This note is a summary of parts of “Introduction to Python for Econometrics, Statistics and Data Analysis” written by Kevin Sheppard, University of Oxford.

# 1 Built-in data types

## 1.1 Variables names

- Variable names can only contain numbers, letters, and underscores (_).
- They are case sensitive.
- Some words are reserved in Python and so cannot be used for variable names.

**Example 1.1**

$$\begin{aligned} x &= 1.0 \\ X &= 1.0 \\ x1 &= 1.0 \\ \_x &= 1.0 \end{aligned}$$

## 1.2 Data types

### 1.2.1 Numeric

- Float (numbers with .)
- Integer
- Complex number

**Example 1.2**

$$\begin{aligned} x &= 1 \\ & type(x) \\ x &= 1.0 \\ & type(x) \\ x &= 1j \\ & type(x) \end{aligned}$$

### 1.2.2 Boolean

The Boolean data type is used to represent true and false, using the reserved keywords True and False.

$$\begin{aligned} x &= True \\ &\quad type(x) \\ x &= bool(1) \\ x &= bool(0) \end{aligned}$$

### 1.2.3 Strings

$$\begin{aligned} x &= \text{‘}dog\prime \\ &\quad type(x) \\ &\quad print(x) \end{aligned}$$

### 1.2.4 List

- A list is a collection of other objects – floats, integers, complex numbers, strings or even other lists.

**Example 1.3**

$$\begin{aligned} x &= [1, 2, 3, 4] \\ x &= [1, 1j, \text{‘}one\text{’}, True] \\ x &= [[1, 2, 3, 4], [5, 6, 7, 8]] \end{aligned}$$

- Python uses 0-based indices, and so the $n$ elements of $x$ can be thought of as $x_0, x_1, ..., x_{n-1}$.
- Slicing lists
  - $x[:]$ Return all $x$
  - $x[i]$ Return $x_i$
  - $x[i:]$ Return $x_i, ..., x_{n-1}$
  - $x[:i]$ Return $x_0, ..., x_{i-1}$
  - $x[i:j]$ Return $x_i, ...x_{j-1}$

**Example 1.4** *x=[1,2,3,4,5,6,7,8,9]*
*x[0]*
*x[5]*
*x[4:]*
*x[:4]*
*x=[[1,2,3,4],[5,6,7,8]]*
*x[0]*

*x[0][0]*
*x[0][1:4]*

- List functions
  - 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.remove(x,value) x.remove(value) Removes the first occurrence of value from the list.
  - list.extend(x,list) x.extend(list) Appends the values in list to the existing list
  - list.pop(x,index) x.pop(index) Removes the value in position index and returns the value.
  - 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.

**Example 1.5** *x=[1,2,3,4,5,6,7,8,9]*
*x.append(10)*
*x.remove(3)*
*y=[10,11]*
*x.extend(y)*
*x.pop(1)*
*x.count(2)*

### 1.2.5 Tuples

- A tuple is virtually identical to a list with one important difference – tuples cannot be changed once created.

**Example 1.6** *x=(1,2)*
*type(x)*

### 1.2.6 Dictionary

- Dictionaries in Python are composed of keys (words) and values (definitions).
- Dictionaries keys must be unique immutable data types (e.g. strings,the most common key, integers, or tuples containing immutable types).
- Values can contain any valid Python data type.

**Example 1.7** *data={'age':34,'children':[1,2],1:'apple'}*
*type(data)*
*data['age']*

### 1.2.7 Sets

- Sets are collections which contain all unique elements of a collection.
- Set functions
  - 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.

**Example 1.8** *x=set([‘MSFT’,‘GOOG’,‘AAPL’,‘HPQ’,‘MSFT’])*
*x.add(’CSCO’)*
*y=set([’XOM’,’GOOG’])*
*x.intersection(y)*
*x.union(y)*
*x.remove(’XOM’)*

### 1.2.8 Range

- range(a,b,i) creates the sequences that follows the pattern a, a+i ,a+2i,...,a+(m-1)i where m = $[\frac{b-a}{i}]$.
- range(a,b) is the same as range(a,b,1) and range(b) is the same as range(0,b,1).

**Example 1.9** *x=range(10)*
*type(x)*
*list(x)*
*x=range(3,10,3)*

## 2 Arrays and Matrices

NumPy provides the core data types for econometrics, statistics, and numerical analysis – arrays and matrices.

- Arrays can have 1, 2, 3 or more dimensions, and matrices always have 2 dimensions.
- Standard mathematical operators on arrays operate element-by-element.
- Arrays can be quickly treated as a matrix using either asmatrix or mat without copying the underlying data.

## 2.1 import and Modules

- Python has access to only a small number of built-in types and functions. The vast majority of functions are located in modules.
- Before a function can be accessed, the module which contains the function must be imported.
- import pylab as pl
  import scipy as sp
  import numpy as np
- sp.sqrt, pl.sqrt, etc.

## 2.2 Arrays

- Arrays are the base data type in NumPy, are in similar to lists or tuples since they both contain collections of elements.
- Arrays, unlike lists, are always rectangular.

**Example 2.1** *from numpy import array*
*x = [0.0, 1, 2, 3, 4]*
*y = array(x)*
*y = array([[0.0, 1, 2, 3, 4], [5, 6, 7, 8, 9]])*
*shape(y)*
*y = array([[[1,2],[3,4]],[[5,6],[7,8]]])*
*shape(y)*

## 2.3 Matrix

- Matrices always have 2 dimensions.
- Matrices follow the rules of linear algebra for *.

**Example 2.2** *x = [0.0, 1, 2, 3, 4] # Any float makes all float.*
*y = array(x)*
*y * y # Element by element*
*z = asmatrix(x)*
*ndim(z)*

## 2.4 1-dimensional Arrays

- A vector x = [1 2 3 4 5] is entered as a 1-dimensional array using
  x=array([1.0, 2.0, 3.0, 4.0, 5.0])
  ndim(x)

- 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]])
  ndim(x)
- 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])
  ndim(x)

## 2.5 2-dimensional Arrays

- Matrices and 2-dimensional arrays are rows of columns
  x = array([[1.0,2.0,3.0],[4.0,5.0,6.0],[7.0,8.0,9.0]])
  x

## 2.6 Concatenation

Concatenation is the process by which one vector or matrix is appended to another.

**Example 2.3** *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 = concatenate((x,y),axis = 1)*

## 2.7 Accessing Elements of an Array

### 2.7.1 Scalar Selection

- Pure scalar selection is the simplest method to select elements from an array, and is implemented using [i] for 1-dimensional arrays and [i , j ] for 2-dimensional arrays.

**Example 2.4** *x = array([[1.0,2,3],[4,5,6]])*
*x[1, 2]*

### 2.7.2 Array Slicing

- Arrays are sliced using the syntax [:,:,. . .,:] (where the number of dimensions of the arrays determines the size of the slice).
- The slice notation a:b:s will select every sth element where the indices i satisfy a≤i< b so that the starting value a is always included in the list and the ending value b is always excluded.
- : 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).

**Example 2.5** *x = array([1.0,2.0,3.0,4.0,5.0])*
*y = x[:]*
*y = x[:2]*
*y = x[1::2]*

**Example 2.6** *y = array([[0.0, 1, 2, 3, 4],[5, 6, 7, 8, 9]])*
*y[:1,:] # Row 0, all columns*
*y[:,:1] # all rows, column 0*
*y[:1,0:3] # Row 0, columns 0 to 2*

#### 2.7.3 Assignment using Slicing

**Example 2.7** *x = array([[0]*3]*3) # *3 repeats the list 3 times*
*x[0,:] = array([1.0, 2.0, 3.0])*

# 3 Basic Math

## 3.1 Operators

- Addition: +
- Subtraction: -
- Multiplication: *
- Division: /
- Integer Division: //
- Exponentiation: **
- Matrix Multiplication: @
- Matrix transpose: .T or transpose(x), x.transpose()

# 4 Basic Functions and Numerical Indexing

## 4.1 Generating Arrays and Matrices

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

- 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(n) is equivalent to arange(0,n,1).
- r_[ start : end : step ] generates an 1-dimensional array, where start and end are the start and end points, and step is a step size.
- reshape rearranges arrays.

**Example 4.1** *x = arange(25.0)*
*z = reshape(x,(5,5))*

## 4.2 Rounding

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

**Example 4.2** *x=randn(3)*
*around(x)*
*around(x,2)*
*x.round(2)*

- floor rounds to the next smallest integer.
- ceil rounds to the next largest integer.

## 4.3 Mathematics

- sum sums elements in an array. By default, it will sum all elements in the array.
- The second argument is normally used to provide the axis to use – 0 to sum downcolumns, 1 to sum across rows.
- cumsum produces the cumulative sum of the values in the array.

**Example 4.3** *x = randn(3,4)*
*sum(x) # all elements*
*sum(x, 0) # Down rows, 4 elements*
*sum(x, 1) # Across columns, 3 elements*
*cumsum(x,0) # Down rows*

- prod and cumprod behave similarly to sum and cumsum except that the product and cumulative product are returned.
- diff computes the finite difference of a vector (also array) and returns an n-1 element vector when used on an n element vector

**Example 4.4** *x= randn(3,4)*
*diff(x, axis=0)*
*diff(x, 2, axis=0) # Double difference, column by column*

- exp returns the element-by-element exponential (e x ) for an array.
- sqrt returns the element-by-element square root for an array.
- square returns the element-by-element square ($x^2$) for an array.
- abs and absolute returns the element-by-element absolute value for an array.
- sign returns the element-by-element sign function, defined as 0 if x = 0.

## 4.4 Sorting and Extreme Values

- sort sorts the elements of an array.

**Example 4.5** *x = randn(4,2)*
*sort(x)*
*sort(x, 0)*
*sort(x, axis=None)*

max and min return the maximum and minimum values from an array

**Example 4.6** *x = randn(3,4)*
*max(x)*
*x.max()*
*x.max(0)*
*x.max(1)*

# 5 Special Arrays

- ones generates an array of 1s and is generally called with one argument, a tuple, containing the size of each dimension.

**Example 5.1** *M, N = 5, 5*
*x = ones((M,N)) # M by N array of 1s*

- zeros produces an array of 0s in the same way ones produces an array of 1s.
- empty produces an empty (uninitialized) array to hold values generated by another procedure.
- eye generates an identity array – an array with ones on the diagonal, zeros everywhere else.

**Example 5.2** *In = eye(N)*

## 6 Array and Matrix Functions

- asmatrix and mat can be used to view an array as a matrix.

**Example 6.1** *x = array([[1,2],[3,4]])*
*mat(x) * mat(x) # Matrix multiplication*

- asarray work in a similar matter as asmatrix, only that the view produced is that of ndarray.
- shape returns the size of all dimensions or an array or matrix as a tuple.

**Example 6.2** *x = randn(4,3)*
*shape(x)*
*x.shape*

- reshape transforms an array with one set of dimensions and to one with a different set, preserving the number of elements.

**Example 6.3** *x = array([[1,2],[3,4]])*
*y = reshape(x,(4,1))*

- size returns the total number of elements in an array or matrix.
- ndim returns the size of all dimensions or an array or matrix as a tuple.

**Example 6.4** *x=randn(4,3)*
*ndim(x)*

- tile replicates an array according to a specified size vector. The generic form of tile is 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

**Example 6.5** *x = array([[1,2],[3,4]])*
*w = tile(x,(2,3))*

- ravel returns a flattened view (1-dimensional) of an array or matrix.

**Example 6.6** *x = array([[1,2],[3,4]])*
*x.ravel()*

- vstack, and hstack stack compatible arrays and matrices vertically and horizontally, respectively

**Example 6.7** *x = array([[1,2],[3,4]])*
*z = hstack((x,x,x))*
*y = vstack((z,z))*

- 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.

**Example 6.8** *x = array([[1,2],[3,4]])*
*z=vsplit(x,2)*
*y=hsplit(x,2)*

- delete removes values from an array. 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).

**Example 6.9** *x = reshape(arange(20),(4,5))*
*delete(x,1,0) # Same as x[[0,2,3]]*
*delete(x,[2,3],1) # Same as x[:,[0,1,4]]*

- fliplr and flipud flip arrays in a left-to-right and up-to-down directions, respectively.

**Example 6.10** *x = reshape(arange(4),(2,2))*
*fliplr(x)*
*flipud(x)*

- diag returns a column vector containing the elements of the diagonal, if the input is a square array.
- diag returns an array containing the elements of the vector along its diagonal, if the input is an vector.

**Example 6.11** *x = array([[1,2],[3,4]])*
*y = diag(x)*
*z = diag(y)*

- triu and tril produce upper and lower triangular arrays, respectively.

**Example 6.12** *x = reshape(arange(20),(4,5))*
*triu(x)*
*tril(x)*

- svd computes the singular value decomposition of a matrix $X$, defined as $X = U\Sigma V$, where $U$ and $V$ are orthonormal and $\Sigma$ is diagonal.

**Example 6.13** *x = np.matrix([[1.0,0.5],[.5,1]])*
*np.linalg.svd(x)*

- cond computes the condition number of a matrix, which measures howclose to singular a matrix is. Lower numbers indicate that the input is better conditioned (further from singular).

- solve solves the system $X\beta = y$ when $X$ is square and invertible so that the solution is exact.

**Example 6.14** *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]])*
*np.linalg.solve(X,y)*

- 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 $\beta$ 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 $X$.

**Example 6.15** *X = np.random.randn(100,2)*
*y = np.random.randn(100)*
*np.linalg.lstsq(X,y)*

- 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 $\Sigma$ is a positive definite matrix.

**Example 6.16** *x = np.matrix([[1,.5],[.5,1]])*
*C = np.linalg.cholesky(x)*

- det computes the determinant of a square matrix or array.
- eig computes the eigenvalues and eigenvectors of a square matrix. When used with one output, the eigenvalues and eigenvectors are returned as a tuple.

**Example 6.17** *x =np.matrix([[1,.5],[.5,1]])*
*val,vec =np.linalg.eig(x)*

- eigh computes the eigenvalues and eigenvectors of a symmetric array.
- inv computes the inverse of an array. inv(x) can alternatively be computed using x**(1) when x is a matrix.

**Example 6.18** *x =np.matrix([[1,.5],[.5,1]])*
*xInv =np.linalg.inv(x)*

- kron computes the Kronecker product of two arrays, x and y, is written as z = kron(x,y).
- trace computes the trace of a square array (sum of diagonal elements).
- matrix_rank computes the rank of an array using a SVD.

# 7 Importing and Exporting Data

## 7.1 Importing Data using pandas

### 7.1.1 CSV and other formatted text files

Comma-separated value (CSV) files can be read using read_csv.

**Example 7.1** *from pandas import read_ csv*
*csv_ data = read_ csv('c:/AAPL.csv')*
*csv_ data = csv_ data.values*

### 7.1.2 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.

**Example 7.2** *from pandas import read_ excel*
*excel_ data = read_ excel('c:/AAPL.xlsx', 'AAPL')*
*excel_ data = excel_ data.values*

### 7.1.3 STATA files

- pandas also contains a method to read STATA files.

**Example 7.3** *from pandas import read_ stata*
*stata_ data = read_ stata('filename.dta')*
*stata_ data = stata_ data.values*

## 7.2 Saving or Exporting Data using pandas

- pandas supports writing to CSV, other delimited text formats, Excel files, json, html tables, HDF5 and STATA.

# 8 Inf, NaN and Numeric Limits

- inf represents infinity.
- nan stands for Not a Number, and nans are created whenever a function produces a result that cannot be clearly evaluated to produce a number or infinity.
- The easiest to understand the upper and lower limits, which are $1.7976\times10^{308}$ and $-1.7976\times10^{308}$. Numbers larger (in absolute value) than these are inf.
- The smallest positive number that can be expressed is $2.2250\times10^{-308}$. Numbers between $-2.2250\times10^{-308}$ and $2.2250\times10^{-308}$ are numerically 0.

# 9 Logical Operators and Find

- Core logical operators

| Symbol | Definition |
|---|---|
| > | greater |
| >= | greater than or equal to |
| < | less |
| <= | less than or equal to |
| == | equal |
| ! = | not equal to |

**Example 9.1** *x = array([[1,2],[3,4]])*
*x>0*
*x==-3*

- Logical expressions can be combined using four logical devices

| Function | Bitwise | True if |
|---|---|---|
| logical_and | & | Bothe true |
| logical_or | | Either or Both True |
| logical_not | ~ | Not True |
| logical_xor | ^ | One True and One False |

**Example 9.2** *x = arange(-2.0,4)*
*y = x >= 0*
*z = x < 2*
*logical_ and(y, z)*
*y & z*
*(x > 0) & (x < 2)*
*~(y & z)*

- all returns True if all logical elements in an array are 1.
- any returns logical(True) if any element of an array is True.

**Example 9.3** *x = array([[1,2][3,4]])*
*y = x <= 2*
*all(y)*
*any(y)*

- allclose can be used to compare two arrays for near equality.
- array_equal tests if two arrays have the same shape and elements.

## 10 Advanced Selection and Assignment

### 10.1 Logical Indexing

- Logical indexing differs from slicing and numeric indexing by using logical indices to select elements, rows or columns.
- nonzero takes logical inputs and returns
- a tuple containing the indices where the logical statement is true.

**Example 10.1** *x = array([[1,2],[3,4]])*
*sel = x <= 3*
*indices = nonzero(sel)*
*x[indices]*

*x = arange(3,3)*
*x[x < 0]*
*x[abs(x) >= 2]*

- argwhere returns an array containing the locations of elements where a logical condition is True.
- extract is similar to argwhere except that it returns the values where the condition is true rather than the indices.

**Example 10.2** *x=randn(3)*
*argwhere(x<0.6)*
*extract(x<0,x)*

## 11 Flow Control, Loops and Exception Handling

### 11.1 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.

### 11.2 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.
- The generic formof 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
or
if logical:
Code to run if logical true
or
if logical:
Code to run if logical true
else:
Code to run if logical false
```

**Example 11.1**
```
x = 5
if x<5:
x=x+1
elif x>5:
x=x-1
else
x=x*2
```

## 11.3 for

- for loops begin with for item in iterable.
- The generic structure of a for loop is

```
for item in iterable:
Code to run
```

**Example 11.2**
```
count = 0
for i in range(1,101):
count += i
```

**Example 11.3**
```
count = 0
x = linspace(0,500,50)
for i in x:
count += i  # count=count+i
```

**Example 11.4**
```
count = 0
for i in range(10):
for j in range(10):
count += j
```

**Example 11.5**
```
returns = randn(100)
count = 0
for ret in returns:
if ret<0:
count += 1
```

- 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.
- 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.

**Example 11.6**
```
x = randn(1000)
for i in x:
print(i)
if i > 2:
break
```

- 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

**Example 11.7**
```
for i in x:
if i >= 0:
continue
else:
print(i)
```

## 11.4 while

- while loops are usefulwhenthe number of iterations needed depends on the outcome of the loop contents.
- The generic structure of a while loop is

  while logical:
  Code to run
  Update logical

**Example 11.8**
```
count = 0
i = 1
while i<10:
count += i
i += 1
```

- break can be used in a while loop to immediately terminate execution.
- continue can be used in a while loop to skip any remaining code in the loop, immediately returning to the top of the loop.

# 12 Dates and Times

## 12.1 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

**Example 12.1** *import datetime as dt*
*yr, mo, dd = 2012, 12, 21*
*dt.date(yr, mo, dd)*

## 12.2 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.

**Example 12.2** *d1 = dt.datetime(yr, mo, dd, hr, mm, ss, ms)*
*d2 = dt.datetime(yr + 1, mo, dd, hr, mm, ss, ms)*
*d2-d1*

# 13 Graphics

- Matplotlib is a complete plotting library capable of high-quality graphics.
- Throughout this chapter, the following modules have been imported.

import matplotlib.pyplot as plt
import scipy.stats as stats

## 13.1 seaborn

- seaborn is a Python package which provides a number of advanced data visualized plots. It also provides a general improvement in the default appearance of matplotlib-produced plots

import seaborn as sns

## 13.2 2D Plotting

- autoscale can be used to set tight limits within a figure's axes and tight_layout will remove wasted space around a figure. These were used in figures that appear in this chapter.
- Basic line plots are produced using plot using a single input containing a 1-dimensional array.

**Example 13.1**
```
import matplotlib.pyplot as plt
import scipy.stats as stats
import numpy as np
y = np.random.randn(100)
plt.plot(y,’g–.’)
plt.autoscale(tight=’x’)
plt.tight_ layout()
```

- scatter produces a scatter plot between 2 1-dimensional arrays.

  scattter(x,y)

- bar produces bar charts using two 1-dimensional arrays .

**Example 13.2**
```
y = np.random.randn(5)
x=np.arange(5)
plt.bar(x,y)
```

- pie produces pie charts using a 1-dimensional array of data
- Histograms can be produced using hist.

### 13.3 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.

**Example 13.3**
```
import matplotlib.pyplot as plt
import scipy.stats as stats
import numpy as np
y = np.random.randn(100)
plt.plot(y,’g–.’)
plt.autoscale(tight=’x’)
plt.tight_ layout()
plt.savefig(’figure.pdf’)
```

## 14 pandas

pandas is a high-performance package that provides a comprehensive set of structures for working with data. pandas excels at handling structured data, such as data sets containingmanyvariables, working with missing values and merging across multiple data sets.

## 14.1 Data Structures

- 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 has an additional column – an index – which is a set of values which are associated with the rows of the Series.

**Example 14.1** *a = array([0.1, 1.2, 2.3, 3.4, 4.5])*
*from pandas import Series*
*s = Series(a)*

**Example 14.2** *s = Series([0.1, 1.2, 2.3, 3.4, 4.5], index = ['a','b','c','d','e'])*
*s['a']*
*s[0]*
*s[['a','c']]*
*s[[0,2]]*

- Series can also be initialized directly from dictionaries.

**Example 14.3** *s = Series({'a':0.1 ,'b': 1.2, 'c': 2.3, 'd':3.4, 'e': 4.5})*

- The underlying NumPy array is accessible through the values property, and the index is accessible the index property, which returns an Index type.

**Example 14.4** *s1 = Series([1.0,2,3])*
*s1.values*
*s1.index*
*s1.index.values*

- head() shows the first 5 rows of a series, and tail() shows the last 5 rows.
- ix is ane advanced indexing function that permits access either by position or index label. For example, s.ix[0:2] is the same as s[0:2].
- describe() returns a simple set of summary statistics

**Example 14.5** *s1 = Series(arange(10.0,20.0))*
*summ = s1.describe()*
*summ['mean']*

- unique() returns the unique elements of a series and nunique() returns the number of unique values in a Series.
- drop(labels) drop elements with the selected labels from a Series.

**Example 14.6** *s1 = Series(arange(1.0,6),index=['a','a','b','c','d'])*
*s1.drop('a')*

- fillna(value) fills all null values in a series with a specific value.

**Example 14.7** *s1 = Series(arange(1.0,4.0),index=['a','b','c'])*
*s2 = Series(arange(1.0,4.0),index=['c','d','e'])*
*s3.fillna(1.0)*

- append(series) appends one series to another, and is similar to list.append.
- 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(series) replaces values in a series with those in another series.

**Example 14.8** *s1 = Series(arange(1.0,4.0),index=['a','b','c'])*
*s2 = Series(1.0*arange(1.0,4.0),index=['c','d','e'])*
*s1.update(s2)*
*s1*

- 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-dimensional NumPy array.

**Example 14.9** *from pandas import DataFrame*
*a = array([[1.0,2],[3,4]])*
*df = DataFrame(a)*
*df*

- 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).

**Example 14.10** *df = DataFrame(array([[1,2],[3,4]]),columns=['a','b'])*
*df*
*df.columns = ['dogs','cats']*
*df*

- Index values are similarly assigned using either the keyword argument index or by setting the index property.

**Example 14.11** *df = DataFrame(array([[1,2],[3,4]]), columns=['dogs','cats'], index=['Alice','Bob'])*

- The final method to create a DataFrame uses a dictionary containing Series, where the keys contain the column names.

**Example 14.12** *s1 = Series(arange(0.0,5))*
*s2 = Series(arange(1.0,3))*
*DataFrame({'one': s1, 'two': s2})*