp

vcv = inv(G.T) * S * inv(G) / T
```

The J-test examines whether the average pricing errors, $\hat{\alpha}$, are zero. The J statistic has an asymptotic χ^2_N distribution, and the model is badly rejected.

```
vcvAlpha = vcv[0:N * K + N:4, 0:N * K + N:4]
J = alpha * inv(vcvAlpha) * alpha.T
J = J[0, 0]
Jpval = 1 - chi2(25).cdf(J)
```

The final block using formatted output to present all of the results in a readable manner.

```
vcvRiskPremia = vcv[N * K + N:, N * K + N:]
annualizedRP = 12 * riskPremia
arp = list(squeeze(annualizedRP.A))
arpSE = list(sqrt(12 * diag(vcvRiskPremia)))
print('        Annualized Risk Premia')
print('           Market       SMB        HML')
print('--------------------------------------')
print('Premia     {0:0.4f}     {1:0.4f}     {2:0.4f}'.format(arp[0], arp[1], arp[2]))
print('Std. Err.  {0:0.4f}     {1:0.4f}     {2:0.4f}'.format(arpSE[0], arpSE[1], arpSE[2]))
print('\n\n')

print('J-test:   {:0.4f}'.format(J))
print('P-value:  {:0.4f}'.format(Jpval))

i = 0
betaSE = []
for j in xrange(5):
    for k in xrange(5):
        a = alpha[0, i]
        b = beta[:, i].A1
        variances = diag(vcv[(K + 1) * i:(K + 1) * (i + 1), (K + 1) * i:(K + 1) * (i + 1)])
        betaSE.append(sqrt(variances))
        s = sqrt(variances)
        c = hstack((a, b))
        t = c / s
        print('Size: {:}, Value:{:}   Alpha   Beta(VWM)   Beta(SMB)   Beta(HML)'.format(j + 1, k + 1))
        print('Coefficients: {:>10,.4f}  {:>10,.4f}  {:>10,.4f}  {:>10,.4f}'.format(a, b[0], b[1], b[2])
        print('Std Err.      {:>10,.4f}  {:>10,.4f}  {:>10,.4f}  {:>10,.4f}'.format(s[0], s[1], s[2], s[
        print('T-stat        {:>10,.4f}  {:>10,.4f}  {:>10,.4f}  {:>10,.4f}'.format(t[0], t[1], t[2], t[
        print('')
```

```
        i += 1
```

The final block converts the standard errors of β to be an array and saves the results.

```
betaSE = array(betaSE)
savez_compressed('Fama-MacBeth_results', alpha=alpha, \
                 beta=beta, betaSE=betaSE, arpSE=arpSE, arp=arp, J=J, Jpval=Jpval)
```

25.3 Estimating the Risk Premia using GMM

The final numeric example estimates examines the same problem, only using GMM rather than 2-stage regression. The first block imports relevant modules and functions. GMM requires non-linear optimization, and `fmin_bfgs` will be used.

```
from __future__ import print_function, division
from numpy import hstack, ones, array, mat, tile, dot, reshape, squeeze, eye, asmatrix
from numpy.linalg import lstsq, inv
from pandas import read_csv
from scipy.linalg import kron
from scipy.optimize import fmin_bfgs
import numpy as np
```

Before defining the problem, the next code block defines a *callback* function that can be called after each iteration of the minimizer. This function will be used to display information about the progress of the optimizer: the current function value, the iteration and the number of function calls. Because the callback only gets 1 input, the current value of the parameters used by the optimizer, it is necessary to use `global` variables to pass information between functions. Three variables have been declared: `iteration`, `lastValue` and `functionCount`. `iteration` is updated only by the callback function since it should report the number of completed iterations of the optimizer, and the callback is called once per iteration. `lastValue` is updated each time the main GMM objective function is called, and `functionCount` is incremented by 1 each time the main GMM objective is called. The variables are all initialized using `global` inside the callback so that their values can be updated.

```
iteration = 0
lastValue = 0
functionCount = 0

def iter_print(params):
    global iteration, lastValue, functionCount
    iteration += 1
    print('Func value: {0:}, Iteration: {1:}, Function Count: {2:}'.format(lastValue, iteration, functio
```

The GMM objective takes the parameters, portfolio returns, factor returns and the weighting matrix and computes the moments, average moments and the objective value. The moments used can be described as

$$\left(r_{it}^2 - \beta_i f_t \right) f_t \quad \forall i = 1, \ldots N$$

and

$$r_{it} - \beta_i \lambda \quad \forall i = 1, \ldots N.$$

```
def gmm_objective(params, pRets, fRets, Winv, out=False):
    global lastValue, functionCount
    T,N = pRets.shape
    T,K = fRets.shape
    beta = squeeze(array(params[:(N*K)]))
    lam = squeeze(array(params[(N*K):]))
    beta = reshape(beta,(N,K))
    lam = reshape(lam,(K,1))
    betalam = dot(beta,lam)
    expectedRet = dot(fRets,beta.T)
    e = pRets - expectedRet
    instr = tile(fRets,N)
    moments1  = kron(e,ones((1,K)))
    moments1 = moments1 * instr
    moments2 = pRets - betalam.T
    moments = hstack((moments1,moments2))

    avgMoment = moments.mean(axis=0)

    J = T * mat(avgMoment)*mat(Winv)*mat(avgMoment).T
    J = J[0,0]
    lastValue = J
    functionCount += 1
    if not out:
        return J
    else:
        return J, moments
```

The final function needed is the Jacobian of the moment conditions. Mathematically it is simply to express the Jacobian using ⊗ (Kronecker product). This code is so literal that it is simple to reverse engineer the mathematical formulas used to implement this estimator.

$$\hat{G} = \left[\begin{array}{cc} I_N \otimes \Sigma_F & 0 \\ I_N \otimes -\lambda & -\beta \end{array} \right]$$

```
def gmm_G(params, pRets, fRets):
    T,N = pRets.shape
    T,K = fRets.shape
    beta = squeeze(array(params[:(N*K)]))
    lam = squeeze(array(params[(N*K):]))
    beta = reshape(beta,(N,K))
    lam = reshape(lam,(K,1))
    G = np.zeros((N*K+K,N*K+N))
    ffp = dot(fRets.T,fRets)/T
    G[:(N*K),:(N*K)]=kron(eye(N),ffp)
    G[:(N*K),(N*K):] = kron(eye(N),-lam)
    G[(N*K):,(N*K):] = -beta.T

    return G
```

The data import step is virtually identical to that in the previous example. Note that only every other portfolio is used to speed up the GMM optimization.

```
data = read_csv('FamaFrench.csv')

# Split using both named colums and ix for larger blocks
dates = data['date'].values
factors = data[['VWMe','SMB','HML']].values
riskfree = data['RF'].values
portfolios = data.ix[:,5:].values

T,N = portfolios.shape
portfolios = portfolios[:,np.arange(0,N,2)]
T,N = portfolios.shape
excessRet = portfolios - np.reshape(riskfree,(T,1))
K = np.size(factors,1)
```

Starting values are important in any optimization problem. The GMM problem is closely related to Fama-MacBeth regression, and so it is sensible to use the output from a FMB regression.

```
betas = []
augFactors = hstack((ones((T,1)),factors))
for i in xrange(N):
    out = lstsq(augFactors,excessRet[:,i])
    betas.append(out[0][1:])

avgReturn = excessRet.mean(axis=0)
avgReturn.shape = N,1
betas = array(betas)
out = lstsq(betas,avgReturn)
riskPremia = out[0]
```

The GMM objective can be minimized using an identity matrix as covariance of the moment conditions along with the starting values computed using a Fama-MacBeth regression. The keyword argument `callback` is used to pass the callback function to the optimizer.

```
riskPremia.shape = 3
startingVals = np.concatenate((betas.flatten(),riskPremia))

Winv = np.eye(N*(K+1))
args = (excessRet, factors, Winv)
iteration = 0
functionCount = 0
step1opt = fmin_bfgs(gmm_objective, startingVals, args=args, callback=iter_print)
```

Once the initial estimates have been computed, these can be used to estimate the covariance of the moment conditions, which is then used to estimate the optimal weighting matrix. The keyword argument `out` is used to return the moments in addition to the objective function value. Note that the variables `iteration` and `lastValue` which are used in the callback are both reset to 0 so that the count will be accurate.

```
out = gmm_objective(step1opt, excessRet, factors, Winv, out=True)
```

```
S = np.cov(out[1].T)
Winv2 = inv(S)
args = (excessRet, factors, Winv2)

iteration = 0
functionCount = 0
step2opt = fmin_bfgs(gmm_objective, step1opt, args=args, callback=iter_print)
```

The final block computes estimates the asymptotic covariance of the parameters using the usual efficient GMM covariance estimator, assuming that the moments are a martingale.

```
out = gmm_objective(step2opt, excessRet, factors, Winv2, out=True)
G = gmm_G(step2opt, excessRet, factors)
S = mat(np.cov(out[1].T))
vcv = inv(G*inv(S)*G.T)/T
```

25.4   Outputting LaTeX

Automatically outputting results to LaTeX or another format can eliminate export errors and avoid tedious work. This example show how two of the tables in the previous Fama-MacBeth example can be exported to a complete LaTeX document, and how, if desired, the document can be compiled to a PDF. The first code block contains the imports needed and defines a flag which determines whether the output LaTeX should be compiled.

```
# imports
from __future__ import print_function
import numpy as np
import subprocess

# Flag to compile output tables
compileLatex = True
```

The next code block loads the npz file created using the output from the Fama-MacBeth example. The loaded data is in a dictionary, and so iterating over the keys and using exec restores the variables with the same name in the main program.

```
# Load variables
f = np.load('Fama-MacBeth_results.npz')
data = f.items()
# Restore the data
for key in f.keys():
    exec(key + " = f['" + key + "']")
f.close()
```

The document will be stored in a list. The first few lines contain the required header for a LaTeX document, including some packages used to improve table display and to select a custom font. All strings are raw r'' so that \ does not have to be escaped (i.e. r'\' rather than '\\' in normal strings).

```
# List to hold table
latex = []
# Initializd LaTeX document
```

```
latex.append(r'\documentclass[a4paper]{article}')
latex.append(r'\usepackage{amsmath}')
latex.append(r'\usepackage{booktabs}')
latex.append(r'\usepackage[adobe-utopia]{mathdesign}')
latex.append(r'\usepackage[T1]{fontenc}')
latex.append(r'\begin{document}')
```

Table 1 will be stored in its own list, and then `extend` will be used to add it to the main list. Building this table is simple string manipulation and use of `format`.

```
# Table 1
table1 = []
table1.append(r'\begin{center}')
table1.append(r'\begin{tabular}{lrrr} \toprule')
# Header
colNames = [r'VWM$^e$','SMB','HML']
header = ''
for cName in colNames:
    header += ' & ' + cName

header += r'\\ \cmidrule{2-4}'
table1.append(header)
# Main row
row = ''
for a,se in zip(arp,arpSE):
    row += r' & $\underset{{({0:0.3f})}}{{{1:0.3f}}}$'.format(se,a)
table1.append(row)
# Blank row
row = r'\\'
table1.append(row)
# J-stat row
row = r'J-stat: $\underset{{({0:0.3f})}}{{{1:0.1f}}}$ \\'.format(float(Jpval),float(J))
table1.append(row)
table1.append(r'\bottomrule \end{tabular}')
table1.append(r'\end{center}')
# Extend latex with table 1
latex.extend(table1)
latex.append(r'\newpage')
```

Table 2 is a bit more complex, and uses loops to iterate over the rows of the arrays containing the βs and their standard errors.

```
# Format information for table 2
sizes = ['S','2','3','4','B']
values = ['L','2','3','4','H']
# Table 2 has the same header as table 1, copy with a slice
table2 = table1[:3]
m = 0
for i in xrange(len(sizes)):
    for j in xrange(len(values)):
        row = 'Size: {:}, Value: {:} '.format(sizes[i],values[j])
        b = beta[:,m]
```

```
        s = betaSE[m,1:]
        for k in xrange(len(b)):
            row += r' & $\underset{{({0:0.3f})}}{{{1: .3f}}}$'.format(s[k],b[k])
        row += r'\\ '
        table2.append(row)
        m += 1
    if i<(len(sizes)-1):
        table2.append(r'\cmidrule{2-4}')

table2.append(r'\bottomrule \end{tabular}')
table2.append(r'\end{center}')
# Extend with table 2
latex.extend(table2)
```

The penultimate block finished the document, and uses write to write the lines to the LaTeX file. write does not automatically insert the newline character, and so \n is added to each line.

```
# Finish document
latex.append(r'\end{document}')
# Write to table
fid = file('latex.tex','w')
for line in latex:
    fid.write(line + '\n')
fid.close()
```

Finally, if the flag is set, subprocess is used to compile the LaTeX. This assumes that pdflatex is on the system path.

```
# Compile if needed
if compileLatex:
    exitStatus = subprocess.call(r'pdflatex latex.tex', shell=True)
```

Chapter 26

Other Interesting Python Packages

To be completed

26.1 statsmodels

statsmodels provides a large range of cross-sectional models as well as some time-series models. statsmodels uses a model descriptive language (provided via the Python package patsy) to formulate the model when working with pandas DataFrames. Models supported include linear regression, generalized linear models, limited dependent variable models, ARMA and VAR models.

26.2 pytz and babel

ptyz and babel provide extended support for time zones and formatting information.

26.3 rpy2

rpy2 provides an interface for calling R 3.0.x in Python, as well as facilities for easily moving data between the two platforms.

26.4 PyTables and h5py

PyTables and h5py both provide access to HDF5 files, a flexible data storage format optimized for numeric data.

Chapter 27

Quick Reference

27.1 Built-ins

import

`import` is used to import modules for use in a program.

from

`from` is used to import selected functions from a module in a program.

def

`def` is used to denote the beginning of a function.

return

`return` is used return a value from a function.

xrange

`xrange` is an iterator commonly used in `for` loops.

tuple

A `tuple` is an immutable collection of other types, and is initialized using parentheses, e.g. `(a,)` for a single element tuple or `(a,b)` for a 2-element tuple.

list

A `list` is a mutable collection of other types, and is initialized using square brackets, e.g. `[a]` or `[a,b]`.

dict

`dict` initialized a dictionary, which is a list of named values where the names are unique.

set, frozenset

set initializes a set which is a unique, mutable collection. frozenset is an immutable counterpart.

for

for being a for loop, and should have the structure for *var* in *iterable*: .

while

while begins a while loop, and should have the structure while *logical* :

break

break terminates a loop prematurely.

continue

continue continues a loop without processing any code below the continue statement.

try

try begin a block which can handle code which may not succeed. It must be followed by an except statement.

except

except catches errors generated in a try statement.

if

if begins an if . . . elif . . . else, and should have the structure if *logical* :

elif

elif stands for else if, and can be used after an if statement to refine the expression. It should have the structure elif *logical* :

else

else finalizes an if block and executes if no previous path was taken.

print

print outputs information to the console. If used with from __future__ import print_function, print behaves like a function.

file

file opens a file for low-level file reading and writing.

27.1.1 `file` Methods

File methods operate on a file object, which is initialized using `file`. For example, `f = file('text.txt','r')` opens the file text.txt for reading, and `close` is used as `f.close()`.

close

`close` closes an open file handle, and flushes any unwritten data to disk.

flush

`flush` flushes any unwritten data to disk without closing the file.

read

`read` reads data from an open file.

readline

`readline` reads a single line from an open file.

readlines

`readlines` reads one or more lines from an open file.

write

`write` writes a single line to a file without appending the new line character.

writelines

`writelines` writes the contents of an iterable (e.g. a list) to a file without appending the new line character to each line.

27.1.2 String (`str`) Methods

String methods operate on strings. For example, `strip` can be used on a string `x = 'abc '` as `x.strip()`, or can be directly used as `' abc '.strip()`.

split

`split` splits a string at every occurrence of another string, left to right.

rsplit

`rsplit` splits a string at every occurrence of another string, right to left.

join

`join` combines an iterable of strings using a given string to combine.

strip

strip removes leading and trailing whitespace from a string.

lstrip

lstrip removes leading whitespace from a string.

rstrip

rstrip removes trailing whitespace from a string.

find

find returns the index of the first occurrence of a substring. -1 is returned if not found.

rfind

rfind returns the index of the first occurrence of a substring, scanning from the right. -1 is returned if not found.

index

index behaves like find, but raises an error when not found.

count

count counts the number of occurrences of a substring.

upper

upper converts a string to upper case.

lower

lower coverts a string to lower case.

ljust

ljust right pads a string with whitespace or a specified character up to a given width.

rjust

rjust left pads a string with whitespace or a specified character up to a given width.

center

center left and right pads a string with whitespace or a specified character up to a given width.

replace

replace returns a copy of a string with all occurrences of a substring replaced with an alternative substring.

format

format formats and inserts formattable objects (e.g. numbers) into a string.

27.1.3   Operating System (os)

os.system

system executes (external) system commands.

os.getcwd

getcwd returns the current working directory.

os.chdir

chdir changes the current working directory.

os.mkdir

mkdir creates a new directory, but requires all higher level directories to exist.

os.makedirs

makedirs creates a directory at any level, and creates higher level directories if needed.

os.rmdir

rmdir removes an empty directory.

os.listdir

listdir returns the contents of a directory. See glob.glob for a more useful form.

os.path.isfile

path.isfile determines whether a string corresponds to a file.

os.path.isdir

path.isdir determines whether a string corresponds to a directory.

27.1.4 Shell Utilities (`shutil`)

`shutil.copy`

copy copies a files using either a file name to use for the copy or a directory to use as a destination, in which case the current file name is used.

`shutil.copyfile`

copyfile copies a file using a file name.

`shutil.copy2`

copy2 is identical to copy only that (some) file meta-data is also copied.

`shutil.copytree`

copytree copies an entire directory tree to a destination that must not exist.

`shutil.move`

move moves a directory tree to a destination which must not exist.

`shutil.make_archive`

make_archive creates zip, gztar and bztar archives.

`shutil.rmtree`

rmtree recursively removes a directory tree.

27.1.5 Regular Expressions (re)

`re.findall`

findall returns all occurrences of a regular expression in a string as a list.

`re.split`

split splits a string on occurrences of a regular expression.

`re.sub`

sub substitutes a string of each occurrence of a regular expression.

`re.finditer`

finditer works similarly to findall, only returning an iterable object rather than a list.

`re.compile`

`compile` compiles a regular expression for repeated use.

27.1.6 Dates and Times (`datetime`)

`datetime.datetime`

`datetime` initializes a date-time object.

`datetime.date`

`date` initializes a date object.

`datetime.time`

`time` initializes a time object.

`datetime.timedelta`

`timedelta` represents the difference between two `datetime`s.

`datetime.datetime.replace`

`replace` replaces fields in a date-time object. `replace` is a method of a date-time, date or time object.

`datetime.datetime.combine`

`combine` combines a date object and a time object and returns a date-time object. `combine` is a method of a date-time, date or time object.

27.1.7 Other

These function all are contained in other modules, as listed to the left of the dot.

`glob.glob`

`glob.glob` returns a directory listing allowing for wildcards.

`subprocess.call`

`subprocess.call` can be used to run external commands.

`textwrap.wrap`

`textwrap.wrap` wraps a long block of text at a fixed width.

27.2 NumPy (`numpy`)

The functions listed in this section are all provided by NumPy. When a function is listed using only the function name, this function appears in the NumPy module, and so can be accessed as `numpy.`*function*, assuming that NumPy was imported using `import numpy`. When a function name contains a dot, for example `linalg.eig`, then the function `eig` is in the `linalg` module of `numpy`, and so the function is accessed as `numpy.linalg.eig`.

27.2.1 Core NumPy Types

dtype

`dtype` is used for constructing data types for use in arrays.

array

`array` is the primary method for constructing arrays from iterables (e.g. list or matrix). Variables created using `array` have type `numpy.ndarray`.

matrix

`matrix` constructs a matrix from an iterable (e.g. list or array) that directly supports matrix mathematics. Variables created using `matrix` have type `numpy.matrixlib.defmatrix.matrix`, which is a subclass of `numpy.ndarray`. Since `matrix` is a subclass of `numpy.ndarray`, it inherits the methods of `numpy.ndarray`.

27.2.2 ndarray

`ndarray` is the core array data type provided by NumPy. Both `array` (`numpy.ndarray`) and `matrix` (`numpy.matrixlib.defma` offer a large number of attributes and methods. Attributes are accessed directly from an array, e.g. `x.dtype`, and methods are calls to functions which operate on an array, e.g. `x.sum()`.

Attributes

T

`T` returns the transpose of an array.

dtype

`dtype` returns the data type of an array.

flat

`flat` returns a 1-dimensional iterator for an array.

imag

`imag` returns the imaginary portion of an array.

real

real returns the real portion of an array.

size

size returns the total number of elements of an array.

ndim

ndim returns the number of dimensions of an array.

shape

shape returns the shape of an array as a tuple with ndim elements. shape can also be used to set the shape using a tuple, e.g. x.shape=(10,5) as long as the number of elements does not change.

Methods

all

all returns True if all elements evaluate to True (i.e. not False, None or 0). axis can be used to compute along a particular axis.

any

any returns True if any element evaluates to True. axis can be used to compute along a particular axis.

argmax

argmax returns the index of the maximum of an array. axis can be used to compute along a particular axis.

argmin

argmin returns the index of the minimum of an array. axis can be used to compute along a particular axis.

argsort

argsort returns the indices needed to sort an array. axis can be used to compute along a particular axis.

astype

astype allows an array to be viewed as another type (e.g. matrix or recarray) and copies the underlying.

conj, conjugate

conj and conjugate both return the complex-conjugate element-by-element.

copy

copy returns a copy of an array.

cumprod

cumprod returns the cumulative product of an array. axis can be used to compute along a particular axis.

cumsum

cumsum return the cumulative sum of an array. axis can be used to compute along a particular axis.

dot

dot computes the dot-product (standard matrix multiplication) or an array with another array.

flatten

flatten returns a copy of an array flattened into a 1-dimensional array.

max

max returns the maximum value of an array. axis can be used to compute along a particular axis.

mean

mean returns the average of an array. axis can be used to compute along a particular axis.

min

min returns the minimum of an array. axis can be used to compute along a particular axis.

nonzero

nonzero returns the indices of the non-zero elements of an array.

prod

prod computes the produce of all elements of an array. axis can be used to compute along a particular axis.

ravel

ravel returns a flattened view of an array without copying data.

repeat

repeat returns an array repeated, element-by-element.

reshape

reshape returns returns an array with a different shape. The number of elements must not change.

resize

resize changes the size shape of an array in-place. If the new size is larger then new entries are 0 filled. If the new size is smaller, then the elements selected are ordered according by their order in `ravel`.

round

round returns an array with each element rounded to a provided number of decimals.

sort

sort sorts an array (in-place). `axis` can be used to compute along a particular axis.

squeeze

squeeze returns an array with any singleton dimensions removed.

std

std returns the standard deviation of an array. `axis` can be used to compute along a particular axis.

sum

sum returns the sum of an array. `axis` can be used to compute along a particular axis.

tolist

tolist returns a list of an array. If the array has more than 1 dimension, the list will be nested.

trace

trace returns the sum of the diagonal elements.

transpose

transpose returns a view of the array transposed. `x.transpose()` is the same as `x.T`.

var

var returns the variance of an array. `axis` can be used to compute along a particular axis.

view

view returns a view of an array without copying data.

Methods and Attributes as functions

Many of the `ndarray` methods can be called as functions and behave identically, aside from taking the array as the first input. For example, `sum(x, 1)` is identical to `x.sum(1)`, and `x.dot(y)` is identical to `dot(x,y)`. The following list of functions are identical to their method: `all`, `any`, `argmax`, `argmin`, `argsort`, `conj`, `cumprod`, `cumsum`, `diagonal`, `dot`, `imag`, `real`, `mean`, `std`, `var`, `prod`, `ravel`, `repeat`, `squeeze`, `reshape`, `std`, `var`, `trace`, `ndim`, and `squeeze`.

Functions with different behavior

round

`around` is the function name for `round`. `around` is preferred as a function name since `round` is also the name of a built-in function which does not operate on arrays.

resize

Using `resize` as a function returns the resized array. Using `resize` as a method performs the resizing in-place.

sort

Using `sort` as a function returns the sorted array. Using `sort` as a method performs the sorting in-place.

size

Using `size` as a function can take an additional argument `axis` specifying the axis to use. When used without an argument `size(x)` is identical to `x.size`.

shape

Using `shape` as a function only returns the shape of an array. The attribute use of `shape` also allows the shape to be directly set, e.g. `x.shape`=3,3.

max

`amax` is the function name for the method `max`. `amax` is preferred as a function name since `max` is also the name of a built-in function which does not operate on arrays.

min

`amin` is the function name for the method `min`. `amin` is preferred as a function name since `min` is also the name of a built-in function which does not operate on arrays.

27.2.3 `matrix`

`matrix` is a derived class of `ndarray` and so inherits its attributes and methods. Members of the `matrix` class have additional attributes.

Attributes

I

I returns the inverse of the matrix. This command is equivalent to `inv(x)` for an invertible matrix x.

A

A returns a view of the matrix as a 2-dimensional array.

A1

A1 returns a view of the matrix as a flattened array.

H

H returns the Hermetian (conjugate-transpose) of a matrix. For real matrices, `x.H` and `x.T` are identical.

Attributes

27.2.4 Array Construction and Manipulation

linspace

`linspace` creates an n-element linearly spaced vector between a lower and upper bound.

logspace

`logspace` creates a logarithmically spaced (base-10) vector between a lower and upper (log-10) bound.

arange

`arange` creates an equally spaced vector between a lower and upper bound using a fixed step size.

ones

`ones` creates an array of 1s.

zeros

`zeros` creates an array of 0s.

empty

`empty` creates an array without initializing the values.

eye

`eye` creates an identity array.

identity

`identify` creates an identity array.

meshgrid

`meshgrid` creates 2-dimensional arrays from 2 1-dimensional arrays which contain all combinations of the original arrays.

tile

`tile` block repeats an array.

broadcast_arrays

`broadcast_arrays` produces the broadcasted version of 2 broadcastable arrays.

vstack

`vstack` vertically stacks 2 or more size compatible arrays.

hstack

`hstack` horizontally stacks 2 or more size compatible arrays.

vsplit

`vsplit` splits an array into 2 or more arrays vertically.

hsplit

`hsplit` splits an array into 2 or more arrays horizontally.

split

`split` splits an array into 2 or more arrays along an arbitrary axis.

concatenate

`concetenate` combines 2 or more arrays along an arbitrary axis.

delete

`delete` deletes elements from an array. `axis` can be used to delete along a particular axis.

flipud

`flipud` flips an array top-to-bottom.

fliplr

fliplrud flips an array left-to-right.

diag

diag returns the diagonal from a 2-dimensional matrix, or returns a diagonal 2-dimensional matrix when used with a 1-dimensional input.

triu

triu returns the upper triangular array from an array.

tril

tril returns the lower triangular array from an array.

27.2.5 Array Functions

kron

kron returns the Kronecker product of 2 arrays.

trace

trace returns the sum of the diagonal elements of an array.

diff

diff returns the 1st difference of an array. An optional second input allows for higher order differencing. axis can be used to compute the difference along a particular axis.

27.2.6 Input/Output

loadtxt

loadtxt loads a rectangular array from a text file. No missing values are allowed. Automatically decompresses gzipped or bzipped text files.

genfromtxt

genfromtxt loads text from a data file and can accept missing values.

load

load loads a npy or npz file.

save

save saves a single array a NumPy data file (npy).

savez

savez saves an array or set of arrays to a NumPy data file (npz).

savez_compressed

savez_compressed saves an array or set of arrays to a NumPy data file (npz) using compression.

savetxt

savetxt saves a single array to a text file.

27.2.7 nan Functions

nansum

nansum returns the sum of an array, ignoring NaN values. axis can be used to compute along a particular axis.

nanmax

nanmax returns the maximum of an array, ignoring NaN values. axis can be used to compute along a particular axis.

nanargmax

nanargmax returns the index of the maximum of an array, ignoring NaN values. axis can be used to compute along a particular axis.

nanmin

nanmin returns the minimum of an array, ignoring NaN values. axis can be used to compute along a particular axis.

nanargmin

nanargmin returns the index of the minimum of an array, ignoring NaN values. axis can be used to compute along a particular axis.

27.2.8 Set Functions

unique

unique returns the set of unique elements of an array.

in1d

in1d returns a Boolean array indicating which elements of one array are in another.

intersect1d

`intersect1d` returns the set of elements of one array which are in another.

union1d

`union1d` returns the set of elements which are in either of 2 arrays, or both.

setdiff1d

`setdiff1d` returns the set of elements on one array which are not in another.

setxor1d

`setxor1d` returns the set of elements which are in either or 2 arrays, but not both.

27.2.9 Logical and Indexing Functions

logical_and

`logical_and` compute the value of applying and to the elements of two broadcastable arrays.

logical_or

`logical_or` compute the value of applying or to the elements of two broadcastable arrays.

logical_xor

`logical_xor` compute the value of applying xor to the elements of two broadcastable arrays.

logical_not

`logical_not` compute the value of applying not to the elements of an array.

allclose

`allclose` returns True if all elements of two arrays differ by less than some tolerance.

array_equal

`array_equal` returns True if two arrays have the same shape and elements.

array_equiv

`array_equiv` returns True if two arrays are equivalent int eh sense that one array can be broadcast to become the other.

find

`find` returns the indices of an array where a logical statement is true. The indices returned correspond to the flattened array.

argwhere

argwhere returns the indices from an array where a logical condition is `True`.

extract

`extract` returns the elements from an array where a logical condition is `True`.

isnan

`isnan` returns a Boolean array indicating whether the elements of the input are `nan` .

isinf

`isinf` returns a Boolean array indicating whether the elements of the input are `inf` .

isfinite

`isfinite` returns a Boolean array indicating whether the elements of the input are not `inf` and not `nan`.

isposinf

isposinf returns a Boolean array indicating whether the elements of the input are `inf`.

isneginf

isneginf returns a Boolean array indicating whether the elements of the input are `-inf`

isreal

`isreal` returns a Boolean array indicating whether the elements of the input are either real or have `0j` complex component.

iscomplex

`iscomplex` returns a Boolean array indicating whether the elements of the input are either have non-zero complex component.

is_string_like

`is_string_like` returns `True` if the input is a string or similar to a string.

isscalar

isscalr returns `True` if the input is not an array or matrix.

is_numlike

`is_numlike` returns `True` if the input is numeric.

isvector

isvector returns True if the input has at most 1 dimension which is not unity.

27.2.10   Numerics

nan

nan represents Not a Number.

inf

inf represents infinity.

finfo

finfo can be used along with a data type to return machine specific information about numerical limits and precision.

27.2.11   Mathematics

log

log returns the natural logarithm of the elements of an array.

log10

log10 returns the bast-10 logarithm of the elements of an array.

sqrt

sqrt returns the square root of the elements of an array.

exp

exp returns the exponential of the elements of an array.

absolute

absolute returns the absolute value of the elements of an array.

sign

sign returns the sign of the elements of an array.

27.2.12   Rounding

floor

floor rounds to next smallest integer.

ceil

ceil round to next largest integer

27.2.13 Views

asmatrix

asmatrix returns a view of an array as a `matrix`.

mat

mat is identical to asmatrix.

asarray

asarray returns a view of a matrix as an `ndarray`.

27.2.14 rec

rec.array

rec.array construct record arrays.

27.2.15 `linalg`

linalg.matrix_power

matrix_power raises a square array to an integer power.

linalg.cholesky

cholesky computes the Cholesky factor of a positive definite array.

linalg.qr

qr computes the QR factorization of an array.

linalg.svd

svd computes the singular value decomposition of an array.

linalg.eig

eig computes the eigenvalues and eigenvectors of an array.

linalg.eigh

eigh computes the eigenvalues and eigenvectors of a Hermitian (symmetric) array.

linalg.cond

cond computes the conditioning number of an array.

linalg.det

det computes the determinant of an array.

linalg.slogdet

slogdet computes the log determinant and the sign of the determinant of an array.

linalg.solve

solve solves a just-identified set of linear equations.

linalg.lstsq

lstsq finds the least squares solutions of an over-identified set of equations.

linalg.inv

inv computes the inverse of a square array.

27.2.16 random

random.rand

rand returns standard uniform pseudo-random numbers. Uses n inputs to produce an n-dimensional array.

random.randn

randn returns standard normal pseudo-random numbers. Uses n inputs to produce an n-dimensional array.

random.randint

randing returns uniform integers on a specified range, exclusive of end point. Uses an n-element tuple to produce an n-dimensional array.

random.random_integers

random_integers returns uniform integers on a specified range, inclusive of end point. Uses an n-element tuple to produce an n-dimensional array.

random.random_sample

random_sample returns standard Uniform pseudo-random numbers. Uses an n-element tuple to produce an n-dimensional array.

random.random

random returns standard Uniform pseudo-random numbers. Uses an n-element tuple to produce an n-dimensional array.

random.standard_normal

standard_normal returns standard normal pseudo-random numbers. Uses an n-element tuple to produce an n-dimensional array.

random.sample

sample returns standard Uniform pseudo-random numbers. Uses an n-element tuple to produce an n-dimensional array.

random.shuffle

shuffle shuffles the elements of an array in-place.

random.permutation

permutation returns a random permutation of an array.

random.RandomState

RandomState is a container for the core random generator. RandomState is used to initialize and control additional random number generators.

random.seed

seed seeds the core random number generator.

random.get_state

get_state gets the state of the core random number generator.

random.set_state

set_state sets the state of the core random number generator.

Random Number Generators

Random number generators are available for distribution in the following list: beta , binomial, chisquare, exponential, f, gamma, geometric, laplace, logistic, lognormal, multinomial, multivariate_normal, negative_binomial, normal, poisson, uniform.

27.3 SciPy

27.3.1 Statistics (`stats`)

27.3.1.1 Continuous Random Variables

Normal (`norm`), Beta (`beta`), Cauchy (`cauchy`), χ^2 (`chi2`), Exponential (`expon`), Exponential Power (`exponpow`), F (`f`), Gamma (`gamma`), Laplace/Double Exponential (`laplace`), Log-Normal (`lognorm`), Student's t (`t`)

stats.*dist*.rvs

`rvs` generates pseudo-random variables.

stats.*dist*.pdf

`pdf` returns the value of the PDF at a point in the support.

stats.*dist*.logpdf

`logpdf` returns the log of the PDF value at a point in the support.

stats.*dist*.cdf

`cdf` returns the value of the CDF at a point in the support.

stats.*dist*.ppf

`ppf` returns the value of the random variable from a point in (`0,1`). PPF is the same as the inverse CDF.

stats.*dist*.fit

`fit` estimates parameters by MLE.

stats.*dist*.median

`median` returns the median of random variables which follow the distribution.

stats.*dist*.mean

mean returns the mean of random variables which follow the distribution.

stats.*dist*.moment

`moment` returns non-central moments of random variables which follow the distribution.

stats.*dist*.var

var returns the variance of random variables which follow the distribution.

stats.*dist*.std

std returns the standard deviation of random variables which follow the distribution.

27.3.1.2 Statistical Functions

stats.mode

mode returns the empirical mode of an array.

stats.moment

moment computes non-central moments of an array.

stats.skew

skew computes the skewness of an array.

stats.kurtosis

kurtosis computes the *excess* kurtosis of an array.

stats.pearsonr

pearsonr computes the correlation of an array.

stats.spearmanr

spearmanr computes the rank correlation of an array.

stats.kendalltau

kendalltau computed Kendall's τ, which is similar to a correlation, from an array.

stats.normaltest

normaltest computes a Jarque-Bera like test for normality.

stats.kstest

kstest computes a Kolmogorov-Smirnov test for a specific distribution.

stats.ks_2samp

ks_2samp computes a Kolmogorov-Smirnov test from directly from two samples.

stats.shapiro

shapire computes the Shapiro-Wilks test of normality.

27.3.2 Optimization (`optimize`)

27.3.2.1 Unconstrained Function Minimization

`optimize.fmin_bfgs`

fmin_bfgs minimizes functions using the BFGS algorithm.

`optimize.fmin_cg`

fmin_cg minimizes functions using a Conjugate Gradient algorithm.

`optimize.fmin_ncg`

fmin_ncg minimizes functions using a Newton-Conjugate Gradient algorithm.

27.3.2.2 Derivative Free Unconstrained Function Minimization

`optimize.fmin`

fmin minimizes a function using a simplex algorithm.

`optimize.fmin_powell`

fmin_powell minimizes a function using Powell's algorithm.

27.3.2.3 Constrained Function Minimization

`optimize.fmin_slsqp`

fmin_slsqp minimizes a function subject to inequality, equality and bounds constraints.

`optimize.fmin_tnc`

fmin_tnc minimizes a function subject to bounds constraints.

`optimize.fmin_l_bfgs_s`

fmin_l_bfgs_s minimizes a function subject to bounds constraints.

`optimize.fmin_colyba`

fmin_colyba minimizes a function subject to inequality and equality constraints.

27.3.2.4 Scalar Function Minimization

`optimize.fmin_bound`

fmin_bound minimizes a function in a bounded region.

`optimize.golden`

golden uses a golden section search to minimize a scalar function.

`optimize.brent`

brent uses Brent's method to minimize a scalar function.

27.3.2.5 Nonlinear Least Squares

`optimize.lstsq`

lstsq performs non-linear least squares minimization of a function which returns a 1-dimensional array of errors.

27.3.3 Input/Output (`io`)

`io.loadmat`

loadmat loads a MATLAB data file.

`io.savemat`

savemat saves an array or set of arrays to a MATLAB data file.

27.4 Matplotlib

27.4.1 2D plotting

`plot`

plot plots a 2-dimensional line or set of lines.

`pie`

pie produces a pie chart.

`hist`

hist computes and plots a histogram.

`scatter`

scatter produces a scatter plot.

`bar`

bar produces a vertical bar chart.

barh

barh produces a horizontal bar chart.

contour

contour produces a 2-dimensional contour representation of 3-dimensional data.

27.4.2   3D plotting

plot

plot using the optional keyword argument zs produces 3-dimensional plots.

plot_wireframe

plot_wireframe plots a 3-dimensional wire-frame surface.

plot_surface

plot_surface plots a 3-dimensional solid surface.

27.4.3   Utility Commands

figure

figure opens a new figure or changes the current figure to an existing figure.

add_axes

add_axes adds a single axes to a figure.

add_subplot

add_subplot adds a subplot to a figure.

show

show updates a figure and pauses the running of a non-interactive program until the figure is closed.

draw

draw updates a figure.

close

close closes figures. close('all') closes all figures.

legend

legend adds a legend to a plot or chart using the information in label keyword arguments.

title

title adds a title to a plot or chart.

savefig

savefig exports figures to common file formats including PDF, EPS, PNG and SVG.

27.4.4 Input/Output

csv2rec

csv2rec reads data in a CSV file and returns a NumPy record array. Columns are automatically types based on the first few rows of the input array, and importing dates is supported.

27.5 Pandas

The format DataFrame.*method* is used to indicate that method is called directly on a DataFrame (e.g. x.*method*()).

27.5.1 Data Structures

Series

Series constructs the 1-dimensional array-like (or column-like) structure underlying a DataFrame.

TimeSeries

TimeSeries is a sub-class of Series where the index is a set of datetimes.

DataFrame

DataFrame constructs the primary 2-dimensional array-like (or table-like) structure in pandas.

27.5.2 Series/DataFrame Methods

drop

drop drops rows or columns using a list of labels.

dropna

dropna drops rows containing null-values.

drop_duplicates

drop_duplicates drops rows that contain duplicate values.

values

values returns the NumPy array underlying the DataFrame.

index

values returns an Index object containing the index labels.

fillna

fillna allows null-values to be filled with another value.

T, transpose

T transposes a DataFrame and is used as a property (i.e. df.T). transpose is identical except that it is used as a method (i.e. df.transpose())

sort

sort sorts a DataFrame using either index or column labels or values in one or more columns.

sort_index

sort_index is identical to sort except for the keyword argument names.

pivot

pivot reshapes a DataFrame

stack

stack creates a single Series representation of a DataFrame.

unstack

unstack converts a single Series representation to a Series.

concat

concat concatenates two DataFrames into a single DataFrame. concat is a function which operated on DataFrames and is not a method.

append

append concatenates a DataFrame to the existing DataFrame.

reindex

reindex changes the index and/or column labels.

reindex_like

reindex_like reindexes a DataFrame using the index of another DataFrame.

reindex_axis

reindex_axis reindexes one axis of a DataFrame.

merge

merge performs a SQL-like join of two DataFrames using index values to construct the union.

join

join performs a SQL-like join of two DataFrames using values in columns to construct the union.

update

update updates the values of cells in one DataFrame using index labels.

groupby

groupby produces a DataFrameGroupBy object that can be used to compute grouped statistics or plots.

apply

apply applies a function column- or row-wise to a DataFrame.

applymap

applymap applies a function element-by-element to a DataFrame.

pivot_table

pivot_table produces an Excel-like grouping of a DataFrame using one or more columns to group.

count

count returns the number of non-null values for each variable in a DataFrame.

describe

describe produces summary statistics about the variables in a DataFrame.

value_counts

value_counts performs histogramming on a DataFrame.

27.5.3 Graphics

DataFrame.plot

plot produces a plot of data in a DataFrame. A keyword argument kind can be used to produce line plots (default), bar ('bar'), horizontal bar ('barh') or kernel density plots ('kde' or 'density').

DataFrame.hist

hist produces a histogram plot.

DataFrame.boxplot

boxplot produces a boxplot of the data in a DataFrame.

DataFrame.scatter_plot

scatter_plot produces a scatter plot of two series.

DataFrame.scatter_matrix

scatter_matrix produces a matrix of bivariate scatter plots with histograms or kernel density estimates along the diagonal.

DataFrame.lag_plot

lag_plot produces a scatter plot of a time series against its lag.

27.5.4 Input/Output

read_excel

read_excel reads both Excel 97/2003 and 2007/10/13 files and requires both the filename and the sheet name.

read_csv

read_csv reads csv and other delimited files.

read_stata

read_stata reads STATA data files.

read_table

read_table reads general delimited files.

read_hdf

read_hdf reads data from HDF5 files (usually created using DataFrame.to_hdf)

read_html

read_html reads html tables.

read_json

read_html reads data from JavaScript Object Notation.

DataFrame.to_excel

to_excel writes data to an Excel file.

DataFrame.to_csv

to_csv writes data to a CSV file.

DataFrame.to_string

to_string writes data to a string variable or buffer (file or StringIO).

DataFrame.to_latex

to_latex writes data to a LaTeX file.

DataFrame.to_stata

to_stata writes data to a STATA data file.

DataFrame.to_hdf

to_hdf writes data to a HDF5 data file.

DataFrame.to_html

to_hdf writes data to a HTML data table.

DataFrame.to_json

to_hdf writes data to a JSON string.

27.6 IPython

?*partial*, ?module.*partial*

?*partial* list any known objects – variables, functions or modules – which match the wild card expression
partial where *partial* can be any string.

*function***?**, *magic***?**

?*function*, ?*magic*, ?*module*, *function*?, *magic*? and *module*? all pretty print the docstring for a function, magic word or module.

*function***??**, *magic***??**

??*function*, ??*magic*, ??*module*, *function*??, *magic*?? and *module*?? all pretty print the entire function, magic word or module, including both the docstring and the code inside the function.

!*command*

!*command* is used to run a system command as if executed in the terminal. For example, !`copy file`.py backup.py will copy file.by to backup.py. An output can be used to capture the command window text printed by the command to a local variable. For example, `dirListing = !dir *.py` will capture the contents to running a directory listing for all py files. This can then be parsed or used for error checking.

%bookmark

%bookmark allows bookmarks to be created to directories. Manage IPython's bookmark system. For example, the current directory can be bookmarked using `%bookmark currentDir`, and can be returned to using `%cd currentDir` (assuming the current directory doesn't contain a directory named currentDir).

%cd

%cd changes the current working directory. In Windows, \, \\ or / can be used as directory separators. In Linux, / should be used to separate directories. `%cd` support tab completion, so that only a few letters are needed before hitting the tab key to complete the directory name.

%clear, **%cls**

%clear and `%cls` both clear the terminal window (but do not remove any variables, functions or modules).

%edit

%edit opens the editor for py files, and is usually used %edit *filename*.py.

%hist

%hist lists command history. %hist -g *searchterm* can be used to search for a specific term or wildcard expression.

%lsmagic

%lsmagic lists all defined magic functions.

%magic

%magic prints detailed information about all magic functions.

%pdb

%pdb controls the use of the Python debugger.

%pdef

%pdef prints only the definition header (the line beginning with def) for functions.

%precision

%precision sets the display precision. For example %precision 3 will show 3 decimal places.

%prun

%prun runs a statement in the profiler, and returns timing and function call information. Use %run -p *filename.py* to profile a standalone file.

%psearch

%psearch searches for variables, functions and loaded modules using wildcards.

%pwd

%pwd shows the current working directory.

%pycat, %more, %less

%pycat, %more and %less all show the contents of a file in the IPython window.

%pylab

%pylab initializes pylab if not initialized from the command line.

%quickref

%quickref shows a reference sheet for magic commands.

%reset

%reset resets the session by removing all variables, functions and imports defined by the user.

%reset_selective

%reset_selective *re* resets all names which match the regular expression *re*.

%run

%run *filename.py* executes a file containing Python code in the IPython environment.

%time

`%time` *code* provides the time needed to run the code once. `%timeit` is often more useful.

%timeit

`%timeit` *code* times a segment of code, and is useful for finding bottlenecks and improving algorithms. If *code* is very fast, multiple runs are used to improve timing accuracy.

```
>>> %timeit x=randn(100,100);dot(x.T,x)
1000 loops, best of 3: 758 us per loop
```

%who

`%who` lists all variables in memory.

%who_ls

`%who_ls` returns a sorted list containing the names of all variables in memory.

%whos

`%whos` provides a detailed list of all variables in memory.

%xdel

`%xdel` *variable* deletes the variable from memory.

Bibliography

Bollerslev, T. & Wooldridge, J. M. (1992), 'Quasi-maximum likelihood estimation and inference in dynamic models with time-varying covariances', *Econometric Reviews* **11**(2), 143–172.

Cochrane, J. H. (2001), *Asset Pricing*, Princeton University Press, Princeton, N. J.

Flannery, B., Press, W., Teukolsky, S. & c, W. (1992), *Numerical recipes in C*, Press Syndicate of the University of Cambridge, New York.

Jagannathan, R., Skoulakis, G. & Wang, Z. (2010), The analysis of the cross section of security returns, *in* Y. Aït-Sahalia & L. P. Hansen, eds, 'Handbook of financial econometrics', Vol. 2, Elsevier B.V., pp. 73–134.

Index