**Introduction**

Data science is an interdisciplinary field encompassing scientific methods, processes, and systems to extract knowledge or insights from data in various forms, either structured or unstructured. It draws principles from mathematics, statistics, information science, computer science, machine learning, visualization, data mining, and predictive analytics. However, it is fundamentally grounded in mathematics.

This book explains and applies the fundamentals of data science crucial for technical professionals such as DBAs and developers who are making career moves toward practicing data science. It is an example-driven book providing complete Python coding examples to complement and clarify data science concepts, and enrich the learning experience. Coding examples include visualizations whenever appropriate. The book is a necessary precursor to applying and implementing machine learning algorithms, because it introduces the reader to foundational principles of the science of data.

The book is self-contained. All the math, statistics, stochastic, and programming skills required to master the content are covered in the book. In-depth knowledge of object-oriented programming isn’t required, because working and complete examples are provided and explained. The examples are in-depth and complex when necessary to ensure the acquisition of appropriate data science acumen. The book helps you to build the foundational skills necessary to work with and understand complex data science algorithms.

*Data Science Fundamentals by Example* is an excellent starting point for those interested in pursuing a career in data science. Like any science, the fundamentals of data science are prerequisite to competency. Without proficiency in mathematics, statistics, data manipulation, and coding, the path to success is “rocky” at best. The coding examples in this book are concise, accurate, and complete, and perfectly complement the data science concepts introduced.

The book is organized into six chapters. Chapter 1 introduces the programming fundamentals with “Python” necessary to work with, transform, and process data for data science applications. Chapter 2 introduces Monte Carlo simulation for decision making, and data distributions for statistical processing. Chapter 3 introduces linear algebra applied with vectors and matrices. Chapter 4 introduces the gradient descent algorithm that minimizes (or maximizes) functions, which is very important because most data science problems are optimization problems. Chapter 5 focuses on munging, cleaning, and transforming data for solving data science problems. Chapter 6 focusing on exploring data by dimensionality reduction, web scraping, and working with large data sets efficiently.  
  
**Python Fundamentals**

Python has several features that make it well suited for learning and doing data science. It’s free, relatively simple to code, easy to understand, and has many useful libraries to facilitate data science problem solving. It also allows quick prototyping of virtually any data science scenario and demonstration of data science concepts in a clear, easy to understand manner.

The goal of this chapter is not to teach Python as a whole, but present, explain, and clarify fundamental features of the language (such as logic, data structures, and libraries) that help prototype, apply, and/or solve data science problems.

Python fundamentals are covered with a wide spectrum of activities with associated coding examples as follows:

1. functions and strings
2. lists, tuples, and dictionaries
3. reading and writing data
4. list comprehension
5. generators
6. data randomization
7. MongoDB and JSON
8. visualization

*Functions and Strings*

Python functions are first-class functions, which means they can be used as parameters, a return value, assigned to variable, and stored in data structures. Simply, functions work like a typical variable. Functions can be either custom or built-in. Custom are created by the programmer, while built-in are part of the language. Strings are very popular types enclosed in either single or double quotes.

The following code example defines custom functions and uses built-in ones:  
  
def num\_to\_str(n):

return str(n)

def str\_to\_int(s):

return int(s)

def str\_to\_float(f):

return float(f)

if \_\_name\_\_ == "\_\_main\_\_":

# hash symbol allows single-line comments

'''

triple quotes allow multi-line comments

'''

float\_num = 999.01

int\_num = 87

float\_str = '23.09'

int\_str = '19'

string = 'how now brown cow'

s\_float = num\_to\_str(float\_num)

s\_int = num\_to\_str(int\_num)

i\_str = str\_to\_int(int\_str)

f\_str = str\_to\_float(float\_str)

print (s\_float, 'is', type(s\_float))

print (s\_int, 'is', type(s\_int))

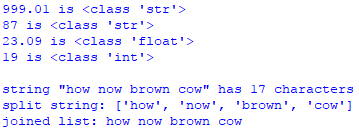
print (f\_str, 'is', type(f\_str))

print (i\_str, 'is', type(i\_str))

print ('\nstring', '"' + string + '" has', len(string), 'characters')

str\_ls = string.split()

print ('split string:', str\_ls)

print ('joined list:', ' '.join(str\_ls))  
  
Output:  
  


A popular coding style is to present library importation and functions first followed by the main block of code. The code example begins with three custom functions that convert numbers to strings, strings to numbers, and strings to float respectively. Each custom function returns a built-in function to let Python do the conversion. The main block begins with comments. Single-line comments are denoted with the # (hash) symbol. Multi-line comments are denoted with three consecutive single quotes. The next five lines assign values to variables. The following four lines convert each variable type to another type. For instance, function num\_to\_str() converts variable float\_num to string type. The next five lines print variables with their associated Python data type. Built-in function type() returns type of given object. The remaining four lines print and manipulate a string variable.

*Lists, Tuples, and Dictionaries*  
  
Lists are ordered collections with comma-separated values between square brackets. Indices start at 0 (zero). Lists items need not be of the same type and can be sliced, concatenated, and manipulated in many ways.

The following code example creates a list, manipulates and slices it, creates a new list and adds elements to it from another list, and creates a matrix from two lists:  
  
import numpy as np

if \_\_name\_\_ == "\_\_main\_\_":

ls = ['orange', 'banana', 10, 'leaf', 77.009, 'tree', 'cat']

print ('list length:', len(ls), 'items')

print ('cat count:', ls.count('cat'), ',', 'cat index:', ls.index('cat'))

print ('\nmanipulate list:')

cat = ls.pop(6)

print ('cat:', cat, ', list:', ls)

ls.insert(0, 'cat')

ls.append(99)

print (ls)

ls[7] = '11'

print (ls)

ls.pop(1)

print (ls)

ls.pop()

print (ls)

print ('\nslice list:')

print ('1st 3 elements:', ls[:3])

print ('last 3 elements:', ls[3:])

print ('start at 2nd to index 5:', ls[1:5])

print ('start 3 from end to end of list:', ls[-3:])

print ('start from 2nd to next to end of list:', ls[1:-1])

print ('\ncreate new list from another list:')

print ('list:', ls)

fruit = ['orange']

more\_fruit = ['apple', 'kiwi', 'pear']

fruit.append(more\_fruit)

print ('appended:', fruit)

fruit.pop(1)

fruit.extend(more\_fruit)

print ('extended:', fruit)

a, b = fruit[2], fruit[1]

print ('slices:', a, b)

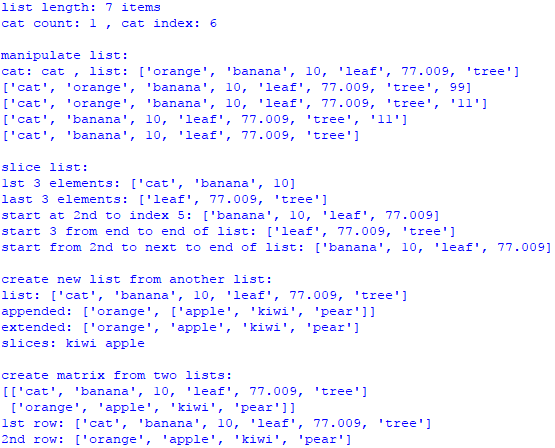
print ('\ncreate matrix from two lists:')

matrix = np.array([ls, fruit])

print (matrix)

print ('1st row:', matrix[0])

print ('2nd row:', matrix[1])  
  
Output:



The code example begins by importing NumPy, which is the fundamental package (library, module) for scientific computing. It is useful for linear algebra, which is fundamental to data science. Think of Python libraries as giant classes with many methods. The main block begins by creating list ls, printing its length, number of elements (items), number of cat elements, and index of the cat element. The code continues by manipulating ls. First, the 7th element (index 6) is popped and assigned to variable cat. Remember, list indices start at 0. Function pop() removes cat from ls. Second, cat is added back to ls at the 1st position (index 0) and 99 is appended to the end of the list. Function append() adds an object to the end of a list. Third, string ‘11’ is substituted for the 8th element (index 7). Finally, the 2nd element and the last element are popped from ls. The code continues by slicing ls. First, print the 1st three elements with ls[:3]. Second, print the last three elements with ls[3:]. Third, print starting with the 2nd element to elements with indices up to 5 with ls[1:5]. Fourth, print starting three elements from the end to the end with ls[-3:]. Fifth, print starting from the 2nd element to next to the last element with ls[1:-1]. The code continues by creating a new list from another. First, create fruit with one element. Second append list more\_fruit to fruit. Notice that append adds list more\_fruit as the 2nd element of fruit, which may not be what you want. So, third, pop 2nd element of fruit and extend more\_fruit to fruit. Function extend() unravels a list before it adds it. This way, fruit now has four elements. Fourth, assign 3rd element to a and 2nd element to b and print slices. Python allows assignment of multiple variables on one line, which is very convenient and concise. The code ends by creating a matrix from two lists – ls and fruit and printing it. A Python matrix is a two-dimensional array consisting of rows and columns, where each row is a list.

A tuple is a sequence of immutable Python objects enclosed by parentheses. Unlike lists, tuples cannot be changed. Tuples are convenient with functions that return multiple values.

The following code example creates a tuple, slices it, creates a list, and creates a matrix from tuple and list:

import numpy as np

if \_\_name\_\_ == "\_\_main\_\_":

tup = ('orange', 'banana', 'grape', 'apple', 'grape')

print ('tuple length:', len(tup))

print ('grape count:', tup.count('grape'))

print ('\nslice tuple:')

print ('1st 3 elements:', tup[:3])

print ('last 3 elements', tup[3:])

print ('start at 2nd to index 5', tup[1:5])

print ('start 3 from end to end of tuple:', tup[-3:])

print ('start from 2nd to next to end of tuple:', tup[1:-1])

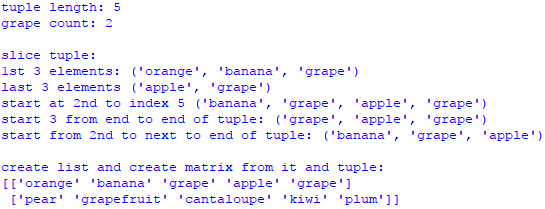
print ('\ncreate list and create matrix from it and tuple:')

fruit = ['pear', 'grapefruit', 'cantaloupe', 'kiwi', 'plum']

matrix = np.array([tup, fruit])

print (matrix)

Output:



The code begins by importing NumPy. The main block begins by creating tuple tup, printing its length, number of elements (items), number of grape elements, and index of grape. The code continues by slicing tup. The code continues by slicing tup. First, print the 1st three elements with tup[:3]. Second, print the last three elements with tup[3:]. Third, print starting with the 2nd element to elements with indices up to 5 with tup[1:5] . Fourth, print starting three elements from the end to the end with tup[-3:]. Fifth, print starting from the 2nd element to next to the last element with tup[1:-1]. The code continues by creating a new fruit and creating a matrix from tup and fruit.

A dictionary is an unordered collection of items identified by a key/value pair. It is an extremely important data structure for working with data. The following example is very simple, but the next section presents a more complex example based on a dataset.

The following code example creates a dictionary, deletes an element, adds an element, creates a list of dictionary elements, and traverses the list:

if \_\_name\_\_ == "\_\_main\_\_":

audio = {'amp':'Linn', 'preamp':'Luxman', 'speakers':'Energy',

'ic':'Crystal Ultra', 'pc':'JPS', 'power':'Equi-Tech',

'sp':'Crystal Ultra', 'cdp':'Nagra', 'up':'Esoteric'}

del audio['up']

print ('dict "deleted" element;')

print (audio, '\n')

print ('dict "added" element;')

audio['up'] = 'Oppo'

print (audio, '\n')

print ('universal player:', audio['up'], '\n')

dict\_ls = [audio]

video = {'tv':'LG 65C7 OLED', 'stp':'DISH', 'HDMI':'DH Labs',

'cable' : 'coax'}

print ('list of dict elements;')

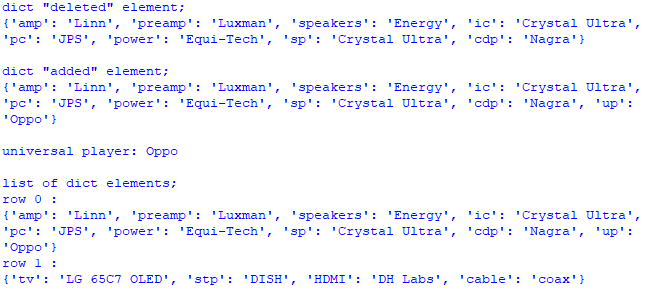
dict\_ls.append(video)

for i, row in enumerate(dict\_ls):

print ('row', i, ':')

print (row)

Output:



The main block begins by creating dictionary audio with several elements. It continues by deleting an element with key up and value Esoteric, and displaying. Next, a new element with key up and element Oppo is added back, and displayed. The next part creates a list with dictionary audio, creates dictionary video, and adds the new dictionary to the list. The final part uses a for loop to traverse the dictionary list and display the two dictionaries. A very useful function that can be used with a loop statement is enumerate(). It adds a counter to an iterable. An iterable is an object that can be iterated. Function enumerate() is very useful because a counter is automatically created and incremented, which means less code.

*Reading and Writing Data*

The ability to read and write data is fundamental to any data science endeavor. The most basic types of data are text and CSV. So, this is where we will start.

The following code example reads a text file and cleans it for processing. It then reads the pre-cleansed text file, saves it as a CSV file, reads the CSV file, converts it to a list of OrderedDict elements, and converts this list to a list of regular dictionary elements:

import csv

def read\_txt(f):

with open(f, 'r') as f:

d = f.readlines()

return [x.strip() for x in d]

def conv\_csv(t, c):

data = read\_txt(t)

with open(c, 'w', newline='') as csv\_file:

writer = csv.writer(csv\_file)

for line in data:

ls = line.split()

writer.writerow(ls)

def read\_csv(f):

contents = ''

with open(f, 'r') as f:

reader = csv.reader(f)

return list(reader)

def read\_dict(f, h):

input\_file = csv.DictReader(open(f), fieldnames=h)

return input\_file

def od\_to\_d(od):

return dict(od)

if \_\_name\_\_ == "\_\_main\_\_":

f = 'data/names.txt'

data = read\_txt(f)

print ('text file data sample:')

for i, row in enumerate(data):

if i < 3:

print (row)

csv\_f = 'data/names.csv'

conv\_csv(f, csv\_f)

r\_csv = read\_csv(csv\_f)

print ('\ntext to csv sample:')

for i, row in enumerate(r\_csv):

if i < 3:

print (row)

headers = ['first', 'last']

r\_dict = read\_dict(csv\_f, headers)

dict\_ls = []

print ('\ncsv to ordered dict sample:')

for i, row in enumerate(r\_dict):

r = od\_to\_d(row)

dict\_ls.append(r)

if i < 3:

print (row)

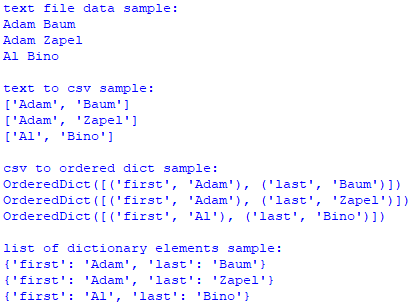
print ('\nlist of dictionary elements sample:')

for i, row in enumerate(dict\_ls):

if i < 3:

print (row)

Output:



The code begins by importing the csv library, which implements classes to read and write tabular data in CSV (Comma Separated Values) format. It continues with five functions. Function read\_txt() reads a text (.txt) file and strips (removes) extraneous characters with list comprehension, which is an elegant way to define and create a list in Python. List comprehension is covered later in the next section. Function conv\_csv converts a text to a CSV file and saves it to disk. Function read\_csv() reads a CSV file and returns it as a list. Function read\_dict() reads a CSV file and returns a list of OrderedDict elements. An OrderedDict is a dictionary subclass that remembers the order in which its contents are added, whereas a regular dictionary doesn’t track insertion order. Finally, function od\_to\_d() converts an OrderedDict element to a regular dictionary element. Working with a regular dictionary element is much more intuitive in my opinion. The main block begins by reading a text file and cleaning it for processing. However, no processing is done with this cleansed file in the code. It is only included in case you want to know how to accomplish this task. The code continues by converting a text file to CSV, which is saved to disk. The CSV file is then read from disk and a few records are displayed. Next, a headers list is created to store keys for a dictionary yet to be created. List dict\_ls is created to hold dictionary element. The code continues by creating an OrderedDict list r\_dict. The OrderedDict list is then iterated so that each element can be converted to a regular dictionary element and appended to dict\_ls. A few records are displayed during iteration. Finally, dict\_ls is iterated and a few records are displayed. I highly recommend that you take some time to familiarize yourself with these data structures as they are used extensively in data science application.

*List Comprehension*

List comprehension provides a concise way to create lists. Its logic is enclosed in square brackets that contain an expression followed by a for clause and can be augmented by more for or if clauses.

The read\_txt() function in the previous section included the following list comprehension:

[x.strip() for x in d]  
  
The logic strips extraneous characters from string in iterable d. In this case, d is a list of strings.

The following code example converts miles to kilometers, manipulates pets, and calculates bonuses with list comprehension:

if \_\_name\_\_ == "\_\_main\_\_":

miles = [100, 10, 9.5, 1000, 30]

kilometers = [x \* 1.60934 for x in miles]

print ('miles to kilometers:')

for i, row in enumerate(kilometers):

print ('{:>4} {:>8}{:>8} {:>2}'.

format(miles[i],'miles is', round(row,2), 'km'))

print ('\npet:')

pet = ['cat', 'dog', 'rabbit', 'parrot', 'guinea pig', 'fish']

print (pet)

print ('\npets:')

pets = [x + 's' if x != 'fish' else x for x in pet]

print (pets)

subset = [x for x in pets if x != 'fish' and x != 'rabbits'

and x != 'parrots' and x != 'guinea pigs']

print ('\nmost common pets:')

print (subset[1], 'and', subset[0])

sales = [9000, 20000, 50000, 100000]

print ('\nbonuses:')

bonus = [0 if x < 10000 else x \* .02 if x >= 10000 and x <= 20000

else x \* .03 for x in sales]

print (bonus)

print ('\nbonus dict:')

people = ['dave', 'sue', 'al', 'sukki']

d = {}

for i, row in enumerate(people):

d[row] = bonus[i]

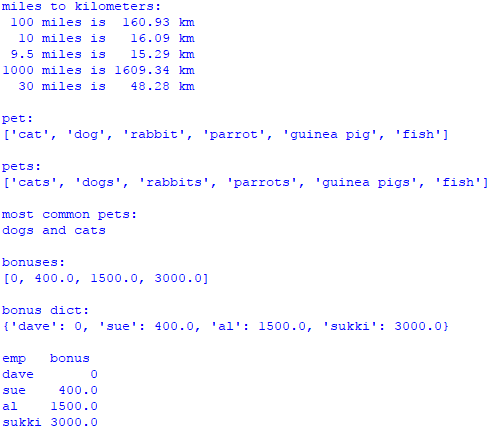
print (d, '\n')

print ('{:<5} {:<5}'.format('emp', 'bonus'))

for k, y in d.items():

print ('{:<5} {:>6}'.format(k, y))

Output:



The main block begins by creating two lists – miles and kilometers. The kilometers list is created with list comprehension, which multiplies each mile value by 1.60934. At first, list comprehension may seem confusing, but practice makes it easier over time. The main block continues by printing miles and associated kilometers. Function format() provides sophisticated formatting options. Each mile value is ({:>4}) with up to four characters right justified. Each string for miles and kilometers is right justified ({:>8}) with up to eight characters. Finally, each string for km is right justified ({:>2}) with up to two characters. This may seem a bit complicated at first, but it is really quite logical (and elegant) once you get used to it. The main block continues by creating a pet and pets list. The pets list is created with list comprehension, which makes a pet plural if it is not a fish. I advise you to study this list comprehension before you go forward because they just get more complex. The code continues by creating a subset list with list comprehension, which only includes dogs and cats. The next part creates two lists – sales and bonus. Bonus is created with list comprehension that calculates bonus for each sales value. If sales are less than 10,000 no bonus is paid. If sales are between 10,000 and 20,000 (inclusive) the bonus is 2% of sales. Finally, if sales if greater than 20,000, the bonus is 3% of sales. At first I was confused with this list comprehension but it makes sense to me now. So, try some of your own and you will get the gist of it. The final part creates a people list to associate with each sales value, continues by creating a dictionary to hold bonus for each person, and ends by iterating dictionary elements. The formatting is quite elegant. The header left justifies emp and bonus properly. Each item is formatted so that the person is left justified with up to five characters ({:<5}) and the bonus is right justified with up to six characters ({:>6}).

*Generators*

A generator is a special type of iterator, but much faster because values are only produced as needed. This process is known as lazy (or deferred) evaluation. Typical iterators are much slower because they are fully built into memory. While regular functions return values, generators yield them. The best way to traverse and access values from a generator is to use a loop. Finally, a list comprehension can be converted to a generator by replacing square brackets with parentheses.

The following code example reads a CSV file and creates a list of OrderedDict elements. It then converts the list elements into regular dictionary elements. The code continues by simulating times for list comprehension, generator comprehension, and generators. During simulation, a list of times for each is created. Simulation is the imitation of a real-world process or system over time, and it is used extensively in data science.

import csv, time, numpy as np

def read\_dict(f, h):

input\_file = csv.DictReader(open(f), fieldnames=h)

return (input\_file)

def conv\_reg\_dict(d):

return [dict(x) for x in d]

def sim\_times(d, n):

i = 0

lsd, lsgc = [], []

while i < n:

start = time.clock()

[x for x in d]

time\_d = time.clock() - start

lsd.append(time\_d)

start = time.clock()

(x for x in d)

time\_gc = time.clock() - start

lsgc.append(time\_gc)

i += 1

return (lsd, lsgc)

def gen(d):

yield (x for x in d)

def sim\_gen(d, n):

i = 0

lsg = []

generator = gen(d)

while i < n:

start = time.clock()

for row in generator:

None

time\_g = time.clock() - start

lsg.append(time\_g)

i += 1

generator = gen(d)

return lsg

def avg\_ls(ls):

return np.mean(ls)

if \_\_name\_\_ == '\_\_main\_\_':

f = 'data/names.csv'

headers = ['first', 'last']

r\_dict = read\_dict(f, headers)

dict\_ls = conv\_reg\_dict(r\_dict)

n = 1000

ls\_times, gc\_times = sim\_times(dict\_ls, n)

g\_times = sim\_gen(dict\_ls, n)

avg\_ls = np.mean(ls\_times)

avg\_gc = np.mean(gc\_times)

avg\_g = np.mean(g\_times)

gc\_ls = round((avg\_ls / avg\_gc), 2)

g\_ls = round((avg\_ls / avg\_g), 2)

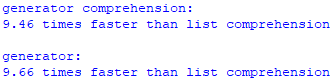
print ('generator comprehension:')

print (gc\_ls, 'times faster than list comprehension\n')

print ('generator:')

print (g\_ls, 'times faster than list comprehension')

Output:



The code begins by importing csv, time, and numpy libraries. Function read\_dict() converts a CSV (.csv) file to a list of OrderedDict elements. Function conv\_reg\_dict() converts a list of OrderedDict elements to a list of regular dictionary elements (for easier processing). Function sim\_times() runs a simulation that creates two lists – lsd and lsgc. List lsd contains n run times for list comprension and list lsgc contains n run times for generator comprehension. Using simulation provides a more accurate picture of the true time it takes for both of these processes by running them over and over (n times). In this case, the simulation is run 1000 times (n =1000). Of course, you can run the simulations as many or few times as you wish. Functions gen() and sim\_gen() work together. Function gen() creates a generator. Function sim\_gen() simulates the generator n times. I had to create these two functions because yielding a generator requires a different process than creating a generator comprehension. Function avg\_ls() returns the mean (average) of a list of numbers. The main block begins by reading a CSV file (the one we created earlier in the chapter) into a list of OrderedDict elements, and converting it to a list of regular dictionary elements. The code continues by simulating run times of list comprehension and generator comprehension 1000 times (n=1000). The 1st simulation calculates 1000 runtimes for traversing the dictionary list created earlier for both list and generator comprehension, and returns a list of those runtimes for each. The 2nd simulation calculates 1000 runtimes for traversing the dictionary list for a generator, and returns a list of those runtimes. The code concludes by calculating the average runtime for each of the three techniques – list comprehension, generator comprehension, and generators, and comparing those averages.

The simulations verify that generator comprehension is more than ten times, and generators are more than eight times faster than list comprehension. This makes sense because list comprehension stores all data in memory, while generators evaluate (lazily) as data is needed. Naturally, the speed advantage of generators because more important with big data sets. Without simulation, runtimes **cannot** be verified because we are randomly getting internal system clock times.

*Data Randomization*

A stochastic process is a family of random variables from some probability space into a state space (whew!). Simply, it is a random process through time. Data randomization is the process of selecting values from a sample in an unpredictable manner with the goal of simulating reality. Simulation allows application of data randomization in data science. The previous section demonstrated how simulation can be used to realistically compare iterables (list comprehension, generator comprehension, and generators).

In Python, pseudo-random numbers are used to simulate data randomness (reality). They are not truly random because the 1st generation has no previous number. We have to provide a seed (or random seed) to initialize a pseudo-random number generator. The random library implements pseudo-random number generators for various data distributions, and random.seed() is used to generate the initial (1st generation) seed number.

The following code example reads a CSV file and converts it to a list of regular dictionary elements. The code continues by creating a random number used to retrieve a random element from the list. Next, a generator of three randomly selected elements is created and displayed. The code continues by displaying three randomly shuffled elements from the list. The next section of code deterministically seeds the random number generator, which means that all generated random numbers will be the same based on the seed. So, the elements displayed will always be the same ones unless the seed is changed. The code then uses the system’s time to non-deterministically generate random numbers and display those three elements. Next, non-deterministic random numbers are generated by another method and those three elements are displayed. The final part creates a names list so random choice and sampling methods can be used to display elements.

import csv, random, time

def read\_dict(f, h):

input\_file = csv.DictReader(open(f), fieldnames=h)

return (input\_file)

def conv\_reg\_dict(d):

return [dict(x) for x in d]

def r\_inds(ls, n):

length = len(ls) - 1

yield [random.randrange(length) for \_ in range(n)]

def get\_slice(ls, n):

return ls[:n]

def p\_line():

print ()

if \_\_name\_\_ == '\_\_main\_\_':

f = 'data/names.csv'

headers = ['first', 'last']

r\_dict = read\_dict(f, headers)

dict\_ls = conv\_reg\_dict(r\_dict)

n = len(dict\_ls)

r = random.randrange(0, n-1)

print ('randomly selected index:', r)

print ('randomly selected element:', dict\_ls[r])

elements = 3

generator = next(r\_inds(dict\_ls, elements))

p\_line()

print (elements, 'randomly generated indicies:', generator)

print (elements, 'elements based on indicies:')

for row in generator:

print (dict\_ls[row])

x = [[i] for i in range(n-1)]

random.shuffle(x)

p\_line()

print ('1st', elements, 'shuffled elements:')

ind = get\_slice(x, elements)

for row in ind:

print (dict\_ls[row[0]])

seed = 1

random\_seed = random.seed(seed)

rs1 = random.randrange(0, n-1)

p\_line()

print ('deterministic seed', str(seed) + ':', rs1)

print ('corresponding element:', dict\_ls[rs1])

t = time.time()

random\_seed = random.seed(t)

rs2 = random.randrange(0, n-1)

p\_line()

print ('non-deterministic time seed', str(t) + ' index:', rs2)

print ('corresponding element:', dict\_ls[rs2], '\n')

print (elements, 'random elements seeded with time:')

for i in range(elements):

r = random.randint(0, n-1)

print (dict\_ls[r], r)

random\_seed = random.seed()

rs3 = random.randrange(0, n-1)

p\_line()

print ('non-deterministic auto seed:', rs3)

print ('corresponding element:', dict\_ls[rs3], '\n')

print (elements, 'random elements auto seed:')

for i in range(elements):

r = random.randint(0, n-1)

print (dict\_ls[r], r)

names = []

for row in dict\_ls:

name = row['last'] + ', ' + row['first']

names.append(name)

p\_line()

print (elements, 'names with "random.choice()":')

for row in range(elements):

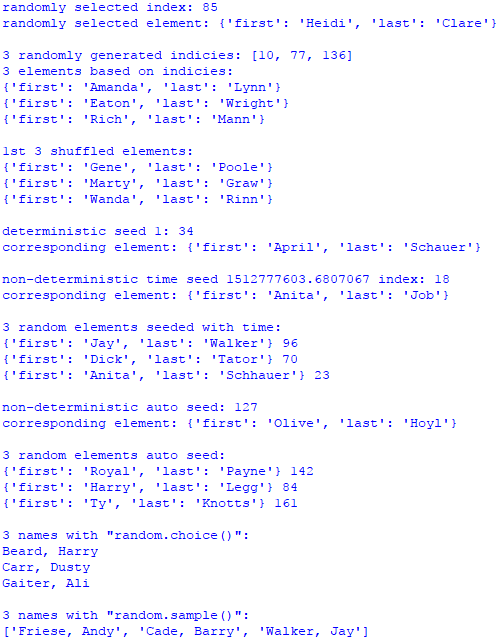
print (random.choice(names))

p\_line()

print (elements, 'names with "random.sample()":')

print (random.sample(names, elements))

Output:



The code begins by importing csv, random, and time libraries. Functions read\_dict() and conv\_reg\_dict() have already been explained. Function r\_inds() generates a random list of n elements from the dictionary list. To get the proper length, one is subtracted because Python lists begin at index zero. Function get\_slice() creates a randomly shuffled list of n elements from the dictionary list. Function p\_line() prints a blank line. The main block begins by reading a CSV file and converting it into a list of regular dictionary elements. The code continues by creating a random number with random.randrange() based on the number of indices from the dictionary list and displays the index and associated dictionary element. Next, a generator is created and populated with three randomly determined elements. The indices and associated elements are printed from the generator. The next part of the code randomly shuffles the indicies and puts them in list x. An index value is created by slicing three random elements based on the shuffled indices stored in list x. The three elements are then displayed. The code continues by creating a deterministic random seed using a fixed number (seed) in the function. So, the random number generated by this seed will be the same each time the program is run. This means that the dictionary element displayed will be also be the same. Next, two methods for creating non-deterministic random numbers are presented – random.seed(t) and random.seed() – where t varies by system time and using no parameter automatically varies random numbers. Randomly generated elements are displayed for each method. The final part of the code creates a list names to hold just first and last names, so random.choice() and random.sample() can be used.

*MongoDB and JSON*

MongoDB is a document-based database classified as NoSQL. NoSQL (Not Only SQL database) is an approach to database design that can accommodate a wide variety of data models, including key-value, document, columnar and graph formats. It uses JSON-like documents with schemas. It integrates extremely well with Python. A MongoDB collection is conceptually like a table in a relational database and a document is conceptually like a row. JSON is a lightweight data-interchange format that is easy for humans to read and write. It is also easy for machines to parse and generate.

Database queries from MongoDB are handled by PyMongo. PyMongo is a Python distribution containing tools for working with MongoDB. It is the most efficient tool for working with MongoDB using the utilities of Python. PyMongo was created to leverage the advantages of Python as a programming language and MongoDB as a database. The pymongo library is a native driver for MongoDB, which means it is it is built into Python language. Since it is native, the pymongo library is automatically available (doesn’t have to be imported into the code).

The following code example reads a CSV file and converts it to a list of regular dictionary elements. The code continues by creating a JSON file from the dictionary list and saving it to disk. Next, the code connects to MongoDB and inserts the JSON data. The final part of the code manipulates data from the MongoDB database. First, all data in the database is queried and a few records are displayed. Second, the database is rewound. Rewind sets the pointer to back to the 1st database record. Finally, various queries are performed.

import json, csv, sys, os

sys.path.append(os.getcwd()+'/classes')

import conn

def read\_dict(f, h):

input\_file = csv.DictReader(open(f), fieldnames=h)

return (input\_file)

def conv\_reg\_dict(d):

return [dict(x) for x in d]

def dump\_json(f, d):

with open(f, 'w') as f:

json.dump(d, f)

def read\_json(f):

with open(f) as f:

return json.load(f)

if \_\_name\_\_ == '\_\_main\_\_':

f = 'data/names.csv'

headers = ['first', 'last']

r\_dict = read\_dict(f, headers)

dict\_ls = conv\_reg\_dict(r\_dict)

json\_file = 'data/names.json'

dump\_json(json\_file, dict\_ls)

data = read\_json(json\_file)

obj = conn.conn('test')

db = obj.getDB()

names = db.names

names.drop()

for i, row in enumerate(data):

row['\_id'] = i

names.insert\_one(row)

n = 3

print('1st', n, 'names:')

people = names.find()

for i, row in enumerate(people):

if i < n:

print (row)

people.rewind()

print('\n1st', n, 'names with rewind:')

for i, row in enumerate(people):

if i < n:

print (row)

print ('\nquery 1st', n, 'names')

first\_n = names.find().limit(n)

for row in first\_n:

print (row)

print ('\nquery last', n, 'names')

length = names.find().count()

last\_n = names.find().skip(length - n)

for row in last\_n:

print (row)

fnames = ['Ella', 'Lou']

lnames = ['Vader', 'Pole']

print ('\nquery Ella:')

query\_1st\_in\_list = names.find( {'first':{'$in':[fnames[0]]}})

for row in query\_1st\_in\_list:

print (row)

print ('\nquery Ella or Lou:')

query\_1st = names.find( {'first':{'$in':fnames}} )

for row in query\_1st:

print (row)

print ('\nquery Lou Pole:')

query\_and = names.find( {'first':fnames[1], 'last':lnames[1]} )

for row in query\_and:

print (row)

print ('\nquery first name Ella or last name Pole:')

query\_or = names.find( {'$or':[{'first':fnames[0]}, {'last':lnames[1]}]} )

for row in query\_or:

print (row)

pattern = '^Sch'

print ('\nquery regex pattern:')

query\_like = names.find( {'last':{'$regex':pattern}} )

for row in query\_like:

print (row)

pid = names.count()

doc = {'\_id':pid, 'first':'Wendy', 'last':'Day'}

names.insert\_one(doc)

print ('\ndisplay added document:')

q\_added = names.find({'first':'Wendy'})

print (q\_added.next())

print ('\nquery last n documents:')

q\_n = names.find().skip((pid-n)+1)

for \_ in range(n):

print (q\_n.next())

Class conn:

class conn:

from pymongo import MongoClient

client = MongoClient('localhost', port=27017)

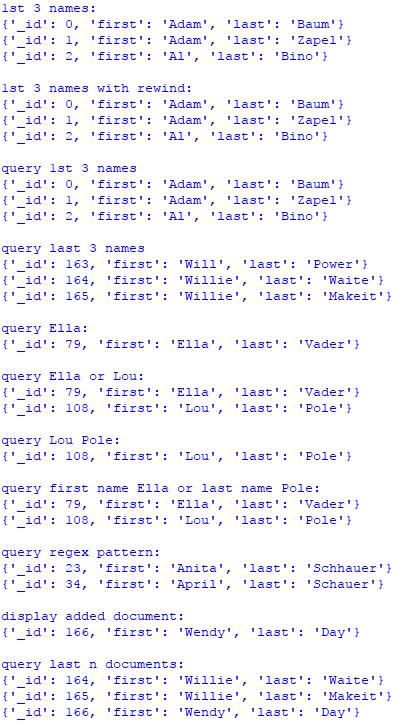
def \_\_init\_\_(self, dbname):

self.db = conn.client[dbname]

def getDB(self):

return self.db

Output:



The code begins by importing json, csv, sys, and os libraries. Next, a path (sys.path.append) to the class conn is established. Method getcwd() (from the os library) gets the current working directory for classes. Class conn is then imported. I built this class to simplify connectivity to the database from any program. The code continues with four functions. Functions read\_dict() and conv\_reg\_dict() were explained earlier. Function dump\_json() writes JSON data to disk. Function read\_json() reads JSON data from disk. The main block begins by reading a CSV file and converting it into a list of regular dictionary elements. Next, the list is dumped to disk as JSON. The code continues by creating a PyMongo connection instance test as an object and assigning it to variable obj. You can create any instance you wish, but test is the default. Next, the database instance is assigned to db by method getDB() from obj. Collection names is then created in MongoDB and assigned to variable names. When prototyping, I always drop the collection before manipulating it. This eliminates duplicate key errors. The code continues by inserting the JSON data into the collection. For each document in a MongoDB collection, I explicitly create primary key values by assigning sequential numbers to \_id. MongoDB exclusively uses \_id as the primary key identifier for each document in a collection. If you don’t name it yourself, a system identifier is automatically created, which is messy to work with in my opinion. The code continues with PyMongo query names.find(), which retrieves all documents from the names collection. Three records are displayed just to verify that the query is working. To reuse a query that has already been accessed, rewind() must be issued. The next PyMongo query accesses and displays three (n=3) documents. The next query accesses and displays the last three documents. Next, we move into more complex queries. First, access documents with first name Ella. Second, access documents with first names Ella or Lou. Third, access document Lou Pole. Fourth, access documents with first name Ella or last name Pole. Next, a regular expression is used to access documents with last names beginning with Sch. A regular expression is a sequence of characters that define a search pattern. Finally, add a new document, display it, and display the last three documents in the collection.

*Visualization*

Visualization is the process of representing data graphically and leveraging these representations to gain insight into the data. Visualization is one of the most important skills in data science because it facilitates the way we process large amounts of complex data.

The following code example creates and plots a normally distributed set of data. It then shifts data to the left (and plots) and shifts the data to the right (and plots). A normal distribution is a probability distribution that is symmetrical about the mean, and is very important to data science because it is an excellent model of how events naturally occur in reality.

import matplotlib.pyplot as plt

from scipy.stats import norm

import numpy as np

if \_\_name\_\_ == '\_\_main\_\_':

x = np.linspace(norm.ppf(0.01), norm.ppf(0.99), num=100)

x\_left = x - 1

x\_right = x + 1

y = norm.pdf(x)

plt.ylim(0.02, 0.41)

plt.scatter(x, y, color='crimson')

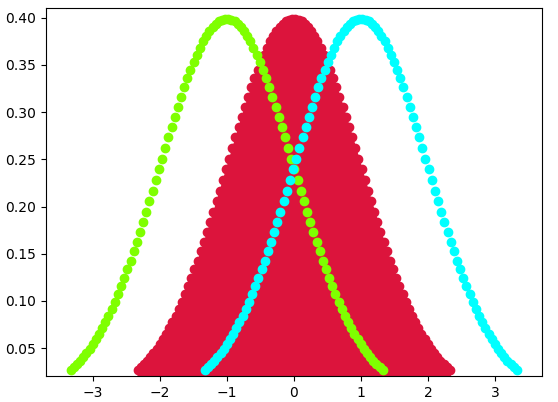
plt.fill\_between(x, y, color='crimson')

plt.scatter(x\_left, y, color='chartreuse')

plt.scatter(x\_right, y, color='cyan')

plt.show()

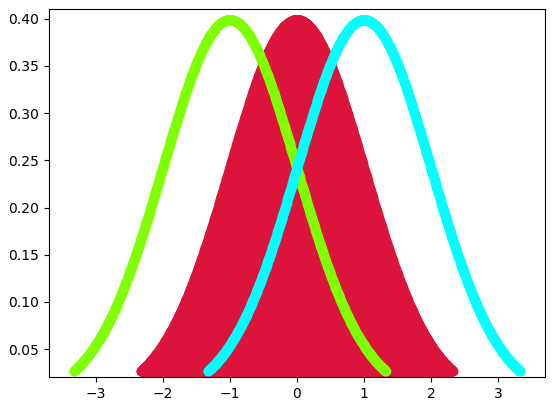
Output:



The code example begins by importing matplotlib, scipy, and numpy libraries. The matplotlib library is a 2D plotting module that produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. The SciPy library provides user-friendly and efficient numerical routings for numerical integration and optimization. The main block begins by creating a sequence of 100 numbers between 0.01 to 0.99. The reason is the normal distribution is based on probabilities, which must be between zero and one. The code continues by shifting the sequence one unit to the left and one to the right for later plotting. The ylim() method is used to pull the chart to the bottom (x-axis). A scatter plot is created for the original data, one unit to the left, and one to the right with different colors for effect.

On the 1st line of the main block in the linespace() function, increase the number of data points from num=100 to num=1000 and see what happens. The result is a smoothing of the normal distribution because more data provides a more realistic picture of the natural world.

Output:



Smoothing works because a normal distribution consists of continuous random variables. A continuous random variable is a random variable with a set of infinite and uncountable values. So, more data creates more predictive realism. Since we cannot add infinite data, we work with as much data as we can. The tradeoff is more data increases computer processing resources and execution time. Data scientists must thereby weigh this tradeoff when conducting their tradecraft.