DST-SERB Sponsored Six-Days FDP on Data Analytics in Healthcare

Python Programming for Data Analytics

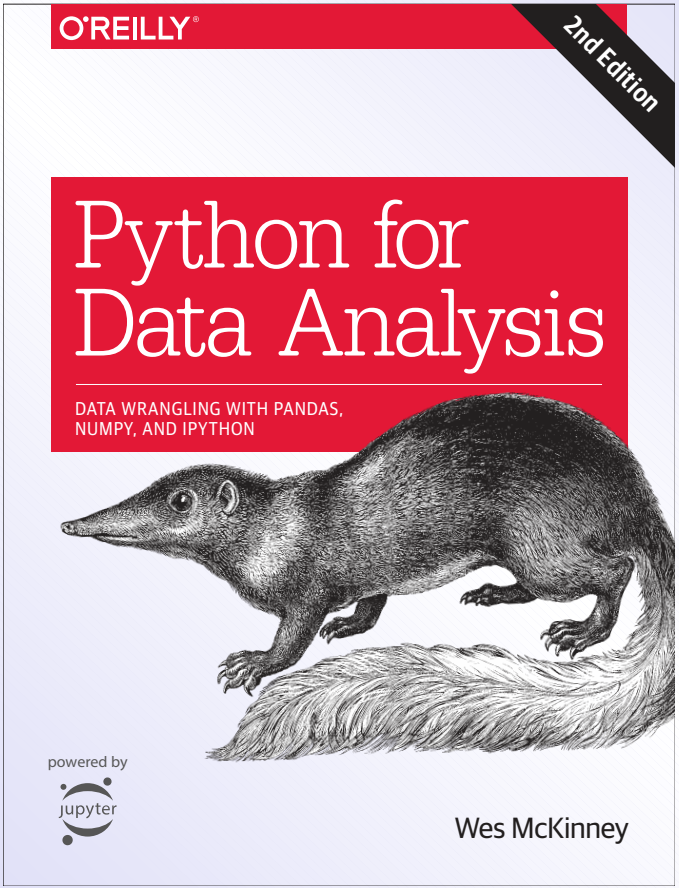
R. S. Milton

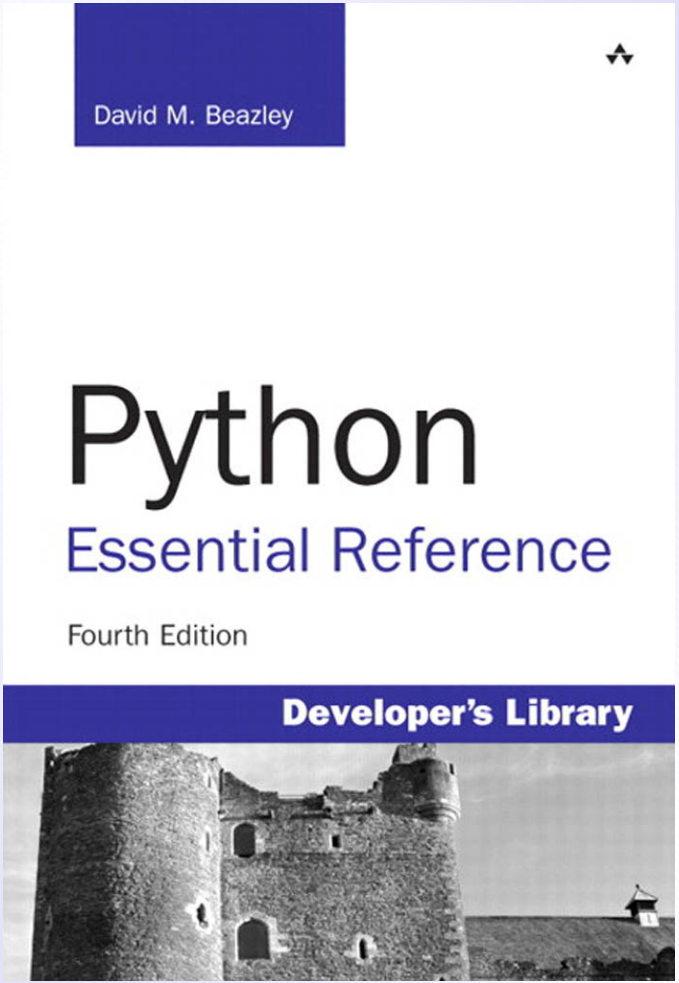
Department of Computer Science and Engineering

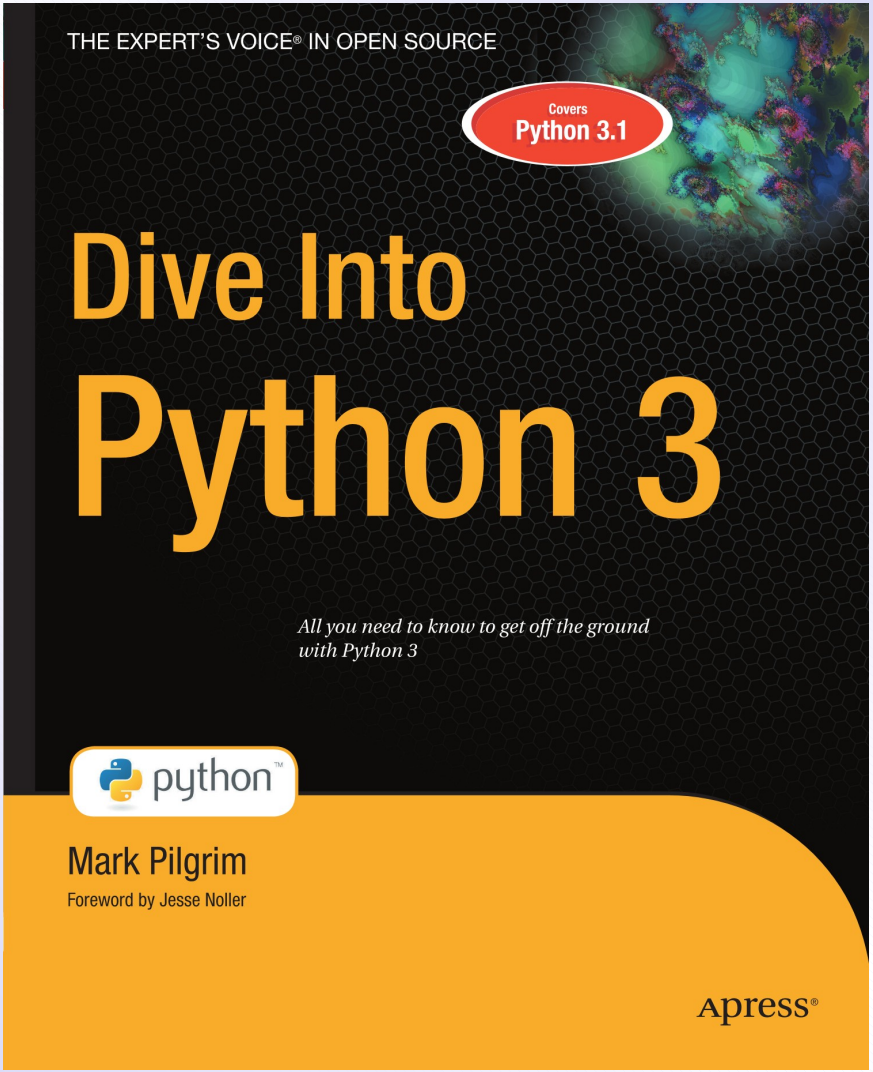
SSN College of Engineering

17 January 2022

1. Reading


O'REILLY
2nd Edition
Python for Data Analysis
DATA WRANGLING WITH PANDAS, NUMPY, AND IPYTHON
powered by
jupyter
Wes McKinney


David M. Beazley
Python
Essential Reference
Fourth Edition
Developer's Library


THE EXPERT'S VOICE® IN OPEN SOURCE
Covers
Python 3.1
Dive Into
Python 3
All you need to know to get off the ground
with Python 3
python
Mark Pilgrim
Foreword by Jesse Noller
Apress®

2. Software for Data Analysis

2.1. Python

- https://www.python.org/
- Very high-level data structures.
- Clean syntax.
- Eco-system (comes with batteries attached!)
- De Facto programming language of ML.
- A large collection of ML packages.
- Python 2.7, Python 3

2.2. Anaconda

- `https://anaconda.org/`
- Most popular Python data science platform.
- Leads open source projects like Anaconda, NumPy and SciPy that form the foundation of modern data science.

2.3. NumPy

Fundamental package for numeric computing with Python.

- `http://www.numpy.org/`
- A powerful N-dimensional array object.
- Sophisticated functions.

- Tools for integrating C/C++ and Fortran code.
- Useful linear algebra, Fourier transform, and random number capabilities.

2.4. SciPy

Python-based ecosystem for mathematics, science, and engineering.

Core packages:

- `https://www.scipy.org/`
- NumPy: Base N-dimensional array package.
- SciPy library: Fundamental library for scientific computing.
- Matplotlib: Comprehensive 2D Plotting.

- Enhanced Interactive Console.
- Pandas: Data structures & analysis.

2.5. Scikit-Learn

- `http://scikit-learn.org/stable/`
- Classification: SVM, Nearest neighbors, Random forests.
- Regression: SVR, Ridge regression, Lasso.
- Clustering: k-means, Spectral clustering.
- PCA, Feature selection, Non-negative matrix factorization.
- Model selection: Grid search, Cross validation, Metrics.
- Preprocessing: Feature extraction.

2.6. Pandas

- `http://pandas.pydata.org/`
- Python Data Analysis Library
- High-performance, easy-to-use data structures and data analysis tools.

3. Program Structure

```
def function_1(arguments):
  ...

def function_2(arguments):
  ...

statement_1
statement_2
...
```

- Code is read from top to bottom.
- Functions are defined, but executed only by function call statements.

4. Variables and Assignment

- Variable: Named box and content.
- We can store a value in a variable.
- We can use the value of a variable in an expression.

```
score = 20
score = score + 4
```

5. Data Types and Operators

Numeric Data

- Arithmetic operators: `+`, `-`, `*`, `/`, `**`
- Integer division: `14//3 = 4`, `14%3 == 2`
- Exponentiation: `2 ** 10 = 1024`

Boolean Data

- `True`, `False`
- Comparison operators: `<`, `<=`, `>`, `>=`, `==`, `!=`
- Logical operators: `and`, `or`, `not`

String Data

- Sequence of characters (letters, digits, etc)
- Concatenate strings
- Examples:

```
username = "Anonymous"
greeting = 'Good morning!'
greeting + username + '.'
```

6. Lists

- Sequence of values

```
fruits = ['Apple', 'Orange', 'Grape', 'Banana']
```

- Access the individual items by 0-based index

```
>>> fruits[3]
'Banana'
```

- Update an item

```
>>> fruits[2] = 'Pomegranate'
>>> fruits
['Apple, 'Orange, ''Pomegranate', 'Banana]
```

7. Control Flow

Four main control flow statements:

- Sequential statement: execute statements in the same order they appear in the text, one after another.
- Conditional statement: if a condition is true, execute one statement; otherwise, execute the other statement.
- Iterative statement: execute the statement repeatedly until a condition becomes true (or as long as a condition is true).
- Function call.

8. Loop

List Loop:

Example 8.1. Find the total score of a team (list loop).

```
scores = [20, 8, 70, 15, 0, 80, 30, 25, 5, 10, 2, 3]
total = 0
for score in scores:
  total = total + score
print (total)
```

Index Loop:

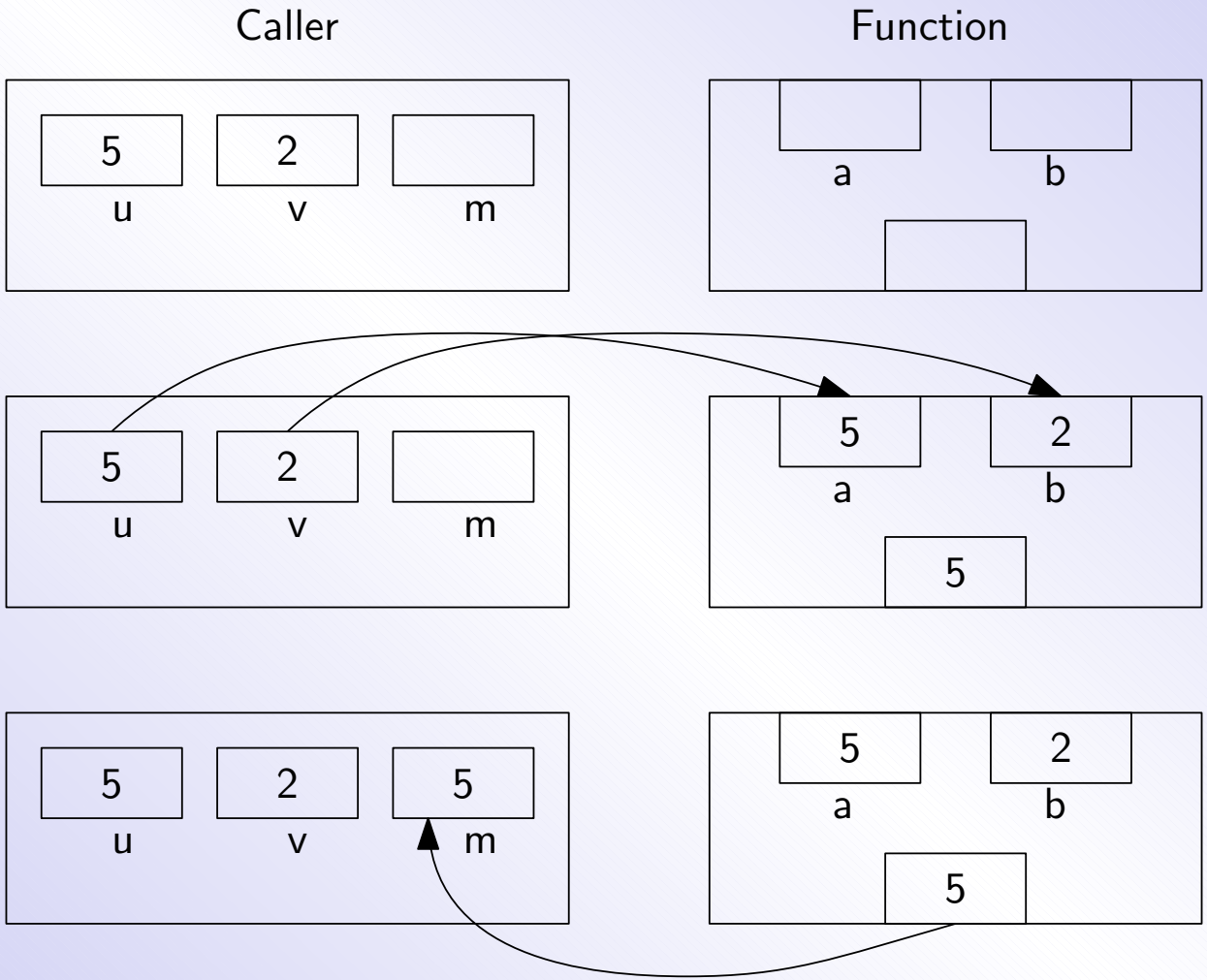
Example 8.2. Find the highest score of a team.

```
scores = [20, 8, 70, 15, 0, 80, 30, 25, 5, 10, 2, 3]
max_pos = 0
n = len(scores)
for i in range(n):
   if scores[i] > scores[max_pos]:
      max_pos = i
print (max_pos)
```

9. Functions

```
def maximum (a, b):
    big = a
    if b > big:
        big = b
    return big

u, v = 5, 2
m = maximum (u, v)
print (m)
```


Caller
Function
5
2
u
v
m
a
b
5
2
u
v
m
5
a
b
5
5
2
u
v
m
5
a
b
5

10. Parameters and Arguments

- Parameters are simple data (numbers, boolean, strings): Function may change parameters. But arguments do not change.
- Parameters are arrays (lists): If function changes array items, changes are made in the arguments also (alias).

```
scores = [20, 8, 70, 15, 0, 80, 30, 25, 5, 10, 2, 3]
def maximum (a):
    n = len(a)
    max_pos = 0
    for i in range(n):
        if scores[i] > scores[max_pos]:
            max_pos = i
    return max_pos
```

Example 10.1. Using `maximum()`, sort the items of an array.

```
def sort_array (a):
   b = []
   while a != []:
      pos = maximum (a)
      b.append(a[pos])
      del a[pos]
   return b
```

11. Files

- Input file

```
Name Shares Price
IBM 50 91.10
MicroSoft 200 51.23
Google 100 490.10
Apple 50 118.22
Yahoo 75 28.34
SCO 500 2.14
RedHat 60 23.45
```

```
f = open("portfolio.dat","r")
lines = f.readlines()
for line in lines:
    print (line, end='')
f.close()
```

```
f = open("portfolio.dat","r")
lines = f.readlines()
for line in lines:
    print (line, end='')
    fields = line.split()
    print (fields)
f.close()
```

- `open (filename, mode)`, `file.close()`
- `string.split()` $\Rightarrow$ a list of strings.

12. Lists: Ordered Data

- Indexed sequence of items.
- Access/change items

```
fields = ['IBM', 50, 91.10]
name = fields[0]
price = fields[2]
fields[1] = 75
```

- `len(list)`

```
len(fields) ⇝ 3
```

- Append/insert item

```
fields.append('2020')
fields.insert(0,'Aadhan)
# fields = ['Aadhan, 'IBM', 50, 91.10, '2020']
```

- Delete item

```
del fields[0]
# fields = ['IBM',50,91.10,'2020']
```

- Can contain mixed types.
- Can contain other lists.

```
portfolio = [ ['IBM',50,91.10],
              ['MicroSoft',200,51.23],
              ['Google,100,490.10] ]
```

```
total = 0.0
f = open("portfolio.dat","r")
lines = f.readlines()
for line in lines:
    fields = line.split()
    name = fields[0]
    shares = int(fields[1])
    price = float(fields[2])
    total += shares*price
    print (name,shares,price)
    # print ("%-10s %8d %10.2f" % (name,shares,price))
f.close()
print ("Total", total)
```

- Type conversion
- String formatting

13. Tuples: Ordered Data, Unchangeable

```
fields = ('IBM', 50, 91.10)
fields[0]
fields[1]
fields[2]
```

```
stocks = []
lines = open('portfolio.dat')
for line in lines:
    fields = line.split()
    name = fields[0]
    shares = int(fields[1])
    price = float(fields[2])
    record = (name, shares, price)
    stocks.append (record)
stocks.sort()
for s in stocks:
    print ("%-10s %8d %10.2f" % s)
```

14. List Comprehension

- Motivated by set comprehension (from math)

$$a = \{x^2 | x \in s, x > 0\}$$

- List comprehensions process list items

```
>>> x = [1, 2, 3, 4]
>>> a = [2*i for i in x]
>>> a
[2,4, 6, 8]
>>>
```

- Shorthand for this code:

```
a = []
for i in x:
    a.append(2*i)
```

- List comprehensions with a condition

```
>>> x = [1, 2, -3, 4, -5]
>>> a = [2*i for i in x if i > 0]
>>> a
[2,4, 8]
>>>
```

- Shorthand for this code:

```
a = []
for i in x:
    if i > 0:
        a.append(2*i)
```

- General form of list comprehensions

```
a = [expression for item in list if condition]
```

- Shorthand for:

```
a = []
for i in s:
    if condition:
        a.append(expression)
```

```
stocks = []
lines = open('portfolio.dat')
for line in lines:
    fields = line.split()
    name = fields[0]
    shares = int(fields[1])
    price = float(fields[2])
    record = (name, shares, price)
    stocks.append (record)
stocks.sort()
for s in stocks:
    print ("%-10s %8d %10.2f" % s)
total = sum([s[1]*s[2] for s in stocks])
```

```
print ("Total", total)
```

```
lines = open("portfolio.dat")
rows = [line.split() for line in lines]
stocks = [(fields[0], int(fields[1]), float(fields[2]))
          for fields in rows]
stocks.sort()
for s in stocks:
   print ("%-10s %8d %10.2f" % s)
total = sum([s[1]*s[2] for s in stocks])
print ("Total", total)
```

15. Dictionaries: Unordered Data

- A collection named fields (key-value pairs).

```
stock = {
  'name' : 'Google',
  'shares' : 100,
  'price' : 490.10
}
cost = stock['shares'] * stock['price']
```

```
prices = {
  'IBM' : 117.88,
  'MicroSoft : 28.48,
  'GE' : 38.75,
  'CAT' : 75.54,
  'Google' : 527.80
}
```

- Getting an item

```
x = prices['IBM']
y = prices.get('IBM',0.0) # w/default if not found
```

- Adding or modifying an item

```
prices['Apple] = 145.14
```

- Deleting an item

```
del prices['SCO']
```

- Membership test (in operator)

```
if 'Google' in prices:
   x = prices['Google]
```

- Create a list of dictionaries.

```
stocks = []
f = open("portfolio.dat"):
for line in f.readlines():
    fields = line.split()
    record = {
        'name' : fields[0],
        'shares' : int(fields[1]),
        'price' : float(fields[2])
    }
    stocks.append(record)
f.close()
```

- Create a dictionary of current prices

```
prices = {}
for line in open("prices.dat"):
    fields = line.split(',')
    prices[fields[0]] = float(fields[1])
```

```
prices = {}
for line in open("prices.dat"):
    fields = line.split(',')
    prices[fields[0]] = float(fields[1])

initial = sum([s['shares']*s['price']
               for s in stocks])

current = sum([s['shares']*prices[s['name']]
                  for s in stocks])

print ("Current value", current)
print ("Gain", current - initial)
```

16. NumPy

- NumPy is the core library for scientific computing in Python.
- It provides a high-performance multidimensional array object, and tools for working with these arrays.
- To use NumPy, first import the numpy package:

```
import numpy as np
```

16.1. Arrays

- A numpy array is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers.

- Rank of the array is the number of dimensions.
- Shape of an array is a tuple of integers giving the size of the array along each dimension.
- We can initialize numpy arrays from nested Python lists, and access elements using square brackets.

```
a = np.array([1, 2, 3]) # Create a rank 1 array
print (type(a), a.shape, a[0], a[1], a[2])
a[0] = 5 # Change an element of the array
print (a)
```

```
b = np.array([[1,2,3],[4,5,6]]) # Create a rank 2 array
print (b)
print (b.shape)
print (b[0, 0], b[0, 1], b[1, 0])
```

Numpy also provides many functions to create arrays:

```
a = np.zeros((2,2)) # Create an array of all zeros
print (a)
b = np.ones((1,2)) # Create an array of all ones
print (b)
c = np.full((2,2), 7) # Create a constant array
print (c)
d = np.eye(2) # Create a 2x2 identity matrix
print (d)
e = np.random.random((2,2)) # An array filled with random val
print (e)
```

16.2. Array indexing

- NumPy offers several ways to index into arrays.
- Slice Indexing: Similar to Python lists, numpy arrays can be sliced. Since arrays may be multidimensional, you must specify a slice for each dimension of the array:

```
import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1 2 3 4]
# [ 5 6 7 8]
# [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
```

```
# Pull out the subarray consisting of
# the first 2 rows and columns 1 and 2;
# b is an array of shape (2, 2):
# [[2 3]
# [6 7]]
print (a)
b = a[:2, 1:3]
print (b)
```

A slice of an array is a view into the same data, so modifying it will modify the original array.

```
print (a[0, 1])
b[0, 0] = 77 # b[0, 0] is the same piece of data as a[0, 1]
print (a[0, 1])
```

We can also mix integer indexing with slice indexing. However, doing so will yield an array of lower rank than the original array.

```
# Create the following rank 2 array with shape (3, 4)
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
print (a)
```

Two ways of accessing the data in the middle row of the array.

- Mixing integer indexing with slices yields an array of lower rank
- while using only slices yields an array of the same rank as the original array:

```
row_r1 = a[1, :] # Rank 1 view of the row 1 of a
row_r2 = a[1:2, :] # Rank 2 view of the row 1 of a
row_r3 = a[[1], :] # Rank 2 view of the row 1 of a
```

```
print (row_r1, row_r1.shape)
print (row_r2, row_r2.shape)
print (row_r3, row_r3.shape)
```

We can make the same distinction when accessing columns of an array:

```
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print (col_r1, col_r1.shape)
print (col_r2, col_r2.shape)
```

Integer array indexing: When you index into numpy arrays using slicing, the resulting array view will always be a subarray of the original array. In contrast, integer array indexing allows you to construct arbitrary arrays using the data from another array.

```
a = np.array([[1,2], [3, 4], [5, 6]])

# integer array indexing.
# The returned array will have shape (3,)
print (a[[0, 1, 2], [0, 1, 0]])

# equivalent to this:
print (np.array([a[0, 0], a[1, 1], a[2, 0]]))
```

When using integer array indexing, you can reuse the same element

from the source array:

```
print (a[[0, 0], [1, 1]])
```

Equivalent to the previous integer array indexing example

```
print (np.array([a[0, 1], a[0, 1]]))
```

One useful trick with integer array indexing is selecting or mutating one element from each row of a matrix:

```
# Create a new array from which we will select elements
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
print (a)
# Create an array of indices
b = np.array([0, 2, 0, 1])

# Select one element from each row of a using indices in b
print (a[np.arange(4), b]) # Prints "[ 1 6 7 11]"

# Mutate one element from each row of a using indices in b
a[np.arange(4), b] += 10
print (a)
```

Boolean array indexing lets you pick out arbitrary elements of an array. This type of indexing is used to select the elements of an array that satisfy some condition.

```
import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]])

# Find the elements of a that are bigger than 2;
# this is an array of Booleans of the same shape as a,
# where each slot of bool_idx tells
# whether that element of a is > 2.
bool_idx = (a > 2)
print (bool_idx)
```

We use boolean array indexing to construct a rank 1 array consisting of the elements of a corresponding to the True values of `bool_idx`

```
print (a[bool_idx])
```

We can do all of the above in a single concise statement:

```
print (a[a > 2])
```

16.3. Datatypes

- Every numpy array is a grid of elements of the same type.
- Numpy provides a large set of numeric datatypes that we can use to construct arrays.
- Numpy tries to guess a datatype when you create an array, but

functions that construct arrays usually also include an optional argument to explicitly specify the datatype.

```
x = np.array([1, 2]) # Let numpy choose the datatype
y = np.array([1.0, 2.0]) # Let numpy choose the datatype
z = np.array([1, 2], dtype=np.int64) # Force a datatype

print (x.dtype, y.dtype, z.dtype)
```

16.4. Array math

Basic mathematical functions operate elementwise on arrays, and are available both as operator overloads and as functions in the numpy module:

```
x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce the array
print (x + y)
print (np.add(x, y))

# Elementwise difference; both produce the array
print (x - y)
print (np.subtract(x, y))

# Elementwise product; both produce the array
print (x * y)
```

```
print (np.multiply(x, y))

# Elementwise division; both produce the array
# [[ 0.2 0.33333333]
# [ 0.42857143 0.5 ]]
print (x / y)
print (np.divide(x, y))

# Elementwise square root; produces the array
# [[ 1. 1.41421356]
# [ 1.73205081 2. ]]
print (np.sqrt(x))
```

Note that * is elementwise multiplication, not matrix multiplication.

We instead use the dot function to compute inner products of vectors, to multiply a vector by a matrix, and to multiply matrices. dot is available both as a function in the numpy module and as an instance method of array objects:

```
x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])

# Inner product of vectors; both produce 219
print (v.dot(w))
print (np.dot(v, w))
```

```
# Matrix/vector product;
# both produce the rank 1 array [29 67]
print (x.dot(v))
print (np.dot(x, v))

# Matrix/matrix product;
# both produce the rank 2 array
# [[19 22]
# [43 50]]
print (x.dot(y))
print (np.dot(x, y))
```

Numpy provides many useful functions for performing computations on arrays; one of the most useful is ‘sum‘:

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

print (np.sum(x)) # sum of all elements; prints "10"
print (np.sum(x, axis=0)) # sum of each column; prints "[4 6]
print (np.sum(x, axis=1)) # sum of each row; prints "[3 7]"
```

Find the full list of mathematical functions provided by numpy in the [documentation](http://docs.scipy.org/doc/numpy/reference/routines.mat

Apart from computing mathematical functions using arrays, we frequently need to reshape or otherwise manipulate data in arrays. The simplest example of this type of operation is transposing a matrix; to transpose a matrix, simply use the T attribute of an array object:

```
print (x)
print (x.T)

v = np.array([[1,2,3]])
print (v)
print (v.T)
```

16.5. Broadcasting

- Broadcasting is a powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations.
- Frequently we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array.
- Suppose that we want to add a constant vector to each row of a matrix.

```
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = np.empty_like(x) # empty matrix with the same shape as x

# Add vector v to each row of the matrix x
# with an explicit loop
for i in range(4):
    y[i, :] = x[i, :] + v
print (y)
```

This works; however when the matrix ‘x‘ is very large, computing an explicit loop in Python could be slow.

Adding vector v to each row of the matrix x is equivalent to forming a matrix vv by stacking multiple copies of v vertically, then performing elementwise summation of x and vv.

```
# Stack 4 copies of v on top of each other
vv = np.tile(v, (4, 1))
print (vv) # Prints "[[1 0 1]
                     # [1 0 1]
                     # [1 0 1]
                     # [1 0 1]]"

y = x + vv # Add x and vv elementwise
print (y)
```

Numpy broadcasting allows us to perform this computation without actually creating multiple copies of v.

```
import numpy as np

# Add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v # Add v to each row of x using broadcasting
print (y)
```

y = x + v works even though x has shape (4, 3) and v has shape (3,) as if v actually had shape (4, 3), where each row was a copy of v, and the sum was performed elementwise.

Broadcasting two arrays together follows these rules:

1. If the arrays do not have the same rank, prepend the shape of the lower rank array with 1s until both shapes have the same length.
2. The two arrays are said to be compatible in a dimension if they have the same size in the dimension, or if one of the arrays has size 1 in that dimension.
3. The arrays can be broadcast together if they are compatible in all dimensions.
4. After broadcasting, each array behaves as if it had shape equal to the elementwise maximum of shapes of the two input arrays.
5. In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension

Functions that support broadcasting are known as universal functions. You can find the list of all universal functions in the [documentation](http://
ufuncs).

Here are some applications of broadcasting:

```
# Compute outer product of vectors
v = np.array([1,2,3]) # v has shape (3,)
w = np.array([4,5]) # w has shape (2,)

# first reshape v to be a column vector of shape (3, 1)
# then broadcast it against w to yield
# an output of shape (3, 2),
# which is the outer product of v and w:
print (np.reshape(v, (3, 1)) * w)
```

```
# Add a vector to each row of a matrix
x = np.array([[1,2,3], [4,5,6]])
# x shape (2, 3), v shape (3,), they broadcast to (2, 3)
print (x + v)
```

Add a vector to each column of a matrix x has shape (2, 3) and w has shape (2,). If we transpose x then it has shape (3, 2) and can be broadcast against w to yield a result of shape (3, 2); transposing this result yields the final result of shape (2, 3) which is the matrix x with the vector w added to each column. Gives the following matrix:

```
print ((x.T + w).T)
```

Another solution is to reshape w to be a row vector of shape (2, 1); we can then broadcast it directly against x to produce the same

output.

```
print (x + np.reshape(w, (2, 1)))
```

Multiply a matrix by a constant: x has shape (2, 3). Numpy treats scalars as arrays of shape (); these can be broadcast together to shape (2, 3), producing the following array:

```
print (x * 2)
```

Broadcasting typically makes your code more concise and faster, so you should strive to use it where possible.

17. Matplotlib

- Matplotlib is a plotting library.
- `matplotlib.pyplot` module provides a plotting system.
- By running this special iPython command, we can display plots inline:

```
import matplotlib.pyplot as plt
get_ipython().run_line_magic('matplotlib', 'inline')
```

17.1. Plotting

The most important function in ‘matplotlib‘ is plot, which allows you to plot 2D data.

```
# Compute the x and y coordinates for points on a sine curve
x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)

# Plot the points using matplotlib
plt.plot(x, y)
```

We can easily plot multiple lines at once, and add a title, legend, and axis labels:

```
y_sin = np.sin(x)
y_cos = np.cos(x)

# Plot the points using matplotlib
plt.plot(x, y_sin)
```

```
plt.plot(x, y_cos)
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.title('Sine and Cosine')
plt.legend(['Sine', 'Cosine'])
```

17.2. Subplots

We can plot different things in the same figure using the subplot function.

```
# Compute the x and y coordinates for points on sine and cosi
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
```

```
y_cos = np.cos(x)

# Set up a subplot grid that has height 2 and width 1,
# and set the first such subplot as active.
plt.subplot(2, 1, 1)

# Make the first plot
plt.plot(x, y_sin)
plt.title('Sine')

# Set the second subplot as active, and make the second plot
plt.subplot(2, 1, 2)
plt.plot(x, y_cos)
plt.title('Cosine')
```

```
# Show the figure.
plt.show()
```