APPENDIX

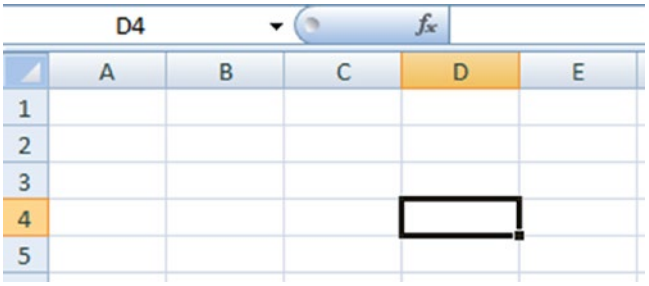
Basics of Excel, R, and Python

This chapter introduces the three tools mentioned often in this book: Microsoft Excel and the two programming languages R and Python.

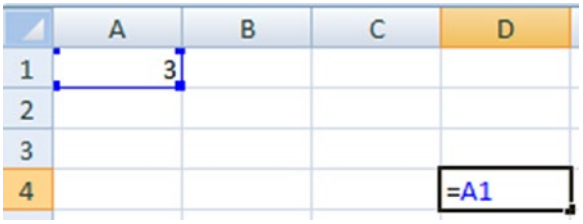
Basics of Excel

In Microsoft Excel, each cell is represented by numbers in rows and letters in columns.

For example, the following highlighted cell is the cell D4:



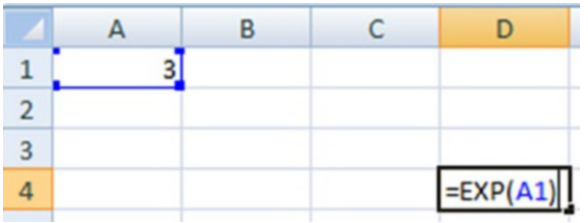
Cells can be referenced by specifying the = symbol followed by the cell we are trying to refer to. For example, if we want the cell D4 to reflect the value in cell A1, we would type it in as follows:



V. K. Ayyadevara, *Pro Machine Learning Algorithms*, https://doi.org/10.1007/978-1-4842-3564-5

Pressing F2 gets you to the formula corresponding to a cell.

We can do multiple manipulations on top of a given cell value with the various functions built-in to Excel. For example, here's how to equate the cell value of D4 to the exponential of the cell value of A1:



Excel provides an optimization tool that comes in handy in various techniques discussed in this book, called the Solver. Excel Solver is an add-in that must be installed. Once installed, it's available in the Data tab in the Excel ribbon at the top.

A typical Solver looks like Figure A-1.

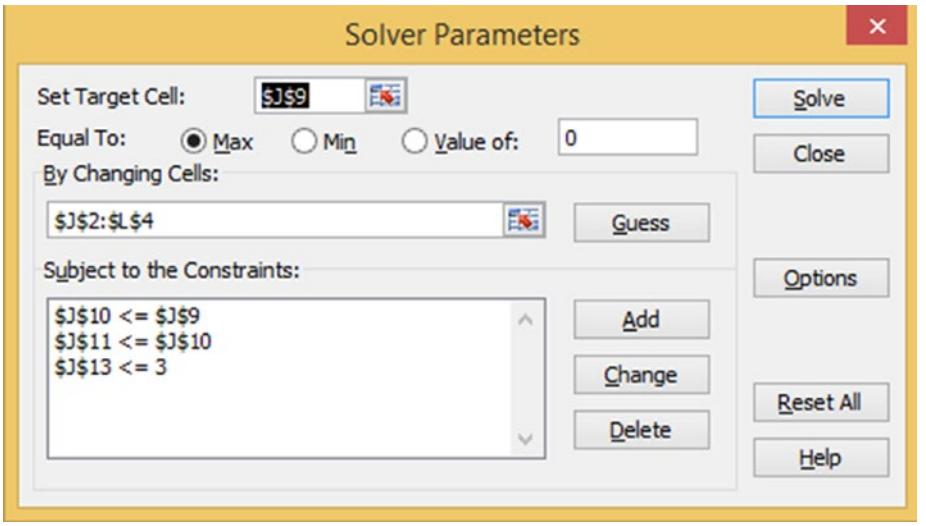


Figure A-1. *A typical Solver*

In the Set Target Cell part of Solver, you can specify the target that needs to be worked on.

In the Equal To section, you specify the objective—whether you want to minimize the target, maximize it, or set it to a value. This comes in very handy in scenarios where the target is the error value and we want to minimize the error.

The next section, "By Changing Cells", specifies the cells that can be changed to achieve the objective.

Finally, the Subject to the Constraints section specifies all the constraints that govern the objective.

Clicking the Solve button gives us the optimal cell values that achieve our objective.

Solver works using multiple algorithms, all of which are based on *back-propagation* (a technique discussed in detail in Chapter 7).

There are a lot more functionalities in Excel that can be very helpful, but for the purposes of this book—to show you how the algorithms work—you are in good shape if you understand Solver and cell linkages well.

Basics of R

The R programming language is an offshoot of a programming language called S. R was developed by Ross Ihaka and Robert Gentleman from the University of Auckland, New Zealand. It was primarily adopted by statisticians and is now the de facto standard language for statistical computing.

- *R is data analysis software*: Data scientists, statisticians, analysts, and others who need to make sense of data use R for statistical analysis, predictive modeling, and data visualization.
- *R is a programming language*: You do data analysis in R by writing scripts and functions in the R programming language. R is a complete, interactive, object-oriented language. The language provides objects, operators, and functions that make the process of exploring, modeling, and visualizing data a natural one. Complete data analyses can often be represented in just a few lines of code.
- *R is an environment for statistical analysis*: Available in the R language are functions for virtually every date manipulation, statistical model, or chart that the data analyst could ever need.
- *R is an open source software project*: Not only does this mean that you can download and use R for free, but the source code is also open for inspection and modification to anyone who wants to see how the methods and algorithms work under the hood.

Downloading R

R works on many operating systems, including Windows, Macintosh, and Linux. Because R is free software, it is hosted on many different servers (mirrors) around the world and can be downloaded from any of them. For faster downloads, you should choose a server close to your physical location. A list of all available download mirrors is available at www.r-project.org/index.html. Click Download R on the front page to choose your mirror for downloading.

You may notice that many of the download URLs include the letters CRAN. *CRAN* stands for the Comprehensive R Archive Network, and it ensures that you have the most recent version of R.

Once you have chosen a mirror, at the top of your screen you should see a list of the versions of R for each operating system. Choose the R version that works on your operating system (also, you should download base R), and then click the download link to download.

Installing and Configuring RStudio

RStudio is an integrated development environment (IDE) dedicated to R development.

RStudio requires a pre-installed R instance, and in RStudio config, an R version must be set (usually it is auto-set by RStudio when it is installed). RStudio is a more user-friendly version of R compared to the native R version.

1. Go to www.rstudio.com/products/rstudio/download/.
2. Click the Download RStudio Desktop button.
3. Select the installation file for your system.
4. Run the installation file.
5. RStudio will be installed on your system. It normally detects your latest installed R version automatically. Ideally, you should be able to use R from within RStudio without extra configuration.

Getting Started with RStudio

RStudio displays the main windows shown in Figure A-2.

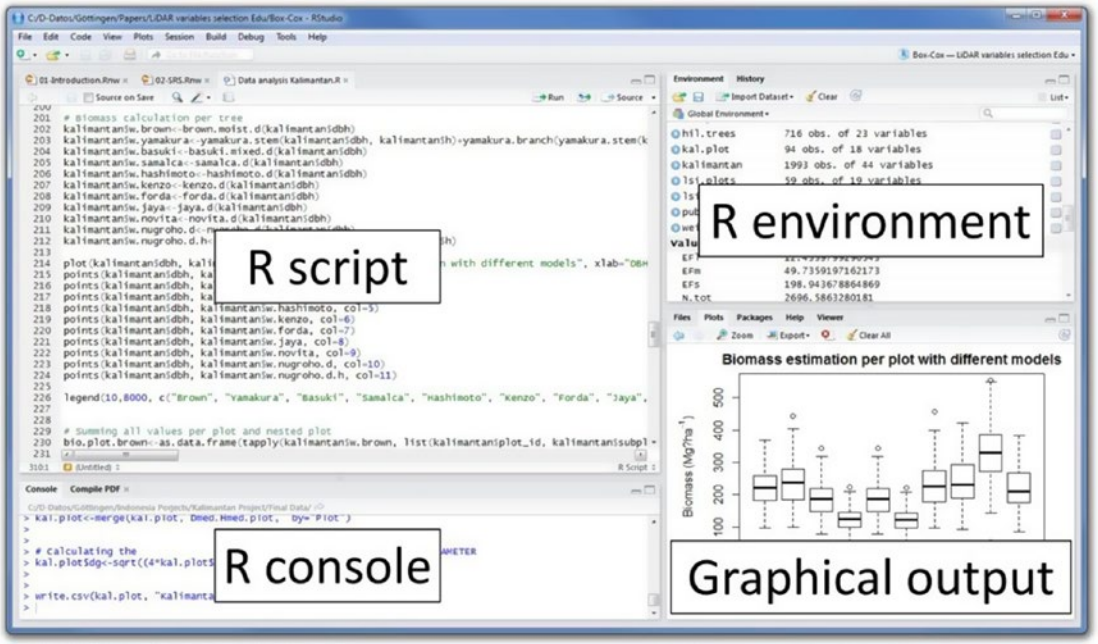


Figure A-2. *RStudio's main windows*

You can perform various functions in R, as follows (code available in github as "R basics.R"):

```
#Basic calculations
1+1
2*2
#Logical operators
1>2
1<2
1&0
1|0
#Creating a vector
n = c(2, 3, 5,6,7)
s = c("aa", "bb", "cc", "dd", "ee")
```

```
s
s = c( "bb", 1)
s
a=c(n,s)

#Creating a list
x = list(n, s)
#Creating a matrix
as.matrix(n)
as.matrix(s)

#as.matrix(c(n,s),nrow=5,ncol=2,byrow=TRUE)
help(as.matrix)
matrix(c(1,2,3, "a","b","c"), nrow = 3, ncol = 2)
matrix(c(n,s), nrow = 5, ncol = 2)

# Importing datasets
help(read)
t=read.csv("D:/in-class/Titanic.csv")
t2=read.table("D:/in-class/credit_default.txt")

#In case there is an issue with importing dataset, consider specifying
quote=, as below
t3=read.csv("D:/in-class/product_search.csv",quote="\"")

# Structure of dataset
t=read.csv("D:/in-class/Titanic.csv")
str(t)  # gives us quite a few information about the dataset
class(t) # typically, the imported datasets as always imported as data.
frame objects
dim(t) # dimension of an abject (data.frame)
nrow(t) # number of rows
ncol(t) # number of columns
colnames(t) # specify column names
class(t$Survived) # class of a variable
head(t) # gives us the first few rows of a dataset
t[1,1] # gives us the value in first row & first column. Note the syntax:
[rows,columns]
```

```
t[1:3,1] # gives us the values in first 3 rows of 1st column
t[c(1,100,500),]
t[1:3,] # gives us the values in first 3 rows of all columns (note that
when we dont specify the filtering condition in column index, the result
includes all columns)
t$Survived[1:3]# gives us the values of first 3 rows of "Survived" variable

t[c(1,4),-c(1,3)] # c() is a function to get a combination of values.
c(1,4) gives us the first & 4th row. -c(1) excludes the first column
t[c(1,4),c("Survived","Fare")] # c() is a function to get a combination of
values. c(1,4) gives us the first & 4th row. -c(1) excludes the first column
t[1:3,"Survived"]
# Data manipulation
t$PassengerId=NULL
summary(t)
t$unknown_age=0
t$unknown_age=ifelse(is.na(t$Age),1,0)  # the syntax here is,
ifelse(condition, value if condition is true, value if condition is false)
# is.na() function helps us in identifying if there any NA values in
dataset. Be sure to remove or impute (replace) NA values within a dataset
unique(t$Embarked) # gives all the unique values within a variable
table(t$Embarked) # table gives a count of all the unique values in dataset
mean(t$Age) # mean as a function
sum(t$Age)
mean(t$Age,na.rm=TRUE) # na.rm=TRUE helps in removing the missing values,
if they exist
t$Age2=ifelse(t$unknown_age==1,mean(t$Age,na.rm=TRUE),t$Age) # one
initializes a new variable within a dataset by using the $ operator & the
new variable name
summary(t) # summary is typically the first step performed after importing
dataset
order(t$Age2) # order function is used to sort dataset. It gives out the
index of rows that a value is sorted to
t=t[order(t$Age2),]  # t is sorted based on the order of age
mean(t$Survived[1:50])
mean(t$Survived[(nrow(t)-50):nrow(t)])
```

```
t2=t[1:50,]
t_male=t[t$Sex=="male",] # one can filter for criterion by specifying the
variable name with a == and the value it is to be filtered to
t_female=t[t$Sex=="female",]

mean(t_male$Survived)
mean(t_female$Survived)

t2=t[t$Sex=="male" & t$Age2<10,]
mean(t2$Survived)
t2=t[t$Sex=="female" | t$Age2<10,]
mean(t2$Survived)

install.packages("sqldf")
library(sqldf)
t3=sqldf("select sex,avg(survived) from t group by sex")
# SQL like filtering or aggregation can be done using the sqldf function
t$age3=ifelse(t$Age2<10,1,0)
t2=t[,c("Age2","Sex","Survived")] # one can filter for the columns of
interest y specifying the c() function with the variables that are needed
help(aggregate)
aggregate(t$Survived,by=list(t$Sex,t$Pclass),sum)
# aggregate function works similar to sqldf where grouping operations can
be done
#seq function is used to generate numbers by a given step size.
seq(0,1,0.2) gives us c(0,0.2,0.4,0.6,0.8,1)
#quantile gives us the values at the various percentiles specified
help(quantile)
summary(t)
t$age3=as.character(t$Age) # as.character function converts a value into a
character variablle
quantile(t$Age2,probs=seq(0,0.5,0.1))[2] # gives us the second value in the
output of quantile function

x=quantile(t$Age2,probs=seq(0,1,0.1))[2]
t2=t[t$Age2<x,]
mean(t2$Survived)
```

```
t$less_than_10=ifelse(t$Age2<x,1,0)
aggregate(t$Survived,by=list(t$Sex,t$less_than_10),mean) # aggregation can
be done over multiple variables by using c() function

t2=t[!t$Age2<x,] # ! is used as a engation statement
mean(t2$Survived)

# Loops
t=read.table("D:/in-class/credit_default.txt")

for(i in 1:3){
  print(i)
}
summary(t)
# a good idea is to note the difference between mean & median values of
variables
mean(t$DebtRatio)
median(t$DebtRatio)
t2=t

# it's a good practcie to test out the code of for loop before looping it
through, by assigning a certain value of i & test out the for loop code
i=2

t2[,i]=ifelse(is.na(t2[,i]),median(t2[,i],na.rm=TRUE),t2[,i])
t2[,i]=ifelse(t2[,i]<median(t2[,i],na.rm=TRUE),"Low","High")
print(aggregate(t2$SeriousDlqin2yrs,by=list(t2[,i]),mean))

for(i in 1:ncol(t2)){
  t2[,i]=ifelse(is.na(t2[,i]),median(t2[,i],na.rm=TRUE),t2[,i])
}

# below is an exercise where we are imputing missing value with median
values & then flagging variables as high when the value is above median
value & low when the value is below median value
for(i in 2:ncol(t2)){
  t2[,i]=ifelse(is.na(t2[,i]),median(t2[,i],na.rm=TRUE),t2[,i])
  t2[,i]=ifelse(t2[,i]<median(t2[,i],na.rm=TRUE),"Low","High")
  print(colnames(t2[i]));
```

```
  print(aggregate(t2$SeriousDlqin2yrs,by=list(t2[,i]),mean))
}
df=data.frame(group=c("a","b"),avg=c(2,2))

#joins
search=read.csv("D:/in-class/product_search.csv",quote="\"")
descriptions=read.csv("D:/in-class/product_descriptions.csv",quote="\"")
summary(search)
colnames(search)
colnames(descriptions)

help(merge)
# in a typical merge function, we have to specify the x (first) table, the
y (second) table to which we are joining the first table
# we would also have to specify the variable based on which we are joining
the datasets using the "by" parameter
# in case the column name of by parameter is different in datasets, we can
use by.x & by.y
# by default merge does an inner join (inner join is when only the values
that are common in both tables are joined)
# all.x=TRUE helps us do a left join (left join is when all the values in x
table are retained even if some of them do not have a match in the second
table)
# all.y = TRUE does a right join where all the values in right (second)
table are retained
# assume that x table has productid as (1,2,3) & right (y) table has
productid as (1,5,6)
# inner join of these two tables gives us the values of only productid =1
(as it is the only one in common)
# left join gives us the information of (1,2,3) however, the info of pid 1
will be full and info of pid 2,3 would be blank, as right table does not
have info about these 2 pids
# right join givs the information of (1,5,6) where we have pid 1 info
completely and info of 5,6 is missing

search_descriptions=merge(search,descriptions,by="product_uid",all.x=TRUE)
search_descriptions1=merge(search,descriptions,by="product_uid",all.y=TRUE)
```

```
search_descriptions2=merge(descriptions,search,by="product_uid",all.x=TRUE)
nrow(search_descriptions)
nrow(search_descriptions1)
nrow(search_descriptions2)

search_descriptions2$missing_id=ifelse(is.na(search_descriptions2$id),1,0)
sum(search_descriptions2$missing_id)
x=search_descriptions2[search_descriptions2$missing_id==0,]
length(unique(x$Product_uid))

system.time(search_descriptions<-merge(search,descriptions,by="product_
uid",all.x=TRUE))

# note the difference in speed between base "merge" statement & fread/
data.table "merge" statment
install.packages("data.table")
library(data.table)
search=fread("D:/in-class/product_search.csv")
descriptions=fread("D:/in-class/product_descriptions.csv")

system.time(descriptions<-read.csv("D:/in-class/product_descriptions.csv"))
system.time(descriptions<-fread("D:/in-class/product_descriptions.csv"))

write.csv(search,"D:/in-class/search_output.csv",row.names=FALSE)

help(merge)

search_descriptions=merge(search,descriptions,by="product_uid",all.x=TRUE)
system.time(search_descriptions<-merge(search,descriptions,by="product_
uid",all.x=TRUE))

# writing custom functions
square = function(x) {x*x}

square(13.5)
square("two")

addition = function(x,y) {x+y}

tt=as.data.frame(quantile(t$Age2,probs=seq(0,1,0.1)))
```

Other functions relevant to various machine learning techniques are discussed in the respective chapters.

Basics of Python

Downloading and installing Python

For the purposes of this book, we will be using Python 3.5 (which is available in archived versions), the Anaconda version of which you can download from `www.continuum.io/downloads`.

Once the file is downloaded, install with all the default conditions in the installer. Once Anaconda is installed, search for "Anaconda prompt," as shown in Figure A-3.

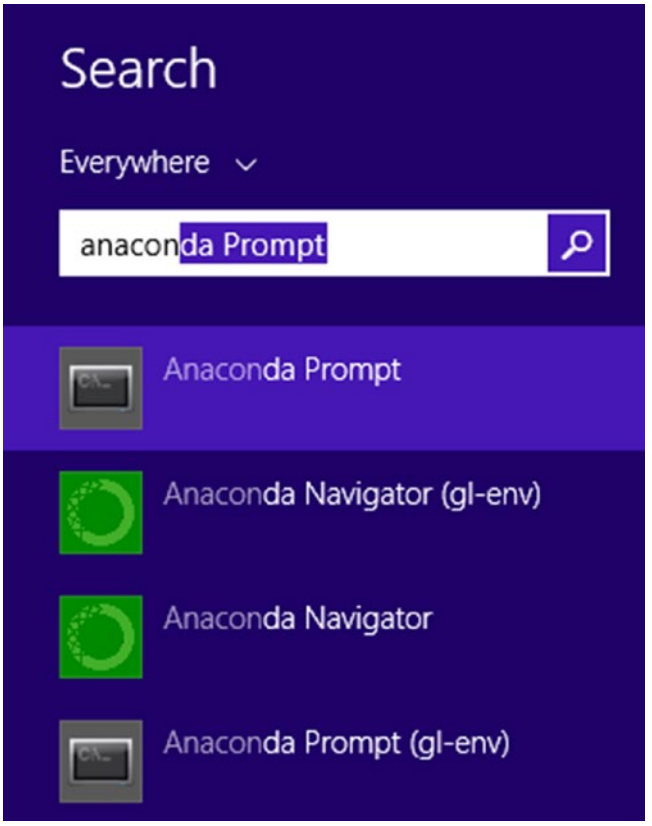


Figure A-3. *Searching for Anaconda prompt*

It takes a while for the prompt to appear (~1 minute). It looks similar to any command line or terminal program, as shown in Figure A-4.

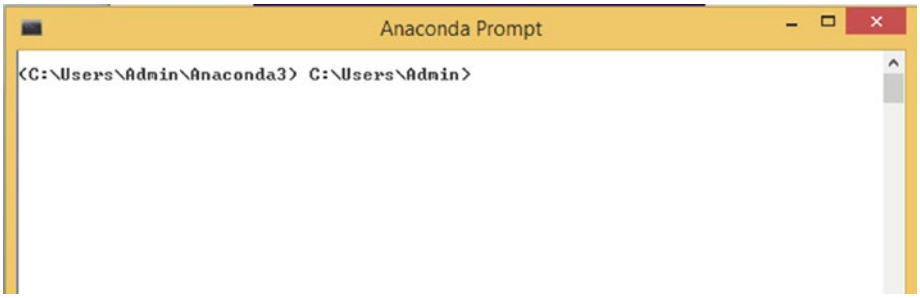


Figure A-4. *The Anaconda prompt*

Once you are able to type in the prompt, type **jupyter notebook**. A new web page opens up, as shown in Figure A-5.

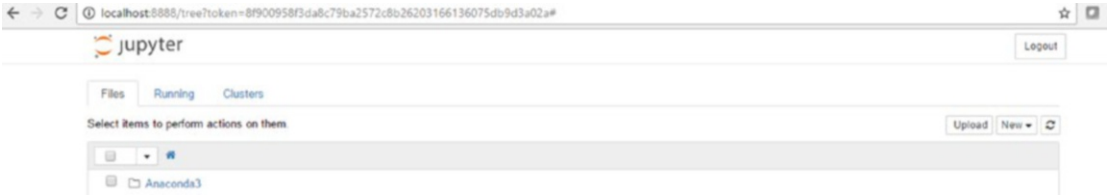


Figure A-5. *The Jupyter web page*

Click the New button and then click Python 3, as shown in Figure A-6.

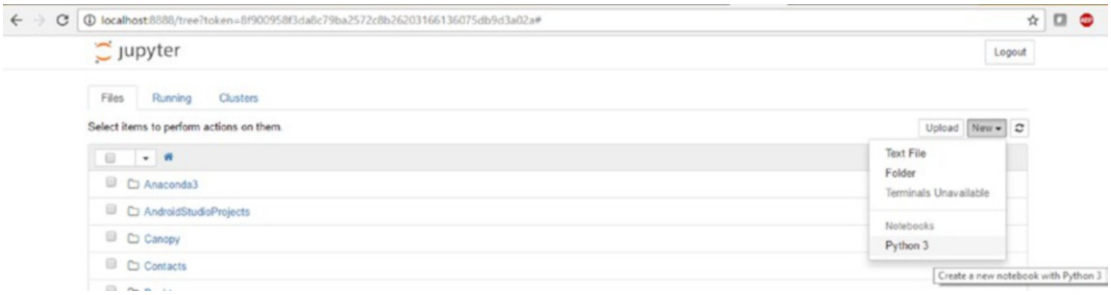


Figure A-6. *Selecting Python 3*

A new code editor page should appear, something like Figure A-7.

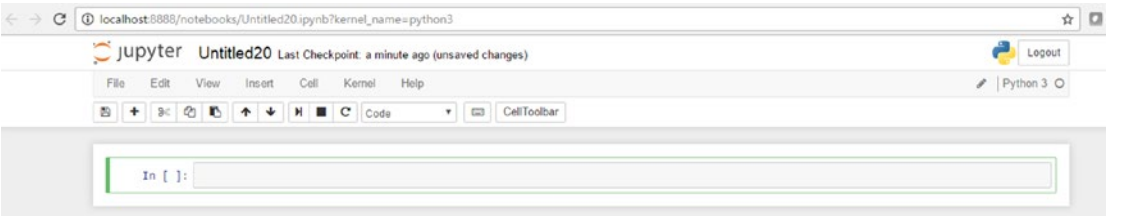


Figure A-7. *The code editor*

Type **1+1** in the space and press Shift+Enter to see if everything is working fine. It should look like Figure A-8.

Figure A-8. *The result of the addition*

Basic operations in Python

The following code shows some basic Python code (available as "Python basics.ipynb" in github).

```
# Python can perform basic calculator type operations
1 + 1
2 * 3
1 / 2
2 ** 4
# exponential
4 % 2
# modulus operator
5 % 2
7//4
```

```
# values can be assigned to variables
name_of_var = 2
x = 2
y = 3
z = x + y
# strings can also be assigned to variables
x = 'hello'
# Lists are very similar to arrays
# They are a combination of numbers
[1,2,3]
# A list can have multiple types of data - numeric or character
# A list can also have another list
['kish',1,[1,2]]
# A list can be assigned to an object, just like a value gets assigned to a
variable
my_list = ['a','b','c']
# just like we have a word and its corresponding value in physical
dictionary
# we have a key in place of word & value in place of meaning in python
dictionary
# Dictionaries help in mapping one value to another
d = {'key1':'item1','key2':'item2'}
d['key1']
d.keys()
# A boolean is a true or false value
True
False
#Basic Python implements all of the usual operators for Boolean logic,
# but uses English words rather than symbols
# A package called "pandas" (we will work on it soon) uses & and | symbols
though for and ,
# or operations
t = True
f = False
```

```
print(type(t)) # Prints "<type 'bool'>"
print(t and f) # Logical AND; prints "False"
print(t or f)  # Logical OR; prints "True"
print(not t)   # Logical NOT; prints "False"
print(t != f)  # Logical XOR; prints "True"
# Sets can help obtain unique values in a collection of elements
{1,2,3}
{1,2,3,1,2,1,2,3,3,3,3,2,2,2,1,1,2}
1 > 2
1 < 2
1 >= 1
1 <= 4
# Please note the usage of == instead of single =
1 == 1
'hi' == 'ahoy'
# Note how we used "and", "or"
(1 > 2) and (2 < 3)
# Writing a for loop
seq = [1,2,3,4,5]
for item in seq:
    print(item)
for i in range(5):
    print(i)
# Writing a function
def square(x):
    return x**2
out = square(2)
st = 'hello my name is Kishore'
st.split()
```

Numpy

Numpy is a fundamental package in Python which has some extremely useful functions for mathematical computations as well as abilities to work on multi dimensional data. Moreover it is very fast. We will go through a small demo of how fast numpy is when compared to traditional way of calculation, in the below code:

```
# In the below code, we are trying to sum up the square of first 10 Million
numbers
# packages can be imported as follows
import numpy as np
a=list(range(10000000))
len(a)
import time
start=time.time()
c=0
for i in range(len(a)):
   c= (c+a[i]**2)
end=time.time()
print(c)
print("Time to execute: "+str(end-start)+"seconds")
a2=np.double(np.array(a))
import time
start=time.time()
c=np.sum(np.square(a2))
end=time.time()
print(c)
print("Time to execute: "+str(end-start)+"seconds")
```

```
333333283333335000000
Time to execute using for loop: 17.9579999447seconds
3.3333328333333443e+20
Time to execute using Numpy: 0.0920000076294seconds
```

Once you implement the code, you should notice that there is a >100X improvement over traditional way of calculation using Numpy.

Number generation using Numpy

```
# notice that np automatically outputted zeroes
np.zeros(3)
# we can also create n dimensional numpy arrays
np.zeros((5,5))
# similar to zeros, we can create arrays with a value of 1
```

```
np.ones(3)
np.ones((3,3))
# not just ones or zeros, we can initialize random numbers too
np.random.randn(5)
ranarr = np.random.randint(0,50,10)
# returns the max value of array
ranarr.max()
# returns the position of max value of the array
ranarr.argmax()
ranarr.min()
ranarr.argmin()
```

Slicing and indexing

```
arr_2d = np.array(([5,10,15],[20,25,30],[35,40,45]))
#Show
arr_2d
#Indexing row
# the below selects the second row, as index starts form 0
arr_2d[1]
# Format is arr_2d[row][col] or arr_2d[row,col]

# Getting individual element value

# the below gives 2nd row first column value
arr_2d[1][0]
# Getting individual element value
# same as above
arr_2d[1,0]
# if, we need the 2nd row & only the first & 3rd column values - the below
will do the job
arr_2d[1,[0,2]]
# 2D array slicing
```

```
#Shape (2,2) from top right corner

# you can read the below as - select all rows till 2nd index & select all
columns from 1st index
arr_2d[:2,1:]
```

Pandas

Pandas is a library that helps us in generating data frames that enable us in working with tabular data. In this section, we will learn about indexing and slicing data frames and also learn about additional functions in the library.

Indexing and slicing using Pandas

```
import pandas as pd
# create a data frame
# a data frame has certain rows and columns as specified
# give the index values of the created data frame
# also, specify the column names of this data frame
df = pd.DataFrame(randn(5,4),index='A B C D E'.split(),columns='W X Y
Z'.split())
# select all the values in a column
df['W']
# select columns by specifying column names
df[['W','Z']]
# selecting certain rows in a dataframe
df.loc[['A']]
# if multiple rows and columns are to be selected - specify the index
df.loc[['A','D'],['W','Z']]
# Create a new column
df['new'] = df['W'] + df['Y']
# drop a column
# not the usage of axis=1 - which stands for doing operation at a column
level
df.drop('new',axis=1)
```

```
# we can specify the condition based on which we want to filter the data
frame
df.loc[df['X']>0]
```

Summarizing data

```
# reading a csv file into dataframe
path="D:/in-class/train.csv"
df=pd.read_csv(path)
# fetching the columns names
print(df.columns)
# if else condition on data frames is accomplished using np.where
# notice the use of == instead of single =
df['Stay_In_Current_City_Years2']=np.where(df['Stay_In_Current_City_
Years']=="4+",4, df['Stay_In_Current_City_Years'])
# specify row filtering conditions
df2=df.loc[df['Marital_Status']==0]
# get the dimension of the dataframe
df2.shape
# extract the unique values of a column
print(df2['Marital_Status'].unique())
# extract the frequency of the unique values of a column
print(df2['Marital_Status'].value_counts())
```

Index

A

B

C

V. K. Ayyadevara, *Pro Machine Learning Algorithms*, https://doi.org/10.1007/978-1-4842-3564-5

D

E

F

G

H

I, J

K

L

M

N

O

P, Q

R

S

T

U

V

W, X, Y, Z