

# Toolbox

Bhaswar Chakma

2021-08-20



# Contents

	5
<b>1 dplyr vs pandas &amp; numpy</b>	<b>7</b>
1.1 select() . . . . .	9
1.2 mutate() . . . . .	11
1.3 filter() . . . . .	12
1.4 group_by() and summarize() . . . . .	13
1.5 arrange() . . . . .	14
<b>2 Helper Functions</b>	<b>17</b>
2.1 case_when() vs pd.cut() vs np.select() . . . . .	17
2.2 if_else() vs np.where() . . . . .	19
<b>3 tidyr vs pandas</b>	<b>21</b>
3.1 pivot_longer() vs melt() . . . . .	21
<b>4 Base Python</b>	<b>23</b>
4.1 map() . . . . .	23
4.2 zip() . . . . .	24
4.3 enumerate() . . . . .	25
<b>5 scikit-learn</b>	<b>27</b>
5.1 Linear Model . . . . .	27
5.2 Train-Test . . . . .	33
5.3 Cross Validation . . . . .	41

<b>6</b>	<b>R Strings</b>	<b>45</b>
6.1	String Manipulation with Base R Functions . . . . .	45
6.2	<code>stringr</code> . . . . .	46
6.3	Regular Expressions . . . . .	46
<b>7</b>	<b>SQL</b>	<b>49</b>
7.1	CREATE . . . . .	49
7.2	DROP . . . . .	50
7.3	ALTER . . . . .	50
7.4	TRUNCATE . . . . .	51
7.5	Guided Exercise: Create table and insert data . . . . .	51
7.6	Guided Exercise: Use the <code>ALTER</code> statement to add, delete, or modify columns in two of the existing tables created in the pre- vious exercise. . . . .	52
7.7	Guided Exercise: TRUNCATE . . . . .	53
7.8	Guided Exercise: DROP . . . . .	53
7.9	Exercise: String Patterns . . . . .	54
7.10	Exercise: Sorting . . . . .	55
7.11	Exercise 3: Grouping . . . . .	56





# Chapter 1

## dplyr vs pandas & numpy

We will use the five dplyr verbs (also pandas' guide) for comparison

- `select()` picks variables based on their names.
- `mutate()` adds new variables that are functions of existing variables
- `filter()` picks cases based on their values.
- `summarise()` reduces multiple values down to a single summary.
- `arrange()` changes the ordering of the rows.

and use the following toy data to apply the verbs.

name	gender	grade
Barney	Male	10
Ted	Male	11
Marshall	Male	13
Lilly	Female	12
Robin	Female	14

### Create Toy Data

dplyr

pandas

```
df <- tibble(
  name = c("Barney", "Ted", "Marshall",
           "Lilly", "Robin"),
  gender = c("Male", "Male", "Male",
             "Female", "Female"),
  grade = c(10, 11, 13, 12, 14)
)
df
```

```
## # A tibble: 5 x 3
##   name      gender grade
##   <chr>    <chr>  <dbl>
## 1 Barney   Male     10
## 2 Ted     Male     11
## 3 Marshall Male     13
## 4 Lilly   Female    12
## 5 Robin   Female    14
```

```
df = pd.DataFrame({
  'name': ["Barney", "Ted", "Marshall",
           "Lilly", "Robin"],
  'gender': ["Male", "Male", "Male",
             "Female", "Female"],
  'grade': [10, 11, 13, 12, 14]
})
df
```

```
##      name gender grade
## 0  Barney   Male    10
## 1    Ted   Male    11
## 2 Marshall  Male    13
## 3   Lilly Female    12
## 4   Robin Female    14
```

### Check Data Structure

dplyr

pandas

```
glimpse(df)
```

```
## Rows: 5
## Columns: 3
```



```
## $ name    <chr> "Barney", "Ted", "Marshall", "Lilly", "Robin"
## $ gender  <chr> "Male", "Male", "Male", "Female", "Female"
## $ grade   <dbl> 10, 11, 13, 12, 14
```

```
df.dtypes
```

```
## name      object
## gender     object
## grade      int64
## dtype: object
```

```
df.shape
```

```
## (5, 3)
```

```
df.info()
```

```
## <class 'pandas.core.frame.DataFrame'>
## RangeIndex: 5 entries, 0 to 4
## Data columns (total 3 columns):
## name      5 non-null object
## gender     5 non-null object
## grade      5 non-null int64
## dtypes: int64(1), object(2)
## memory usage: 248.0+ bytes
```

## 1.1 select()

**Task:** Pick the variables `name` and `grade`.

dplyr

pandas

```
df %>%
  select(name, grade)
```

```
## # A tibble: 5 x 2
##   name      grade
##   <chr>    <dbl>
## 1 Barney      10
## 2 Ted          11
## 3 Marshall    13
## 4 Lilly       12
## 5 Robin       14
```

```
df[['name', 'grade']]
```

```
##      name  grade
## 0  Barney    10
## 1    Ted     11
## 2 Marshall    13
## 3   Lilly    12
## 4   Robin    14
```

```
# or
df.drop(columns = ['gender'])
```

```
##      name  grade
## 0  Barney    10
## 1    Ted     11
## 2 Marshall    13
## 3   Lilly    12
## 4   Robin    14
```

```
# or
df.drop(['gender'], axis = 1)
```

```
##      name  grade
## 0  Barney    10
## 1    Ted     11
## 2 Marshall    13
## 3   Lilly    12
## 4   Robin    14
```

Using positions of columns:

```
df[df.columns[[0,2]]]
```

```
##      name  grade
## 0  Barney    10
## 1    Ted     11
## 2 Marshall    13
## 3   Lilly    12
## 4   Robin    14
```

```
df.iloc[:, [0,2]]
```

```
##      name  grade
## 0   Barney    10
## 1     Ted     11
## 2  Marshall    13
## 3    Lilly    12
## 4    Robin    14
```

## 1.2 mutate()

**Task:** Generate a variable `grade_p`, expressing grade out of 100.

dplyr

pandas

```
df %>%
  mutate(grade_p = grade/20*100)
```

```
## # A tibble: 5 x 4
##   name      gender grade grade_p
##   <chr>    <chr>  <dbl>  <dbl>
## 1 Barney   Male      10      50
## 2 Ted     Male      11      55
## 3 Marshall Male      13      65
## 4 Lilly   Female     12      60
## 5 Robin   Female     14      70
```

```
df['grade_p'] = df['grade']/20*100
df
```

```
##      name  gender  grade  grade_p
## 0   Barney   Male     10    50.0
## 1     Ted   Male     11    55.0
## 2  Marshall   Male     13    65.0
## 3    Lilly  Female     12    60.0
## 4    Robin  Female     14    70.0
```

```
# now drop the newly created variable
df.drop(columns = 'grade_p', inplace = True)
```

### 1.3 filter()

**Task: Keep Barney or females.**

dplyr

pandas

```
df %>%
  filter(name == "Barney" |
         gender == "Female")
```

```
## # A tibble: 3 x 3
##   name  gender grade
##   <chr> <chr>  <dbl>
## 1 Barney Male      10
## 2 Lilly Female     12
## 3 Robin Female     14
```

```
# similar to base R
df[(df["name"] == "Barney") |
   (df["gender"] == "Female")]
```

```
##      name  gender  grade
## 0  Barney   Male    10
## 3   Lilly  Female    12
## 4   Robin  Female    14
```

```
# query with ''; need to use "" for conditions
df.query('name == "Barney"|gender == "Female"')
```

```
##      name  gender  grade
## 0  Barney   Male    10
## 3   Lilly  Female    12
## 4   Robin  Female    14
```

```
# query with ""; need to use '' for conditions
df.query("name == 'Barney'| gender == 'Female'")
```

```
##      name  gender  grade
## 0  Barney   Male    10
## 3   Lilly  Female    12
## 4   Robin  Female    14
```

## 1.4 group\_by() and summarize()

**Task:** Grouped by gender, find mean grade.

dplyr

pandas

```
df %>%
  group_by(gender) %>%
  summarize(avg_grade = mean(grade))
```

```
## # A tibble: 2 x 2
##   gender avg_grade
##   <chr>     <dbl>
## 1 Female      13
## 2 Male       11.3
```

*# returns a series*

```
df.groupby("gender")["grade"].mean()
```

```
## gender
## Female    13.000000
## Male      11.333333
## Name: grade, dtype: float64
```

*# returns a data frame*

```
df[["gender", "grade"]].groupby("gender").mean()
```

```
##           grade
## gender
## Female  13.000000
## Male    11.333333
```

**Task:** Grouped by gender, find mean, median, minimum, and maximum grade.

dplyr

pandas

```
df %>%
  group_by(gender) %>%
  summarize(mean = mean(grade),
            median = median(grade),
            min = min(grade),
            max = max(grade))
```

```
## # A tibble: 2 x 5
##   gender mean median   min   max
##   <chr>   <dbl>   <dbl> <dbl> <dbl>
## 1 Female    13      13    12    14
## 2 Male     11.3     11    10    13
```

```
df.groupby("gender")["grade"].agg(
  # provide a dictionary
  {# variable name followed by function
    'mean': 'mean',
    'median': 'median',
    'min': 'min',
    'max': 'max'
  }
)
```

```
##           mean median min max
## gender
## Female 13.000000     13  12  14
## Male   11.333333     11  10  13
```

```
##
```

```
## C:/Users/Bhaswar/AppData/Local/r-miniconda/envs/r-reticulate/python.exe:7: FutureWarning:
## is deprecated and will be removed in a future version. Use named aggregation
```

```
##
```

```
##      >>> grouper.agg(name_1=func_1, name_2=func_2)
```

## 1.5 arrange()

**Task:** Arrange grade in ascending order.

dplyr

pandas

```
df %>%
  arrange(grade)
```

```
## # A tibble: 5 x 3
##   name    gender grade
##   <chr>   <chr>   <dbl>
## 1 Barney  Male      10
## 2 Ted     Male      11
## 3 Lilly   Female     12
## 4 Marshall Male      13
## 5 Robin   Female     14
```

```
df.sort_values('grade')
```

```
##      name  gender  grade
## 0  Barney   Male    10
## 1    Ted    Male    11
## 3   Lilly  Female    12
## 2 Marshall   Male    13
## 4   Robin  Female    14
```

**Task:** Arrange grade in ascending order.

dplyr

pandas

```
df %>% arrange(desc(grade))
```

```
## # A tibble: 5 x 3
##   name      gender grade
##   <chr>    <chr>  <dbl>
## 1 Robin   Female    14
## 2 Marshall Male     13
## 3 Lilly   Female    12
## 4 Ted     Male     11
## 5 Barney  Male     10
```

```
df.sort_values('grade', ascending = False)
```

```
##      name  gender  grade
## 4   Robin  Female    14
## 2 Marshall   Male    13
## 3   Lilly  Female    12
## 1    Ted    Male    11
## 0  Barney   Male    10
```

**Task:** Arrange gender in ascending order then arrange grade in descending order.

dplyr

pandas

```
df %>%
  arrange(gender, desc(grade))
```

```
## # A tibble: 5 x 3
##   name      gender grade
##   <chr>    <chr>  <dbl>
## 1 Robin    Female    14
## 2 Lilly    Female    12
## 3 Marshall Male      13
## 4 Ted      Male      11
## 5 Barney   Male      10
```

```
df.sort_values(['gender', 'grade'],
               ascending = [True, False])
```

```
##      name gender grade
## 4   Robin  Female    14
## 3   Lilly  Female    12
## 2 Marshall  Male     13
## 1     Ted   Male     11
## 0  Barney   Male     10
```



## Chapter 2

# Helper Functions

### 2.1 `case_when()` vs `pd.cut()` vs `np.select()`

*Suppose we have a data frame with a variable called `age`. We want to create a variable `age_cat` with the following conditions:*

- $\text{age} < 18$ : “Kids”
- $18 \leq \text{age} < 31$ : “18-30”
- $31 \leq \text{age}$ : “31 and above”

With `dplyr::case_when()` we can do it in the following way:

```
# example data
df <- tibble(age = c(9, 10, 18, 21, 29, 31, 45))
# case_when() in action
df %>%
  mutate(age_cat = case_when(
    age < 18 ~ "Kids",
    age >= 18 & age < 31 ~ "18-30",
    age >= 31 ~ "31 and above"
  ))
```

```
## # A tibble: 7 x 2
##   age age_cat
##   <dbl> <chr>
## 1     9 Kids
## 2    10 Kids
```

```
## 3    18 18-30
## 4    21 18-30
## 5    29 18-30
## 6    31 31 and above
## 7    45 31 and above
```

We can achieve the same result in Python using

- `np.select()`
- `pd.cut()`

```
df = pd.DataFrame({'age': [9, 10, 18, 21, 29, 31, 45]})
```

numpy

pandas

```
# Step 1: Create conditions
conditions = [
    (df['age'].lt(18)),
    (df['age'].ge(18) & (df['age'].lt(31))),
    (df['age'].ge(31))
]
# Step 2: Assign names based on the conditions
cat_names = ['Kids', '18-30', '30 and above']

# Step 3: Use np.select() to generate the variable
df['age_cat'] = np.select(conditions, cat_names)
df
```

```
##    age    age_cat
## 0     9      Kids
## 1    10      Kids
## 2    18    18-30
## 3    21    18-30
## 4    29    18-30
## 5    31  30 and above
## 6    45  30 and above
```

```
# Step 1: Create bin condition
bin_cond = [0, 17, 30, np.inf]
# note: instead of 0,
#      -np.inf will also work
```

```
# Step 2: Assign bin labs
bin_lab = ['Kids', '18-30',
           '30 and above']

# Step 3: Use pd.cut()
df["age_cat"] = pd.cut(df["age"],
                       bins=bin_cond,
                       labels=bin_lab)

df
```

```
##    age    age_cat
## 0    9      Kids
## 1   10      Kids
## 2   18    18-30
## 3   21    18-30
## 4   29    18-30
## 5   31  30 and above
## 6   45  30 and above
```

## 2.2 if\_else() vs np.where()

Given prices of shirts *price*, how do we create a variable *price\_cat* with the following conditions?

- when price is less than 50, we label it as “Cheap”
- when price is 50 or more, we label it as “Expensive”

dplyr

numpy

```
# toy data
prices <- c(25, 30, 45, 80, 100, 125)
df <- tibble(price = prices)
# case_when in action
df %>%
  mutate(price_cat = if_else(
    price < 50, "Cheap", "Expensive"
  ))
```

```
## # A tibble: 6 x 2
##   price price_cat
```

```
##      <dbl> <chr>
## 1      25 Cheap
## 2      30 Cheap
## 3      45 Cheap
## 4      80 Expenseive
## 5     100 Expenseive
## 6     125 Expenseive
```

```
# toy data
prices = {'price': [25, 30, 45, 80, 100, 125]}
df = pd.DataFrame(prices)
# np.where() in action
df['price_cat'] = np.where(
    df.price < 50, "Cheap", "Expenseive"
)
df
```

```
##      price  price_cat
## 0      25      Cheap
## 1      30      Cheap
## 2      45      Cheap
## 3      80  Expenseive
## 4     100  Expenseive
## 5     125  Expenseive
```

## Chapter 3

# tidyr vs pandas

### 3.1 pivot\_longer() vs melt()

Example data: Life expectancy

country	1997	2007
Bangladesh	59.4	64.1
Portugal	76.0	78.1

*How do we make the table long?*

Desired output:

country	year	life_exp
Bangladesh	1997	59.4
Bangladesh	2007	64.1
Portugal	1997	76.0
Portugal	2007	78.1

tidyr::pivot\_longer

pandas::melt()

```
# create df
df <- tibble(
  country = c("Bangladesh", "Portugal"),
  `1997` = c(59.4, 76.0),
  `2007` = c(64.1, 78.1)
```

```

)

# pivot_longer in action
df %>%
  pivot_longer(
    cols = c(`1997`, `2007`),
    names_to = "year",
    values_to = "life_exp"
  )

```

```

## # A tibble: 4 x 3
##   country    year life_exp
##   <chr>      <chr>   <dbl>
## 1 Bangladesh 1997     59.4
## 2 Bangladesh 2007     64.1
## 3 Portugal   1997     76
## 4 Portugal   2007     78.1

```

```

# create df
data = {
  'country': ["Bangladesh", "Portugal"],
  '1997': [59.4, 76.0],
  '2007': [64.1, 78.1]
}
df = pd.DataFrame(data)

# melt() in action
df.melt(
  id_vars = 'country',
  value_vars = ['1997', '2007'],
  value_name = 'life_exp',
  var_name = 'year'
)

```

```

##      country  year  life_exp
## 0  Bangladesh 1997    59.4
## 1   Portugal 1997    76.0
## 2  Bangladesh 2007    64.1
## 3   Portugal 2007    78.1

```

## Chapter 4

# Base Python

### 4.1 map()

map() lets you apply a function to each element of a list.

```
# toy list
toy_list = [1, 200, 3, 400]

# Create toy function
def smaller_than_100(k):
    if k < 100:
        return True
    else:
        False
```

```
# test the function
smaller_than_100(2)
```

```
## True
```

```
# apply it to toy_list
mapped = map(smaller_than_100, toy_list)
print(mapped) # doesn't provide the desired output; use loop
```

```
## <map object at 0x000000003031CEB8>
```

```
for i in mapped:
    print(i)

## True
## None
## True
## None

# Extract mapping into new list
mapped_list = [*map(smaller_than_100, toy_list)]
type(mapped_list)

## <class 'list'>

print(mapped_list)

## [True, None, True, None]

# Use map() with lambda function
[*map(lambda x: x < 100, toy_list)]

## [True, False, True, False]
```

## 4.2 zip()

Use `zip()` to iterables into tuples

- elementwise
- make separate lists into tuples

```
x = [1, 3, 7, 9]
y = [1, 9, 49, 81]
[*zip(x, y)]

## [(1, 1), (3, 9), (7, 49), (9, 81)]

# can operate on more than two inputs
z = [10, 11, 12, 13]
[*zip(x, y, z)]
```



```
## [(1, 1, 10), (3, 9, 11), (7, 49, 12), (9, 81, 13)]
```

```
# zip() will continue upto the length of the shortest input
short_list = [1, 2]
long_list = [16, 7, 8, 9]
[*zip(short_list, long_list)]
```

```
## [(1, 16), (2, 7)]
```

```
# If you want to keep all the items, use itertools.zip_longest()
from itertools import zip_longest
[*zip_longest(short_list, long_list, fillvalue = None)]
```

```
## [(1, 16), (2, 7), (None, 8), (None, 9)]
```

## 4.3 enumerate()

enumerate() returns a sequence of tuples: (index, item).

```
toy_names = ["Robin", "Barney", "Ted", "Lilly", "Marshall"]
enumerate(toy_names) # creates object
```

```
## <enumerate object at 0x000000003031AB40>
```

```
list(enumerate(toy_names)) # get a list of tuples
```

```
## [(0, 'Robin'), (1, 'Barney'), (2, 'Ted'), (3, 'Lilly'), (4, 'Marshall')]
```

```
# Use enumerate() in a for loop
for i, j in enumerate(toy_names):
    print(i, j)
```

```
## 0 Robin
## 1 Barney
## 2 Ted
## 3 Lilly
## 4 Marshall
```

**Example:** Suppose there are duplicates in a given list. You want to create a dictionary with names as keys; index numbers as values.

```
dup_names_list = ["Robin", "Barney", "Robin", "Ted", "Lilly", "Marshall", "Robin", "Ted"]
# create dictionary, keys:names; values: empty
names_dic = {name:[] for name in set(dup_names_list)}
print(names_dic)
```

```
## {'Barney': [], 'Robin': [], 'Marshall': [], 'Lilly': [], 'Ted': []}
```

```
# use enumerate() to store the index for each occurrence
for index, name in enumerate(dup_names_list):
    names_dic[name].append(index)
print(names_dic)
```

```
## {'Barney': [1, 8], 'Robin': [0, 2, 6], 'Marshall': [5], 'Lilly': [4], 'Ted': [3, 7]}
```

# Chapter 5

## scikit-learn

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

```
import sklearn
# check version
sklearn.__version__
```

```
## '0.24.2'
```

### 5.1 Linear Model

```
from sklearn import linear_model
from sklearn.metrics import mean_squared_error, r2_score
```

```
# data
olympic = pd.read_csv("https://raw.githubusercontent.com/sdrogers/fcmlcode/master/R/data/olympics")
```

```
olympic.head()
```

```
##      year  time
## 0  1896   12.0
## 1  1900   11.0
## 2  1904   11.0
## 3  1906   11.2
## 4  1908   10.8
```

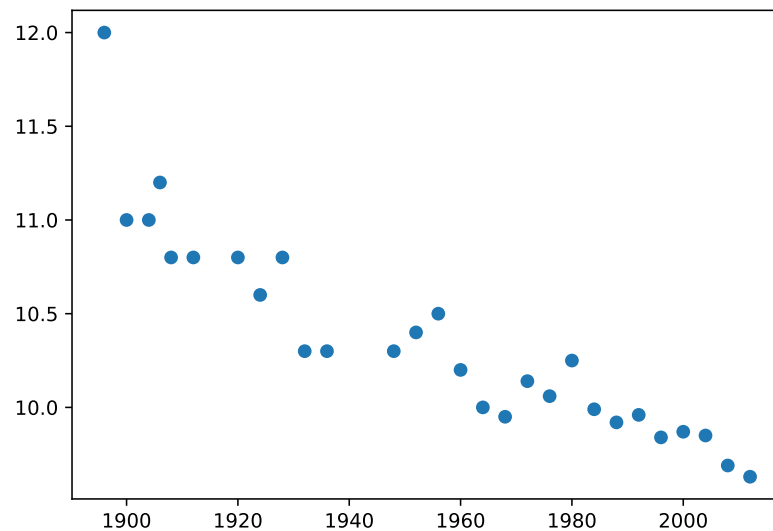
```
olympic.tail()
```

```
##      year  time
## 23  1996  9.84
## 24  2000  9.87
## 25  2004  9.85
## 26  2008  9.69
## 27  2012  9.63
```

```
plt.scatter('year', 'time', data = olympic)
```

```
## <matplotlib.collections.PathCollection object at 0x000000003F036470>
```

```
plt.show()
```



```
# create an instance of a linear regression model where we will estimate the intercept
model = linear_model.LinearRegression(fit_intercept = True)
```

scikit-learn requires that the features (x) be a matrix and the response y be a one-dimension array.

### 5.1.1 Prepare X

```
# Create an X matrix using the x values  
x = olympic.year.values  
x.shape
```

```
## (28,)
```

```
type(x)
```

```
## <class 'numpy.ndarray'>
```

```
X = x.reshape([-1, 1]) # here - 1 means "I don't know how many..."
```

```
# if you know the dimensions  
X = x.reshape((28, 1))
```

```
# Check the shape  
print(X.shape)
```

```
## (28, 1)
```

Alternative? Try the following

```
X2 = olympic[['year']]  
X2.shape
```

```
## (28, 1)
```

### 5.1.2 Prepare y

```
y = olympic.time  
y.shape
```

```
## (28,)
```

```

type(y) # fine! note the difference between year and time; we had to reshape year

## <class 'pandas.core.series.Series'>

```

### 5.1.3 Fit

```

# Now fit the model
model.fit(X, y)

## LinearRegression()

print(model.coef_) # coefficient

## [-0.01327532]

print(model.intercept_) # intercept

## 36.30912040967222

```

### 5.1.4 Prediction

```

# New X as np array
prediction_x = np.linspace(1900, 2000, 101)
# reshape it
prediction_x = prediction_x.reshape([-1, 1]) # recall -1 stands for "i don't know"

model.predict(prediction_x)

## array([11.08600515, 11.07272982, 11.0594545 , 11.04617918, 11.03290385,
##        11.01962853, 11.00635321, 10.99307788, 10.97980256, 10.96652723,
##        10.95325191, 10.93997659, 10.92670126, 10.91342594, 10.90015061,
##        10.88687529, 10.87359997, 10.86032464, 10.84704932, 10.833774 ,
##        10.82049867, 10.80722335, 10.79394802, 10.7806727 , 10.76739738,
##        10.75412205, 10.74084673, 10.72757141, 10.71429608, 10.70102076,
##        10.68774543, 10.67447011, 10.66119479, 10.64791946, 10.63464414,
##        10.62136881, 10.60809349, 10.59481817, 10.58154284, 10.56826752,
##        10.5549922 , 10.54171687, 10.52844155, 10.51516622, 10.5018909 ,
##        10.48861558, 10.47534025, 10.46206493, 10.4487896 , 10.43551428,

```

```
##      10.42223896, 10.40896363, 10.39568831, 10.38241299, 10.36913766,
##      10.35586234, 10.34258701, 10.32931169, 10.31603637, 10.30276104,
##      10.28948572, 10.2762104 , 10.26293507, 10.24965975, 10.23638442,
##      10.2231091 , 10.20983378, 10.19655845, 10.18328313, 10.1700078 ,
##      10.15673248, 10.14345716, 10.13018183, 10.11690651, 10.10363119,
##      10.09035586, 10.07708054, 10.06380521, 10.05052989, 10.03725457,
##      10.02397924, 10.01070392,  9.99742859,  9.98415327,  9.97087795,
##      9.95760262,  9.9443273 ,  9.93105198,  9.91777665,  9.90450133,
##      9.891226 ,  9.87795068,  9.86467536,  9.85140003,  9.83812471,
##      9.82484939,  9.81157406,  9.79829874,  9.78502341,  9.77174809,
##      9.75847277])
```

### 5.1.5 Scatter Plot: Actual vs Fitted

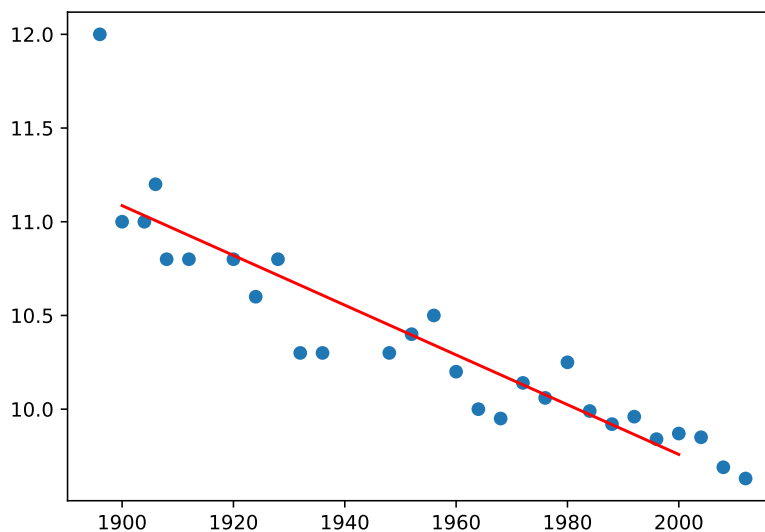
```
plt.scatter(x, y)
```

```
## <matplotlib.collections.PathCollection object at 0x0000000041C79048>
```

```
plt.plot(prediction_x, model.predict(prediction_x), color = 'red')
```

```
## [<matplotlib.lines.Line2D object at 0x0000000041C79358>]
```

```
plt.show()
```



### 5.1.6 Residual Plot

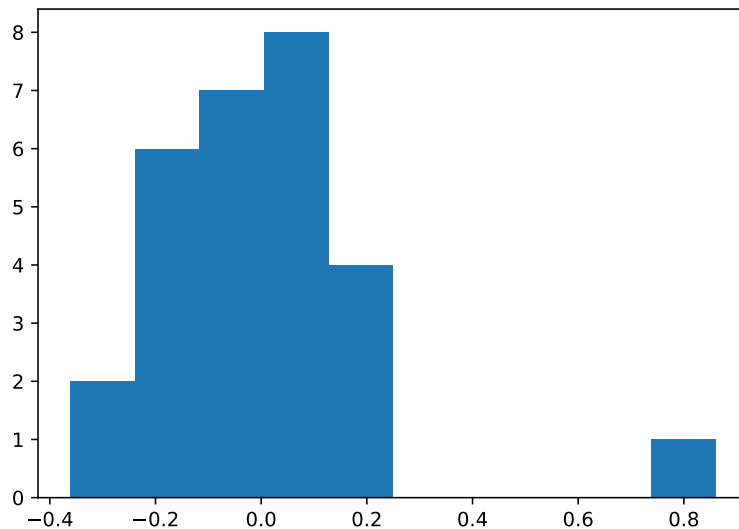
```
# find residuals
residuals = y - model.predict(X)
np.mean(residuals) # check mean
```

```
## 1.9032394707859825e-16
```

```
plt.hist(residuals)
```

```
## (array([2., 6., 7., 8., 4., 0., 0., 0., 1.]), array([-0.36119479, -0.23898595, -0.24984939, 0.37205822, 0.49426705, 0.61647589, 0.73868472, 0.86089356]), <a list of 10 Patch objects>)
```

```
plt.show()
```

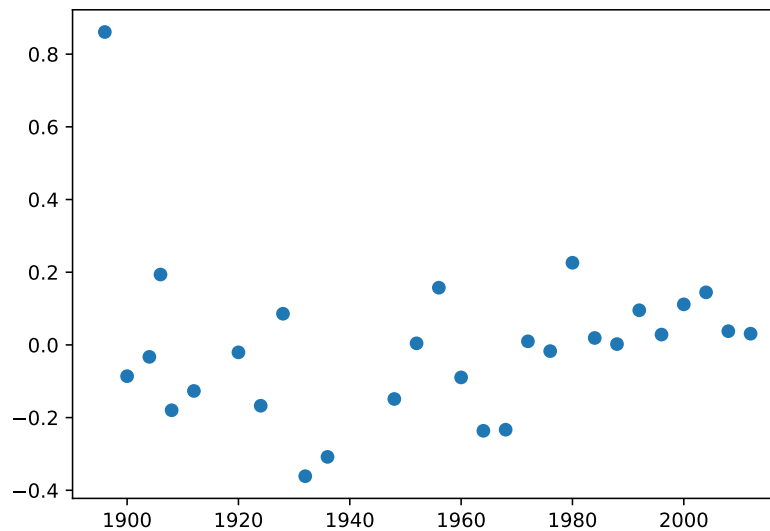


```
plt.plot(x, residuals, "o")
```

```
## [<matplotlib.lines.Line2D object at 0x0000000041D14F60>]
```



```
plt.show()
```



## 5.2 Train-Test

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import linear_model, preprocessing, model_selection
from sklearn.model_selection import train_test_split, cross_val_score
```

`train_test_split()` takes a list of arrays and splits each array into two arrays (a training set and a test set) by randomly selecting rows or values.

### 5.2.1 Example

```
# x is our predictor matrix
X = np.arange(20).reshape((2, -1)).T
print(X)
```

```
## [[ 0 10]
##   [ 1 11]
##   [ 2 12]
##   [ 3 13]
##   [ 4 14]
##   [ 5 15]
##   [ 6 16]
##   [ 7 17]
##   [ 8 18]
##   [ 9 19]]
```

```
# y is a numeric output - for regression methods
y = np.arange(10)
print(y)
```

```
## [0 1 2 3 4 5 6 7 8 9]
```

```
# z is a categorical output - for classification methods
z = np.array([0,0,0,0,0,1,1,1,1,1])
print(z)
```

```
## [0 0 0 0 0 1 1 1 1 1]
```

We can use `train_test_split()` on each array **individually**.

What happens?

```
train_test_split(X, test_size = 1/4, random_state = 1)
```

```
## [array([[ 4, 14],
##        [ 0, 10],
##        [ 3, 13],
##        [ 1, 11],
##        [ 7, 17],
##        [ 8, 18],
##        [ 5, 15]]), array([[ 2, 12],
##        [ 9, 19],
##        [ 6, 16]])]
```

```
type(train_test_split(X, test_size = 1/4, random_state = 1))
```

```
## <class 'list'>
```

Store them

```
X_train, X_test = train_test_split(X, test_size = 1/4, random_state = 1)
print(X_train)
```

```
## [[ 4 14]
##   [ 0 10]
##   [ 3 13]
##   [ 1 11]
##   [ 7 17]
##   [ 8 18]
##   [ 5 15]]
```

```
print(X_test)
```

```
## [[ 2 12]
##   [ 9 19]
##   [ 6 16]]
```

```
y_train, y_test = train_test_split(y, test_size = 1/4, random_state = 1)
print(y_train)
```

```
## [4 0 3 1 7 8 5]
```

```
print(y_test)
```

```
## [2 9 6]
```

We can also apply it to multiple arrays **simultaneously**.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 1/4, random_state = 1)
print(X_train)
```

```
## [[ 4 14]
##   [ 0 10]
##   [ 3 13]
##   [ 1 11]
##   [ 7 17]
##   [ 8 18]
##   [ 5 15]]
```

```
print(X_test)
```

```
## [[ 2 12]
##   [ 9 19]
##   [ 6 16]]
```

```
print(y_train)
```

```
## [4 0 3 1 7 8 5]
```

```
print(y_test)
```

```
## [2 9 6]
```

If you have a **categorical** variable, the **stratify** argument ensures that you'll get an appropriate number of each category in the resulting split. For this purpose, we previously created **z**.

```
X_train, X_test, z_train, z_test = train_test_split(
    X, z, test_size = 1/4, random_state = 1, stratify = z
)
```

```
print(X_train)
```

```
## [[ 4 14]
##   [ 0 10]
##   [ 5 15]
##   [ 7 17]
##   [ 1 11]
##   [ 9 19]
##   [ 2 12]]
```

```
print(X_test)
```

```
## [[ 3 13]
##   [ 8 18]
##   [ 6 16]]
```

```
print(z_train)
```

```
## [0 0 1 1 0 1 0]
```

```
print(z_test)
```

```
## [0 1 1]
```

### 5.2.2 Another Example

```
# Example data: ironslag  
iron = pd.read_csv('https://raw.githubusercontent.com/bhaswar-chakma/toolbox/main/data/ironslag.csv')
```

```
iron.head()
```

```
##      chemical  magnetic  
## 0          24         25  
## 1          16         22  
## 2          24         17  
## 3          18         21  
## 4          18         20
```

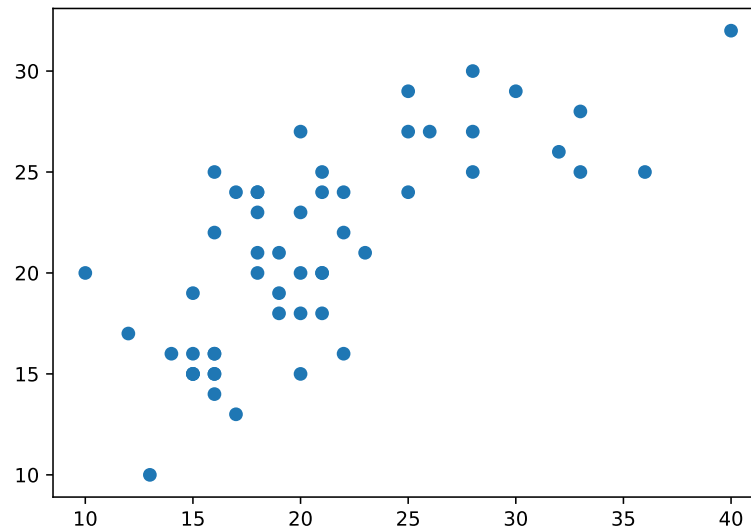
```
iron.shape
```

```
## (53, 2)
```

*Magnetic test is cheaper; chemical test is more accurate. Can we use the magnetic test to predict the chemical test result?*

- $X$  = magnetic test result
- $y$  = chemical test

```
plt.scatter(iron.magnetic, iron.chemical)
```



Create a hold-out set using train-test split

```
train, test = train_test_split(  
    iron, test_size = 1/5, random_state = 1  
)
```

```
train.shape
```

```
## (42, 2)
```

```
train.head()
```

```
##      chemical  magnetic  
## 3          18         21  
## 21         13         17  
## 49         25         36  
## 38         23         18  
## 41         15         16
```

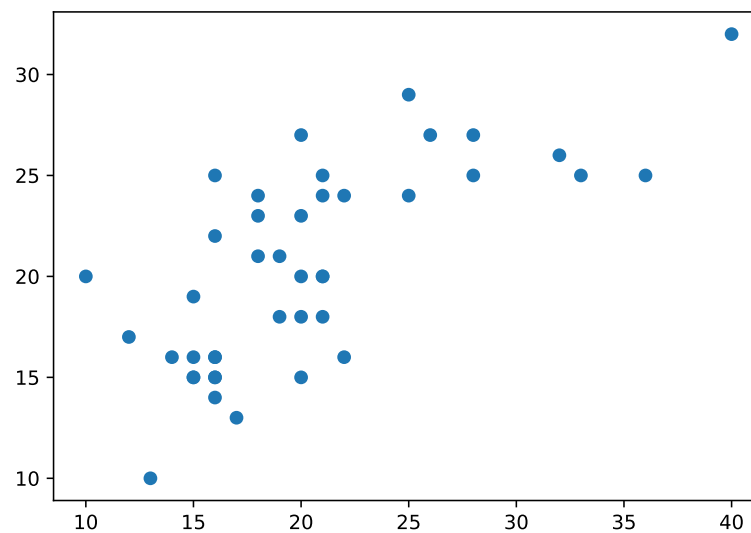
```
test.shape
```

```
## (11, 2)
```

```
test.head()
```

```
##      chemical  magnetic
## 30         27         25
## 2         24         17
## 51         28         33
## 32         20         18
## 31         22         22
```

```
plt.scatter(train.magnetic, train.chemical)
```



Use only the training data to try out possible models

```
# sklearn requires our predictor variables to be in a two dimensional array
# reshape to have 1 column
# the -1 in reshape means I don't want to figure out all the necessary dimensions
# i want 1 column, and numpy, you figure out how many rows I need
X = train.magnetic.values.reshape(-1,1)
X.shape
```

```
## (42, 1)
```

```
y = train.chemical.values
y.shape
```

```
## (42,)
```

```
np.corrcoef(train.magnetic.values, train.chemical.values)
```

```
## array([[1.          , 0.70876994],
##        [0.70876994, 1.          ]])
```

```
# r-squared
```

```
np.corrcoef(train.magnetic.values, train.chemical.values)[0,1] ** 2
```

```
## 0.5023548215592254
```

Fit a linear model between x and y

```
linear = linear_model.LinearRegression()
linear.fit(X, y)
```

```
## LinearRegression()
```

linear.score() is the  $R^2$  value.

```
# linear.score is the R^2 value
# how much error is reduced from no model (variance or MSE)
# vs having the regression model
linear.score(X, y)
```

```
## 0.5023548215592256
```

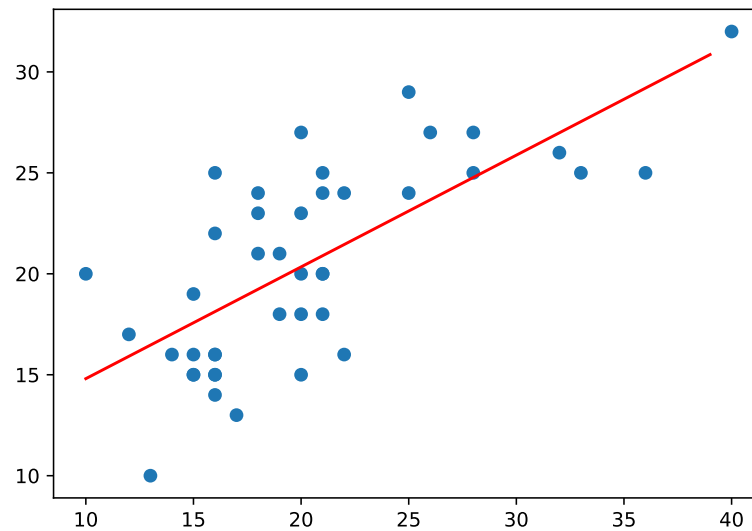
```
x_predict = np.arange(10, 40).reshape(-1,1) # values to be used for prediction
lin_y_hat = linear.predict(x_predict) # use the values and predict
```

```
plt.scatter(X, y)
```

```
## <matplotlib.collections.PathCollection object at 0x0000000004BFF2B0>
```



```
plt.plot(x_predict, lin_y_hat, c = 'red')
```



## 5.3 Cross Validation

### Linear Model

```
# shuffle split says 'shuffle the data' and split it into 5 equal parts
cv = model_selection.ShuffleSplit(n_splits = 5, test_size = 0.3, random_state=0)
cv_linear = model_selection.cross_val_score(linear, X, y, cv = cv)
print(cv_linear)
```

```
## [0.5811901  0.5322723  0.45145614 0.13698027 0.65315849]
```

```
print(np.mean(cv_linear))
```

```
## 0.4710114602819653
```

### Polynomial Fit - Quadratic

```
# preprocessing polynomial features creates a polynomial based on X
quad = preprocessing.PolynomialFeatures(2)
quadX = quad.fit_transform(X)
quad_model = linear_model.LinearRegression()
quad_model.fit(quadX, y)
```

```
## LinearRegression()
```

```
cv_quad = model_selection.cross_val_score(quad_model, quadX, y, cv = cv)
print(cv_quad)
```

```
## [ 0.20489832  0.39310396  0.24068822 -0.11114126  0.58637808]
```

```
print(np.mean(cv_quad))
```

```
## 0.2627854641006778
```

### Cubic Fit

```
cube = preprocessing.PolynomialFeatures(3)
cubeX = cube.fit_transform(X)
cube_model = linear_model.LinearRegression()
cube_model.fit(cubeX, y)
```

```
## LinearRegression()
```

```
cv_cube = model_selection.cross_val_score(cube_model, cubeX, y, cv = cv)
print(cv_cube)
```

```
## [-0.01197637 -0.80626221  0.08937258 -0.2144141  -0.62784165]
```

```
print(np.mean(cv_cube))
```

```
## -0.3142243517318586
```

### $Y \sim \log X$ model

```
log_transform = preprocessing.FunctionTransformer(np.log)
logX = log_transform.fit_transform(X)
logX_model = linear_model.LinearRegression()
logX_model.fit(logX, y)
```

```
## LinearRegression()
```

```
cv_logX = model_selection.cross_val_score(logX_model, logX, y, cv = cv)
print(cv_logX)
```

```
## [ 0.47681194  0.52326867  0.36527782 -0.08925939  0.61651106]
```

```
print(np.mean(cv_logX))
```

```
## 0.3785220199147977
```



## Chapter 6

# R Strings

### 6.1 String Manipulation with Base R Functions

There are many functions in base R for basic string manipulation.

Function

Example

**nchar()**

```
y <- c("Hello", "World", "Hello", "Universe")  
nchar(y) # Returns number of characters
```

```
## [1] 5 5 5 8
```

**tolower()**

```
tolower(y)
```

```
## [1] "hello" "world" "hello" "universe"
```

**toupper()**

```
toupper(y)
```

```
## [1] "HELLO" "WORLD" "HELLO" "UNIVERSE"
```

**chartr()**

```
chartr("oe", "$#", y) #o becomes $; e becomes #
```

```
## [1] "H#ll$"      "W$rld"      "H#ll$"      "Univ#rs#"
```

```
substr()
```

```
x <- "1t345s?"
```

```
substr(x, 2, 6) # provides strings from 2 to 6
```

```
## [1] "t345s"
```

```
strsplit()
```

```
x <- "R#Rocks#!"
```

```
strsplit(x, split = "#")
```

```
## [[1]]
```

```
## [1] "R"      "Rocks" "!"
```

## 6.2 stringr

```
library(stringr)
```

Job	stringr	Base R
String concatenation	<code>str_c()</code>	<code>paste()</code>
Number of characters	<code>str_length()</code>	<code>nchar()</code>
Extracts substrings	<code>str_sub()</code>	<code>substr()</code>
Duplicates characters	<code>str_dup()</code>	
Removes leading and trailing whitespace	<code>str_trim()</code>	
Pads a string	<code>str_pad()</code>	
Wraps a string paragraph	<code>str_wrap()</code>	

## 6.3 Regular Expressions

A **regular expression** (or **regex**) is a set of symbols that describes a text pattern. More formally, a regular expression is a pattern that describes a set of strings.

Regular expressions are a formal language in the sense that the symbols have a defined set of rules to specify the desired patterns.

**6.3.1 stringr Functions for Regular Expressions**

Function	Job
<code>str_detect(str, pattern)</code>	Detects the presence of a pattern and returns TRUE if it is found
<code>str_locate(str, pattern)</code>	Locate the 1st position of a pattern and return a matrix with start & end.s
<code>str_extract(str, pattern)</code>	Extracts text corresponding to the first match.
<code>str_match(str, pattern)</code>	Extracts capture groups formed by () from the first match.
<code>str_split(str, pattern)</code>	Splits string into pieces and returns a list of character vectors.





# Chapter 7

## SQL

### 7.1 CREATE

The general syntax to create a table:

```
create table TABLENAME (  
    COLUMN1 datatype,  
    COLUMN2 datatype,  
    COLUMN3 datatype,  
    ... );
```

To create a table called **TEST** with two columns - **ID** of type integer, and **NAME** of type varchar, we could create it using the following SQL statement:

```
create table TEST(  
    ID int  
    NAME varchar(30)  
);
```

To create a table called **COUNTRY** with an **ID** column, a two letter country code column **CCODE**, and a variable length country name column **NAME**:

```
create table COUNTRY(  
    ID int,  
    CCODE char(2),  
    NAME varchar(60)  
);
```

Sometimes you may see additional keywords in a create table statement:

```
create table COUNTRY(  
    ID int NOT NULL,  
    CCODE char(2),  
    NAME varchar(60),  
    PRIMARY KEY(ID)  
);
```

- In the above example the ID column has the **NOT NULL** constraint added after the datatype - meaning that *it cannot contain a NULL or an empty value*.
- If you look at the last row in the create table statement above you will note that we are using ID as a **Primary Key** and the database **does not allow** Primary Keys to have **NULL** values. *A Primary Key is a unique identifier in a table, and using Primary Keys can help speed up your queries significantly.*
- If the table you are trying to create already exists in the database, you will get an error indicating table XXX.YYY already exists. To circumvent this error, either create a table with a different name or first DROP the existing table. It is quite common to issue a DROP before doing a CREATE in test and development scenarios.

## 7.2 DROP

The general syntax to drop a table:

```
drop table TABLENAME;
```

For example, to drop the table COUNTRY, we can use the following code:

```
drop table COUNTRY;
```

## 7.3 ALTER

```
ALTER TABLE table_name  
ADD COLUMN column_name data_type column_constraint;  
  
ALTER TABLE table_name  
DROP COLUMN column_name;
```

```
ALTER TABLE table_name
ALTER COLUMN column_name SET DATA TYPE data_type;

ALTER TABLE table_name
RENAME COLUMN current_column_name TO new_column_name;
```

## 7.4 TRUNCATE

```
TRUNCATE TABLE table_name;
```

## 7.5 Guided Exercise: Create table and insert data

You will to create two tables

1. PETSALe
2. PET.

```
CREATE TABLE PETSALe (
    ID INTEGER NOT NULL,
    PET CHAR(20),
    SALEPRICE DECIMAL(6,2),
    PROFIT DECIMAL(6,2),
    SALEDATE DATE
);

CREATE TABLE PET (
    ID INTEGER NOT NULL,
    ANIMAL VARCHAR(20),
    QUANTITY INTEGER
);
```

*Now insert some records into the two newly created tables and show all the records of the two tables.*

```
INSERT INTO PETSALe VALUES
(1, 'Cat', 450.09, 100.47, '2018-05-29'),
(2, 'Dog', 666.66, 150.76, '2018-06-01'),
```

```
(3, 'Parrot', 50.00, 8.9, '2018-06-04'),
(4, 'Hamster', 60.60, 12, '2018-06-11'),
(5, 'Goldfish', 48.48, 3.5, '2018-06-14');

INSERT INTO PET VALUES
(1, 'Cat', 3),
(2, 'Dog', 4),
(3, 'Hamster', 2);

SELECT * FROM PETALE;
SELECT * FROM PET;
```

## 7.6 Guided Exercise: Use the ALTER statement to add, delete, or modify columns in two of the existing tables created in the previous exercise.

*Add a new QUANTITY column to the PETALE table and show the altered table.*

```
ALTER TABLE PETALE
ADD COLUMN QUANTITY INTEGER;

SELECT * FROM PETALE;
```

*Now update the newly added QUANTITY column of the PETALE table with some values and show all the records of the table.*

```
UPDATE PETALE SET QUANTITY = 9 WHERE ID = 1;
UPDATE PETALE SET QUANTITY = 3 WHERE ID = 2;
UPDATE PETALE SET QUANTITY = 2 WHERE ID = 3;
UPDATE PETALE SET QUANTITY = 6 WHERE ID = 4;
UPDATE PETALE SET QUANTITY = 24 WHERE ID = 5;

SELECT * FROM PETALE;
```

*Delete the PROFIT column from the PETALE table and show the altered table.*

```
ALTER TABLE PETALE
DROP COLUMN PROFIT;

SELECT * FROM PETALE;
```

*Change the data type to VARCHAR(20) type of the column PET of the table PETSale and show the altered table.*

```
ALTER TABLE PETSale
ALTER COLUMN PET SET DATA TYPE VARCHAR(20);

SELECT * FROM PETSale;
```

If you are using IBM db2: Now verify if the data type of the column PET of the table PETSale changed to VARCHAR(20) type or not. Click on the 3 bar menu icon in the top left corner and click Explore > Tables. Find the PETSale table from Schemas by clicking Select All. Click on the PETSale table to open the Table Definition page of the table. Here, you can see all the current data type of the columns of the PETSale table.

*Rename the column PET to ANIMAL of the PETSale table and show the altered table.*

```
ALTER TABLE PETSale
RENAME COLUMN PET TO ANIMAL;

SELECT * FROM PETSale;
```

## 7.7 Guided Exercise: TRUNCATE

In this exercise, you will use the TRUNCATE statement to remove all rows from an existing table created in exercise 1 without deleting the table itself.

*Remove all rows from the PET table and show the empty table.*

```
TRUNCATE TABLE PET IMMEDIATE;

SELECT * FROM PET;
```

## 7.8 Guided Exercise: DROP

In this exercise, you will use the DROP statement to delete an existing table created in the previous exercise.

*Delete the PET table and verify if the table still exists or not (SELECT statement won't work if a table doesn't exist).*

```
DROP TABLE PET;

SELECT * FROM PET;
```

## 7.9 Exercise: String Patterns

In this exercise, you will go through some SQL problems on String Patterns.

Here is EMPLOYEES table.

EMP_ID	F_NAME	L_NAME	SSN	B_DATE
E1001	John	Thomas	123456	1976-01-09
E1002	Alice	James	123457	1972-07-31
E1003	Steve	Wells	123458	1980-08-10
E1004	Santosh	Kumar	123459	1985-07-20
E1005	Ahmed	Hussain	123410	1981-01-04
E1006	Nancy	Allen	123411	1978-02-06
E1007	Mary	Thomas	123412	1975-05-05
E1008	Bharath	Gupta	123413	1985-05-06
E1009	Andrea	Jones	123414	1990-07-09
E1010	Ann	Jacob	123415	1982-03-30

SEX	ADDRESS	JOB_ID	SALARY	MANAGER_ID	DEP_ID
M	5631 Rice, OakPark,IL	100	100000	30001	2
F	980 Berry Ln, Elgin,IL	200	80000	30002	5
M	291 Springs, Gary,IL	300	50000	30002	5
M	511 Aurora Av, Aurora,IL	400	60000	30004	5
M	216 Oak Tree, Geneva,IL	500	70000	30001	2
F	111 Green Pl, Elgin,IL	600	90000	30001	2
F	100 Rose Pl, Gary,IL	650	65000	30003	7
M	145 Berry Ln, Naperville,IL	660	65000	30003	7
F	120 Fall Creek, Gary,IL	234	70000	30003	7
F	111 Britany Springs,Elgin,IL	220	70000	30004	5

### 7.9.1 Retrieve all employees whose address is in Elgin,IL.

[Click here for the solution](#)

```
SELECT F_NAME , L_NAME
FROM EMPLOYEES
WHERE ADDRESS LIKE '%Elgin,IL%';
```

### 7.9.2 Retrieve all employees who were born during the 1970's..

[Click here for the solution](#)

```
SELECT F_NAME , L_NAME  
FROM EMPLOYEES  
WHERE B_DATE LIKE '197%';
```

**7.9.3** *Retrieve all employees in department 5 whose salary is between 60000 and 70000..*

[Click here for the solution](#)

```
SELECT F_NAME , L_NAME  
FROM EMPLOYEES  
WHERE DEP_ID = 5 and (SALARY BETWEEN 60000 AND 70000);  
--Notice the "=" and "and"
```

## 7.10 Exercise: Sorting

**7.10.1** *Retrieve a list of employees ordered by department ID..*

[Click here for the solution](#)

```
SELECT F_NAME, L_NAME, DEP_ID  
FROM EMPLOYEES  
ORDER BY DEP_ID;
```

**7.10.2** *Retrieve a list of employees ordered in descending order by department ID and within each department ordered alphabetically in descending order by last name..*

[Click here for the solution](#)

```
SELECT F_NAME, L_NAME, DEP_ID  
FROM EMPLOYEES  
ORDER BY DEP_ID DESC, L_NAME DESC;
```

**7.10.3** *In the previous problem, use department name instead of department ID. Retrieve a list of employees ordered by department name, and within each department ordered alphabetically in descending order by last name..*

Here is the DEPARTMENTS table.

DEPT_ID	DEP_NAME	MANAGER_ID	LOC_ID
2	Architect Group	30001	L0001
5	Software Group	30002	L0002
7	Design Team	30003	L0003

[Click here for the solution](#)

```
SELECT D.DEP_NAME , E.F_NAME, E.L_NAME
FROM EMPLOYEES as E, DEPARTMENTS as D
WHERE E.DEP_ID = D.DEPT_ID_DEP
ORDER BY D.DEP_NAME, E.L_NAME DESC;
```

In the SQL Query above, D and E are aliases for the table names. Once you define an alias like D in your query, you can simply write D.COLUMN\_NAME rather than the full form DEPARTMENTS.COLUMN\_NAME.

## 7.11 Exercise 3: Grouping

**7.11.1** *For each department ID retrieve the number of employees in the department..*

[Click here for the solution](#)

```
SELECT DEP_ID, COUNT(*)
FROM EMPLOYEES
GROUP BY DEP_ID;
```

**7.11.2** *For each department retrieve the number of employees in the department, and the average employee salary in the department..*

[Click here for the solution](#)



```
SELECT DEP_ID, COUNT(*), AVG(SALARY)
FROM EMPLOYEES
GROUP BY DEP_ID;
```

**7.11.3** *Label the computed columns in the result set of the last SQL problem as NUM\_EMPLOYEES and AVG\_SALARY..*

[Click here for the solution](#)

```
SELECT DEP_ID, COUNT(*) AS "NUM_EMPLOYEES", AVG(SALARY) AS "AVG_SALARY"
FROM EMPLOYEES
GROUP BY DEP_ID;
```

**7.11.4** *In the previous SQL problem , order the result set by Average Salary..*

[Click here for the solution](#)

```
SELECT DEP_ID, COUNT(*) AS "NUM_EMPLOYEES", AVG(SALARY) AS "AVG_SALARY"
FROM EMPLOYEES
GROUP BY DEP_ID
ORDER BY AVG_SALARY;
```

**7.11.5** *In SQL problem 4 (Exercise 3 Problem 4), limit the result to departments with fewer than 4 employees..*

[Click here for the solution](#)

```
SELECT DEP_ID, COUNT(*) AS "NUM_EMPLOYEES", AVG(SALARY) AS "AVG_SALARY"
FROM EMPLOYEES
GROUP BY DEP_ID
HAVING count(*) < 4
ORDER BY AVG_SALARY;
```