Course Code: Four-year B.Sc.(Hons)
Domain Subject: Computer Science IV Year B. Sc.(Hons)
Semester – V
Course 6C: DATA SCIENCE

Max Marks: 100 + 50

UNIT I (10 hours)

Introduction: The Ascendance of Data, What is Data Science? , Finding key Connectors, Data Scientists You May Know, Salaries and Experience, Paid Accounts, Topics of Interest, Onward.

Python: Getting Python, The Zen of Python, Whitespace Formatting, Modules, Arithmetic, Functions, Strings, Exceptions, Lists, Tuples, Dictionaries, Sets, Control Flow, Truthiness, Sorting, List Comprehensions, Generators and Iterators, Randomness, Object – Orienting Programming

Visualizing Data: matplotlib, Bar charts, Line charts, Scatterplots

UNIT II (10 hours)

Statistics: Describing a Single Set of Data, Correlation, Simpson's Paradox Correlation and Causation.

Probability: Dependence and Independence, Conditional Probability, Bayes's Theorem, Random Variables, Continuous Distributions, The Normal Distribution, The Central Limit Theorem.

UNIT III (10 hours)

Getting Data: stdin and stdout, Reading Files – The Basics of Text Files, Delimited Files, Scraping the Web - HTML and the parsing Thereof, Example: O'Reilly Books About Data, Using APIs – JSON (and XML), Using an Unauthenticated API, Finding APIs.

Working with Data: Exploring Your Data, Exploring One-Dimensional Data, Two Dimensions Many Dimensions, Cleaning and Munging

Machine Learning: Modeling, What Is Machine Learning? Over fitting and under fitting, Correctness, The Bias-Variance Trade-off, Feature Extraction and Selection

UNIT IV (10 hours)

K-Nearest Neighbors: The Model, Example: Favorite Languages, The Curse of Dimensionality.

Naive Bayes: A Really Dumb Spam Filter, A More Sophisticated Spam Filter, Implementation, Testing Our Model.

Simple Linear Regression: The Model, Using Gradient Descent, Maximum Likelihood Estimation.

UNIT V (10 hours)

Logistic Regression: The Problem, The Logistic Function, Applying the Model, Goodness of Fit Support Vector Machines.

Decision Trees: What Is a Decision Tree? Entropy, The Entropy of a Partition, Creating a Decision Tree, Putting It All Together, Random Forests.

Neural Networks: Perceptron, Feed-Forward Neural Networks And Back propagation, Example: Defeating a CAPTCHA.

UNIT – I

The Ascendance of Data

- We live in a world that's drowning in data.
- Websites track every user's every click.
- Your smartphone is building up a record of your location and speed every second of every day.
- "Quantified selfers" wear pedometers-on-steroids that are ever recording their heart rates, movement habits, diet, and sleep patterns.
- Smart cars collect driving habits, smart homes collect living habits, and smart marketers collect purchasing habits.
- The Internet itself represents a huge graph of knowledge that contains an enormous cross-referenced encyclopedia; domain-specific databases about movies, music, sports results, pinball machines, memes, and cocktails; and too many government statistics from too many governments to wrap your head around.

What is Data Science?

Ans:

Data science is a deep study of the massive amount of data, which involves extracting meaningful insights from raw, structured, and unstructured data that is processed using the scientific method, different technologies, and algorithms.

It is a multidisciplinary field that uses tools and techniques to manipulate the data so that you can find something new and meaningful.

Data science uses the most powerful hardware, programming systems, and most efficient algorithms to solve the data related problems. It is the future of artificial intelligence.

In short, we can say that data science is all about:

- Asking the correct questions and analyzing the raw data.
- Modeling the data using various complex and efficient algorithms.
- Visualizing the data to get a better perspective.
- Understanding the data to make better decisions and finding the final result.

Data science uses the most powerful hardware, programming systems, and most efficient algorithms to solve the data related problems. It is the future of artificial intelligence.

In short, we can say that data science is all about:

- Asking the correct questions and analyzing the raw data.
- Modeling the data using various complex and efficient algorithms.
- Visualizing the data to get a better perspective.
- Understanding the data to make better decisions and finding the final result.

Finding Key Connectors

It's your first day on the job at DataSciencester, and the VP of Networking is full of questions about your users. Until now he's had no one to ask, so he's very excited to haveyou aboard.

In particular, he wants you to identify who the "key connectors" are among data scientists.To this end, he gives you a dump of the entire DataSciencester network.

What does this data dump look like? It consists of a list of users, each represented by a dictthat contains for each user his or her id(which is a number) and name(which, in oneof the great cosmic coincidences, rhymes with the user's id):

```
users = [
{       "id": 0,  "name":  "Hero" },
{       "id": 1,  "name":  "Dunn" },
{       "id": 2,  "name":  "Sue" },
{       "id": 3,  "name":  "Chi" },
{       "id": 4,  "name":  "Thor" },
{       "id": 5,  "name":  "Clive" },
{       "id": 6,  "name":  "Hicks" },
{       "id": 7,  "name":  "Devin" },
{       "id": 8,  "name":  "Kate" },
{       "id": 9,  "name":  "Klein" }
]
```

He also gives you the "friendship" data, represented as a list of pairs of IDs:
friendships=[(0,1), (0,2), (1, 2), (1, 3), (2, 3), (3, 4), (4, 5), (5, 6), (5, 7), (6, 8), (7, 8),(8, 9)]

For example, the tuple (0, 1)indicates that the data scientist with id0 (Hero) and the data scientist with id1 (Dunn) are friends. The network is illustrated in Figure 1-1.

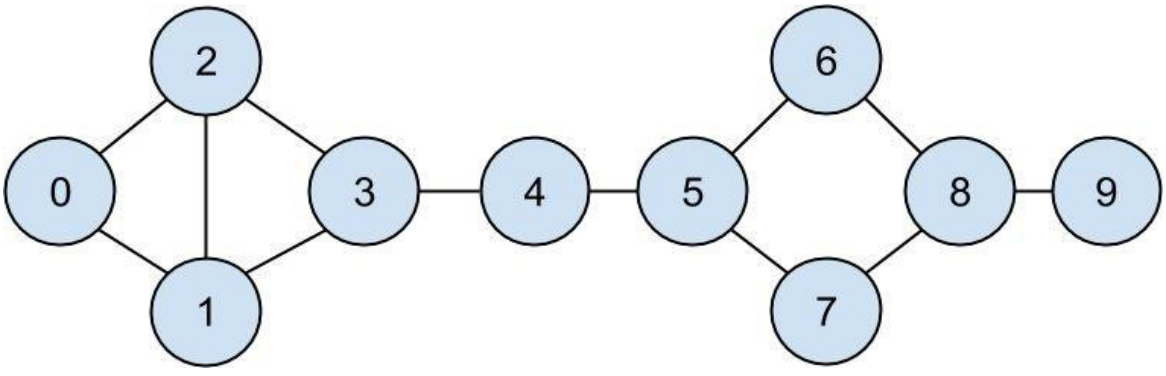


Figure 1-1. The DataSciencester network

Since we represented our users as dicts, it's easy to augment them with extra data.

For example, we might want to add a list of friends to each user. First we set each user's friends property to an empty list:

```
for user in users: user["friends"] = []
```

And then we populate the lists using the friendships data:

```
for i, j in friendships:
users[i]["friends"].append(users[j]) # add i as a friend of j
users[j]["friends"].append(users[i]) # add j as a friend of i
```

Once each user dict contains a list of friends, we can easily ask questions of our graph, like "what's the average number of connections?"

First we find the total number of connections, by summing up the lengths of all the friends lists:

```
def number_of_friends(user):
return len(user["friends"])  # length of friend_ids list
total_connections = sum(number_of_friends(user)
for user in users)      # 24
```

And then we just divide by the number of users:

```
from  future import division       # integer division is lame
num_users = len(users)      # length of the users list
avg_connections = total_connections / num_users       # 2.4
```

It's also easy to find the most connected people — they're the people who have the largest number of friends.

Since there aren't very many users, we can sort them from "most friends" to "least friends":

```
# create a list (user_id, number_of_friends)
num_friends_by_id = [(user["id"], number_of_friends(user)) for user in users]
sorted(num_friends_by_id,          # get it sorted key=lambda (user_id, num_friends):
num_friends,# by num_friends reverse=True)  # largest to smallest

# each pair is (user_id, num_friends)
# [(1, 3), (2, 3), (3, 3), (5, 3), (8, 3),
# (0, 2), (4, 2), (6, 2), (7, 2), (9, 1)]
```

One way to think of what we've done is as a way of identifying people who are somehow central to the network. In fact, what we've just computed is the network metric degree centrality (Figure 1-2).

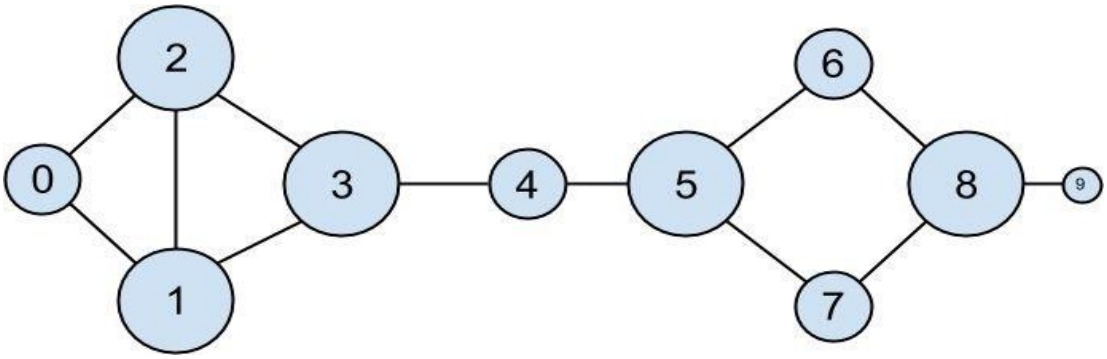


Figure 1-2. The DataSciencester network sized by degree

This has the virtue of being pretty easy to calculate, but it doesn't always give the results you'd want or expect. For example, in the DataSciencester network Thor (id 4) only has two connections while Dunn (id 1) has three. Yet looking at the network it intuitively seems like Thor should be more central.

Data Scientists You May Know

While you're still filling out new-hire paperwork, the VP of Fraternization comes by your desk. She wants to encourage more connections among your members, and she asks you to design a "Data Scientists You May Know" suggester.

Your first instinct is to suggest that a user might know the friends of friends. These are easy to compute: for each of a user's friends, iterate over that person's friends, and collect all the results:

```
def friends_of_friend_ids_bad(user):
        return [foaf["id"]
for friend in user["friends"]          # for each of user's friends
for foaf in friend["friends"]]         # get each of _their_ friends
```

When we call this on users[0] (Hero), it produces:

[0, 2, 3, 0, 1, 3]

It includes user 0 (twice), since Hero is indeed friends with both of his friends. It includes users 1 and 2, although they are both friends with Hero already. And it includes user 3 twice, as Chi is reachable through two different friends:

print [friend["id"] for friend in users[0]["friends"]] # [1, 2] print [friend["id"] for friend in users[1]["friends"]] # [0, 2, 3] print [friend["id"] for friend in users[2]["friends"]] # [0, 1, 3]

Knowing that people are friends-of-friends in multiple ways seems like interesting information, so maybe instead we should produce a count of mutual friends. And we definitely should use a helper function to exclude people already known to the user:

```
from collections import Counter   # not loaded by default
def not_the_same(user, other_user):
        return user["id"] != other_user["id"]
def not_friends(user, other_user):
return all(not_the_same(friend, other_user)
for friend in user["friends"])
```

```
def friends_of_friend_ids(user):
        return Counter(foaf["id"]
for friend in user["friends"]        # for each of my friends for foaf in
friend["friends"]    # count *their* friends if not_the_same(user, foaf)    # who
aren't me
and not_friends(user, foaf))         # and aren't my friends

print friends_of_friend_ids(users[3])      # Counter({0: 2, 5: 1})
```

This correctly tells Chi (id 3) that she has two mutual friends with Hero (id 0) but only one mutual friend with Clive (id 5).

As a data scientist, you know that you also might enjoy meeting users with similar interests. After asking around, you manage to get your hands on this data, as a list of pairs (user_id, interest):

```
interests = [
(0, "Hadoop"), (0, "Big Data"), (0, "HBase"), (0, "Java"),
(0, "Spark"), (0, "Storm"), (0, "Cassandra"),
(1, "NoSQL"), (1, "MongoDB"), (1, "Cassandra"), (1, "HBase"),
(1, "Postgres"), (2, "Python"), (2, "scikit-learn"), (2, "scipy"),
(2, "numpy"), (2, "statsmodels"), (2, "pandas"), (3, "R"), (3, "Python"),
(3, "statistics"), (3, "regression"), (3, "probability"),
(4, "machine learning"), (4, "regression"), (4, "decision trees"),
(4, "libsvm"), (5, "Python"), (5, "R"), (5, "Java"), (5, "C++"),
(5, "Haskell"), (5, "programming languages"), (6, "statistics"),
(6, "probability"), (6, "mathematics"), (6, "theory"),
(7, "machine learning"), (7, "scikit-learn"), (7, "Mahout"),
(7, "neural networks"), (8, "neural networks"), (8, "deep learning"),
(8, "Big Data"), (8, "artificial intelligence"), (9, "Hadoop"),
(9, "Java"), (9, "MapReduce"), (9, "Big Data")
]
```

For example, Thor (id 4) has no friends in common with Devin (id 7), but they share an interest in machine learning.

It's easy to build a function that finds users with a certain interest:

```
def data_scientists_who_like(target_interest):
return [user_id
for user_id, user_interest in interests
if user_interest == target_interest]
```

This works, but it has to examine the whole list of interests for every search. If we have a lot of users and interests, we're probably better off building an index from interests to users:

```
from collections import defaultdict

        # keys are interests, values are lists of user_ids with that interest
            user_ids_by_interest = defaultdict(list)
```

for user_id, interest in interests:

user_ids_by_interest[interest].append(user_id)

And another from users to interests:

interests_by_user_id = defaultdict(list)

for user_id, interest in interests: interests_by_user_id[user_id].append(interest)

Now it's easy to find who has the most interests in common with a given user: Iterate over the user's interests.

For each interest, iterate over the other users with that interest. Keep count of how many times we see each other user.

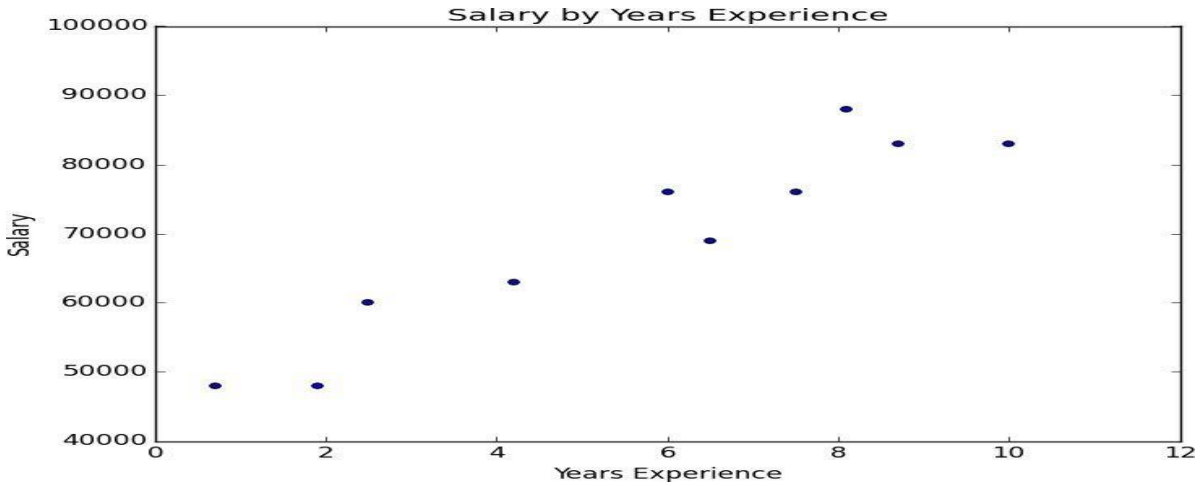
def most_common_interests_with(user):
return Counter(interested_user_id
for interest in interests_by_user_id[user["id"]]
for interested_user_id in user_ids_by_interest[interest]
if interested_user_id != user["id"])

Salaries and Experience

Right as you're about to head to lunch, the VP of Public Relations asks if you can provide some fun facts about how much data scientists earn. Salary data is of course sensitive, but he manages to provide you an anonymous data set containing each user's salary (in dollars) and tenure as a data scientist (in years):

salaries_and_tenures = [(83000, 8.7), (88000, 8.1),(48000, 0.7), (76000, 6),(69000, 6.5), (76000, 7.5), (60000, 2.5), (83000, 10),(48000, 1.9), (63000, 4.2)]

The natural first step is to plot the data .



You can see the results in Figure 1-3.

It seems pretty clear that people with more experience tend to earn more. How can you turn this into a fun fact? Your first idea is to look at the average salary for each tenure:

 # keys are years, values are lists of the salaries for each tenure
salary_by_tenure = defaultdict(list)
for salary, tenure in salaries_and_tenures: salary_by_tenure[tenure].append(salary)
# keys are years, each value is average salary for that tenure
average_salary_by_tenure = {tenure : sum(salaries) / len(salaries)
for tenure, salaries in salary_by_tenure.items()}

This turns out to be not particularly useful, as none of the users have the same tenure, which means we're just reporting the individual users' salaries:
{0.7: 48000.0, 1.9: 48000.0, 2.5: 60000.0, 4.2: 63000.0, 6: 76000.0, 6.5: 69000.0,
7.5: 76000.0, 8.1: 88000.0, 8.7: 83000.0, 10: 83000.0}
It might be more helpful to bucket the tenures:
def tenure_bucket(tenure):
if tenure < 2:
return "less than two"
elif tenure < 5:
return "between two and five"
else:
return "more than five"

Then group together the salaries corresponding to each bucket:
salary_by_tenure_bucket = defaultdict(list)

for salary, tenure in salaries_and_tenures: bucket = tenure_bucket(tenure) salary_by_tenure_bucket[bucket].append(salary)

And finally compute the average salary for each group:

average_salary_by_bucket = {
tenure_bucket : sum(salaries) / len(salaries)
for tenure_bucket, salaries in salary_by_tenure_bucket.iteritems()
}
which is more interesting:
{'between two and five': 61500.0, 'less than two': 48000.0,
'more than five': 79166.66666666667}

And you have your sound bite: "Data scientists with more than five years experience earn 65% more than data scientists with little or no experience!"

But we chose the buckets in a pretty arbitrary way. What we'd really like is to make some sort of statement about the salary effect — on average — of having an additional year of experience. In addition to making for a snappier fun fact, this allows us to make predictions about salaries that we don't know.

Paid Accounts

When you get back to your desk, the VP of Revenue is waiting for you. She wants to better understand which users pay for accounts and which don't.

You notice that there seems to be a correspondence between years of experience and paid accounts:

0.7 paid
1.9 unpaid
2.5 paid
4.2 unpaid
6 unpaid
6.5 unpaid
7.5 unpaid
8.1 unpaid
8.7 paid
10 paid

Users with very few and very many years of experience tend to pay; users with average amounts of experience don't.

Accordingly, if you wanted to create a model — though this is definitely not enough data to base a model on — you might try to predict "paid" for users with very few and very many years of experience, and "unpaid" for users with middling amounts of experience:

```
def predict_paid_or_unpaid(years_experience):
    if years_experience < 3.0:
    return "paid"
    elif years_experience < 8.5:
    return "unpaid"
    else:
    return "paid"
```

Topics of Interest

As you're wrapping up your first day, the VP of Content Strategy asks you for data about what topics users are most interested in, so that she can plan out her blog calendar accordingly. You already have the raw data from the friend-suggester project:

```
interests = [
(0, "Hadoop"), (0, "Big Data"), (0, "HBase"), (0, "Java"),
(0, "Spark"), (0, "Storm"), (0, "Cassandra"),
(1, "NoSQL"), (1, "MongoDB"), (1, "Cassandra"), (1, "HBase"),
(1, "Postgres"), (2, "Python"), (2, "scikit-learn"), (2, "scipy"),
(2, "numpy"), (2, "statsmodels"), (2, "pandas"), (3, "R"), (3, "Python"),
(3, "statistics"), (3, "regression"), (3, "probability"),
(4, "machine learning"), (4, "regression"), (4, "decision trees"),
(4, "libsvm"), (5, "Python"), (5, "R"), (5, "Java"), (5, "C++"),
(5, "Haskell"), (5, "programming languages"), (6, "statistics"),
(6, "probability"), (6, "mathematics"), (6, "theory"),
(7, "machine learning"), (7, "scikit-learn"), (7, "Mahout"),
```

(7, "neural networks"), (8, "neural networks"), (8, "deep learning"),
(8, "Big Data"), (8, "artificial intelligence"), (9, "Hadoop"),
(9, "Java"), (9, "MapReduce"), (9, "Big Data")]

One simple way to find the most popular interests is simply to count the words:

1. Lowercase each interest (since different users may or may not capitalize their interests).
2. Split it into words.
3. Count the results.

Onward

It's been a successful first day! Exhausted, you slip out of the building before anyone else can ask you for anything else. Get a good night's rest, because tomorrow is new employeeorientation.

The Zen of Python

- Python has a somewhat Zen description of its design principles, which you can also findinside the Python interpreter itself by typing import this.
- One of the most discussed of these is:
  There should be one — and preferably only one — obvious way to do it.
- Code written in accordance with this "obvious" way is often described as "Pythonic."
- We will occasionally contrast Pythonic and non-Pythonic ways of accomplishing the same things, and we will generally favor Pythonic solutions to our problems.

Whitespace Formatting

- Many languages use curly braces to delimit blocks of code. Python uses indentation. This makes Python code very readable, but it also means that you have to be very careful with your formatting.
- Whitespace is ignored inside parentheses and brackets, which can be helpful for long-winded computations:

  long_winded_computation = (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12
  +13 + 14+ 15 + 16 + 17 + 18 + 19 + 20)

  and for making code easier to read:

  list_of_lists = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

  easier_to_read_list_of_lists = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

  You can also use a backslash to indicate that a statement continues onto the next line, although we'll rarely do this:

  ```
  two_plus_three = 2 + \
                   3
  ```

  One consequence of whitespace formatting is that it can be hard to copy and paste code into the Python shell. For example, if you tried to paste the code:

  ```
  for i in [1, 2, 3, 4, 5]:
  print i
  ```

  into the ordinary Python shell, you would get a:

  IndentationError: expected an indented block

  because the interpreter thinks the blank line signals the end of the for loop's block.

Modules

A Python module is a file containing Python definitions and statements. A module can define functions, classes, and variables. A module can also include runnable code. Grouping related code into a module makes the code easier to understand and use. It also makes the code logically organized.

Create a simple Python module

```
# A simple module, calc.py
def add(x, y):
        return (x+y)

def subtract(x, y):
        return (x-y)
```

Import Module in Python

We can import the functions, and classes defined in a module to another module using the **import statement** in some other Python source file.

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches for importing a module. For example, to import the module calc.py, we need to put the following command at the top of the script.

Syntax: import module

Importing modules in Python

Now, we are importing the **calc** that we created earlier to perform add operation.

```
# importing module calc.py
import calc
print(calc.add(10, 2))
```

The from-import Statement in Python

Python's *from* statement lets you import specific attributes from a module without importing the module as a whole.

Eg: Importing specific attributes from the module

Here, we are importing specific sqrt and factorial attributes from the math module.

```
# importing sqrt() and factorial from the
# module math
from math import sqrt, factorial
print(sqrt(16))
print(factorial(6))
```

Import all Names

The * symbol used with the from import statement is used to import all the names from a module to a current namespace.

Syntax: from module_name import *

Eg: From import *

The use of * has its advantages and disadvantages. If you know exactly what you will be needing from the module, it is not recommended to use *, else do so.

```
# importing sqrt() and factorial from the
# module math
from math import *
print(sqrt(16))
print(factorial(6))
```

Locating Python Modules

Whenever a module is imported in Python the interpreter looks for several locations. First, it will check for the built-in module, if not found then it looks for a list of directories defined in the sys.path. Python interpreter searches for the module in the following manner –

- First, it searches for the module in the current directory.
- If the module isn't found in the current directory, Python then searches each directory in the shell variable PYTHONPATH. The PYTHONPATH is an environment variable, consisting of a list of directories.
- If that also fails python checks the installation-dependent list of directories configured at the time Python is installed.

Directories List for Modules

Here, sys.path is a built-in variable within the sys module. It contains a list of directories that the interpreter will search for the required module.

```
# importing sys module
import sys
# importing sys.path
print(sys.path)
```

Renaming the Python module

We can rename the module while importing it using the keyword.

Syntax: *Import* ***Module_name*** *as* ***Alias_name***

```
# importing sqrt() and factorial from the
# module math
import math as mt
# if we simply do "import math", then
# math.sqrt(16) and math.factorial()
# are required.
print(mt.sqrt(16))
print(mt.factorial(6))
```

Python built-in modules

There are several built-in modules in Python, which you can import whenever you like.

```
# importing built-in module math
import math
print(math.sqrt(25))
print(math.pi)

# 2 radians = 114.59 degrees
print(math.degrees(2))

# 60 degrees = 1.04 radians
print(math.radians(60))
print(math.sin(2))
print(math.cos(0.5))
print(math.tan(0.23))
print(math.factorial(4))

# importing built in module random
import random
print(random.randint(0, 5))

# print random floating point number between 0 and 1
print(random.random())
# random number between 0 and 100
print(random.random() * 100)
List = [1, 4, True, 800, "python", 27, "hello"]
print(random.choice(List))

# importing built in module datetime
import datetime
from datetime import date
import time
# Returns the number of seconds since the Unix Epoch, January 1st 1970
print(time.time())
# Converts a number of seconds to a date object
print(date.fromtimestamp(454554))
```

Arithmetic

- Python 2.7 uses integer division by default, so that 5 / 2 equals 2. Almost always this is not what we want, so we will always start our files with:

  ```
  from future import division
  ```

  after which 5 / 2 equals 2.5. Every code example in this book uses this new-style division. In the handful of cases where we need integer division, we can get it with a double slash: 5 // 2.

Functions

- A function is a collection of related assertions that performs a mathematical, analytical, or evaluative operation.
- Python functions are simple to define and essential to intermediate-level programming.
- The exact criteria hold to function names as they do to variable names. The goal is to group up certain often performed actions and define a function. Rather than rewriting the same code block over and over for varied input variables, we may call the function and repurpose the code included within it with different variables.
- The functions are broad of two types, **user-defined** and **built-in functions.**

  Syntax: def name_of_function(parameters):
  # code block

  The following elements make up define a function, as seen above.
  - ⇨ The beginning of a function header is indicated by a keyword called def.
  - ⇨ name_of_function is the function's name that we can use to separate it from others. We will use this name to call the function later in the program. The same criteria apply to naming functions as to naming variables in Python.
  - ⇨ We pass arguments to the defined function using parameters. They are optional, though.
  - ⇨ The function header is terminated by a colon (:).
  - ⇨ The body of the function is made up of several valid Python statements. The indentation depth of the whole code block must be the same (usually 4 spaces).
  - ⇨ We can use a return expression to return a value from a defined function.

Example of a User-Defined Function

We will define a function that when called will return the square of the number passed to it as an argument.

```
def square( num ):
   return num**2
object_ = square(9)
print( "The square of the number is: ", object_ )
```

Output:

```
The square of the number is:  81
```

Calling a Function

- A function is defined by using the def keyword and giving it a name, specifying the arguments that must be passed to the function, and structuring the code block.
- After a function's fundamental framework is complete, we can call it from anywhere in the program.

```
def a_function( string ):
        return len(string)
print( "Length of the string Functions is: ", a_function( "Functions" ) )
print( "Length of the string Python is: ", a_function( "Python" ) )
```

Output:

```
Length of the string Functions is:  9
Length of the string Python is:  6
```

Function Arguments

The following are the types of arguments that we can use to call a function:

1. Default arguments
2. Keyword arguments
3. Required arguments
4. Variable-length arguments

Default Arguments

- A default argument is a kind of parameter that takes as input a default value if no value is supplied for the argument when the function is called.

```
# Python code to demonstrate the use of default arguments
def function( num1, num2 = 40 ):
   print("num1 is: ", num1)
   print("num2 is: ", num2)
# Calling the function and passing only one argument
print( "Passing one argument" )
function(10)
# Now giving two arguments to the function
print( "Passing two arguments" )
function(10,30)
```

Output:

```
Passing one argument
num1 is:  10
num2 is:  40
Passing two arguments
num1 is:  10
num2 is:  30
```

Keyword Arguments

- The arguments in a function called are connected to keyword arguments. If we provide keyword arguments while calling a function, the user uses the parameter label to identify which parameters value it is.
- Since the Python interpreter will connect the keywords given to link the values with its parameters, we can omit some arguments or arrange them out of order.

```
def function( num1, num2 ):
   print("num1 is: ", num1)
   print("num2 is: ", num2)
# Calling function and passing arguments without using keyword
print( "Without using keyword" )
function( 50, 30)
# Calling function and passing arguments using keyword
print( "With using keyword" )
function( num2 = 50, num1 = 30)
```

Output:

```
Without using keyword
num1 is:  50
num2 is:  30
With using keyword
num1 is:  30
num2 is:  50
```

Required Arguments

- The arguments given to a function while calling in a pre-defined positional sequence are required arguments. The count of required arguments in the method call must be equal to the count of arguments provided while defining the function.
- We must send two arguments to the function function() in the correct order, or it will return a syntax error, as seen below.

```
# Python code to demonstrate the use of default arguments
def function( num1, num2 ):
   print("num1 is: ", num1)
   print("num2 is: ", num2)
print( "Passing out of order arguments" )
function( 30, 20 )
# Calling function and passing only one argument
print( "Passing only one argument" )
try:
   function( 30 )
except:
   print( "Function needs two positional arguments" )
```

Output:

```
Passing out of order arguments
num1 is:  30
num2 is:  20
Passing only one argument
Function needs two positional arguments
```

Variable-Length Arguments

We can use special characters in Python functions to pass as many arguments as we want in a function. There are two types of characters that we can use for this purpose:

- *args -These are Non-Keyword Arguments
- **kwargs - These are Keyword Arguments.

```
# Python code to demonstrate the use of variable-length arguments
def function( *args_list ):
   ans = []
   for l in args_list:
      ans.append( l.upper() )
   return ans
# Passing args arguments
object = function('Python', 'Functions', 'tutorial')
print( object )
# defining a function
def function( **kargs_list ):
   ans = []
   for key, value in kargs_list.items():
      ans.append([key, value])
   return ans
```

 # Passing kwargs arguments
object = function(First = "Python", Second = "Functions", Third = "Tutorial")
print(object)
Output:
['PYTHON', 'FUNCTIONS', 'TUTORIAL']
[['First', 'Python'], ['Second', 'Functions'], ['Third', 'Tutorial']]

return Statement
We write a return statement in a function to leave a function and give the calculated value when a defined function is called.

Syntax: return < expression to be returned as output >

An argument, a statement, or a value can be used in the return statement, which is given as output when a specific task or function is completed. If we do not write a return statement, then None object is returned by a defined function.

Eg: # Python code to demonstrate the use of return statements
```
def square( num ):
  return num**2
# Calling function and passing arguments.
print( "With return statement" )
print( square( 39 ) )

# Defining a function without return statement
def square( num ):
   num**2
# Calling function and passing arguments.
print( "Without return statement" )
print( square( 39 ) )
```
Output:
With return statement
1521
Without return statement
None

The Anonymous Functions

- These types of Python functions are anonymous since we do not declare them, as we declare usual functions, using the def keyword. We can use the lambda keyword to define the short, single output, anonymous functions.
- Lambda expressions can accept an unlimited number of arguments; however, they only return one value as the result of the function. They can't have numerous expressions or instructions in them. Since lambda needs an expression, an anonymous function cannot be directly called to print.
- Lambda functions contain their unique local domain, meaning they can only reference variables in their argument list and the global domain name.
- Although lambda expressions seem to be a one-line representation of a function, they are not like inline expressions in C and C++, which pass function stack allocations at execution for efficiency concerns.

Syn: lambda [argument1 [,argument2... .argumentn]] : expression

Eg:

 # Defining a function

```
lambda_ = lambda argument1, argument2: argument1 + argument2;
```

 # Calling the function and passing values

```
print( "Value of the function is : ", lambda_( 20, 30 ) )
print( "Value of the function is : ", lambda_( 40, 50 ) )
```

Output:

Value of the function is : 50
Value of the function is : 90

Strings

- The string can be defined as the sequence of characters represented in the quotation marks.
- In Python, we can use single, double, or triple quotes to define a string.
- String handling in Python is a straightforward task since Python provides built-in functions and operators to perform operations in the string.
- In the case of string handling, the operator + is used to concatenate two strings as the operation "*hello*"+" *python*" returns "*hello python*".
- The operator * is known as a repetition operator as the operation "Python" *2 returns 'Python Python'.

Types of Strings:

There are two types of Strings supported in Python:

1. **Single-line String**
   Strings that are terminated within a single-line are known as Single line Strings.
   Example: text1='hello'
2. **Multi-line String**
   A piece of text that is written in multiple lines is known as multiple lines string. There are two ways to create multiline strings:
   1) **Adding black slash at the end of each line.**

```
text1='hello\
      user'
print(text1)
Output: hellouser
```

   2) **Using triple quotation marks:-**

```
str2='''welcome
      to SSSIT'''
      print str2
```

Output:

Welcome
to SSSIT

Exceptions

- An exception in Python is an incident that happens while executing a program that causes the regular course of the program's commands to be disrupted.
- When a Python code comes across a condition it can't handle, it raises an exception.
- An object in Python that describes an error is called an exception.
- When a Python code throws an exception, it has two options: handle the exception immediately or stop and quit.

Exceptions versus Syntax Errors

When the interpreter identifies a statement that has an error, syntax errors occur.

```
#Python code after removing the syntax error
string = "Python Exceptions"
for s in string:
if (s != o:
print( s )
```

Output:

```
  if (s != o:
        ^
SyntaxError: invalid syntax
```

Try and Except Statement - Catching Exceptions

In Python, we catch exceptions and handle them using try and except code blocks. The try clause contains the code that can raise an exception, while the except clause contains the code lines that handle the exception. Let's see if we can access the index from the array, which is more than the array's length, and handle the resulting exception.

```
# Python code to catch an exception and handle it using try and except code blocks
a = ["Python", "Exceptions", "try and except"]
try:
   for i in range( 4 ):
     print( "The index and element from the array is", i, a[i] )
except:
  print ("Index out of range")
```

Output:

```
The index and element from the array is 0 Python
The index and element from the array is 1 Exceptions
The index and element from the array is 2 try and except
Index out of range
```

How to Raise an Exception

If a condition does not meet our criteria but is correct according to the Python interpreter, we can intentionally raise an exception using the raise keyword. We can use a customized exception in conjunction with the statement.

If we wish to use raise to generate an exception when a given condition happens, we may do so as follows:

Eg: #Python code to show how to raise an exception in Python

```
num = [3, 4, 5, 7]
if len(num) > 3:
   raise Exception( f"Length of the given list must be less than or equal to 3 but is {len(num)}" )
```

Output:

```
    1 num = [3, 4, 5, 7]
    2 if len(num) > 3:
----> 3     raise Exception( f"Length of the given list must be less than or equal to 3 but is {len(num)}" )
```

List

- Python Lists are similar to arrays in C. However, the list can contain data of different types. The items stored in the list are separated with a comma (,) and enclosed within square brackets [].
- We can use slice [:] operators to access the data of the list.
- The concatenation operator (+) and repetition operator (*) works with the list in the same way as they were working with the strings.

  Eg:

```
list1  = [1, "hi", "Python", 2]
print(type(list1))    #Checking type of given list
print (list1)  #Printing the list1
print (list1[3:])  # List slicing
print (list1[0:2])   # List slicing
print (list1 + list1)  # List Concatenation using + operator
print (list1 * 3)  # List repetation using * operator
```

  Output:

```
[1, 'hi', 'Python', 2]
[2]
[1, 'hi']
[1, 'hi', 'Python', 2, 1, 'hi', 'Python', 2]
[1, 'hi', 'Python', 2, 1, 'hi', 'Python', 2, 1, 'hi', 'Python', 2]
```

Tuple

- A tuple is similar to the list in many ways.
- Like lists, tuples also contain the collection of the items of different data types.
- The items of the tuple are separated with a comma (,) and enclosed in parentheses ().
- A tuple is a read-only data structure as we can't modify the size and value of the items of a tuple.

  Eg:

```
tup  = ("hi", "Python", 2)
print (tup)    #Printing the tuple
print (tup[1:])    # Tuple slicing
print (tup[0:1])
print (tup + tup)    # Tuple concatenation using + operator
print (tup * 3)      # Tuple repatation using * operator
tup[2] = "hi"  # Adding value to tup. It will throw an error.
```

Output:

```
 ('hi', 'Python', 2)
('Python', 2)
('hi',)
('hi', 'Python', 2, 'hi', 'Python', 2)
('hi', 'Python', 2, 'hi', 'Python', 2, 'hi', 'Python', 2)
Traceback (most recent call last):
  File "main.py", line 14, in <module>
    tup[2] = "hi";
TypeError: 'tuple' object does not support item assignment
```

Dictionary

- Dictionary is an unordered set of a key-value pair of items.
- It is like an associative array or a hash table where each key stores a specific value. Key can hold any primitive data type, whereas value is an arbitrary Python object.
- The items in the dictionary are separated with the comma (,) and enclosed in the curly braces {}.

Eg:

```
d = {1:'Jimmy', 2:'Alex', 3:'john', 4:'mike'}
print (d)        # Printing dictionary
print("1st name is "+d[1])   # Accesing value using keys
print("2nd name is "+ d[4])
print (d.keys())
print (d.values())
```

Output:

```
{1: 'Jimmy', 2: 'Alex', 3: 'john', 4: 'mike'}
1st name is Jimmy
2nd name is mike
dict_keys([1, 2, 3, 4])
dict_values(['Jimmy', 'Alex', 'john', 'mike'])
```

Set

- Python Set is the unordered collection of the data type.
- It is iterable, mutable(can modify after creation), and has unique elements.
- In set, the order of the elements is undefined; it may return the changed sequence of the element.
- The set is created by using a built-in function set(), or a sequence of elements is passed in the curly braces and separated by the comma.
- It can contain various types of values. Consider the following example.

```
# Creating Empty set
set1 = set()
set2 = {'James', 2, 3,'Python'}
#Printing Set value
print(set2)
# Adding element to the set
set2.add(10)
print(set2)
```

```
#Removing element from the set
set2.remove(2)
print(set2)
```

Output:

```
{3, 'Python', 'James', 2}
{'Python', 'James', 3, 2, 10}
{'Python', 'James', 3, 10}
```

Counter

- A Counter turns a sequence of values into a defaultdict(int)-like object mapping keys to counts. We will primarily use it to create histograms:

```
from collections import Counter
c = Counter([0, 1, 2, 0])       # c is (basically) { 0 : 2, 1 : 1, 2 : 1 }
```

This gives us a very simple way to solve our word_counts problem:

```
word_counts = Counter(document)
```

- A Counter instance has a most_common method that is frequently useful:

```
# print the 10 most common words and their counts
for word, count in word_counts.most_common(10):
print word, count
```

Control Flow

The conditional statements are used to control the flow of execution of statements of a program. Python support different conditional statements. They are

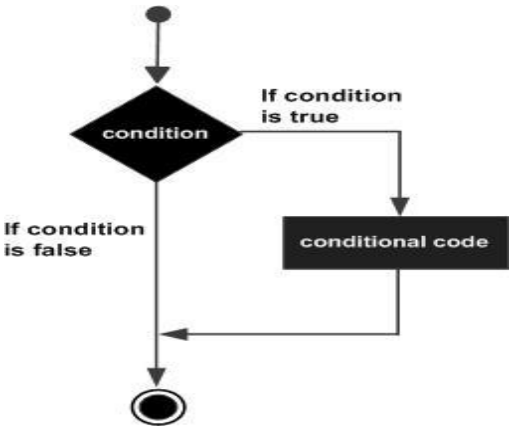
1. Simple if
2. If..else
3. If..elif..else
4. Nested if

1) Simple if:

- This conditional statement executes true statements only when the condition is true, otherwise it skips the if statement
- The if statement checks the condition first.
- If the condition evaluates to True, it executes the statements in the if-block. Otherwise, it ignores the statements.
- Note that the colon (:) that follows the condition is very important. If you forget it, you'll get a syntax error.

Syn:

```
if expression:
    Statement1;
    Statement2;
```

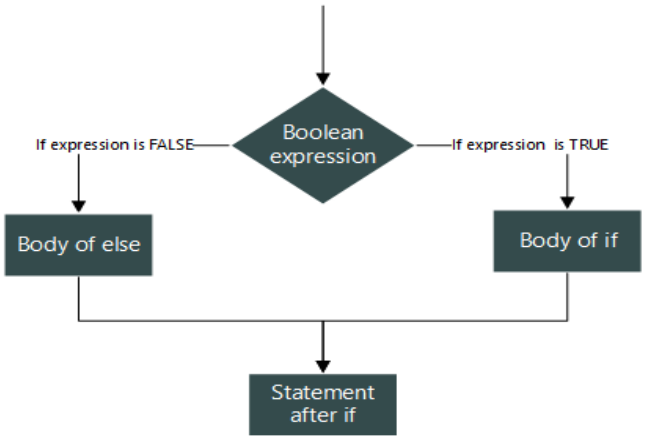


2) if .. else:

This conditional statement can execute true statements only when the condition is true otherwise it executes false statements.

Syn:

```
if expression:
        statement1;
        statement2;
else:
        statement1;
        statement2;
```



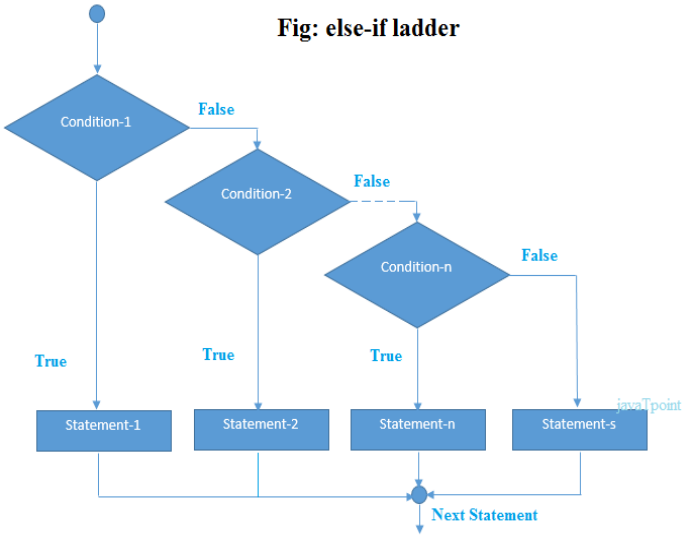
3) if .. elif.. else:

It is also called as a branching statement or Ladder statement. This conditional statement executes statements based on its respective condition.

Syn:

```
if expression1:
        statement1;
        statement2;
 elif expression2:
        statement1;
        statement2;
else:
        statement1;
        statement2;
```

Fig: else-if ladder

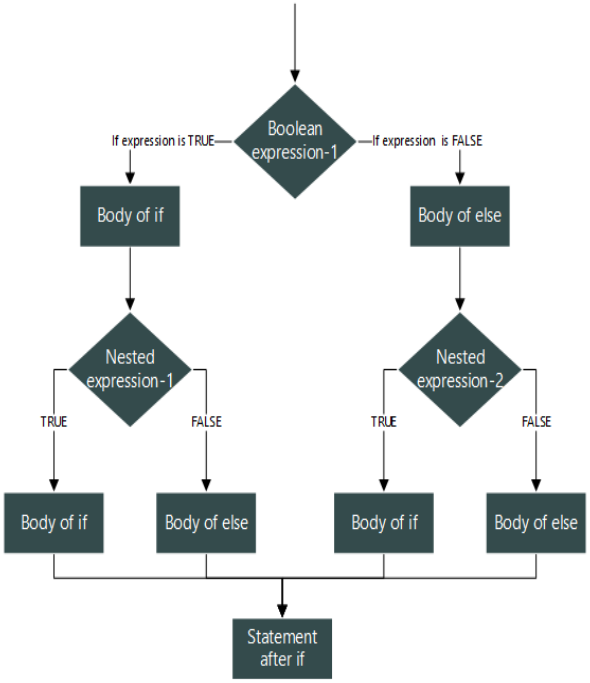


4) Nested if:

A if statement which can execute within if statement itself it is called as nested if conditional statement. It is also called as a multi-level branching statement. In a nested **if** construct, you can have an **if...elif...else** construct inside another **if...elif...else** construct.

Syn:

```
if exp1:
        if exp1.1:
                statement1
                statement2
        elif exp1.2:
                statement1
                statement2
          else:
                statement1;
                statement2;
elif exp2:
     if exp2.1:
                statement1;
                statement2;
        elif exp1.2:
                statement1;
                statement2;
```



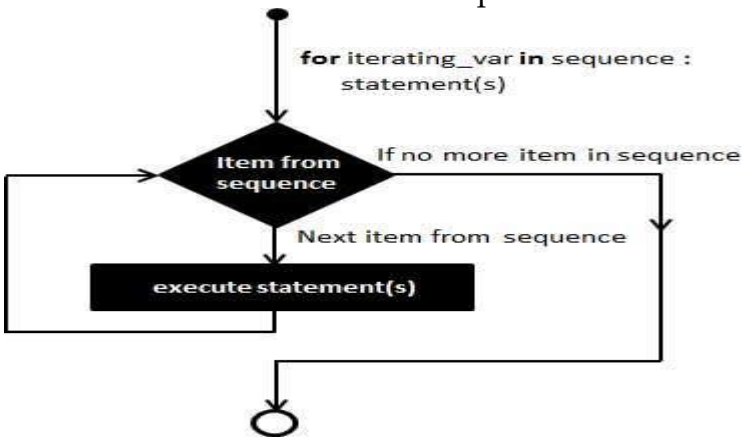
Loops

A loop statement allows us to execute a statement or group of statements multiple times. Python programming language provides following types of loops to handle looping requirements.

1) For..loop
2) While..loop
3) Nested loop

1. For..loop

- Python's for loop is designed to repeatedly execute a code block while iterating through a list, tuple, dictionary, or other iterable objects of Python. The process of traversing a sequence is known as iteration.
  Syn: for value in sequence:
  code block
- In this case, the variable value is used to hold the value of every item present in the sequence before the iteration begins until this particular iteration is completed.
- Loop iterates until the final item of the sequence are reached.



Examples:

Print the text "Hello, World!" 5 times.

```
list = [1, 2, 3, 4, 5]
for num in list:
   print("Hello, World!")
```

Note: The variable num is not used in the code, so we can use the below syntax (use underscore):

```
list = [1, 2, 3, 4, 5]
for _ in list:
   print("Hello, World!")
```

 # Prints out the numbers 0,1,2,3,4

```
for x in range(5):
   print(x)
# Prints out 3,4,5
for x in range(3, 6):
   print(x)
# Prints out 3,5,7
for x in range(3, 8, 2):
   print(x)
```

 # Program to find the sum of all numbers stored in a list

```
numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]
sum = 0
# iterate over the list
for val in numbers:
   sum = sum+val
print("The sum is", sum)
```

 # Program to Create a list of all the even numbers between 1 and 10

```
even_nums = []
for i in range(1, 11):
   if i % 2 == 0:
      even_nums.append(i)
print("Even Numbers: ", even_nums)
for letter in 'Python':     # First Example
  print 'Current Letter :', letter
fruits = ['banana', 'apple',  'mango']
for fruit in fruits:        # Second Example
  print 'Current fruit :', fruit
print "Good bye!"
```

o/p:

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : h
Current Letter : o
Current Letter : n
Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!
```

Eg:

```
fruits = ['banana', 'apple',  'mango']
for index in range(len(fruits)):
   print 'Current fruit :', fruits[index]
print "Good bye!"
```

o/p:

```
Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!
```

for loop with else

- A for loop can have an optional else block as well. The else part is executed if the items in the sequence used in for loop exhausts.
- The break keyword can be used to stop a for loop. In such cases, the else part is ignored.
- Hence, a for loop's else part runs if no break occurs.

Eg: to Use else with a for loop

```
iterator = (1, 2, 3, 4)
for item in iterator:
   print(item)
else:
   print("No more items in the iterator")
```

The range() function

- We can generate a sequence of numbers using range() function. range(10) will generate numbers from 0 to 9 (10 numbers).
- We can also define the start, stop and step size as range(start, stop,step_size). step_size defaults to 1 if not provided.
- The range object is "lazy" in a sense because it doesn't generate every number that it "contains" when we create it. However, it is not an iterator since it supports in, len and __getitem__ operations.
- This function does not store all the values in memory; it would be inefficient. So it remembers the start, stop, step size and generates the next number on the go.
- To force this function to output all the items, we can use the function list().

  Eg:

```
print(range(10))
print(list(range(10)))
print(list(range(2, 8)))
print(list(range(2, 20, 3)))
```

2. While Loop

While loops are used in Python to iterate until a specified condition is met. However, the statement in the program that follows the while loop is executed once the condition changes to false.

Syn: while <condition>:

code block

 # Python program to show how to use a while loop

```
counter = 0
while counter < 10:
   counter = counter + 3
   print("Python Loops")
```

Output:

```
Python Loops
Python Loops
Python Loops
Python Loops
```

Using else Statement with while Loops

We can use the else statement with the while loop also. It has the same syntax.

Syn: while <condition>:
code block
else:
statements

#Python program to show how to use else statement with the while loop

```
counter = 0
while (counter < 10):
   counter = counter + 3
   print("Python Loops") # Executed until condition is met
else:
   print("Code block inside the else statement")
```

Output:

```
Python Loops
Python Loops
Python Loops
Python Loops
Code block inside the else statement
```

Single statement while Block

- The loop can be declared in a single statement, as seen below.
- This is similar to the if-else block, where we can write the code block in a single line.

 # Python program to show how to write a single statement while loop

```
counter = 0
while (count < 3): print("Python Loops")
```

3. Nested loop

Python programming language allows to use one loop inside another loop.

Syn: for iterating_var in sequence:
for iterating_var in sequence:
statements(s)
statements(s)

Syn: while expression:
while expression:
statement(s)

```
        statement(s)
```

A final note on loop nesting is that you can put any type of loop inside of any other type of loop. For example a for loop can be inside a while loop or vice versa.

Truthiness

Booleans in Python work as in most other languages, except that they're capitalized:

```
one_is_less_than_two = 1 < 2        # equals True
true_equals_false = True == False # equals False
```

Python uses the value None to indicate a nonexistent value. It is similar to other languages' null:

```
x = None
print x == None      # prints True, but is not Pythonic
print x is None      # prints True, and is Pythonic
```

Python lets you use any value where it expects a Boolean. The following are all "Falsy":

- False
- None
- [] (an empty list)
- {} (an empty dict)
- ""
- set()
- 0
- 0.0

Pretty much anything else gets treated as True. This allows you to easily use if statements to test for empty lists or empty strings or empty dictionaries or so on. It also sometimes causes tricky bugs if you're not expecting this behavior:

```
s = some_function_that_returns_a_string()
if s:
first_char = s[0]
else:
first_char = ""
```

A simpler way of doing the same is:

```
first_char = s and s[0]
```

since and returns its second value when the first is "truthy," the first value when it's not. Similarly, if x is either a number or possibly None:

```
safe_x = x or 0
```

is definitely a number.

Python has an all function, which takes a list and returns True precisely when every element is truthy, and an any function, which returns True when at least one element is truthy:

```
all([True, 1, { 3 }])    # True
all([True, 1, {}])       # False, {} is falsy
any([True, 1, {}])       # True, True is truthy
all([])  # True, no falsy elements in the list
any([])# False, no truthy elements in the list
```

Sorting

- Every Python list has a sort method that sorts it in place. If you don't want to mess up your list, you can use the sorted function, which returns a new list:

```
x = [4,1,2,3]
y = sorted(x)
x.sort()
```

- By default, sort (and sorted) sort a list from smallest to largest based on naively comparing the elements to one another.
- If you want elements sorted from largest to smallest, you can specify a reverse=True parameter. And instead of comparing the elements themselves, you can compare the results of a function that you specify with key:

```
# sort the list by absolute value from largest to smallest
x = sorted([-4,1,-2,3], key=abs, reverse=True) # is [-4,3,-2,1]
# sort the words and counts from highest count to lowest
wc = sorted(word_counts.items(),
key=lambda (word, count): count, reverse=True)
```

List Comprehensions

A Python list comprehension consists of brackets containing the expression, which is executed for each element along with the for loop to iterate over each element in the Python list.

Advantages of List Comprehension

- More time-efficient and space-efficient than loops.
- Require fewer lines of code.
- Transforms iterative statement into a formula.

Syn: ***newList*** *= [* ***expression(element)*** *for* ***element*** *in* ***oldList*** *if* ***condition*** *]*

Example 1: Iteration with List comprehension

```
# Using list comprehension to iterate through loop
List = [character for character in [1, 2, 3]]
# Displaying list
print(List)
```

Example 2: Even list using list comprehension

```
list = [i for i in range(11) if i % 2 == 0]
print(list)
```

Example 3: Matrix using List comprehension

```
matrix = [[j for j in range(3)] for i in range(3)]
print(matrix)
```

Nested List Comprehensions

Nested List Comprehensions are nothing but a list comprehension within another list comprehension which is quite similar to nested for loops. Below is the program which implements nested loop:

```
matrix = []
for i in range(3):
        matrix.append([])
        for j in range(5):
                matrix[i].append(j)
print(matrix)
```

Iterators

Iterator in Python is an object that is used to iterate over iterable objects like lists, tuples, dicts, and sets. The iterator object is initialized using the **iter()** method. It uses the **next()** method for iteration.

1. **__iter__():** The iter() method is called for the initialization of an iterator. This returns an iterator object
2. **__next__():** The next method returns the next value for the iterable. When we use a for loop to traverse any iterable object, internally it uses the iter() method to get an iterator object, which further uses the next() method to iterate over. This method raises a StopIteration to signal the end of the iteration.

#Python iter() Example

```
string = "GFG"
ch_iterator = iter(string)
print(next(ch_iterator))
print(next(ch_iterator))
print(next(ch_iterator))
```

Output

```
G  F  G
```

Ex1: Creating and looping over an iterator using iter() and next()

```
class Test:
        # Constructor
        def __init__(self, limit):
                self.limit = limit
        def __iter__(self):
                self.x = 10
                return self
        def __next__(self):
                x = self.x
                if x > self.limit:
                        raise StopIteration
                self.x = x + 1;
                return x
# Prints numbers from 10 to 15
for i in Test(15):
        print(i)
# Prints nothing
for i in Test(5):
        print(i)
```

Output

```
10
11
12
13
14
15
```

Generator-Function:

A generator-function is defined like a normal function, but whenever it needs to generate a value, it does so with the yield keyword rather than return. If the body of a def contains yield, the function automatically becomes a generator function.

```
# A generator function that yields 1 for first time,
# 2 second time and 3 third time
def simpleGeneratorFun():
    yield 1
    yield 2
    yield 3
# Driver code to check above generator function
for value in simpleGeneratorFun():
    print(value)
```

Output

1
2
3

Generator-Object:

Generator functions return a generator object. Generator objects are used either by calling the next method on the generator object or using the generator object in a "for in" loop

```
# A Python program to demonstrate use of generator object with next()
def simpleGeneratorFun():
    yield 1
    yield 2
    yield 3
# x is a generator object
x = simpleGeneratorFun()
# Iterating over the generator object using next
print(next(x)) # In Python 3, __next__()
print(next(x))
print(next(x))
```

Output

1
2
3

So a generator function returns an generator object that is iterable, i.e., can be used as an Iterators .

```
# A simple generator for Fibonacci Numbers
def fib(limit):
    a, b = 0, 1
    while a < limit:
        yield a
        a, b = b, a + b
# Create a generator object
x = fib(5)
```

```
# Iterating over the generator object using next
print(next(x)) # In Python 3, __next__()
print(next(x))
print(next(x))
print(next(x))
print(next(x))
# Iterating over the generator object using for in loop.
print("\nUsing for in loop")
for i in fib(5):
        print(i)
```

Output

```
0
1
1
2
3
Using for in loop
0
1
1
2
3
```

Randomness

- Python defines a set of functions that are used to generate or manipulate random numbers through the **random module.**
- Functions in the random module rely on a pseudo-random number generator function **random()**, which generates a random float number between 0.0 and 1.0. These particular type of functions is used in a lot of games, lotteries, or any application requiring a random number generation.

Eg: for Generating Random Number using random() function

```
import random
num = random.random()
print(num)
```

Output:

```
0.30078080420602904
```

```
# Python3 program to demonstrate the use of# choice() method
import random
list1 = [1, 2, 3, 4, 5, 6]
print(random.choice(list1))
string = "striver"
print(random.choice(string))
```

Output

```
5
T
```

Method 2: Generating random number list in Python randrange(beg, end, step)

The random module offers a function that can generate random numbers from a specified range and also allows room for steps to be included, called **randrange().**

 # Python code to demonstrate the working of choice() and randrange()

```
import random
print("A random number from list is : ", end="")
print(random.choice([1, 4, 8, 10, 3]))
print("A random number from range is : ", end="")
print(random.randrange(20, 50, 3))
```

Output:

```
A random number from list is : 4
A random number from range is : 41
```

Method 3: Generating random number list in Python using seed()

The seed function is used to save the state of a random function so that it can generate some random numbers on multiple executions of the code on the same machine or on different machines (for a specific seed value). The seed value is the previous value number generated by the generator. For the first time when there is no previous value, it uses the current system time.

 # Python code to demonstrate the working of random() and seed()

```
import random
print("A random number between 0 and 1 is : ", end="")
print(random.random())
random.seed(5)
print("The mapped random number with 5 is : ", end="")
print(random.random())
random.seed(7)
print("The mapped random number with 7 is : ", end="")
print(random.random())
random.seed(5)
print("The mapped random number with 5 is : ", end="")
print(random.random())
random.seed(7)
print("The mapped random number with 7 is : ", end="")
print(random.random())
```

Output:

```
A random number between 0 and 1 is : 0.510721762520941
The mapped random number with 5 is : 0.6229016948897019
The mapped random number with 7 is : 0.32383276483316237
The mapped random number with 5 is : 0.6229016948897019
The mapped random number with 7 is : 0.32383276483316237
```

Method 4: Generating random number list in Python using shuffle()

- It is used to shuffle a sequence (list).
- Shuffling means changing the position of the elements of the sequence. Here, the shuffling operation is in place.

Eg:

```
import random
sample_list = ['A', 'B', 'C', 'D', 'E']
print("Original list : ")
print(sample_list)
random.shuffle(sample_list)
print("\nAfter the first shuffle : ")
print(sample_list)
random.shuffle(sample_list)
print("\nAfter the second shuffle : ")
print(sample_list)
```

Output:

```
Original list:
['A', 'B', 'C', 'D', 'E']
After the first shuffle :
['A', 'B', 'E', 'C', 'D']
After the second shuffle :
['C', 'E', 'B', 'D', 'A']
```

Regular Expressions

A regular expression is a set of characters with highly specialized syntax that we can use to find or match other characters or groups of characters. In short, regular expressions, or Regex, are widely used in the UNIX world.

The re-module in Python gives full support for regular expressions of Pearl style. The re module raises the re.error exception whenever an error occurs while implementing or using a regular expression.

We'll go over two crucial functions utilized to deal with regular expressions. But first, a minor point: many letters have a particular meaning when utilized in a regular expression.

re.match()

Python's re.match() function finds and delivers the very first appearance of a regular expression pattern. In Python, the RegEx Match function solely searches for a matching string at the beginning of the provided text to be searched. The matching object is produced if one match is found in the first line. If a match is found in a subsequent line, the Python RegEx Match function gives output as null.

Examine the implementation for the re.match() method in Python. The expressions ".w*" and ".w*?" will match words that have the letter "w," and anything that does not has the letter "w" will be ignored. The for loop is used in this Python re.match() illustration to inspect for matches for every element in the list of words.

Matching Characters

The majority of symbols and characters will easily match. (A case-insensitive feature can be enabled, allowing this RE to match Python or PYTHON.) The regular expression check, for instance, will match exactly the string check.

There are some exceptions to this general rule; certain symbols are special metacharacters that don't match. Rather, they indicate that they must compare something unusual, or they have an effect on other parts of the RE by recurring or modifying their meaning.

Here's the list of the metacharacters;

. ^ $ * + ? { } [ ] \ | ( )

Repeating Things

The ability to match different sets of symbols will be the first feature regular expressions can achieve that's not previously achievable with string techniques. On the other hand, Regexes isn't much of an improvement if that had been their only extra capacity. We can also define that some sections of the RE must be reiterated a specified number of times.

The first metacharacter we'll examine for recurring occurrences is *. Instead of matching the actual character '*,' * signals that the preceding letter can be matched 0 or even more times, rather than exactly one.

Ba*t, for example, matches 'bt' (zero 'a' characters), 'bat' (one 'a' character), 'baaat' (three 'a' characters), etc.

Greedy repetitions, such as *, cause the matching algorithm to attempt to replicate the RE as many times as feasible. If later elements of the sequence fail to match, the matching algorithm will retry with lesser repetitions.

Syn: re.match(pattern, string, flags=0)

Parameters

pattern:- this is the expression that is to be matched. It must be a regular expression

string:- This is the string that will be compared to the pattern at the start of the string.

flags:- Bitwise OR (|) can be used to express multiple flags. These are modifications, and the table below lists them.

Eg:

```
import re
line = "Learn Python through tutorials on javatpoint"
match_object = re.match( r'.w* (.w?) (.w*?)', line, re.M|re.I)
if match_object:
   print ("match object group : ", match_object.group())
   print ("match object 1 group : ", match_object.group(1))
   print ("match object 2 group : ", match_object.group(2))
else:
   print ( "There isn't any match!!" )
```

Output: There isn't any match!!

re.search()

The re.search() function will look for the first occurrence of a regular expression sequence and deliver it. It will verify all rows of the supplied string, unlike Python's re.match(). If the pattern is matched, the re.search() function produces a match object; otherwise, it returns "null."

To execute the search() function, we must first import the Python re-module and afterward run the program. The "sequence" and "content" to check from our primary string are passed to the Python re.search() call.

Syn: re.search(pattern, string, flags=0)

Here is the description of the parameters -

pattern:- this is the expression that is to be matched. It must be a regular expression

string:- The string provided is the one that will be searched for the pattern wherever within it.

flags:- Bitwise OR (|) can be used to express multiple flags. These are modifications, and the table below lists them.

```
import re
line = "Learn Python through tutorials on Futuresoft";
search_object = re.search( r' .*t? (.*t?) (.*t?)', line)
if search_object:
   print("search object group : ", search_object.group())
   print("search object group 1 : ", search_object.group(1))
   print("search object group 2 : ", search_object.group(2))
else:
   print("Nothing found!!")
```

Output:

```
search object group :   Python through tutorials on Futuresoft
search object group 1 :  on
search object group 2 :  Futuresoft
```

Object-Oriented Programming

Python is a multi-paradigm programming language. It supports different programming approaches. One of the popular approaches to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP). The concept of OOP in Python focuses on creating reusable code.

An object has two characteristics:

- attributes
- behavior

Let's take an example:

A parrot is an object, as it has the following properties:

name, age, color as attributes

singing, dancing as behavior

In Python, the concept of OOP follows some basic principles:

Class

- A class is a blueprint for the object.
- We can think of class as a sketch of a parrot with labels. It contains all the details about the name, colors, size etc. Based on these descriptions, we can study about the parrot. Here, a parrot is an object.

Object

An object (instance) is an instantiation of a class. When class is defined, only the description for the object is defined. Therefore, no memory or storage is allocated.

The example for object of parrot class can be:

obj = Parrot()

Here, obj is an object of class Parrot.

Suppose we have details of parrots. Now, we are going to show how to build the class and objects of parrots.

Example 1: Creating Class and Object in Python

```
class Parrot:
    species = "bird"
    # instance attribute
    def __init__(self, name, age):
        self.name = name
        self.age = age
# instantiate the Parrot class
blu = Parrot("Blu", 10)
woo = Parrot("Woo", 15)
# access the class attributes
print("Blu is a {}".format(blu.__class__.species))
print("Woo is also a {}".format(woo.__class__.species))
# access the instance attributes
print("{} is {} years old".format( blu.name, blu.age))
print("{} is {} years old".format( woo.name, woo.age))
```

Output

```
Blu is a bird
Woo is also a bird
Blu is 10 years old
Woo is 15 years old
```

Methods

Methods are functions defined inside the body of a class. They are used to define the behaviors of an object.

Example 2 : Creating Methods in Python

```
class Parrot:
    # instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age
    # instance method
```

```
    def sing(self, song):
        return "{} sings {}".format(self.name, song)
    def dance(self):
        return "{} is now dancing".format(self.name)
# instantiate the object
blu = Parrot("Blu", 10)
# call our instance methods
print(blu.sing("'Happy'"))
print(blu.dance())
```

Output

Blu sings 'Happy'
Blu is now dancing

Inheritance

Inheritance is a way of creating a new class for using details of an existing class without modifying it. The newly formed class is a derived class (or child class). Similarly, the existing class is a base class (or parent class).

Example 3: Use of Inheritance in Python

```
# parent class
class Bird:
    def __init__(self):
        print("Bird is ready")
    def whoisThis(self):
        print("Bird")
    def swim(self):
        print("Swim faster")
# child class
class Penguin(Bird):
    def __init__(self):
        # call super() function
        super().__init__()
        print("Penguin is ready")
    def whoisThis(self):
        print("Penguin")
    def run(self):
        print("Run faster")
peggy = Penguin()
peggy.whoisThis()
peggy.swim()
peggy.run()
```

Output

Bird is ready
Penguin is ready
Penguin
Swim faster
Run faster

Encapsulation

Using OOP in Python, we can restrict access to methods and variables. This prevents data from direct modification which is called encapsulation. In Python, we denote private attributes using underscore as the prefix i.e single _ or double __.

Example 4: Data Encapsulation in Python

```
class Computer:
    def __init__(self):
        self.__maxprice = 900
    def sell(self):
        print("Selling Price: {}".format(self.__maxprice))
    def setMaxPrice(self, price):
        self.__maxprice = price
c = Computer()
c.sell()
# change the price
c.__maxprice = 1000
c.sell()
# using setter function
c.setMaxPrice(1000)
c.sell()
```

Output

```
Selling Price: 900
Selling Price: 900
Selling Price: 1000
```

Polymorphism

Polymorphism is an ability (in OOP) to use a common interface for multiple forms (data types). Suppose, we need to color a shape, there are multiple shape options (rectangle, square, circle). However we could use the same method to color any shape. This concept is called Polymorphism.

Example 5: Using Polymorphism in Python

```
class Parrot:
    def fly(self):
        print("Parrot can fly")
    def swim(self):
        print("Parrot can't swim")
class Penguin:
    def fly(self):
        print("Penguin can't fly")
     def swim(self):
        print("Penguin can swim")
# common interface
def flying_test(bird):
    bird.fly()
#instantiate objects
blu = Parrot()
peggy = Penguin()
# passing the object
flying_test(blu)
flying_test(peggy)
```

Output: Parrot can fly

Matplotlib

A wide variety of tools exists for visualizing data. We will be using the matplotlib library, which is widely used (although sort of showing its age). If you are interested in producing elaborate interactive visualizations for the Web, it is likely not the right choice, but for simple bar charts, line charts, and scatterplots, it works pretty well.

In particular, we will be using the matplotlib.pyplot module. In its simplest use, pyplot maintains an internal state in which you build up a visualization step by step. Once you're done, you can save it (with savefig()) or display it (with show()).
For example, making simple plots (like Figure 3-1) is pretty simple:

```
from matplotlib import pyplot as plt

years = [1950, 1960, 1970, 1980, 1990, 2000, 2010]
gdp = [300.2, 543.3, 1075.9, 2862.5, 5979.6, 10289.7, 14958.3]
# create a line chart, years on x-axis, gdp on y-axis
plt.plot(years, gdp, color='green', marker='o', linestyle='solid')
# add a title
plt.title("Nominal GDP")
plt.ylabel("Billions of $")
plt.show()
```

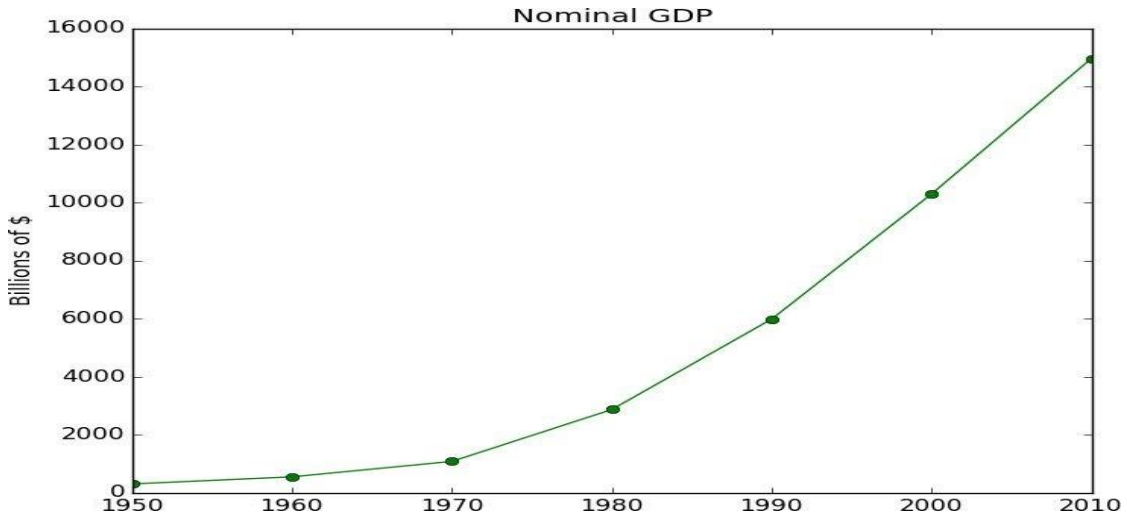


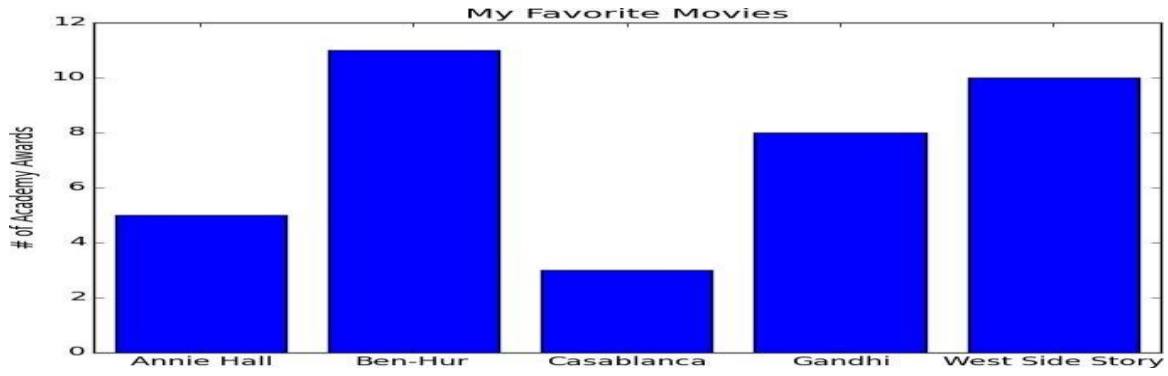
Figure 3-1. A simple line chart

Making plots that look publication-quality good is more complicated and beyond the scope of this chapter. There are many ways you can customize your charts with (for example) axis labels, line styles, and point markers. Rather than attempt a comprehensivetreatment of these options,

Bar Charts

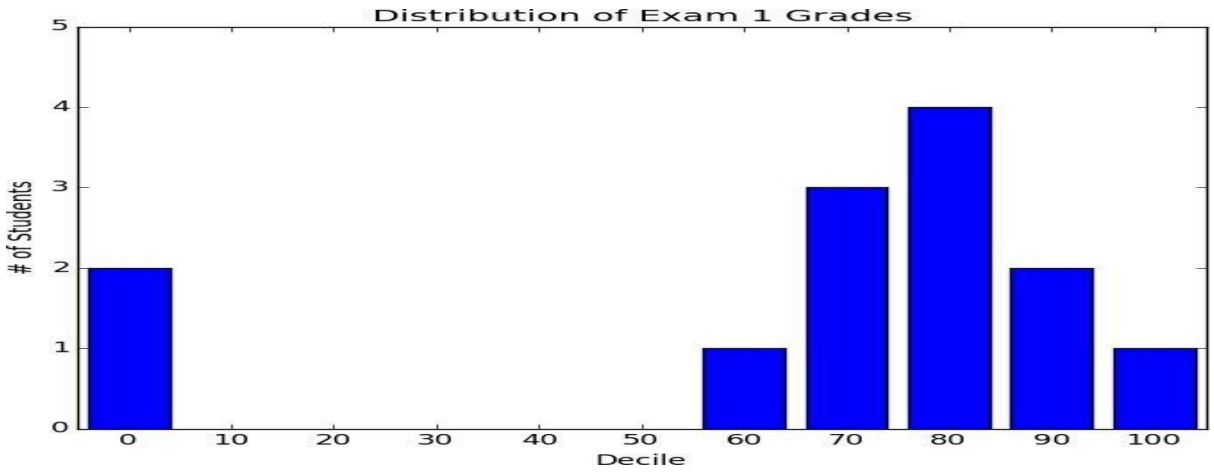
A bar chart is a good choice when you want to show how some quantity varies among some discrete set of items. For instance, Figure 3-2 shows how many Academy Awards were won by each of a variety of movies:

```
movies = ["Annie Hall", "Ben-Hur", "Casablanca", "Gandhi", "West Side Story"]
num_oscars = [5, 11, 3, 8, 10]

xs = [i + 0.1 for i, _ in enumerate(movies)]
plt.bar(xs, num_oscars)

plt.ylabel("# of Academy Awards")
plt.title("My Favorite Movies")
plt.xticks([i + 0.5 for i, _ in enumerate(movies)], movies)
plt.show()
```



A bar chart can also be a good choice for plotting histograms of bucketed numeric values, in order to visually explore how the values are distributed, as in Figure 3-3:

```
grades = [83,95,91,87,70,0,85,82,100,67,73,77,0]
decile = lambda grade: grade // 10 * 10
histogram = Counter(decile(grade) for grade in grades)
plt.bar([x - 4 for x in histogram.keys()], histogram.values(),8)
plt.axis([-5, 105, 0, 5])          # x-axis from -5 to 105, # y-axis from 0 to 5
plt.xticks([10 * i for i in range(11)])          # x-axis labels at 0, 10, ..., 100
plt.xlabel("Decile") plt.ylabel("# of Students")
plt.title("Distribution of Exam 1 Grades")
plt.show();
```

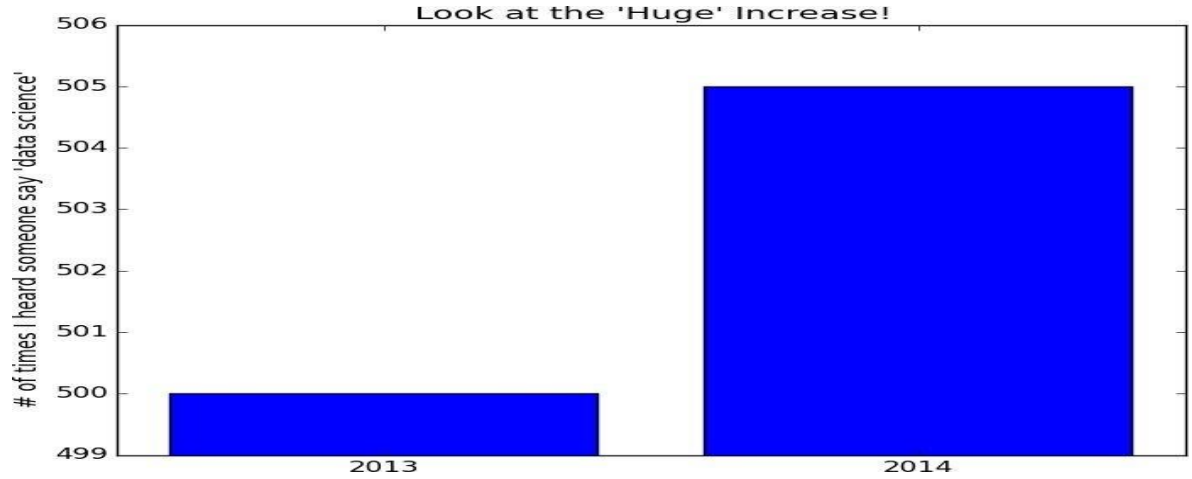


The third argument to plt.bar specifies the bar width. Here we chose a width of 8. And we shifted the bar left by 4, so that (for example) the "80" bar has its left and right sides at 76 and 84, and (hence) its center at 80.

The call to plt.axis indicates that we want the x-axis to range from -5 to 105 (so that the "0" and "100" bars are fully shown), and that the y-axis should range from 0 to 5. And the call to plt.xticks puts x-axis labels at 0, 10, 20, …, 100.

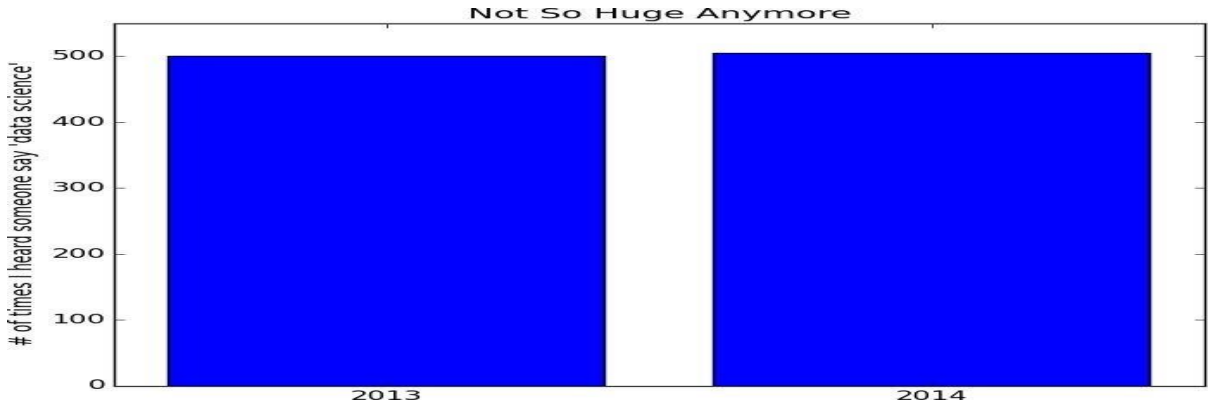
Be judicious when using plt.axis(). When creating bar charts it is considered especially bad form for your y-axis not to start at 0, since this is an easy way to mislead people (Figure 3-4):

```
mentions = [500, 505]
years = [2013, 2014]
plt.bar([2012.6, 2013.6], mentions, 0.8) plt.xticks(years)
plt.ylabel("# of times I heard someone say 'data science'")
plt.ticklabel_format(useOffset=False)
plt.axis([2012.5,2014.5,499,506])
plt.title("Look at the 'Huge' Increase!")
plt.show()
```



In Figure 3-5, we use more-sensible axes, and it looks far less impressive:

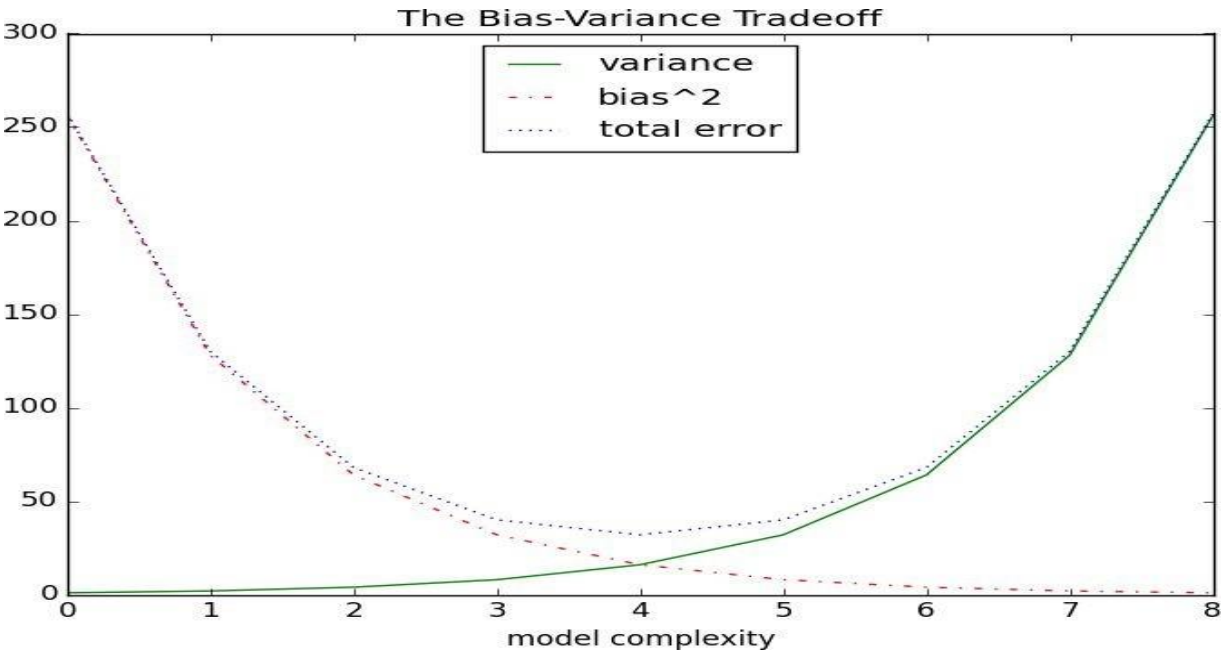
```
plt.axis([2012.5,2014.5,0,550])
plt.title("Not So Huge Anymore") plt.show()
```



Line Charts

As we saw already, we can make line charts using plt.plot(). These are a good choice for showing trends, as illustrated in Figure 3-6:

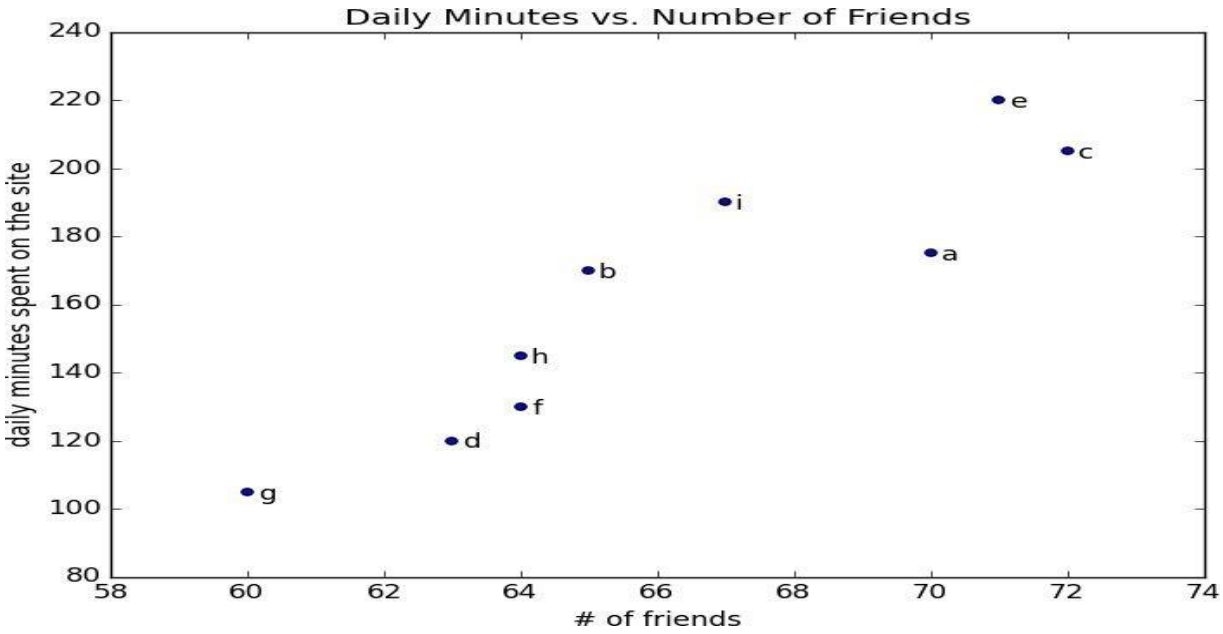
```
variance     = [1, 2, 4, 8, 16, 32, 64, 128, 256]
bias_squared = [256, 128, 64, 32, 16, 8, 4, 2, 1]
total_error = [x + y for x, y in zip(variance, bias_squared)] xs = [i for i, _ in enumerate(variance)]

plt.plot(xs, variance,     'g-', label='variance')
plt.plot(xs, bias_squared, 'r-.', label='bias^2')
plt.plot(xs, total_error, 'b:', label='total error')
plt.legend(loc=9) plt.xlabel("model complexity")
plt.title("The Bias-Variance Tradeoff") plt.show()
```



Scatterplots

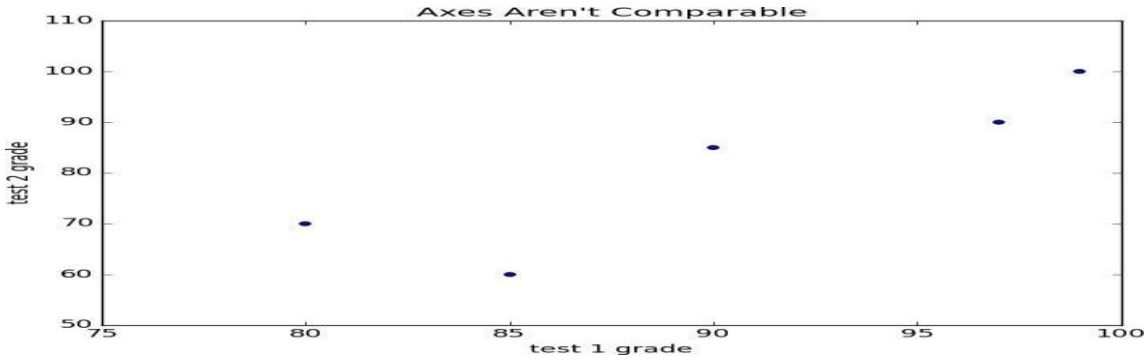
A scatterplot is the right choice for visualizing the relationship between two paired sets of data. For example, Figure 3-7 illustrates the relationship between the number of friends your users have and the number of minutes they spend on the site every day:

```
friends = [ 70, 65, 72, 63, 71, 64, 60, 64, 67]
minutes = [175, 170, 205, 120, 220, 130, 105, 145, 190]
labels = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i'] plt.scatter(friends, minutes)
for label, friend_count, minute_count in zip(labels, friends, minutes): plt.annotate(label,
xy=(friend_count, minute_count), # put the label with its point xytext=(5, -5),     # but slightly offset textcoords='offset points')
plt.title("Daily Minutes vs. Number of Friends") plt.xlabel("# of friends")
plt.ylabel("daily minutes spent on the site") plt.show()
```

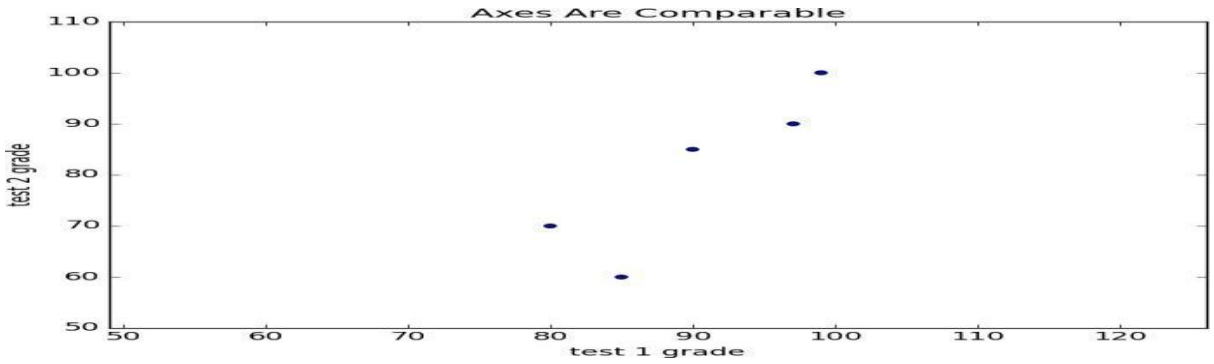


If you're scattering comparable variables, you might get a misleading picture if you let matplotlibchoose the scale, as in Figure 3-8

```
test_1_grades = [ 99, 90, 85, 97, 80]
test_2_grades = [100, 85, 60, 90, 70]
plt.scatter(test_1_grades, test_2_grades)plt.title("Axes Aren't Comparable")
plt.xlabel("test 1 grade")
plt.ylabel("test 2 grade")
plt.show()
```



If we include a call to `plt.axis("equal")`, the plot (Figure 3-9) more accurately showsthat most of the variation occurs on test 2.



UNIT – II

Describing a Single Set of Data

Through a combination of word-of-mouth and luck, DataSciencester has grown to dozensof members, and the VP of Fundraising asks you for some sort of description of how manyfriends your members have that he can include in his elevator pitches.

Using techniques from Chapter 1, you are easily able to produce this data. But now youare faced with the problem of how to *describe* it.

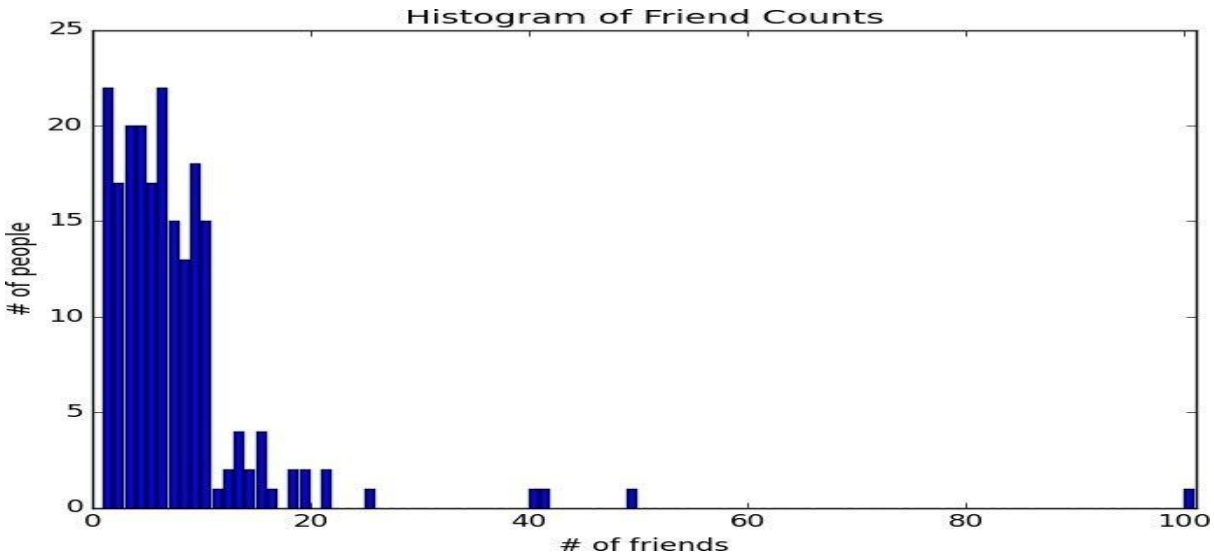
One obvious description of any data set is simply the data itself:

```
num_friends = [100, 49, 41, 40, 25,

                # ... and lots more

              ]
```

For a small enough data set this might even be the best description. But for a larger data set, this is unwieldy and probably opaque. For that reason we use statistics to distill and communicate relevant features of our data.

As a first approach you put the friend counts into a histogram using Counter and plt.bar() (Figure 5-1):

```
friend_counts = Counter(num_friends)
xs = range(101)                         # largest value is 100
ys = [friend_counts[x] for x in xs]     # height is just # of friends
plt.bar(xs, ys) plt.axis([0, 101, 0, 25])
plt.title("Histogram of Friend Counts")
plt.xlabel("# of  friends")
plt.ylabel("# of people")
plt.show()
```



Unfortunately, this chart is still too difficult to slip into conversations. So you start generating some statistics. Probably the simplest statistic is simply the number of datapoints:

```
num_points = len(num_friends)              # 204
```

You're probably also interested in the largest and smallest values:

```
largest_value = max(num_friends)           # 100
smallest_value = min(num_friends)          # 1
```

which are just special cases of wanting to know the values in specific positions:

```
sorted_values = sorted(num_friends)
smallest_value = sorted_values[0]          # 1
second_smallest_value = sorted_values[1] #1
second_largest_value = sorted_values[-2]  # 49
```

But we're only getting started.

Correlation

DataSciencester's VP of Growth has a theory that the amount of time people spend on thesite is related to the number of friends they have on the site and she's asked you to verify this.

After digging through traffic logs, you've come up with a list daily_minutes that shows how many minutes per day each user spends on DataSciencester, and you've ordered it so that its elements correspond to the elements of our previous num_friends list. We'd like to investigate the relationship between these two metrics.

We'll first look at *covariance*, the paired analogue of variance. Whereas variance measures how a single variable deviates from its mean, covariance measures how twovariables vary in tandem from their means:

```
def covariance(x, y):n = len(x)
        return dot(de_mean(x), de_mean(y)) / (n - 1)covariance(num_friends,
    daily_minutes) # 22.43
```

Recall that dot sums up the products of corresponding pairs of elements. When corresponding elements of x and y are either both above their means or both below their means, a positive number enters the sum. When one is above its mean and the other below, a negative number enters the sum. Accordingly, a "large" positive covariance means that xtends to be large when yis large and small when yis small. A "large" negative covariance means the opposite — that x tends to be small when y is large and vice versa. A covariance close to zero means that no such relationship exists.

Nonetheless, this number can be hard to interpret, for a couple of reasons:

- Its units are the product of the inputs' units (e.g., friend-minutes-per-day), which can behard to make sense of. (What's a "friend-minute-per-day"?)
- If each user had twice as many friends (but the same number of minutes), the covariance would be twice as large. But in a sense the variables would be just as interrelated. Said differently, it's hard to say what counts as a "large" covariance.

For this reason, it's more common to look at the *correlation*, which divides out the standard deviations of both variables:

```
def correlation(x, y):
        stdev_x = standard_deviation(x)
        stdev_y = standard_deviation(y)
        if stdev_x > 0 and stdev_y > 0:
            return covariance(x, y) / stdev_x / stdev_y
        else:
            return 0     # if no variation, correlation is zero
    correlation(num_friends, daily_minutes) # 0.25
```

The correlation is unitless and always lies between -1 (perfect anti-correlation) and 1 (perfect correlation). A number like 0.25 represents a relatively weak positive correlation.

However, one thing we neglected to do was examine our data. Check out Figure 5-2.

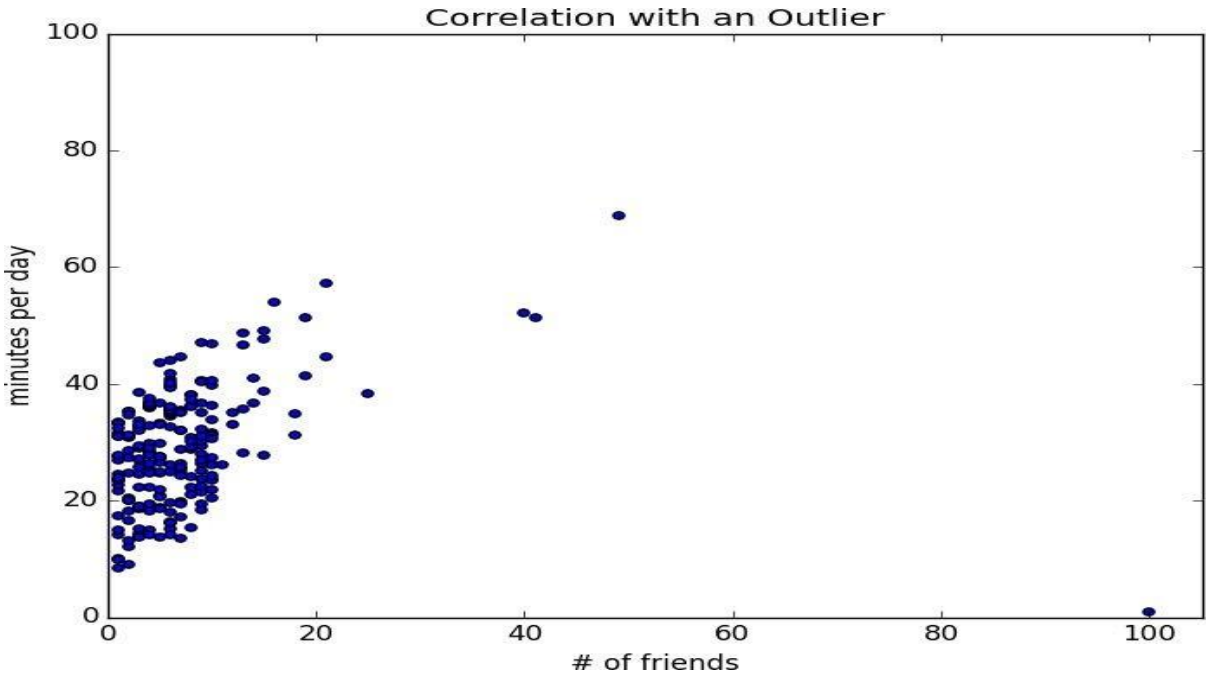


Figure 5-2. Correlation with an outlier

The person with 100 friends (who spends only one minute per day on the site) is a huge outlier, and correlation can be very sensitive to outliers. What happens if we ignore him?

```
outlier = num_friends.index(100)   # index of outlier
num_friends_good = [x
                    for i, x in enumerate(num_friends)
                    if i != outlier]
daily_minutes_good = [x
                      for i, x in enumerate(daily_minutes)
                      if      i      !=      outlier]
correlation(num_friends_good,
daily_minutes_good) # 0.57
```

Without the outlier, there is a much stronger correlation (Figure 5-3).

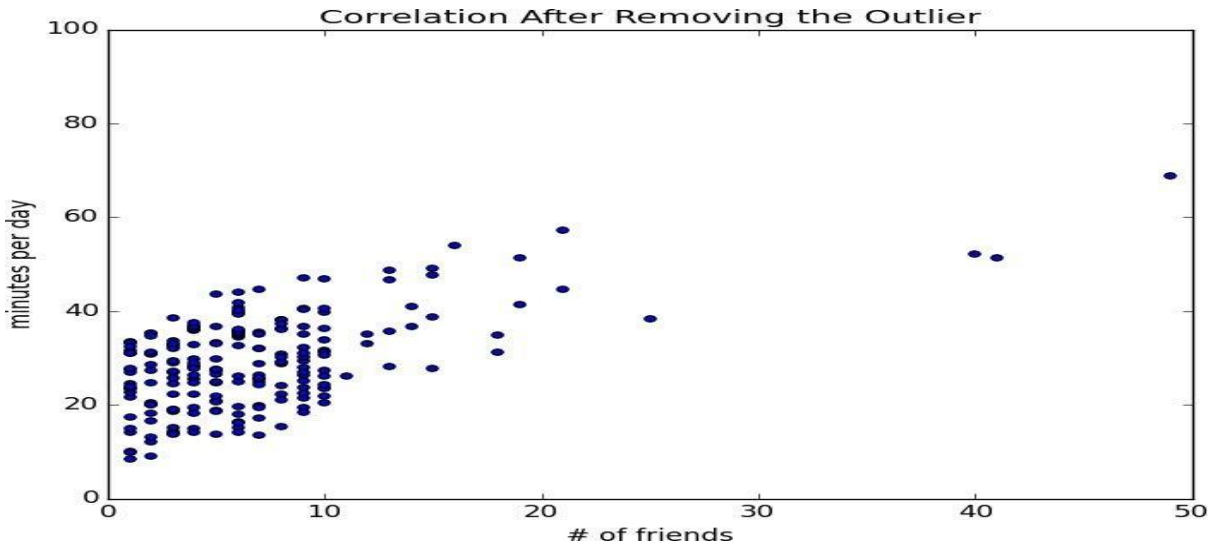


Figure 5-3. Correlation after removing the outlier

You investigate further and discover that the outlier was actually an internal *test* accountthat no one ever bothered to remove. So you feel pretty justified in excluding it.

Simpson's Paradox

One not uncommon surprise when analyzing data is Simpson's Paradox, in which correlations can be misleading when *confounding* variables are ignored.

For example, imagine that you can identify all of your members as either East Coast data scientists or West Coast data scientists. You decide to examine which coast's data scientists are friendlier:

coast	# of members	avg. # of friends
West Coast	101	8.2
East Coast	103	6.5

It certainly looks like the West Coast data scientists are friendlier than the East Coast data scientists. Your coworkers advance all sorts of theories as to why this might be: maybe it'sthe sun, or the coffee, or the organic produce, or the laid-back Pacific vibe?

When playing with the data you discover something very strange. If you only look at people with PhDs, the East Coast data scientists have more friends on average. And if youonly look at people without PhDs, the East Coast data scientists also have more friends onaverage!

coast	degree	# of members	avg. # of friends
West Coast	PhD	35	3.1
East Coast	PhD	70	3.2
West Coast	no PhD	66	10.9
East Coast	no PhD	33	13.4

Once you account for the users' degrees, the correlation goes in the opposite direction! Bucketing the data as East Coast/West Coast disguised the fact that the East Coast datascientists skew much more heavily toward PhD types.

This phenomenon crops up in the real world with some regularity. The key issue is that correlation is measuring the relationship between your two variables *all else being equal*. If your data classes are assigned at random, as they might be in a well-designed experiment, "all else being equal" might not be a terrible assumption. But when there is a deeper pattern to class assignments, "all else being equal" can be an awful assumption.

The only real way to avoid this is by *knowing your data* and by doing what you can to make sure you've checked for possible confounding factors. Obviously, this is not always possible. If you didn't have the educational attainment of these 200 data scientists, you might simply conclude that there was something inherently more sociable about the WestCoast

Some Other Correlational Caveats

A correlation of zero indicates that there is no linear relationship between the two variables. However, there may be other sorts of relationships.

For example, if:

x = [-2, -1, 0, 1, 2]
y = [2, 1, 0, 1, 2]

then x and y have zero correlation. But they certainly have a relationship — each element of y equals the absolute value of the corresponding element of x. What they don't have is a relationship in which knowing how x_i compares to mean(x) gives us information about how y_i compares to mean(y). That is the sort of relationship that correlation looks for.

In addition, correlation tells you nothing about how large the relationship is. The variables:

x = [-2, 1, 0, 1, 2]
y = [99.98, 99.99, 100, 100.01, 100.02]

are perfectly correlated, but (depending on what you're measuring) it's quite possible thatthis relationship isn't all that interesting.

Correlation and Causation

You have probably heard at some point that "correlation is not causation," most likely by someone looking at data that posed a challenge to parts of his worldview that he was reluctant to question. Nonetheless, this is an important point — if x and y are strongly correlated, that might mean that x causes y, that y causes x, that each causes the other, that some third factor causes both, or it might mean nothing.

Consider the relationship between num_friends and daily_minutes. It's possible that having more friends on the site *causes* DataSciencester users to spend more time on thesite. This might be the case if each friend posts a certain amount of content each day, which means that the more friends you have, the more time it takes to stay current with their updates.

However, it's also possible that the more time you spend arguing in the DataSciencester forums, the more you encounter and befriend like-minded people. That is, spending moretime on the site *causes* users to have more friends.

A third possibility is that the users who are most passionate about data science spend moretime on the site (because they find it more interesting) and more actively collect data science friends (because they don't want to associate with anyone else).

One way to feel more confident about causality is by conducting randomized trials. If you can randomly split your users into two groups with similar demographics and give one of the groups a slightly different experience, then you can often feel pretty good that the different experiences are causing the different outcomes.

For instance, if you don't mind being angrily accused of experimenting on your users, youcould randomly choose a subset of your users and show them content from only a fractionof their friends. If this subset subsequently spent less time on the site, this would give you some confidence that having more friends *causes* more time on the site.

Dependence and Independence

Roughly speaking, we say that two events E and F are *dependent* if knowing something about whether E happens gives us information about whether F happens (and vice versa).Otherwise they are *independent*.

For instance, if we flip a fair coin twice, knowing whether the first flip is Heads gives us no information about whether the second flip is Heads. These events are independent. On the other hand, knowing whether the first flip is Heads certainly gives us information about whether both flips are Tails. (If the first flip is Heads, then definitely it's not the casethat both flips are Tails.) These two events are dependent.

Mathematically, we say that two events E and F are independent if the probability thatthey both happen is the product of the probabilities that each one happens:

$$P(E, F) = P(E)P(F)$$

In the example above, the probability of "first flip Heads" is 1/2, and the probability of "both flips Tails" is 1/4, but the probability of "first flip Heads *and* both flips Tails" is 0.

Conditional Probability

When two events *E* and *F* are independent, then by definition we have:

$$P(E, F) = P(E)P(F)$$

If they are not necessarily independent (and if the probability of *F* is not zero), then wedefine the probability of *E* "conditional on *F*" as:

$$P(E \mid F) = P(E, F) / P(F)$$

You should think of this as the probability that *E* happens, given that we know that *F*happens.

We often rewrite this as:

$$P(E, F) = P(E \mid F)P(F)$$

When *E* and *F* are independent, you can check that this gives:

$$P(E \mid F) = P(E)$$

which is the mathematical way of expressing that knowing *F* occurred gives us noadditional information about whether *E* occurred.

One common tricky example involves a family with two (unknown) children. If we assume that:

1. Each child is equally likely to be a boy or a girl
2. The gender of the second child is independent of the gender of the first child

then the event "no girls" has probability 1/4, the event "one girl, one boy" has probability1/2, and the event "two girls" has probability 1/4.

Now we can ask what is the probability of the event "both children are girls" (*B*) conditional on the event "the older child is a girl" (*G*)? Using the definition of conditionalprobability:

$$P(B \mid G) = P(B, G) / P(G) = P(B) / P(G) = 1 / 2$$

since the event *B* and *G* ("both children are girls *and* the older child is a girl") is just theevent *B*.

Most likely this result accords with your intuition.

We could also ask about the probability of the event "both children are girls" conditional on the event "at least one of the children is a girl" (*L*). Surprisingly, the answer is differentfrom before!

As before, the event *B* and *L* ("both children are girls *and* at least one of the children is agirl") is just the event *B*. This means we have:

$$P(B \mid L) = P(B, L) / P(L) = P(B) / P(L) = 1 / 3$$

How can this be the case? Well, if all you know is that at least one of the children is a girl,then it is twice as likely that the family has one boy and one girl than that it has both girls.

We can check this by "generating" a lot of families:

```
def random_kid():
    return random.choice(["boy", "girl"])
both_girls = 0
older_girl = 0
either_girl = 0

random.seed(0)
for _ in range(10000):
younger = random_kid()
older = random_kid()
if older == "girl":
        older_girl += 1
    if older == "girl" and younger == "girl":both_girls += 1
    if older == "girl" or younger == "girl":either_girl += 1
print "P(both | older):", both_girls / older_girl       # 0.514 ~ 1/2
print "P(both | either): ", both_girls / either_girl     # 0.342 ~ 1/3
```

Bayes's Theorem

One of the data scientist's best friends is Bayes's Theorem, which is a way of "reversing" conditional probabilities. Let's say we need to know the probability of some event *E* conditional on some other event *F* occurring. But we only have information about the probability of *F* conditional on *E* occurring. Using the definition of conditional probabilitytwice tells us that:

$$P(E \mid F) = P(E, F) / P(F) = P(F \mid E)P(E) / P(F)$$

The event *F* can be split into the two mutually exclusive events "*F* and *E*" and "*F* and not E." If we write $\neg E$ for "not *E*" (i.e., "*E* doesn't happen"), then:

$$P(F) = P(F, E) + P(F, \neg E)$$

so that:

$$P(E \mid F) = P(F \mid E)P(E) / [P(F \mid E)P(E) + P(F \mid \neg E)P(\neg E)]$$

which is how Bayes's Theorem is often stated.

This theorem often gets used to demonstrate why data scientists are smarter than doctors. Imagine a certain disease that affects 1 in every 10,000 people. And imagine that there is a test for this disease that gives the correct result ("diseased" if you have the disease, "nondiseased" if you don't) 99% of the time.

What does a positive test mean? Let's use *T* for the event "your test is positive" and *D* for the event "you have the disease." Then Bayes's Theorem says that the probability that you have the disease, conditional on testing positive, is:

$$P(D \mid T) = P(T \mid D)P(D) / [P(T \mid D)P(D) + P(T \mid \neg D)P(\neg D)]$$

Here we know that $P(T \mid D)$, the probability that someone with the disease tests positive, is 0.99. $P(D)$, the probability that any given person has the disease, is 1/10,000 = 0.0001. $P(T \mid \neg D)$, the probability that someone without the disease tests positive, is 0.01. And $P(\neg D)$, the probability that any given person doesn't have the disease, is 0.9999. If you substitute these numbers into Bayes's Theorem you find

$$P(D \mid T) = 0.98\ \%$$

That is, less than 1% of the people who test positive actually have the disease.

While this is a simple calculation for a data scientist, most doctors will guess that $P(D \mid T)$ is approximately 2.

A more intuitive way to see this is to imagine a population of 1 million people. You'd expect 100 of them to have the disease, and 99 of those 100 to test positive. On the otherhand, you'd expect 999,900 of them not to have the disease, and 9,999 of those to test positive. Which means that you'd expect only 99 out of (99 + 9999) positive testers to actually have the disease.

Random Variables

A *random variable* is a variable whose possible values have an associated probability distribution. A very simple random variable equals 1 if a coin flip turns up heads and 0 if the flip turns up tails. A more complicated one might measure the number of heads observed when flipping a coin 10 times or a

value picked from `range(10)` where each number is equally likely.

The associated distribution gives the probabilities that the variable realizes each of its possible values. The coin flip variable equals 0 with probability 0.5 and 1 with probability 0.5. The `range(10)` variable has a distribution that assigns probability 0.1 to each of thenumbers from 0 to 9.

We will sometimes talk about the *expected value* of a random variable, which is the average of its values weighted by their probabilities. The coin flip variable has an expected value of 1/2 (= 0 * 1/2 + 1 * 1/2), and the `range(10)` variable has an expectedvalue of 4.5.

Random variables can be *conditioned* on events just as other events can. Going back to thetwo-child example from "Conditional Probability", if X is the random variable representing the number of girls, X equals 0 with probability 1/4, 1 with probability 1/2, and 2 with probability 1/4.

We can define a new random variable Y that gives the number of girls conditional on at least one of the children being a girl. Then Y equals 1 with probability 2/3 and 2 with probability 1/3. And a variable Z that's the number of girls conditional on the older childbeing a girl equals 1 with probability 1/2 and 2 with probability 1/2.

For the most part, we will be using random variables *implicitly* in what we do without calling special attention to them. But if you look deeply you'll see them.

Continuous Distributions

A coin flip corresponds to a *discrete distribution* — one that associates positive probability with discrete outcomes. Often we'll want to model distributions across a continuum of outcomes. (For our purposes, these outcomes will always be real numbers, although that's not always the case in real life.) For example, the *uniform distribution* puts*equal weight* on all the numbers between 0 and 1.

Because there are infinitely many numbers between 0 and 1, this means that the weight itassigns to individual points must necessarily be zero. For this reason, we represent a continuous distribution with a *probability density function* (pdf) such that the probability of seeing a value in a certain interval equals the integral of the density function over the interval.

The density function for the uniform distribution is just:

```
def uniform_pdf(x):
    return 1 if x >= 0 and x < 1 else 0
```

The probability that a random variable following that distribution is between 0.2 and 0.3 is1/10, as you'd expect. Python's random.random() is a [pseudo]random variable with a uniform density.

We will often be more interested in the *cumulative distribution function* (cdf), which gives the probability that a random variable is less than or equal to a certain value. It's not hard to create the cumulative distribution function for the uniform distribution (Figure 6-1):

```
def uniform_cdf(x):
    if x < 0:
    return 0
    elif x < 1:
    return x
   else:   return 1
```

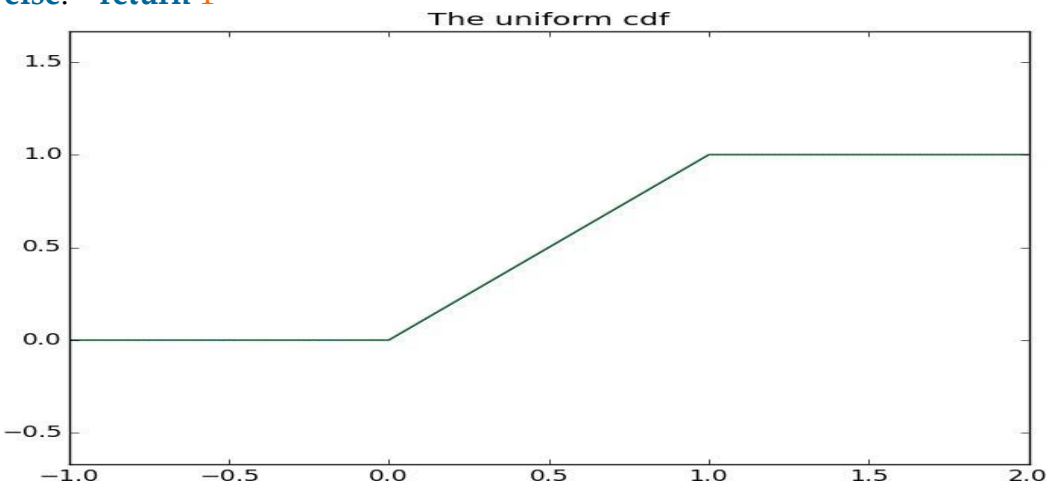


Figure 6-1. The uniform cdf

The Normal Distribution

The normal distribution is the king of distributions. It is the classic bell curve–shaped distribution and is completely determined by two parameters: its mean μ (mu) and itsstandard deviation (sigma). The mean indicates where the bell is centered, and the standard deviation σ how "wide" it is.

It has the distribution function:

$$f(x \mid \mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

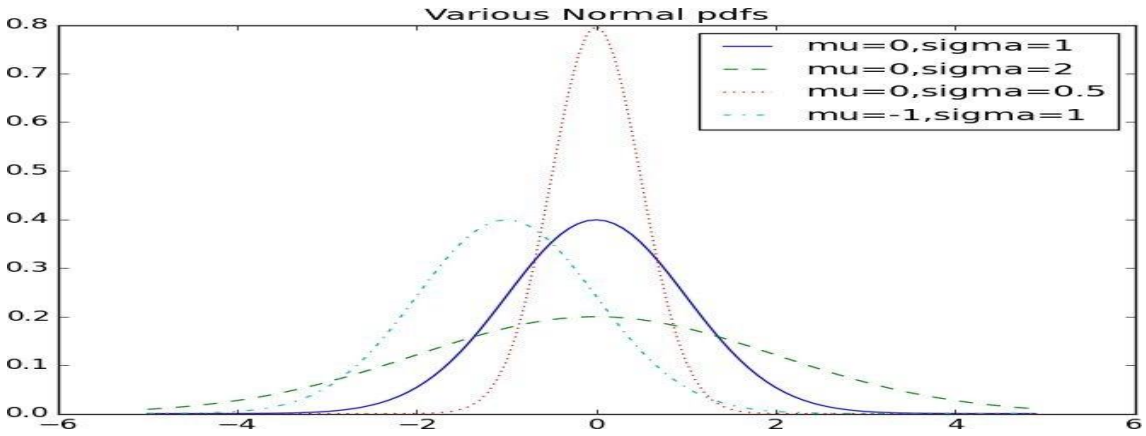
which we can implement as:

```
def normal_pdf(x, mu=0,
    sigma=1):
    sqrt_two_pi =
    math.sqrt(2 *
    math.pi)
    return (math.exp(-(x-mu) ** 2 / 2 / sigma ** 2) / (sqrt_two_pi * sigma))
```

In Figure 6-2, we plot some of these pdfs to see what they look like:

```
xs = [x / 10.0 for x in range(-50, 50)] plt.plot(xs,[normal_pdf(x,sigma=1)
for x in xs],'-',label='mu=0,sigma=1') plt.plot(xs,[normal_pdf(x,sigma=2)
for x in xs],'--',label='mu=0,sigma=2') plt.plot(xs,[normal_pdf(x,sigma=0.5)
```

```
for x in xs],':',label='mu=0,sigma=0.5')plt.plot(xs,[normal_pdf(x,mu=-1)
for x in xs],'-.',label='mu=-1,sigma=1')
plt.legend()
plt.title("Various Normal pdfs")
plt.show()
```



When $\mu = 0$ and $\sigma = 1$, it's called the *standard normal distribution*. If Z is a standardnormal random variable, then it turns out that:

$$X = \sigma Z + \mu$$

is also normal but with mean μ and standard deviation σ. Conversely, if X is a normalrandom variable with mean μ and standard deviation σ,

$$Z = (X - \mu) / \sigma$$

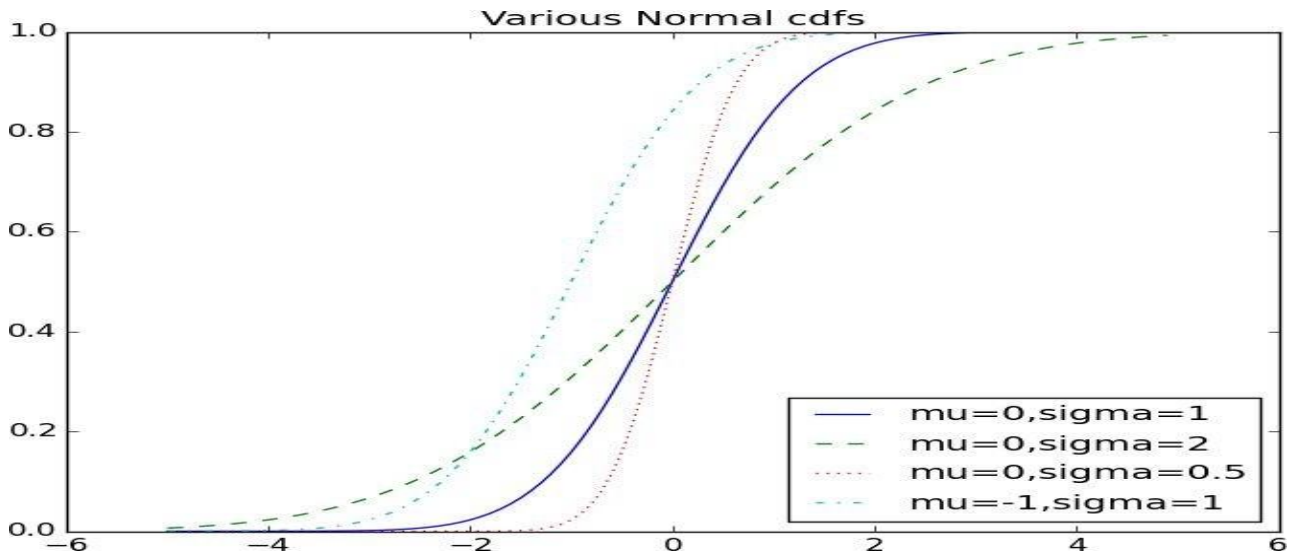
is a standard normal variable.

The cumulative distribution function for the normal distribution cannot be written in an"elementary" manner, but we can write it using Python's math.erf:

```
def normal_cdf(x, mu=0,sigma=1):
    return (1 + math.erf((x - mu) / math.sqrt(2) / sigma)) / 2
```

Again, in Figure 6-3, we plot a few:

```
xs = [x / 10.0 for x in range(-50, 50)]
plt.plot(xs,[normal_cdf(x,sigma=1) for x in xs],'',label='mu=0,sigma=1')
plt.plot(xs,[normal_cdf(x,sigma=2) for x in xs],'--',label='mu=0,sigma=2')
plt.plot(xs,[normal_cdf(x,sigma=0.5) for x in xs],':',label='mu=0,sigma=0.5')
plt.plot(xs,[normal_cdf(x,mu=-1) for x in xs],'-.',label='mu=-1,sigma=1')
plt.legend(loc=4) # bottom right
plt.title("Various Normal cdfs")
plt.show()
```



Sometimes we'll need to invert normal_cdf to find the value corresponding to a specified probability. There's no simple way to compute its inverse, but normal_cdfis continuous and strictly increasing, so we can use a binary search:

```
def inverse_normal_cdf(p, mu=0, sigma=1, tolerance=0.00001):
    if mu != 0 or sigma != 1:
        return mu + sigma * inverse_normal_cdf(p, tolerance=tolerance)
    low_z, low_p = -10.0, 0
    hi_z,  hi_p  =  10.0, 1
    while hi_z - low_z > tolerance:
        mid_z = (low_z + hi_z) / 2
        mid_p = normal_cdf(mid_z)
        if mid_p < p:
            low_z, low_p = mid_z, mid_p
        elif mid_p > p:
            hi_z, hi_p = mid_z, mid_p
    else:
    break return mid_z
```

The function repeatedly bisects intervals until it narrows in on a Z that's close enough to the desired probability

The Central Limit Theorem

One reason the normal distribution is so useful is the *central limit theorem*, which says (in essence) that a random variable defined as the average of a large number of independent and identically distributed random variables is itself approximately normally distributed.

In particular, if $x_1, \ldots, x_n$ are random variables with mean μ and standard σ deviation,and if n is large, then:

$$\frac{1}{n}(x_1 + \ldots + x_n)$$

is approximately normally distributed with mean μ and standard deviation $\sigma/\sqrt{n}$.Equivalently,

$$\frac{(x_1 + \ldots + x_n) - \mu n}{\sigma\sqrt{n}}$$

is approximately normally distributed with mean 0 and standard deviation 1.
An easy way to illustrate this is by looking at *binomial* random variables, which have two parameters n and p. A Binomial(n,p) random variable is simply the sum of n independent Bernoulli(p) random variables, each of which equals 1 with probability p and 0 with probability $1 - p$:

```
def bernoulli_trial(p):
    return 1 if random.random() < p else 0
def binomial(n, p):
    return sum(bernoulli_trial(p) for _ in range(n))
```

The mean of a Bernoulli(p) variable is p, and its standard deviation is $\sqrt{p(1-p)}$. The central limit theorem says that as n gets large, a Binomial(n,p) variable is approximately a normal random variable with mean $\mu = np$ and standard deviation $\sigma = \sqrt{np(1-p)}$ If we plot both, you can easily see the resemblance:

```
def make_hist(p, n, num_points):
    data = [binomial(n, p) for _ in range(num_points)]
    histogram = Counter(data)
    plt.bar([x - 0.4 for x in histogram.keys()],
            [v / num_points for v in
            histogram.values()],0.8,
            color='0.75')
    mu = p * n
    sigma = math.sqrt(n * p * (1 - p))
    xs = range(min(data), max(data) + 1)
    ys = [normal_cdf(i + 0.5, mu, sigma) - normal_cdf(i - 0.5, mu, sigma)
          for i in xs]plt.plot(xs,ys)
    plt.title("Binomial Distribution vs. Normal Approximation")plt.show()
```

For example, when you call make_hist(0.75, 100, 10000), you get the graph in Figure 6-4.

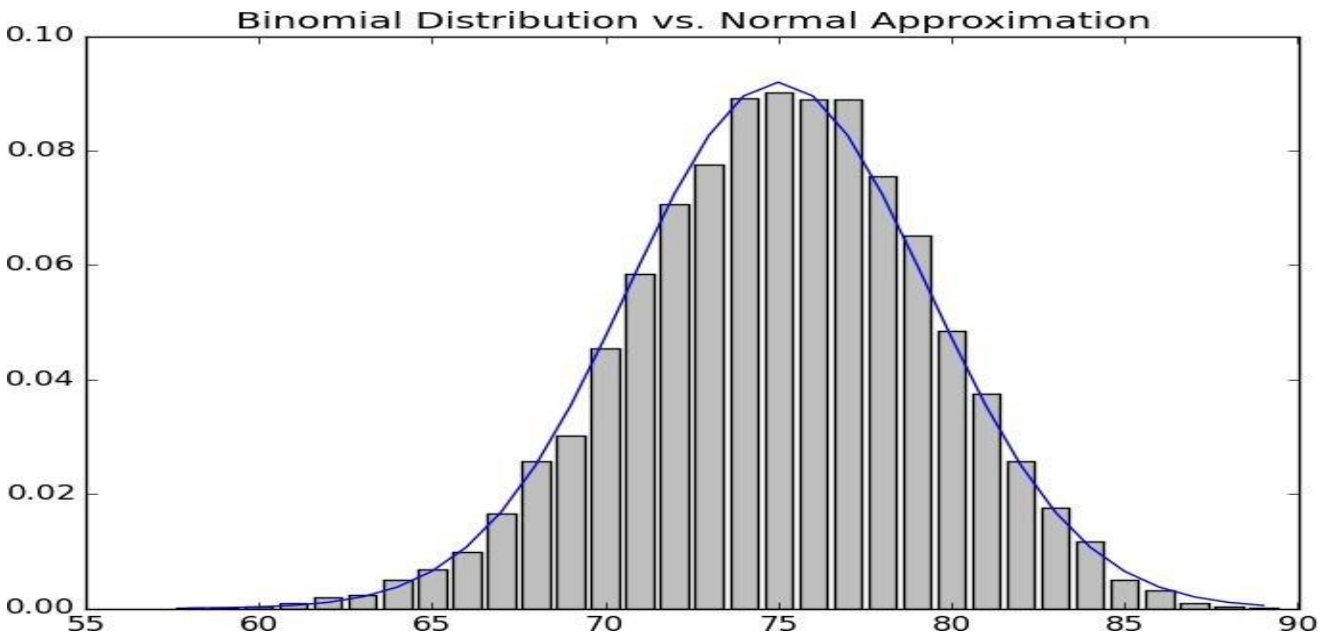


Figure 6-4. The output from `make_hist`

The moral of this approximation is that if you want to know the probability that (say) a fair coin turns up more than 60 heads in 100 flips, you can estimate it as the probability that a Normal(50,5) is greater than 60, which is easier than computing the Binomial(100,0.5) cdf.

UNIT – III

1Q) Write about Stdin, Stdout and Stderr?

Ans: Python's sys module provides us with all three file objects for stdin, stdout, and stderr. For the input file object, we use sys.stdin. This is similar to a file, where you can open and close it, just like any other file.

1. **Standard input**
   - This is the *file-handle* that a user program reads to get information from the user. We give input to the standard input (**stdin**).

     Eg:

```
import sys
stdin_fileno = sys.stdin
for line in stdin_fileno:
    if 'exit' == line.strip():
        print('Found exit. Terminating the program')
        exit(0)
    else:
        print('Message from sys.stdin: ---> {} <---'.format(line))
```

     Output

```
Hi
Message from sys.stdin: ---> Hi
 <---
Hello from AskPython
Message from sys.stdin: ---> Hello from AskPython
 <---
exit
Found exit. Terminating the program
```

2. **sys.stdout**
   - For the output file object, we use sys.stdout. It is similar to sys.stdin, but it directly displays anything written to it to the Console.
   - The below snippet shows that we get the Output to the Console if we write to sys.stdout.

     Eg:

```
import sys
stdout_fileno = sys.stdout
sample_input = ['Hi', 'Hello from AskPython', 'exit']
for ip in sample_input:
    stdout_fileno.write(ip + '\n')
```

     Output

```
Hi
Hello from AskPython
Exit
```

3. **Standard error**
   - The user program writes error information to this file-handle. Errors are returned via the Standard error (**stderr**).
   - sys.stderr is similar to sys.stdout because it also prints directly to the Console. But the difference is that it only prints Exceptions and Error messages. (Which is why it is called Standard Error).

Eg:

```
import sys
stdout_fileno = sys.stdout
stderr_fileno = sys.stderr
sample_input = ['Hi', 'Hello from AskPython', 'exit']
for ip in sample_input:
   stdout_fileno.write(ip + '\n')
   try:
      ip = ip + 100
   except:
      stderr_fileno.write('Exception Occurred!\n')
```

Output

```
Hi
Exception Occurred!
Hello from AskPython
Exception Occurred!
exit
Exception Occurred!
```

2Q) Reading a file

Ans:

- To read the contents of a file, we have to open a file in reading mode. Open a file using the built-in function called open(). In addition to the file name, we need to pass the file mode specifying the **purpose of opening the file**.
- The following are the different modes for reading the file. We will see each one by one.

File Mode	Definition
r	The default mode for opening a file to read the contents of a text file.
r+	Open a file for both reading and writing. The file pointer will be placed at the beginning of the file.
rb	Opens the file for reading a file in binary format. The file pointer will be placed at the beginning of the file.
w+	Opens a file for both writing as well as reading. The file pointer will be placed in the beginning of the file. For an existing file, the content will be overwritten.
a+	Open the file for both the reading and appending. The pointer will be placed at the end of the file and new content will be written after the existing content.

Steps for Reading a File in Python

1. **Find the path of a file**
   - We can read a file using both relative path and absolute path. The path is the location of the file on the disk.
   - An **absolute path** contains the complete directory list required to locate the file.
   - A **relative path** contains the current directory and then the file name.

2. **Open file in Read Mode**
   To open a file Pass file path and access mode to the open() function. The access mode specifies the operation you wanted to perform on the file, such as reading or writing.
   For example, fp= open(r'File_Path', 'r')
3. **Read content from a file.**
   Once opened, we can read all the text or content of the file using the read() method. You can also use the readline() to read file line by line or the readlines() to read all lines.
   For example, content = fp.read()
4. **Close file after completing the read operation**
   We need to make sure that the file will be closed properly after completing the file operation. Use fp.close() to close a file.

Access Modes for Reading a file

To read the contents of a file, we have to open a file in reading mode. Open a file using the built-in function called open(). In addition to the file name, we need to pass the file mode specifying the **purpose of opening the file**.

The following are the different modes for reading the file. We will see each one by one.

File Mode	Definition
r	The default mode for opening a file to read the contents of a text file.
r+	Open a file for both reading and writing. The file pointer will be placed at the beginning of the file.
rb	Opens the file for reading a file in binary format. The file pointer will be placed at the beginning of the file.
w+	Opens a file for both writing as well as reading. The file pointer will be placed in the beginning of the file. For an existing file, the content will be overwritten.
a+	Open the file for both the reading and appending. The pointer will be placed at the end of the file and new content will be written after the existing content.

File Read Methods

Python provides three different methods to read the file. We don't have to import any module for that.. Below are the three methods

Method	When to Use?
read()	Returns the entire file content and it accepts the optional size parameter that mentions the bytes to read from the file.
readline()	The readline() method reads a single line from a file at a time. . Accepts optional size parameter that mentions how many bytes to return from the file.
readlines()	The readlines() method returns a list of lines from the file.

3Q) Delimited Files

Ans

- The hypothetical email addresses file we just processed had one address per line. More frequently you'll work with files with lots of data on each line. These files are very often either comma-separated or tab-separated.
- Each line has several fields, with a comma (or a tab) indicating where one field ends and the next field starts.
- This starts to get complicated when you have fields with commas and tabs and newlines in them. For this reason, it's pretty much always a mistake to try to parse them yourself. Instead, you should use Python's csv module
- For technical reasons that you should feel free to blame on Microsoft, you should always work with csv files in binary mode by including a b after the r or w (see Stack Overflow).
- If your file has no headers (which means you probably want each row as a list, and which places the burden on you to know what's in each column), you can use csv.reader to iterate over the rows, each of which will be an appropriately split list.
- **For example,** if we had a tab-delimited file of stock prices:

  6/20/2014 AAPL 90.91
  6/20/2014 MSFT 41.68
  6/20/2014 FB 64.5
  6/19/2014 AAPL 91.86
  6/19/2014 MSFT 41.51
  6/19/2014 FB 64.34

  we could process them with:

```
import csv
with open('tab_delimited_stock_prices.txt', 'rb') as f:
reader = csv.reader(f, delimiter='\t')
for row in reader:
date = row[0]
symbol = row[1]
closing_price = float(row[2])
process(date, symbol, closing_price)
```

  If your file has headers:

```
date:symbol:closing_price
6/20/2014:AAPL:90.91
6/20/2014:MSFT:41.68
6/20/2014:FB:64.5
```

- you can either skip the header row or get each row as a dict (with the headers as keys) by using csv.DictReader:

  Eg:

```
with open('colon_delimited_stock_prices.txt', 'rb') as f:
reader = csv.DictReader(f, delimiter=':')
for row in reader:
date = row["date"]
symbol = row["symbol"]
closing_price = float(row["closing_price"])
process(date, symbol, closing_price)
```

4Q) Scraping the Web

Ans:

Another way to get data is by scraping it from web pages. Fetching web pages, it turns out, is pretty easy; getting meaningful structured information out of them less so.

HTML and the Parsing Thereof

Pages on the Web are written in HTML, in which text is (ideally) marked up into elements and their attributes:

Eg:

```
<html>
<head>
<title>A web page</title>
</head>
<body>
<p id="author">Joel Grus</p>
<p id="subject">Data Science</p>
</body></html>
```

In a perfect world, where all web pages are marked up semantically for our benefit, we would be able to extract data using rules like "find the <p> element whose id is subject and return the text it contains."

In the actual world, HTML is not generally well-formed, let alone annotated. This means we'll need help making sense of it.

To get data out of HTML, we will use the Beautiful Soup library, which builds a tree out of the various elements on a web page and provides a simple interface for accessing them. As I write this, the latest version is Beautiful Soup 4.3.2 (pip install beautifulsoup4), which is what we'll be using. We'll also be using the requests library (pip install requests), which is a much nicer way of making HTTP requests than anything that's built into Python.

Python's built-in HTML parser is not that lenient, which means that it doesn't always cope well with HTML that's not perfectly formed. For that reason, we'll use a different parser, which we need to install:

```
pip install html5lib
```

To use Beautiful Soup, we'll need to pass some HTML into the BeautifulSoup() function. In our examples, this will be the result of a call to requests.get:

```
from bs4 import BeautifulSoup
import requests
html = requests.get("http://www.example.com").text
soup = BeautifulSoup(html, 'html5lib')
```

Example: O'Reilly Books About Data

- A potential investor in DataSciencester thinks data is just a fad. To prove him wrong, you decide to examine how many data books O'Reilly has published over time. After digging through its website, you find that it has many pages of data books (and videos), reachable through 30-items-at-a-time directory pages with URLs like:

http://shop.oreilly.com/category/browse-subjects/data.do?sortby=publicationDate&page=1

- Unless you want to be a jerk, whenever you want to scrape data from a website you should first check to see if it has some sort of access policy.

  Looking at: http://oreilly.com/terms/

  there seems to be nothing prohibiting this project. In order to be good citizens, we should also check for a robots.txt file that tells webcrawlers how to behave. The important lines in http://shop.oreilly.com/robots.txt are:

  Crawl-delay: 30
  Request-rate: 1/30

  The first tells us that we should wait 30 seconds between requests, the second that we should request only one page every 30 seconds. So basically they're two different ways of saying the same thing.

  To figure out how to extract the data, let's download one of those pages and feed it to Beautiful Soup:

```
url = "http://shop.oreilly.com/category/browse-subjects/" + \
      "data.do?sortby=publicationDate&page=1"
soup = BeautifulSoup(requests.get(url).text, 'html5lib')
```

  If you view the source of the page (in your browser, right-click and select "View source" or "View page source" or whatever option looks the most like that), you'll see that each book (or video) seems to be uniquely contained in a <td> table cell element whose class is thumbtext. Here is (an abridged version of) the relevant **HTML for one book:**

```
<td class="thumbtext">
<div class="thumbcontainer">
<div class="thumbdiv">
<a href="/product/9781118903407.do">
<img src="..."/></a>
</div></div>
<div class="widthchange">
<div class="thumbheader">
<a href="/product/9781118903407.do">Getting a Big Data Job For
Dummies</a></div>
<div class="AuthorName">By Jason Williamson</div>
<span class="directorydate"> December 2014 </span>
<div style="clear:both;"><div id="146350">
<span class="pricelabel">
Ebook:<span class="price"> $29.99</span>
</span></div></div></div></td>
```

Using APIs

Many websites and web services provide application programming interfaces (APIs), which allow you to explicitly request data in a structured format. This saves you the trouble of having to scrape them!

5Q) JSON (and XML)

Ans:

- HTTP is a protocol for transferring text, the data you request through a web API needs to be serialized into a string format. Often this serialization uses JavaScript Object Notation (JSON). JavaScript objects look quite similar to Python dicts, which makes their string representations easy to interpret:

```
{ "title" : "Data Science Book","author" : "Joel Grus",
"publicationYear" : 2014,"topics" : [ "data", "science", "data science"] }
```

- We can parse JSON using Python's json module. In particular, we will use its loads function, which deserializes a string representing a JSON object into a Python object:

```
import json
serialized = """{ "title" : "Data Science Book",
"author" : "Joel Grus",
"publicationYear" : 2014,
"topics" : [ "data", "science", "data science"] }"""
# parse the JSON to create a Python dict
deserialized = json.loads(serialized)
if "data science" in deserialized["topics"]:
print deserialized
```

- Sometimes an API provider hates you and only provides responses in XML:

```
<Book>
<Title>Data Science Book</Title>
<Author>Joel Grus</Author>
<PublicationYear>2014</PublicationYear>
<Topics>
<Topic>data</Topic>
<Topic>science</Topic>
<Topic>data science</Topic>
</Topics>
</Book>
```

Using an Unauthenticated API

Most APIs these days require you to first authenticate yourself in order to use them. While we don't be grudge them this policy, it creates a lot of extra boilerplate that muddies up our exposition. Accordingly, we'll first take a look at GitHub's API, with which you can do some simple things unauthenticated:

```
import requests, json
endpoint = "https://api.github.com/users/joelgrus/repos"
repos = json.loads(requests.get(endpoint).text)
```

At this point repos is a list of Python dicts, each representing a public repository in my GitHub account.

We can use this to figure out which months and days of the week I'm most likely to create a repository. The only issue is that the dates in the response are (Unicode) strings: u'created_at': u'2013-07-05T02:02:28Z'

Python doesn't come with a great date parser, so we'll need to install one:

```
pip install python-dateutil
```

from which you'll probably only ever need the dateutil.parser.parse function:

```
from dateutil.parser import parse
dates = [parse(repo["created_at"]) for repo in repos]
month_counts = Counter(date.month for date in dates)
weekday_counts = Counter(date.weekday() for date in dates)
```

Similarly, you can get the languages of my last five repositories:

```
last_5_repositories = sorted(repos, key=lambda r: r["created_at"], reverse=True)[:5]
last_5_languages = [repo["language"]
for repo in last_5_repositories]
```

Finding APIs

- If you need data from a specific site, look for a developers or API section of the site for details, and try searching the Web for "python__api" to find a library. There is a Rotten Tomatoes API for Python. There are multiple Python wrappers for the Klout API, for the Yelp API, for the IMDB API, and so on.
- If you're looking for lists of APIs that have Python wrappers, two directories are at Python API and Python for Beginners.
- If you want a directory of web APIs more broadly, a good resource is Programmable Web, which has a huge directory of categorized APIs. And if after all that you can' find what you need, there's always scraping, the last refuge of the data scientist.

6Q) Exploring One-Dimensional Data

Ans:

The simplest case is when you have a one-dimensional data set, which is just a collection of numbers. For example, these could be the daily average number of minutes each user spends on your site, the number of times each of a collection of data science tutorial videos was watched, or the number of pages of each of the data science books in your data science library.

An obvious first step is to compute a few summary statistics. You'd like to know how many data points you have, the smallest, the largest, the mean, and the standard deviation.

But even these don't necessarily give you a great understanding. A good next step is to create a histogram, in which you group your data into discrete buckets and count how many points fall into each bucket:

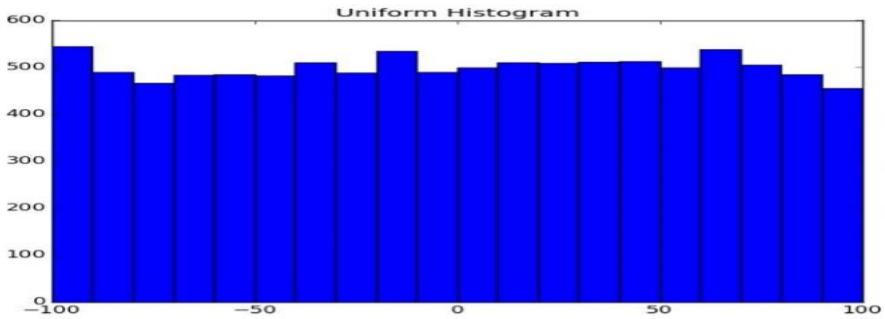
```
Def bucketize(point,bucket_size):
    return bucket_size * math.floor(point/bucket_size)
def make_histogram(points, bucket_size):
    return Counter(bucketize(point, bucket_size) for point in points)
def plot_histogram(points, bucket_size, title=""):
    histogram=make_histogram(points,bucket_size)
    plt.bar(histogram.keys(),histogram.values(),width=bucket_size)
    plt.title(title)
    plt.show()
```

For example, consider the two following sets of data:

```
random.seed(0)
uniform=[200*random.random() - 100 for _ in range(10000)]
normal=[57 * inverse_normal_cdf(random.random())
        for _ in range(10000)]
```

Both have means close to 0 and standard deviations close to 58. However, they have very different distributions. This shows the distribution of uniform:

```
plot_histogram(uniform, 10, "Uniform Histogram")
```



while Figure 10-2 shows the distribution of normal:

```
plot_histogram(normal, 10, "Normal Histogram")
```

In this case, both distributions had pretty different max and min, but even knowing that wouldn't have been sufficient to understand how they differed.

Two Dimensions

Now imagine you have a data set with two dimensions. Maybe in addition to daily minutes you have years of data science experience. Of course you'd want to understand each dimension individually. But you probably also want to scatter the data. For example, consider another fake data set:

```
def random_normal():
    return inverse_normal_cdf(random.random())
xs=[random_normal()
for _ in range(1000)]
ys1 = [x + random_normal() / 2 for x in xs]
ys2 = [-x + random_normal() / 2 for x in xs]
```

If you were to run plot_histogram on ys1 and ys2 you'd get very similar looking plots

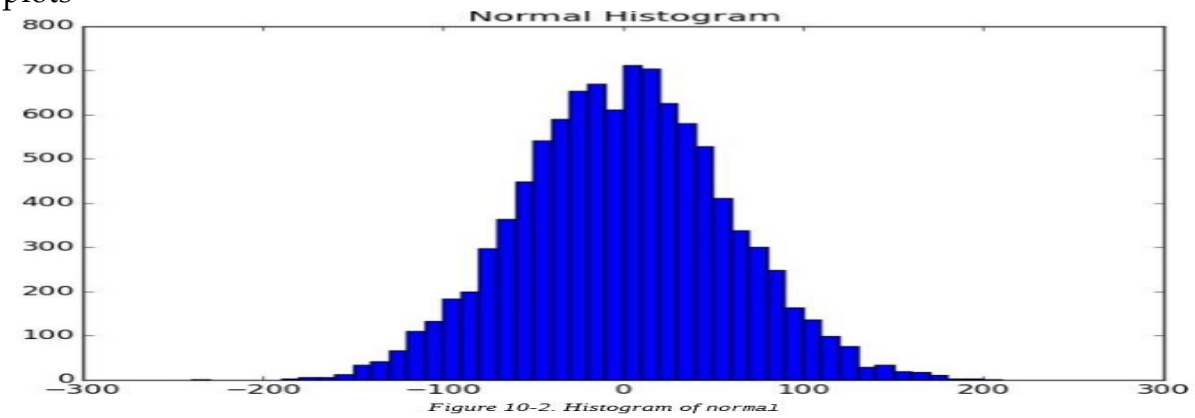


Figure 10-2. Histogram of normal

But each has a very different joint distribution with xs, as shown in Figure 10-3:

```
plt.scatter(xs, ys1, marker='.', color='black', label='ys1')
plt.scatter(xs, ys2, marker='.', color='gray', label='ys2')
plt.xlabel('xs')
plt.ylabel('ys')
plt.legend(loc=9)
plt.title("Very Different Joint Distributions")
plt.show()
```

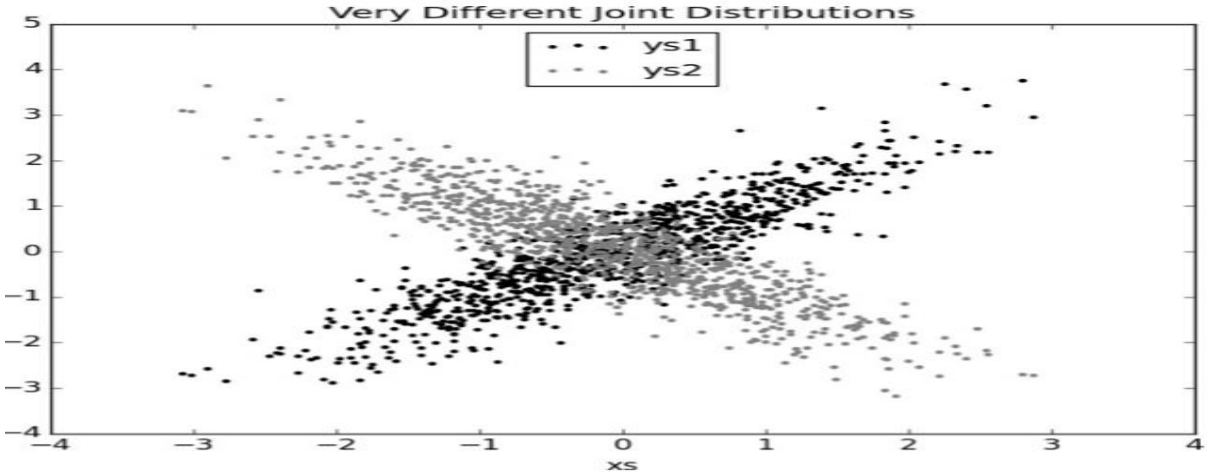


Figure 10-3. Scattering two different ys

This difference would also be apparent if you looked at the correlations:

```
print correlation(xs,ys1)
print correlation(xs,ys2)
```

Many Dimensions

With many dimensions, you'd like to know how all the dimensions relate to one another. A simple approach is to look at the correlation matrix, in which the entry in row i and column j is the correlation between the ith dimension and the jth dimension of the data:

```
def correlation_matrix(data):
    _,num_columns = shape(data)
def matrix_entry(i, j):
    return correlation(get_column(data, i), get_column(data, j))
    return make_matrix(num_columns, num_columns, matrix_entry)
```

A more visual approach is to make a scatterplot matrix (Figure 10-4) showing all the pairwise scatterplots. To do that we'll use plt.subplots(), which allows us to create subplots of our chart. We give it the number of rows and the number of columns, and it returns a figure object and a two-dimensional array of axes objects

```
import matplotlib.pyplot as plt
    _,num_columns=shape(data)
    fig, ax = plt.subplots(num_columns, num_columns)
    for i in range(num_columns):
        for j in range(num_columns):
            if i != j:
```

```
 ax[i][j].scatter(get_column(data,j),get_column(data,i))
else:
ax[i][j].annotate("series"+str(i),(0.5,0.5),xycoords='axesfraction',
ha="center",  va="center")
if i < num_columns  - 1: ax[i][j].xaxis.set_visible(False)
if j > 0: ax[i][j].yaxis.set_visible(False)
ax[-1][-1].set_xlim(ax[0][-1].get_xlim())
ax[0][0].set_ylim(ax[0][1].get_ylim())
plt.show()
```

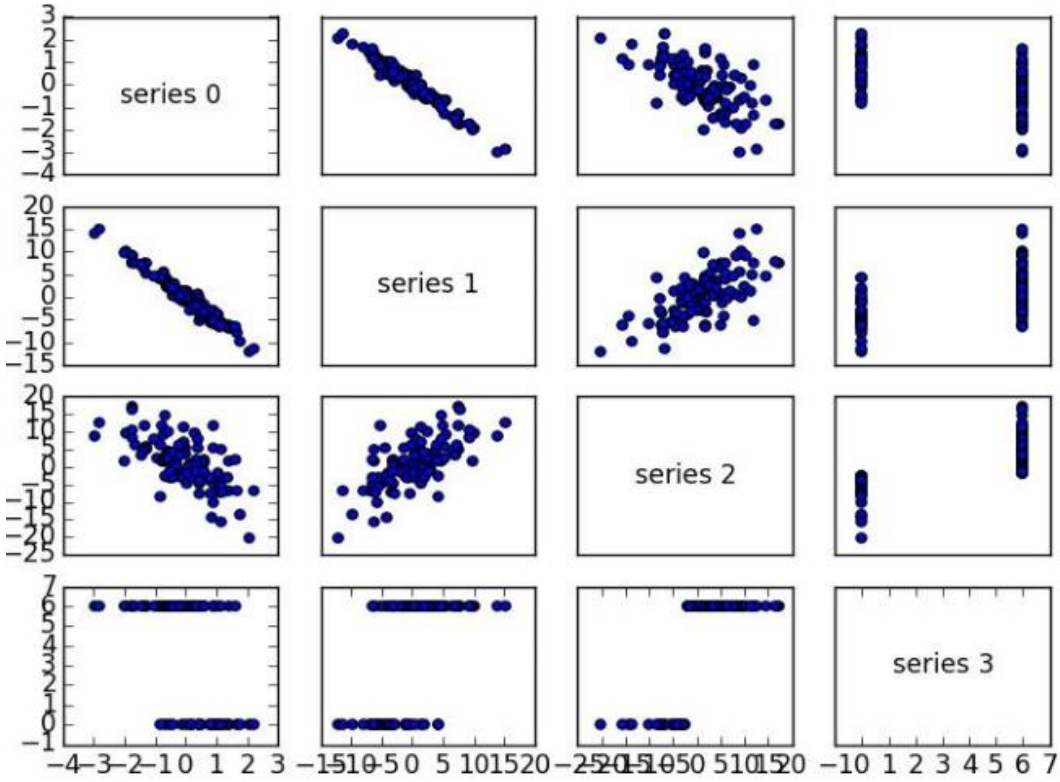


Figure 10-4. Scatterplot matrix

Looking at the scatterplots,you can see that series 1 is very negatively correlated with series 0, series 2 is positively correlated with series 1, and series 3 only takes on the values 0 and 6, with 0 corresponding to small values of series 2 and 6 corresponding to large values. This is a quick way to get a rough sense of which of your variables are correlated

7Q) Cleaning and Munging:

Ans:

Real-world data is dirty. Often you'llhave to do some work on it before you can use it. We have to convert strings to floats or ints before we can use them. Previously, we did that right before using the data:

```
closing_price = float(row[2])
```

But it's probably less error-prone to do the parsing on the way in, which we can do by creating a function that wraps csv.reader. We'll give it a list of parsers, each specifying how to parse one of the columns. And we'll use None to represent "don't do anything to this column":

```
def parse_row(input_row, parsers):
    return [parser(value) if parser is not None else value
            for value, parser in zip(input_row, parsers)]
def parse_rows_with(reader, parsers):
    for row in reader:
        yield parse_row(row, parsers)
```

What if there's bad data?

A "float" value that doesn't actually represent a number? We'd usually rather get a None than crash our program. We can do this with a helper function:

```
def try_or_none(f):
    def f_or_none(x):
    try:
        return f(x)
    except:
        return None
    return f_or_none
```

after which we can rewrite parse_row to use it:

```
def parse_row(input_row, parsers):
    return [try_or_none(parser)(value) if parser is not None else value
        for value, parser in zip(input_row, parsers)]
```

8Q) What is Machine Learning

Ans: Machine Learning is said as a subset of **artificial intelligence** that is mainly concerned with the development of algorithms which allow a computer to learn from the data and past experiences on their own.

The term machine learning was first introduced by **Arthur Samuel** in **1959**. We can define it in a summarized way as:

> **"Machine learning enables a machine to automatically learn from data, improve performance from experiences, and predict things without being explicitly programmed"**

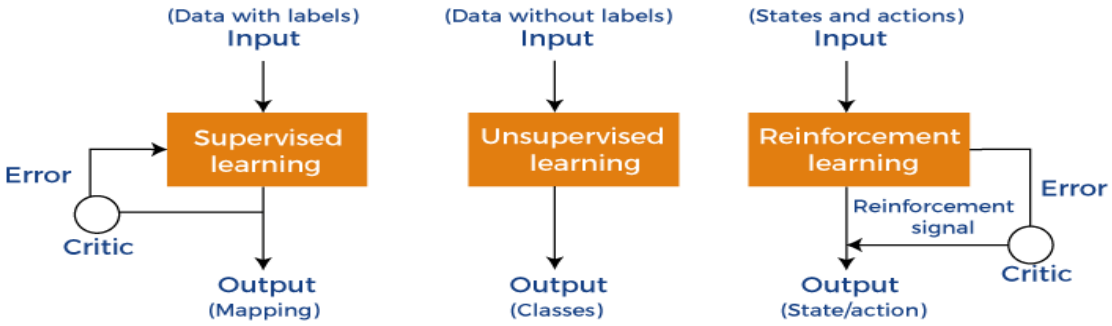
Machine Learning Models

- ***A machine learning model is defined as a mathematical representation of the output of the training process.***
- Machine learning is the study of different algorithms that can improve automatically through experience & old data and build the model.
- The learning algorithm discovers patterns within the training data, and it outputs an ML model which captures these patterns and makes predictions on new data.
- There are various types of machine learning models available based on different business goals and data sets.

Classification of Machine Learning Models:

Based on different business goals and data sets, there are three learning models for algorithms. Each machine learning algorithm settles into one of the three models:

1. Supervised Learning
2. Unsupervised Learning
3. Reinforcement Learning



1. Supervised Machine Learning Models

- Supervised Learning is the simplest machine learning model to understand in which input data is called training data and has a known label or result as an output. So, it works on the principle of input-output pairs.
- It requires creating a function that can be trained using a training data set, and then it is applied to unknown data and makes some predictive performance.
- Supervised learning is task-based and tested on labeled data sets.
- We can implement a supervised learning model on simple real-life problems. For example, we have a dataset consisting of age and height; then, we can build a supervised learning model to predict the person's height based on their age.
- Supervised Learning models are further classified into two categories:

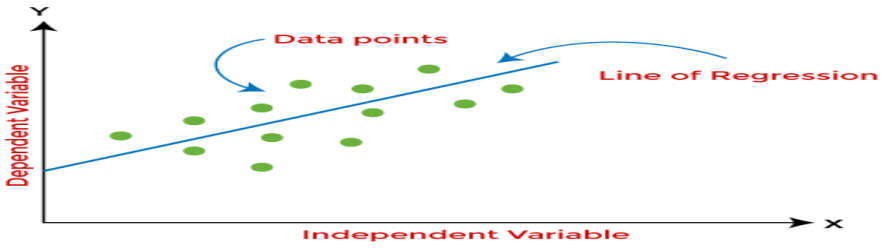
1) Regression

In regression problems, the output is a continuous variable. Some commonly used Regression models are as follows:

a) Linear Regression

Linear regression is the simplest machine learning model in which we try to predict one output variable using one or more input variables. The representation of linear regression is a linear equation, which combines a set of input values(x) and predicted output(y) for the set of those input values. It is represented in the form of a line:

Y = bx+ c.



The main aim of the linear regression model is to find the best fit line that best fits the data points. Linear regression is extended to multiple linear regression.

b) Decision Tree

- ⇨ Decision trees are the popular machine learning models that can be used for both regression and classification problems.
- ⇨ A decision tree uses a tree-like structure of decisions along with their possible consequences and outcomes. In this, each internal node is used to represent a test on an attribute; each branch is used

to represent the outcome of the test. The more nodes a decision tree has, the more accurate the result will be.

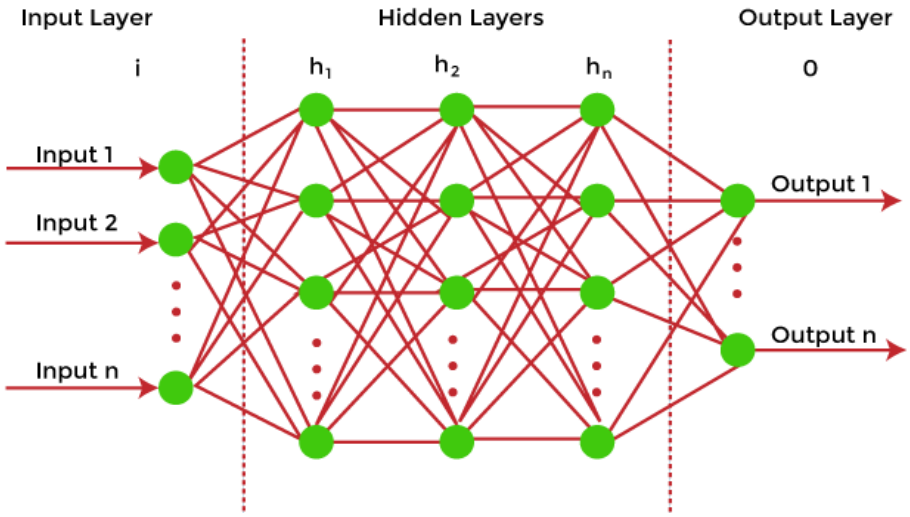
- ⇨ The advantage of decision trees is that they are intuitive and easy to implement, but they lack accuracy.
- ⇨ Decision trees are widely used in **operations research, specifically in decision analysis, strategic planning**, and mainly in machine learning.

c) Random Forest

- ⇨ Random Forest is the ensemble learning method, which consists of a large number of decision trees. Each decision tree in a random forest predicts an outcome, and the prediction with the majority of votes is considered as the outcome.
- ⇨ A random forest model can be used for both regression and classification problems.
- ⇨ For the classification task, the outcome of the random forest is taken from the majority of votes. Whereas in the regression task, the outcome is taken from the mean or average of the predictions generated by each tree.

d) Neural Networks

- Neural networks are the subset of machine learning and are also known as artificial neural networks.
- Neural networks are made up of artificial neurons and designed in a way that resembles the human brain structure and working.
- Each artificial neuron connects with many other neurons in a neural network, and such millions of connected neurons create a sophisticated cognitive structure.



- Neural networks consist of a multilayer structure, containing one input layer, one or more hidden layers, and one output layer. As each neuron is connected with another neuron, it transfers data from one layer to the other neuron of the next layers. Finally, data reaches the last layer or output layer of the neural network and generates output.

- Neural networks depend on training data to learn and improve their accuracy. However, a perfectly trained & accurate neural network can cluster data quickly and become a powerful machine learning and AI tool. One of the best-known neural networks is **Google's search algorithm.**

2) Unsupervised Machine learning models

Unsupervised Machine learning models implement the learning process opposite to supervised learning, which means it enables the model to learn from the unlabeled training dataset. Based on the unlabeled dataset, the model predicts the output. Using unsupervised learning, the model learns hidden patterns from the dataset by itself without any supervision.

Unsupervised learning models are mainly used to perform three tasks, which are as follows:

- **Clustering**
  - ⇨ Clustering is an unsupervised learning technique that involves clustering or groping the data points into different clusters based on similarities and differences. The objects with the most similarities remain in the same group, and they have no or very few similarities from other groups.
  - ⇨ Clustering algorithms can be widely used in different tasks such as **Image segmentation, Statistical data analysis, Market segmentation**, etc
  - ⇨ .Some commonly used Clustering algorithms are *K-means Clustering, hierarchal Clustering, DBSCAN*, etc.

- **Association Rule Learning**
  - ⇨ Association rule learning is an unsupervised learning technique, which finds interesting relations among variables within a large dataset.
  - ⇨ The main aim of this learning algorithm is to find the dependency of one data item on another data item and map those variables accordingly so that it can generate maximum profit.
  - ⇨ This algorithm is mainly applied in **Market Basket analysis, Web usage mining, continuous production**, etc.
  - ⇨ Some popular algorithms of Association rule learning are ***Apriori Algorithm, Eclat, FP-growth algorithm.***
- **Dimensionality Reduction**
  - ⇨ The number of features/variables present in a dataset is known as the dimensionality of the dataset, and the technique used to reduce the dimensionality is known as the dimensionality reduction technique.

- ⇨ Although more data provides more accurate results, it can also affect the performance of the model/algorithm, such as overfitting issues. In such cases, dimensionality reduction techniques are used. "***It is a process of converting the higher dimensions dataset into lesser dimensions dataset ensuring that it provides similar information***."
- ⇨ Different dimensionality reduction methods such ***as PCA(Principal Component Analysis), Singular Value Decomposition, etc.***

3) Reinforcement Learning

- ⇨ In reinforcement learning, the algorithm learns actions for a given set of states that lead to a goal state.
- ⇨ It is a feedback-based learning model that takes feedback signals after each state or action by interacting with the environment. This feedback works as a reward and the agent's goal is to maximize the positive rewards to improve their performance.
- ⇨ The behavior of the model in reinforcement learning is similar to human learning, as humans learn things by experiences as feedback and interact with the environment.
- ⇨ Below are some popular algorithms that come under reinforcement learning:
  - **Q-learning:**
    - Q-learning is one of the popular model-free algorithms of reinforcement learning, which is based on the Bellman equation.
    - It aims to learn the policy that can help the AI agent to take the best action for maximizing the reward under a specific circumstance.
    - It incorporates Q values for each state-action pair that indicate the reward to following a given state path, and it tries to maximize the Q-value.
  - **State-Action-Reward-State-Action (SARSA):**
    - SARSA is an On-policy algorithm based on the Markov decision process.
    - It uses the action performed by the current policy to learn the Q-value.
    - The SARSA algorithm stands **for State Action Reward State Action, which symbolizes the tuple (s, a, r, s', a').**
  - **Deep Q Network:**
    - DQN **or Deep Q Neural network is Q-learning** within the neural network.
    - It is basically employed in a big state space environment where defining a Q-table would be a complex task. So, in such a case, rather than using Q-table, the neural network uses Q-values for each action based on the state.

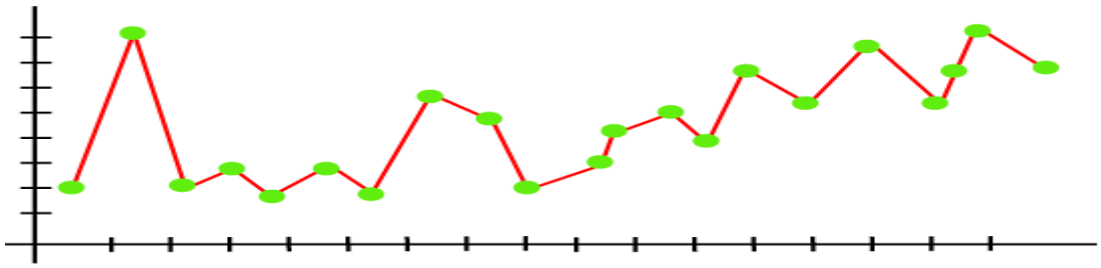
9Q) Overfitting and Underfitting in Machine Learning

Ans:

- Overfitting and Underfitting are the two main problems that occur in machine learning and degrade the performance of the machine learning models.
- The main goal of each machine learning model is **to generalize well**. Here **generalization** defines the ability of an ML model to provide a suitable output by adapting the given set of unknown input.
- It means after providing training on the dataset, it can produce reliable and accurate output. Hence, the underfitting and overfitting are the two terms that need to be checked for the performance of the model and whether the model is generalizing well or not.
- Before understanding the overfitting and underfitting, let's understand some basic terms:
  - **Signal:** It refers to the true underlying pattern of the data that helps the machine learning model to learn from the data.
  - **Noise:** Noise is unnecessary and irrelevant data that reduces the performance of the model.
  - **Bias:** Bias is a prediction error that is introduced in the model due to oversimplifying the machine learning algorithms. Or it is the difference between the predicted values and the actual values.
  - **Variance:** If the machine learning model performs well with the training dataset, but does not perform well with the test dataset, then variance occurs.

Overfitting

- Overfitting occurs when our machine learning model tries to cover all the data points or more than the required data points present in the given dataset. Because of this, the model starts caching noise and inaccurate values present in the dataset, and all these factors reduce the efficiency and accuracy of the model. The overfitted model has **low bias** and **high variance.**
- The chances of occurrence of overfitting increase as much we provide training to our model. It means the more we train our model, the more chances of occurring the overfitted model.
- Overfitting is the main problem that occurs in supervised learning.
  Example: The concept of the overfitting can be understood by the below graph of the linear regression output:



As we can see from the above graph, the model tries to cover all the data points present in the scatter plot. It may look efficient, but in reality, it is not so. Because the goal of the regression model to find the best fit line, but here we have not got any best fit, so, it will generate the prediction errors.

How to avoid the Overfitting in Model

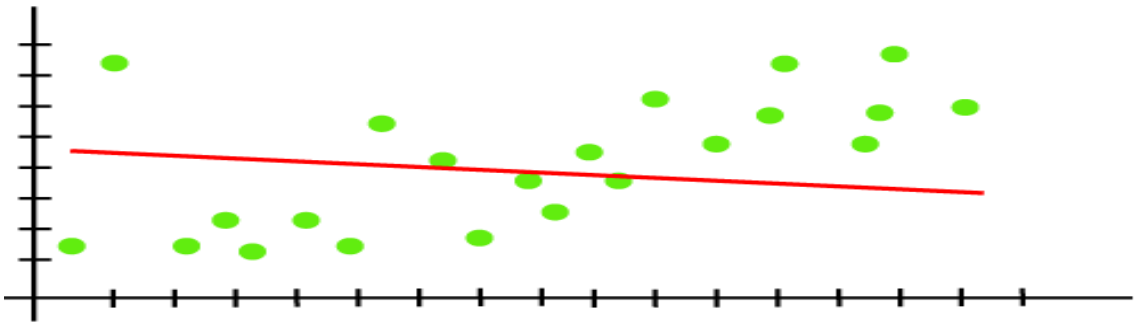
Both overfitting and underfitting cause the degraded performance of the machine learning model. But the main cause is overfitting, so there are some ways by which we can reduce the occurrence of overfitting in our model.

- **Cross-Validation**
- **Training with more data**
- **Removing features**
- **Early stopping the training**
- **Regularization**
- **Ensembling**

Underfitting

- Underfitting occurs when our machine learning model is not able to capture the underlying trend of the data. To avoid the overfitting in the model, the fed of training data can be stopped at an early stage, due to which the model may not learn enough from the training data. As a result, it may fail to find the best fit of the dominant trend in the data.
- In the case of underfitting, the model is not able to learn enough from the training data, and hence it reduces the accuracy and produces unreliable predictions.
- An underfitted model has high bias and low variance.

Example: We can understand the underfitting using below output of the linear regression model:



As we can see from the above diagram, the model is unable to capture the data points present in the plot.

How to avoid underfitting:

- By increasing the training time of the model.
- By increasing the number of features.

Goodness of Fit

The "Goodness of fit" term is taken from the statistics, and the goal of the machine learning models to achieve the goodness of fit. In statistics modeling, *it defines how closely the result or predicted values match the true values of the dataset.*

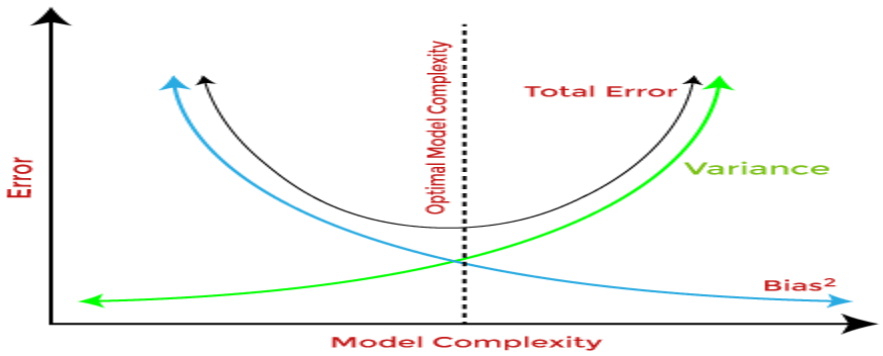
The model with a good fit is between the underfitted and overfitted model, and ideally, it makes predictions with 0 errors, but in practice, it is difficult to achieve it.

As when we train our model for a time, the errors in the training data go down, and the same happens with test data. But if we train the model for a long duration, then the performance of the model may decrease due to the overfitting, as the model also learn the noise present in the dataset. The errors in the test dataset start increasing, *so the point, just before the raising of errors, is the good point, and we can stop here for achieving a good model.*

10Q) Bias-Variance Trade-Off

Ans:

While building the machine learning model, it is really important to take care of bias and variance in order to avoid overfitting and underfitting in the model. If the model is very simple with fewer parameters, it may have low variance and high bias. Whereas, if the model has a large number of parameters, it will have high variance and low bias. So, it is required to make a balance between bias and variance errors, and this balance between the bias error and variance error is known as **the Bias-Variance trade-off.**



For an accurate prediction of the model, algorithms need a low variance and low bias. But this is not possible because bias and variance are related to each other:

- If we decrease the variance, it will increase the bias.
- If we decrease the bias, it will increase the variance.

Bias-Variance trade-off is a central issue in supervised learning. Ideally, we need a model that accurately captures the regularities in training data and simultaneously generalizes well with the unseen dataset. Unfortunately, doing this is not possible simultaneously. Because a high variance algorithm may perform well with training data, but it may lead to overfitting to noisy data. Whereas, high bias algorithm generates a much simple model that may not even capture important regularities in the data. So, we need to find a sweet spot between bias and variance to make an optimal model.

Hence, the ***Bias-Variance trade-off is about finding the sweet spot to make a balance between bias and variance errors.***

UNIT - IV

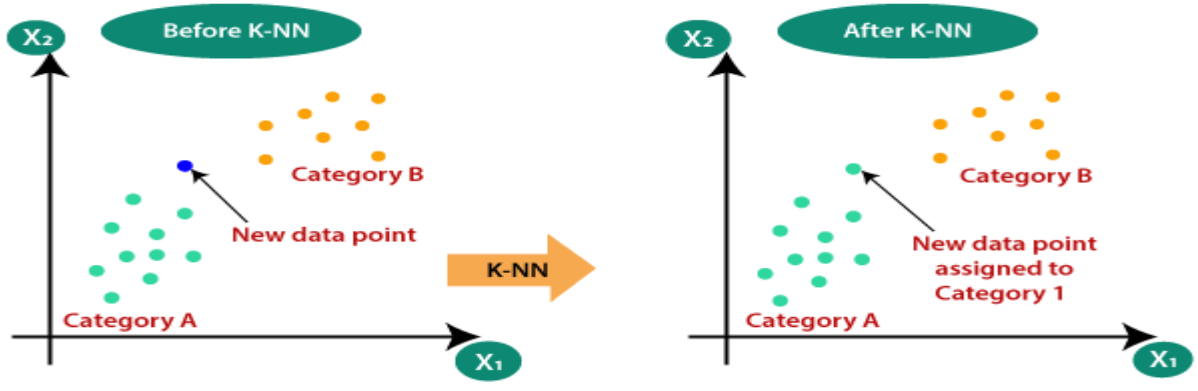
1Q) K-Nearest Neighbors

Ans:

- K-Nearest Neighbour is one of the simplest Machine Learning algorithms based on Supervised Learning technique.
- K-NN algorithm assumes the similarity between the new case/data and available cases and put the new case into the category that is most similar to the available categories.
- K-NN algorithm stores all the available data and classifies a new data point based on the similarity. This means when new data appears then it can be easily classified into a well suite category by using K- NN algorithm.
- K-NN algorithm can be used for Regression as well as for Classification but mostly it is used for the Classification problems.
- K-NN is a **non-parametric algorithm**, which means it does not make any assumption on underlying data.
- It is also called a **lazy learner algorithm** because it does not learn from the training set immediately instead it stores the dataset and at the time of classification, it performs an action on the dataset.
- KNN algorithm at the training phase just stores the dataset and when it gets new data, then it classifies that data into a category that is much similar to the new data.

K-NN Model

Suppose there are two categories, i.e., Category A and Category B, and we have a new data point x1, so this data point will lie in which of these categories. To solve this type of problem, we need a K-NN algorithm. With the help of K-NN, we can easily identify the category or class of a particular dataset. Consider the below diagram:

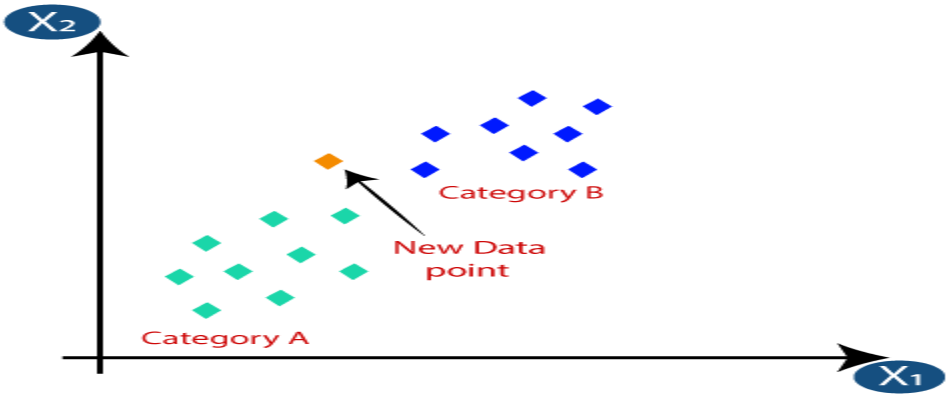


How does K-NN work?

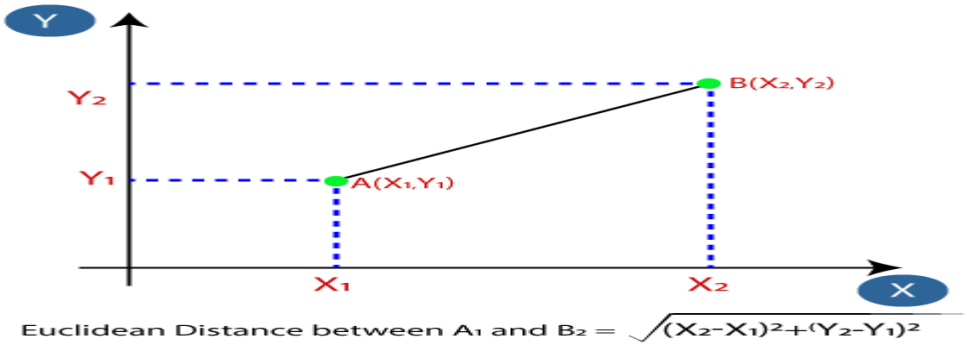
- **Step-1:** Select the number K of the neighbors
- **Step-2:** Calculate the Euclidean distance of **K number of neighbors**
- **Step-3:** Take the K nearest neighbors as per the calculated Euclidean distance.
- **Step-4:** Among these k neighbors, count the number of the data points in each category.

- **Step-5:** Assign the new data points to that category for which the number of the neighbor is maximum.
- **Step-6:** Our model is ready.

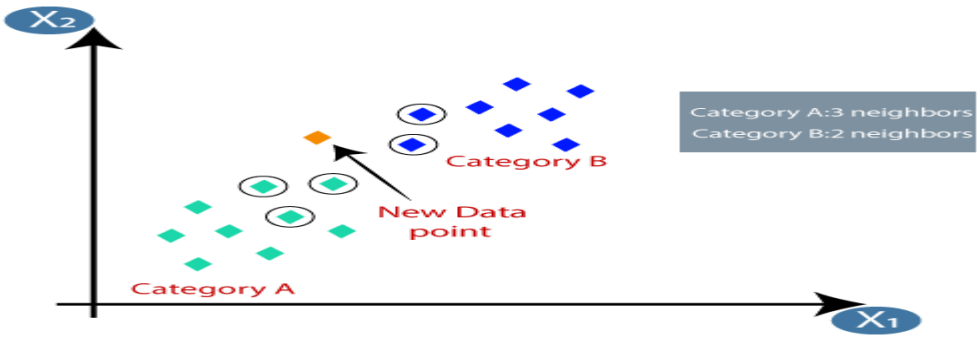
Suppose we have a new data point and we need to put it in the required category. Consider the below image:



- Firstly, we will choose the number of neighbors, so we will choose the k=5.
- Next, we will calculate the **Euclidean distance** between the data points. The Euclidean distance is the distance between two points, which we have already studied in geometry. It can be calculated as:



- By calculating the Euclidean distance we got the nearest neighbors, as three nearest neighbors in category A and two nearest neighbors in category B. Consider the below image:



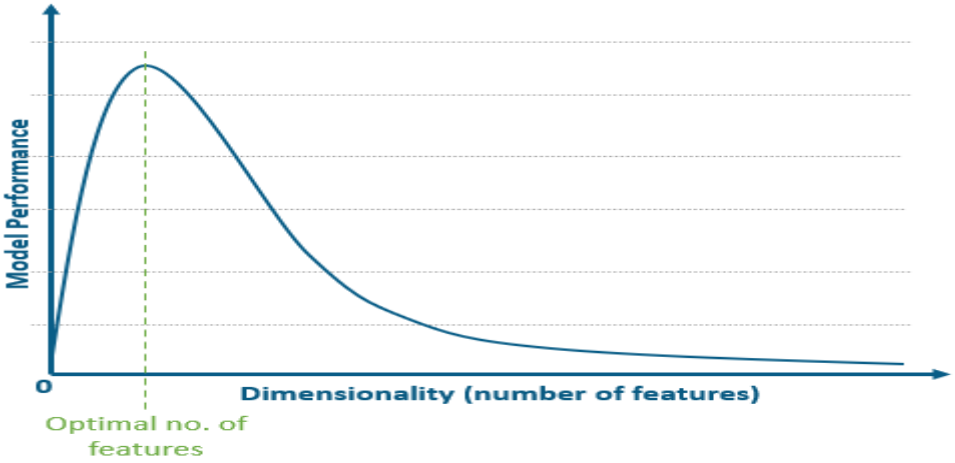
- As we can see the 3 nearest neighbors are from category A, hence this new data point must belong to category A.
-

2Q) What is the Curse of Dimensionality?

Ans:

The curse of dimensionality basically refers to the difficulties a machine learning algorithm faces when working with data in the higher dimensions, that did not exist in the lower dimensions. This happens because when you add dimensions, the minimum data requirements also increase rapidly.

This means, that as the number of features (columns) increases, you need an exponentially growing number of samples (rows) to have all combinations of feature values well-represented in our sample.



With the increase in the data dimensions, your model –

- would also increase in complexity.
- would become increasingly dependent on the data it is being trained on.

This leads to overfitting of the model, so even though the model performs really well on training data, it fails drastically on any real data.

Quite a few algorithms work only on *tall, svelte datasets* with fewer features and more samples. Hence, to remove the curse afflicting your model, you might need to put your data on a diet – i.e., reduce its dimensions through feature selection and feature engineering techniques. Let's see how this is done!

Dimensionality Reduction to the Rescue

What you need to understand first is that data features are usually correlated. Hence, the higher dimensional data is dominated by a rather small number of features. If we can find a subset of the *super features* that can represent the information just as well as the original dataset, we can remove the curse of dimensionality!

This is what dimensionality reduction is – a process of reducing the dimension of your data to a few principal features.

Fewer input dimensions often correspond to a simpler model, referred to as its **degrees of freedom**. A model with larger degrees of freedom is more prone to overfitting. So, it is desirable to have more generalized models, and input data with fewer features.

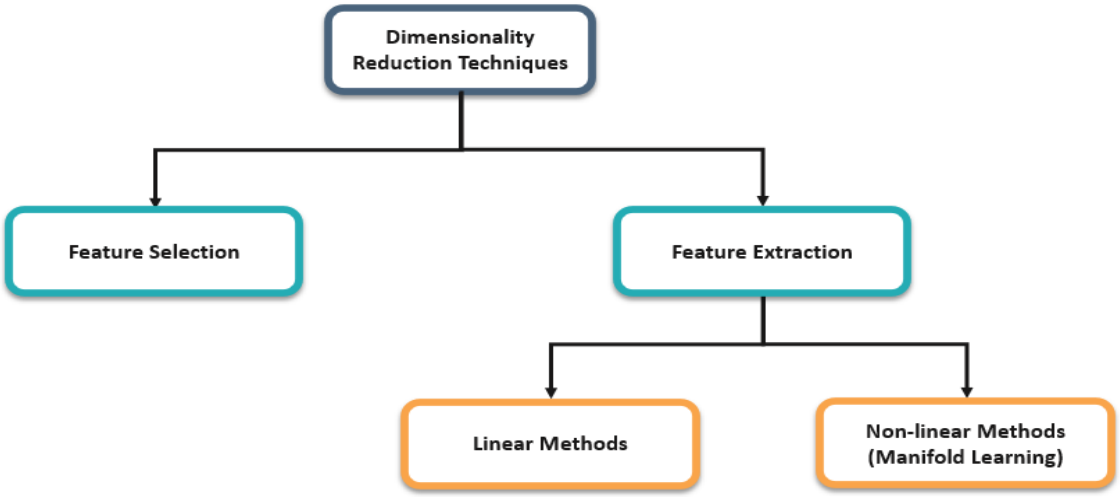
Why is Dimensionality Reduction necessary?

- **Avoids overfitting** – the lesser assumptions a model makes, the simpler it will be.
- **Easier computation** – the lesser the dimensions, the faster the model trains.
- **Improved model performance** – removes redundant features and noise, lesser misleading data improves model accuracy.
- Lower dimensional data requires **less storage space**.
- Lower dimensional data can **work with other algorithms** that were unfit for larger dimensions.

How is Dimensionality Reduction done?

Several techniques can be employed for dimensionality reduction depending on the problem and the data. These techniques are divided into two broad categories:

1. **Feature Selection:** Choosing the most important features from the data
2. **Feature Extraction:** Combining features to create new *superfeatures*.



Now, we are going to demonstrate how to get rid of the curse of dimensionality. We will be performing dimensionality reduction through a common linear method – Principal Component Analysis (PCA).

3Q) A Really Dumb Spam Filter

Ans: Imagine a "universe" that consists of receiving a message chosen randomly from all possible messages. Let S be the event "the message is spam" and V be the event "the message contains the word viagra." Then Bayes's Theorem tells us that the probability that the message is spam conditional on containing the word Viagra is:

$$P(S \mid V) = [P(V \mid S)P(S)] / [P(V \mid S)P(S) + P(V \mid \neg S)P(\neg S)]$$

The numerator is the probability that a message is spam and contains viagra, while the denominator is just the probability that a message contains viagra. Hence youcan think of this calculation as simply representing the proportion of Viagra messages that are spam. If we have a large collection of messages we know are spam, and a large collection of messages we know are not spam, then we can easily estimate

$P(V \mid S)$ and $P(V \mid \neg S)$. If we further assume that any message is equally likely to be spam or not spam(so that $P(S) = P(\neg S) = 0.5$), then:

$$P(S \mid V) = P(V \mid S) / [P(V \mid S) + P(V \mid \neg S)]$$

For example, if 50% of spam messages have the word viagra, but only 1% of nonspam messages do, then the probability that any given viagra-containing email is spam is:

$$0.5 / (0.5 + 0.01) = 98\ \%$$

4Q) A More Sophisticated Spam Filter

Ans: Imagine now that we have a vocabulary of many words $w_1, \ldots, w_n$. To move this into the realm of probability theory, we'll write X_i for the event "a message contains the word w_i." Also imagine that we've come up with an estimate $P(X_i \mid S)$ for the probability that a spam message contains the *i*th word, and a similar estimate $P(X_i \mid \neg S)$ for the probability that a nonspam message containsthe *i*th word.

The key to Naive Bayes is making the (big) assumption that the presences (or absences) of each word are independent of one another, conditional on a message being spam or not.

Intuitively, this assumption means that knowing whether a certain spam message contains the word "viagra" gives you no information about whether that same message contains theword "rolex." In math terms, this means that:

$$P(X_1 = x_1, \ldots, X_n = x_n \mid S) = P(X_1 = x_1 \mid S) \times \cdots \times P(X_n = x_n \mid S)$$

This is an extreme assumption. Imagine that our vocabulary consists *only* of the words "viagra" and "rolex," and that half of all spam messages are for "cheap viagra" and that the other half are for "authentic rolex." In this case, the Naive Bayes estimate that a spam message contains both "viagra" and "rolex" is:

$$P(X_1 = 1, X_2 = 1 \mid S) = P(X_1 = 1 \mid S) P(X_2 = 1 \mid S) = .5 \times .5 = .25$$

since we've assumed away the knowledge that "viagra" and "rolex" actually never occur together. Despite the unrealisticness of this assumption, this model often performs well and is used in actual spam filters.

The same Bayes's Theorem reasoning we used for our "viagra-only" spam filter tells usthat we can calculate the probability a message is spam using the equation:

$$P(S \mid X = x) = P(X = x \mid S) / [P(X = x \mid S) + P(X = x \mid \neg S)]$$

The Naive Bayes assumption allows us to compute each of the probabilities on the right simply by multiplying together the individual probability estimates for each vocabulary word.

In practice, you usually want to avoid multiplying lots of probabilities together, to avoid a problem called *underflow*, in which computers don't deal well with floating-point numbersthat are too close to zero. Recalling from algebra that $\log(ab) = \log a + \log b$ and that $\exp(\log x) = x$, we usually compute $p_1 * \cdots * p_n$ as the equivalent

$$\exp(\log(p_1) + \cdots + \log(p_n))$$

The only challenge left is coming up with estimates for $P(X_i \mid S)$ and $P(X_i \mid \neg S)$, the probabilities that a spam message (or nonspam message) contains the word w_i. If we have a fair number of "training" messages labeled as spam and not-spam, an obvious first try is to estimate $P(X_i \mid S)$ simply as the fraction of spam messages containing word w_i.

This causes a big problem, though. Imagine that in our training set the vocabulary word "data" only occurs in nonspam messages. Then we'd estimate $P(\text{"data"} \mid S) = 0$. The result is that our Naive Bayes classifier would always assign spam probability 0 to *any* message containing the word "data," even a message like "data on cheap viagra and authentic rolex watches." To avoid this problem, we usually use some kind of smoothing.

In particular, we'll choose a *pseudocount* — k — and estimate the probability of seeingthe ith word in a spam as:

$$P(X_i \mid S) = (k + \text{number of spams containing } w_i) / (2k + \text{number of spams})$$

Similarly for $P(X_i \mid \neg S)$. That is, when computing the spam probabilities for the ith word, we assume we also saw k additional spams containing the word and k additional spams not containing the word.

For example, if "data" occurs in 0/98 spam documents, and if k is 1, we estimate $P(\text{"data"} \mid S)$ as 1/100 = 0.01, which allows our classifier to still assign some nonzerospam probability to messages that contain the word "data."

Implementation

Now we have all the pieces we need to build our classifier. First, let's create a simple function to tokenize messages into distinct words. We'll first convert each message to lowercase; use re.findall()to extract "words" consisting of letters, numbers, and apostrophes; and finally use set()to get just the distinct words:

```
def tokenize(message):
    message = message.lower()
    all_words = re.findall("[a-z0-9']+", message)
    return set(all_words)
```

Our second function will count the words in a labeled training set of messages. We'll have it return a dictionary whose keys are words, and whose values are two-element lists [spam_count, non_spam_count] corresponding to how many times we saw that word in both spam and nonspam messages:

```
def count_words(training_set):
    counts = defaultdict(lambda: [0, 0])
    for message, is_spam in training_set:
        for word in
            tokenize(message):
            counts[word][0 if
            is_spam else 1] += 1
    return counts
```

Our next step is to turn these counts into estimated probabilities using the smoothing we described before. Our function will return a list of triplets containing each word, the probability of seeing that word in a spam message, and the probability of seeing that wordin a nonspam message:

```
def word_probabilities(counts, total_spams, total_non_spams, k=0.5):
    return [(w,
             (spam + k) / (total_spams
             + 2 * k), (non_spam + k) /
             (total_non_spams + 2 * k))
             for w, (spam, non_spam) in counts.iteritems()]
```

The last piece is to use these word probabilities (and our Naive Bayes assumptions) toassign probabilities to messages:

```
def
    spam_probability(word_pro
    bs, message):
    message_words =
    tokenize(message)
    log_prob_if_spam =
    log_prob_if_not_spam = 0.0
    for word, prob_if_spam, prob_if_not_spam in word_probs:
        if word in message_words:
            log_prob_if_spam += math.log(prob_if_spam) log_prob_if_not_spam
            += math.log(prob_if_not_spam)
          else:
```

```
log_prob_if_spam += math.log(1.0 - prob_if_spam) log_prob_if_not_spam +=
math.log(1.0 - prob_if_not_spam)

prob_if_spam = math.exp(log_prob_if_spam) prob_if_not_spam =
math.exp(log_prob_if_not_spam) return prob_if_spam / (prob_if_spam +
prob_if_not_spam)
```

We can put this all together into our Naive Bayes Classifier:

```
class  NaiveBayesClassifier: def__init__(self, k=0.5):
self.k = k self.word_probs = []

def train(self, training_set):
num_spams = len([is_spam
for message, is_spam in training_set
if is_spam])
num_non_spams = len(training_set) - num_spams

word_counts = count_words(training_set) self.word_probs =
word_probabilities(word_counts,
num_spams, num_non_spams,self.k)

def classify(self, message):
return spam_probability(self.word_probs, message)
```

5Q) Testing Our Model

- A good data set is the SpamAssassin public corpus. We'll look at the files prefixed with *20021010*. (On Windows, you might need a program like 7-Zip to decompress and extract them.)
- After extracting the data (to, say, *C:\spam*) you should have three folders: *spam*, *easy_ham*, and *hard_ham*. Each folder contains many emails, each contained in a single file. To keep things *really* simple, we'll just look at the subject lines of each email.
- How do we identify the subject line? Looking through the files, they all seem to start with "Subject:". So we'll look for that:

```
import glob, re
path = r"C:\spam\*\*"data = []
for fn in glob.glob(path): is_spam = "ham" not in fn
    with open(fn,'r') as file:
        for line in file:
            if line.startswith("Subject:"):
                subject = re.sub(r"^Subject: ", "", line).strip() data.append((subject,
                is_spam))
```

- Now we can split the data into training data and test data, and then we're ready to build a classifier:

```
random.seed(0)
train_data, test_data = split_data(data, 0.75)
classifier = NaiveBayesClassifier()classifier.train(train_data)
```

And then we can check how our model does:

```
classified = [(subject, is_spam, classifier.classify(subject))
              for subject, is_spam in test_data]
Counter((is_spam, spam_probability > 0.5)
                for _, is_spam, spam_probability in classified)
```

This gives 101 true positives (spam classified as "spam"), 33 false positives (ham classified as "spam"), 704 true negatives (ham classified as "ham"), and 38 false negatives(spam classified as "ham"). This means our precision is 101 / (101 + 33) = 75%, and our recall is 101 / (101 + 38) = 73%, which are not bad numbers for such a simple model.

7Q) Simple Linear Regression

Ans:

Simple Linear Regression is a type of Regression algorithms that models the relationship between a dependent variable and a single independent variable. The relationship shown by a Simple Linear Regression model is linear or a sloped straight line, hence it is called Simple Linear Regression.

The key point in Simple Linear Regression is that the ***dependent variable must be a continuous/real value***. However, the independent variable can be measured on continuous or categorical values.

Simple Linear regression algorithm has mainly two objectives:

- **Model the relationship between the two variables.** Such as the relationship between Income and expenditure, experience and Salary, etc.
- **Forecasting new observations.** Such as Weather forecasting according to temperature, Revenue of a company according to the investments in a year, etc.

Simple Linear Regression Model:

The Simple Linear Regression model can be represented using the below equation:

$$y= a_0+a_1x+ \varepsilon$$

Where,

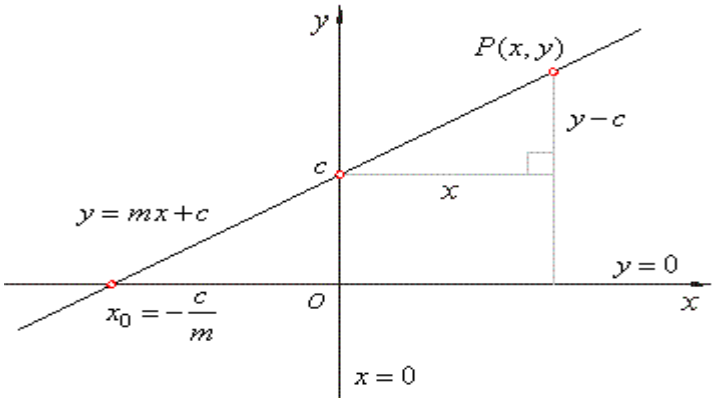
a0= It is the intercept of the Regression line (can be obtained putting x=0)
a1= It is the slope of the regression line, which tells whether the line is increasing or decreasing.
ε = The error term. (For a good model it will be negligible)

8Q) Linear Regression using Gradient Descent

Ans:

In statistics, linear regression is a linear approach to modelling the relationship between a dependent variable and one or more independent variables. Let **X** be the independent variable and **Y** be the dependent variable. We will define a linear relationship between these two variables as follows:

$$Y = mX + c$$



This is the equation for a line that you studied in high school. **m** is the slope of the line and **c** is the y intercept. Today we will use this equation to train our model with a given dataset and predict the value of **Y** for any given value of **X**. Our challenge today is to determine the value of **m** and **c**, such that the line corresponding to those values is the best fitting line or gives the minimum error.

Loss Function

The loss is the error in our predicted value of **m** and **c**. Our goal is to minimize this error to obtain the most accurate value of **m** and **c**. We will use the Mean Squared Error function to calculate the loss. There are three steps in this function:

1. Find the difference between the actual y and predicted y value(y = mx + c), for a given x.
2. Square this difference.
3. Find the mean of the squares for every value in X.

$$E = \frac{1}{n}\sum_{i=0}^{n}(y_i - \bar{y}_i)^2$$

Mean Squared Error Equation

Here y_i is the actual value and $\bar{y}_i$ is the predicted value. Lets substitute the value of $\bar{y}_i$:

$$E = \frac{1}{n}\sum_{i=0}^{n}(y_i - (mx_i + c))^2$$

Substituting the value of $\bar{y}_i$

So we square the error and find the mean. hence the name Mean Squared Error. Now that we have defined the loss function, lets get into the interesting part — minimizing it and finding **m** and **c.**

9Q) Maximum Likelihood Estimation

Ans

Maximum likelihood estimation or otherwise noted as MLE is a popular mechanism which is used to estimate the model parameters of a regression model. Other than regression, it is very often used in statics to estimate the parameters of various distribution models.

Maximum likelihood estimation (MLE) is a technique used for estimating the parameters of a given distribution, using some observed data. For example, if a population is known to follow a "normal distribution" but the "mean" and "variance" are unknown, MLE can be used to estimate them using a limited sample of the population. MLE does that by finding particular values for the parameters (mean and variance) so that the resultant model with those parameters (mean and variance) would have generated the data.

So generally, likelihood expression is in the form of: ***L(parameters | data).*** Meaning of this is, "*likelihood of having these parameters, once the data are these*".

Likelihood and Probability are two different things although they look and behaves same. We talk about probability when we know the model parameters and when predicting a value from that model. So there we talk about how probable is the resultant value to be come out from that model. So probability is: ***P(data | parameters)***

Now we can see that Likelihood is other side of probability. That is we are going to guess the model parameters from the data. So there we know the results well and we know for sure that they have occured (probability = 1).