Toolbox

Bhaswar Chakma

2021-09-23

Contents

Chapter 1

Grammar: 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()

1.1.1 *Example: 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()`

1.2.1 *Example: 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()

1.3.1 *Example: Keep if the student is Barney or female.*

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()

1.4.1 *Example: 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
```

Option 1:

```
# returns a series
df.groupby("gender")['grade'].mean()
```

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

Option 2:

```
# returns a data frame
df[['gender', 'grade']].groupby("gender").mean()
```

```
##             grade
```

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

Option 3:

```
# useful for multiple groups and stat
df.groupby(['gender']).agg(
    {'grade': ['mean']}
    # Here key: variable name; value: stat function
)
# here [] is unnecessary but
# required for multiple groups and stats
```

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

1.4.2 *Example: 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"]).agg(
  # provide a dictionary
  #   key: variable name
  #   value: stat function
   {'grade':['mean',
             'median',
             'min',
             'max']}
```

```
)
```

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

1.5 arrange()

1.5.1 *Example: 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
```

1.5.2 *Example: Arrange grade in descending 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
```

1.5.3 *Example: 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() and 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:

- `age` < 18: "Kids"
- $18 \leq$ `age` < 31: "18-30"
- $31 \leq$ `age`: "31 and above"

Example:

age	age_cat
9	Kids
10	Kids
18	18-30
21	18-30
29	18-30
31	31 and above
45	31 and above

dplyr::case_when()

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]})
```

np.select()

pd.cut()

```
# Step 1: Create conditions
cond = [
  (df['age'].lt(18)),
  (df['age'].ge(18) & (df['age'].lt(31))),
  (df['age'].ge(31))
]

# Step 2: Assign labels
cond_labs = [
  'Kids', '18-30', '30 and above'
]

# Step 3: apply np.select()
df['age_cat'] = np.select(cond, cond_labs)
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
#
# 0: greater than 0
# 17: upper limit is 17

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

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

np.where()

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

```
## # 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
```

```python
# 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
```

2.3 %in% vs isin, @, and in

code	capital
BD	Dhaka
PT	Lisbon
ES	Madrid

code	capital
FR	Paris

How to keep observations that belong to BD or DE (without using the | operator)?

R

Python

```
# toy data
df <- tibble(
  # country code
  code = c(
    "BD", "PT",
    "ES", "FR"
  ),
  # capital
  capital = c(
    "Dhaka", "Lisbon",
    "Madrid", "Paris"
  )
)
df
```

```
## # A tibble: 4 x 2
##   code  capital
##   <chr> <chr>
## 1 BD    Dhaka
## 2 PT    Lisbon
## 3 ES    Madrid
## 4 FR    Paris
```

```
# %in% in action
df %>%
  filter(code %in% c("BD", "PT"))
```

```
## # A tibble: 2 x 2
##   code  capital
##   <chr> <chr>
## 1 BD    Dhaka
## 2 PT    Lisbon
```

```
# toy data
data = {
    # country code
    'code':[
```

```
        "BD", "PT",
        "ES", "FR"
    ],
    # capital
    'capital':[
        "Dhaka", "Lisbon",
        "Madrid", "Paris"
    ]
}
df = pd.DataFrame(data)
df
```

```
##   code capital
## 0   BD   Dhaka
## 1   PT  Lisbon
## 2   ES  Madrid
## 3   FR   Paris
```

isin

```
# isin in action
df[df["code"].isin(["BD", "PT"])]
```

```
##   code capital
## 0   BD   Dhaka
## 1   PT  Lisbon
```

@

```
country_list = ["BD", "PT"]
# @ in action
df.query('code == @country_list')
# note: you must create a list first
#   @["BD", "PT"] doesn't work
#   but, @list(["BD", "PT"] works
```

```
##   code capital
## 0   BD   Dhaka
## 1   PT  Lisbon
```

in

```
# in in action
df.query('code in ["BD", "PT"]')
```

```
##   code capital
## 0   BD   Dhaka
## 1   PT  Lisbon
```

2.4 stringr::str_detect() vs str.contains()

Example:

info	amount
XYZ Deposit 2020	0
Cash Deposit	1
ATM	2
XYZ Fee 2021	3
XYZ Deposit 2021	4

How to keep or drop only those observations where `info` is about XYZ?

R `stringr::str_detect()`

Python `str.contains()`

```
# toy data
df <- tibble(
  info = c(
    "XYZ Deposit 2020",
    "Cash Deposit",
    "ATM",
    "XYZ Fee 2021",
    "XYZ Deposit 2021"
  ),
  amount = seq(1,5) - 1
)

df
```

```
## # A tibble: 5 x 2
##   info             amount
##   <chr>             <dbl>
## 1 XYZ Deposit 2020      0
## 2 Cash Deposit          1
## 3 ATM                   2
## 4 XYZ Fee 2021          3
## 5 XYZ Deposit 2021      4
```

Keep:

```
# str_detect() in action
df %>%
  filter( # keeps
    stringr::str_detect(
      info, "XYZ"
```

```
    )
  )
```

```
## # A tibble: 3 x 2
##   info             amount
##   <chr>             <dbl>
## 1 XYZ Deposit 2020      0
## 2 XYZ Fee 2021          3
## 3 XYZ Deposit 2021      4
```

Drop:

```
# str_detect() in action
df %>%
  filter( # drops
    ! stringr::str_detect(
      info, "XYZ"
    )
  )
```

```
## # A tibble: 2 x 2
##   info         amount
##   <chr>         <dbl>
## 1 Cash Deposit      1
## 2 ATM               2
```

```
# toy data
df = pd.DataFrame({
    'info':
    [
    "XYZ Deposit 2020",
    "Cash Deposit",
    "ATM",
    "XYZ Fee 2021",
    "XYZ Deposit 2021"
],
    'amount': np.arange(5)
})
df
```

```
##                 info  amount
## 0  XYZ Deposit 2020       0
## 1      Cash Deposit       1
## 2               ATM       2
## 3      XYZ Fee 2021       3
## 4  XYZ Deposit 2021       4
```

Keep:

```
# str.contains in action: keep
df[df['info'].str.contains("XYZ")]
```

```
##                 info  amount
## 0  XYZ Deposit 2020       0
## 3      XYZ Fee 2021       3
## 4  XYZ Deposit 2021       4
```

Drop:

```
# str.contains in action: drop
df[~ df['info'].str.contains("XYZ")]
```

```
##             info  amount
## 1  Cash Deposit       1
## 2           ATM       2
```

2.5 distinct() vs unique()

name	year
A	2020
B	2020
C	2020
A	2021
B	2021
D	2021

What are the distinct names?

dplyr::distinct()

pd.unique()

```
# toy data
df <- tibble(
    name = c(
        'A', 'B', 'C',
        'A', 'B', 'D'
    ),
    year = c(
        2020, 2020, 2020,
        2021, 2021, 2021
    )
)
```

```r
# dplyr::distinct in action
df %>% distinct(name)
```

```
## # A tibble: 4 x 1
##   name
##   <chr>
## 1 A
## 2 B
## 3 C
## 4 D
```

```r
# if you want to count
df %>% distinct(name) %>% count()
```

```
## # A tibble: 1 x 1
##       n
##   <int>
## 1     4
```

```python
# toy data
df = pd.DataFrame({
    'name':[
        'A', 'B', 'C',
        'A', 'B', 'D'
    ],
    'year':[
        2020, 2020, 2020,
        2021, 2021, 2021
    ]
})
```

```python
# pd.unique() in action
df['name'].unique()
```

```
## array(['A', 'B', 'C', 'D'], dtype=object)
```

```python
# if you want to count
len(df['name'].unique())
```

```
## 4
```

2.6 slice() vs iloc()

age	id
20	1
21	1
22	1

age	id
23	1
24	1

How to slice single or multiple rows?

dplyr::slice()

pd.iloc()

```
# toy data
df <- tibble(
  age = seq(20,24),
  id = seq(1,5)
)
df
```

```
## # A tibble: 5 x 2
##     age    id
##   <int> <int>
## 1    20     1
## 2    21     2
## 3    22     3
## 4    23     4
## 5    24     5
```

```
# slice row 1
df %>% slice(1)
```

```
## # A tibble: 1 x 2
##     age    id
##   <int> <int>
## 1    20     1
```

```
# slice rows 3 and 4
df %>% slice(3:4)
```

```
## # A tibble: 2 x 2
##     age    id
##   <int> <int>
## 1    22     3
## 2    23     4
```

```
# toy data
df = pd.DataFrame({
    'age' : np.arange(20, 25),
    'id': np.arange(1, 6)
})
```

```
df
```

```
##    age  id
## 0   20   1
## 1   21   2
## 2   22   3
## 3   23   4
## 4   24   5
```

```
# slice row 1
df.iloc[0:1]
```

```
##    age  id
## 0   20   1
```

```
# slice row 3 and 4
df.iloc[2:4]
```

```
##    age  id
## 2   22   3
## 3   23   4
```

Chapter 3

Join and Bind

3.1 Join Data Frames: `*_join` vs `merge()`

Toy Data

`df1`:

id	first_name
hiRS	Robin
hiTM	Ted
bbP	Penny
bbSC	Sheldon

`df2`:

id	last_name
hiRS	Robin
hiTM	Ted
bbSC	Cooper
bbLH	Hofstadter

R

Python

```
# toy data
df1 = tibble(
    'id'= c("hiRS", "hiTM",
            "bbP", "bbSC"),
```

```r
    'first_name'= c(
        'Robin',
        'Ted',
        'Penny',
        'Sheldon'
    )
)



df2 = tibble(
    'id'= c("hiRS", "hiTM",
            "bbSC", "bbLH"),
    'last_name'= c(
        'Robin',
        'Ted',
        'Cooper',
        'Hofstadter'
    )
)
```

```python
# toy data
df1 = pd.DataFrame({
    'id': ["hiRS", "hiTM",
           "bbP", "bbSC"],
    'first_name':[
        'Robin',
        'Ted',
        'Penny',
        'Sheldon'
    ]
})



df2 = pd.DataFrame({
    'id': ["hiRS", "hiTM",
           "bbSC", "bbLH"],
    'last_name':[
        'Scherbatsky',
        'Mosby',
        'Cooper',
        'Hofstadter'
    ]
})
```

3.1.1 Inner Join

dplyr::inner_join()

pandas::merge(how = "inner")

```
# inner_join in action
# note: by argument is not needed here
df1 %>% inner_join(df2, by = "id")
```

```
## # A tibble: 3 x 3
##   id    first_name last_name
##   <chr> <chr>      <chr>
## 1 hiRS  Robin      Robin
## 2 hiTM  Ted        Ted
## 3 bbSC  Sheldon    Cooper
```

```
# merge(how = "inner") in action
df1.merge(df2, how = 'inner', on = 'id')
# note: default is inner join
# i.e. df1.merge(df2) does inner join
```

```
##      id first_name    last_name
## 0  hiRS      Robin  Scherbatsky
## 1  hiTM        Ted        Mosby
## 2  bbSC    Sheldon       Cooper
```

3.1.2 Left Join

dplyr::inner_join()

pandas::merge(how = "left")

```
# left_join in action
# note: by argument is not needed here
# needed when you have more keys
df1 %>% left_join(df2, by = "id")
```

```
## # A tibble: 4 x 3
##   id    first_name last_name
##   <chr> <chr>      <chr>
## 1 hiRS  Robin      Robin
## 2 hiTM  Ted        Ted
## 3 bbP   Penny      <NA>
## 4 bbSC  Sheldon    Cooper
```

```
# merge(how = "left") in action
df1.merge(df2, how = 'left', on = 'id')
```

```
##      id first_name    last_name
```

```
## 0  hiRS      Robin  Scherbatsky
## 1  hiTM        Ted        Mosby
## 2   bbP      Penny          NaN
## 3  bbSC    Sheldon       Cooper
```

3.1.3 Right Join

dplyr::right_join()

pandas::merge(how = "right")

```
# left_join in action
# note: by argument is not needed here
# needed when you have more keys
df1 %>% right_join(df2, by = "id")
```

```
## # A tibble: 4 x 3
##   id    first_name last_name
##   <chr> <chr>      <chr>
## 1 hiRS  Robin      Robin
## 2 hiTM  Ted        Ted
## 3 bbSC  Sheldon    Cooper
## 4 bbLH  <NA>       Hofstadter
```

```
# merge(how = "right") in action
df1.merge(df2, how = 'right', on = 'id')
```

```
##      id first_name    last_name
## 0  hiRS      Robin  Scherbatsky
## 1  hiTM        Ted        Mosby
## 2  bbSC    Sheldon       Cooper
## 3  bbLH        NaN   Hofstadter
```

3.1.4 Full Join

dplyr::full_join()

pandas::merge(how = "outer")

```
# full_join in action
# note: by argument is not needed here
# needed when you have more keys
df1 %>% full_join(df2, by = "id")
```

```
## # A tibble: 5 x 3
##   id    first_name last_name
##   <chr> <chr>      <chr>
## 1 hiRS  Robin      Robin
## 2 hiTM  Ted        Ted
```

```
## 3 bbP   Penny      <NA>
## 4 bbSC  Sheldon    Cooper
## 5 bbLH  <NA>       Hofstadter
```

```
# merge(how = "outer") in action
df1.merge(df2, how = 'outer', on = 'id')
```

```
##       id first_name    last_name
## 0   hiRS      Robin  Scherbatsky
## 1   hiTM        Ted        Mosby
## 2    bbP      Penny          NaN
## 3   bbSC    Sheldon       Cooper
## 4   bbLH        NaN   Hofstadter
```

3.2 Bind Rows: `bind_rows()` vs `append()`

df1:

first_name	last_name
Steve	Vai
Joe	Satriani

df2:

first_name	last_name
Paul	Gilbert
Eric	Johnson

How do we combine the rows of `df1` and `df2`?

dplyr::bind_rows()

pandas::append()

```
# toy data
df1 <- tibble(
    first_name = c('Steve', 'Joe'),
    last_name = c('Vai', 'Satriani')
)

df2 <- tibble(
    first_name = c('Paul', 'Eric'),
    last_name = c('Gilbert', 'Johnson')
)
```

```
# bind_rows() in action
bind_rows(df1, df2)
```

```
## # A tibble: 4 x 2
##   first_name last_name
##   <chr>      <chr>
## 1 Steve      Vai
## 2 Joe        Satriani
## 3 Paul       Gilbert
## 4 Eric       Johnson
```

```
# toy data
df1 = pd.DataFrame({
    'first_name': ['Steve', 'Joe'],
    'last_name': ['Vai', 'Satriani']
})

df2 = pd.DataFrame({
    'first_name': ['Paul', 'Eric'],
    'last_name': ['Gilbert', 'Johnson']
})
```

```
# append in action: ignoring the index
df1.append(df2, ignore_index = True)
```

```
##   first_name last_name
## 0      Steve       Vai
## 1        Joe  Satriani
## 2       Paul   Gilbert
## 3       Eric   Johnson
```

```
# append in action: maintaining the index
df1.append(df2)
```

```
##   first_name last_name
## 0      Steve       Vai
## 1        Joe  Satriani
## 0       Paul   Gilbert
## 1       Eric   Johnson
```

Chapter 4

Reshape: tidyr vs pandas

4.1 `pivot_longer()` vs `melt()`

4.1.1 *Example: Life Expectancy data in "wide" format*

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()

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

```
  )
df
```

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

pivot_longer() in action!

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

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

```
##       country  1997  2007
## 0  Bangladesh  59.4  64.1
## 1    Portugal  76.0  78.1
```

melt() in action!

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

4.2 pivot_wider vs pivot

4.2.1 *Example: Life Expectancy data in "long" format*

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

How do we make the table "wide"?

Desired output:

country	1997	2007
Bangladesh	59.4	64.1
Portugal	76.0	78.1

tidyr::pivot_wider()

pandas::pivot()

```r
# toy data

country <- c(
  "Bangladesh", "Bangladesh",
  "Portugal", "Portugal"
)

year <- c(
  "1997", "2007",
  "1997", "2007"
)

life_exp <- c(
  59.4, 64.1,
  76, 78.1
)

df <- tibble(country, year, life_exp)
df
```

```
## # 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
```

pivot_wider() in action!

```
# pivot_wider in action
df %>%
  pivot_wider(
    names_from = "year",
    values_from = "life_exp"
  )
```

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

```
# toy data

country =[
    "Bangladesh", "Bangladesh",
    "Portugal", "Portugal"
]

year = [
    "1997", "2007",
    "1997", "2007"
]

life_exp = [
    59.4, 64.1,
    76, 78.1
]

df = pd.DataFrame(
   {'country': country,
    'year': year,
    'life_exp': life_exp}
)
df
```

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

pivot() in action!

```
# pivot in action
df_wide = df.pivot(
    index = 'country',
    columns = 'year', # names from
    values = 'life_exp' # vasles from
)
df_wide
```

```
## year        1997  2007
## country
## Bangladesh  59.4  64.1
## Portugal    76.0  78.1
```

```
# Reset the names
df_wide.index.name = None
df_wide.columns.name = None
```

```
df_wide
```

```
##             1997  2007
## Bangladesh  59.4  64.1
## Portugal    76.0  78.1
```

4.3 pandas:: stack()

```
# toy data
df = pd.DataFrame({
    'year': np.arange(2020,2025),
    'Fall': np.linspace(10,15,5),
    'Spring': np.linspace(1, 5,5)
})
df
```

```
##    year   Fall  Spring
## 0  2020  10.00     1.0
## 1  2021  11.25     2.0
## 2  2022  12.50     3.0
## 3  2023  13.75     4.0
## 4  2024  15.00     5.0
```

How to create MultiIndex series?

```
# step 1: set year as index
df.set_index('year', inplace = True)
df
```

```
##        Fall  Spring
```

```
## year
## 2020  10.00     1.0
## 2021  11.25     2.0
## 2022  12.50     3.0
## 2023  13.75     4.0
## 2024  15.00     5.0
```

```
# step 2: apply stack()
df_stacked = df.stack()
df_stacked
```

```
## year
## 2020  Fall      10.00
##       Spring     1.00
## 2021  Fall      11.25
##       Spring     2.00
## 2022  Fall      12.50
##       Spring     3.00
## 2023  Fall      13.75
##       Spring     4.00
## 2024  Fall      15.00
##       Spring     5.00
## dtype: float64
```

```
# check type
type(df_stacked)
```

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

```
# check index
df_stacked.index
```

```
## MultiIndex([(2020,   'Fall'),
##             (2020, 'Spring'),
##             (2021,   'Fall'),
##             (2021, 'Spring'),
##             (2022,   'Fall'),
##             (2022, 'Spring'),
##             (2023,   'Fall'),
##             (2023, 'Spring'),
##             (2024,   'Fall'),
##             (2024, 'Spring')],
##            names=['year', None])
```

4.4 pandas:: unstack()

```
# toy data
year = [2010, 2010, 2010, 2020, 2020, 2020]
name = ["X", "Y", "Z", "X", "Y", "Z"]
gender = ["M", "F", "F", "M", "F", "F"]
grade = [10, 10, 20, 20, 12.5, 17.5]
df = pd.DataFrame(
    {
    'year': year,
    'name': name,
    'gender': gender,
    'grade': grade
    }
)
df
```

```
##    year name gender  grade
## 0  2010    X      M   10.0
## 1  2010    Y      F   10.0
## 2  2010    Z      F   20.0
## 3  2020    X      M   20.0
## 4  2020    Y      F   12.5
## 5  2020    Z      F   17.5
```

Suppose we want to find mean `grade` grouped by `year` and `gender`.

```
# grouped by year and gender:
# find mean grade
df_stat = df.groupby(
    ['year', 'gender']
).agg({
    'grade': ['mean']
})
df_stat
```

```
##              grade
##               mean
## year gender
## 2010 F        15.0
##      M        10.0
## 2020 F        15.0
##      M        20.0
```

How to get `F` and `M` as columns?

- Just apply `unstack()`

```
df_unstacked = df_stat.unstack()
df_unstacked
```

```
##        grade
##         mean
## gender     F     M
## year
## 2010    15.0  10.0
## 2020    15.0  20.0
```

```
# To change the names

# reset index
df_unstacked2 = df_unstacked.reset_index()
# rename
df_unstacked2.columns = ['year', 'M', 'F']
df_unstacked2
```

```
##    year     M     F
## 0  2010  15.0  10.0
## 1  2020  15.0  20.0
```

Chapter 5

Missing Values and Duplicates

5.1 Missing Values

5.1.1 `tidyr::drop_na()` vs `pd.dropna()` and `pd.notpna()`

name	age
A	21
B	NA
NA	NA
D	40

How to drop missing rows or columns?

tidyr::drop_na()

pd.dropna() and pd.notpna()

```
# toy data
df <- tibble(
  name = c('A', 'B',
           NA, 'D'),
  age  =  c(21, NA,
           NA, 40)
)
df
```

```
## # A tibble: 4 x 2
##   name    age
```

```
##   <chr> <dbl>
## 1 A        21
## 2 B        NA
## 3 <NA>     NA
## 4 D        40
```

```
# drop missing rows
df %>% drop_na()
```

```
## # A tibble: 2 x 2
##   name    age
##   <chr> <dbl>
## 1 A        21
## 2 D        40
```

```
# drop specific variable
# with NaN
df %>% drop_na(name)
```

```
## # A tibble: 3 x 2
##   name    age
##   <chr> <dbl>
## 1 A        21
## 2 B        NA
## 3 D        40
```

```
# toy data
# toy data
df = pd.DataFrame({
    'name': [
        'A', 'B', np.nan, 'D'
    ],
    'age': [
        21, np.nan, np.nan, 40
    ]
})
df
```

```
##   name   age
## 0    A  21.0
## 1    B   NaN
## 2  NaN   NaN
## 3    D  40.0
```

```
# drop missing rows
df.dropna()
```

```
##   name   age
## 0    A  21.0
```

```
## 3    D  40.0
```

```
# drop specific variable
# with NaN
df[df['name'].notna()]
```

```
##   name   age
## 0    A  21.0
## 1    B   NaN
## 3    D  40.0
```

Chapter 6

Base Python

6.1 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 0x0000000030945E80>
```

```
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]
```

6.2 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)]
```

6.3 enumerate()

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

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

```
## <enumerate object at 0x000000003094F630>
```

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

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

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

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

Chapter 7

Graphics

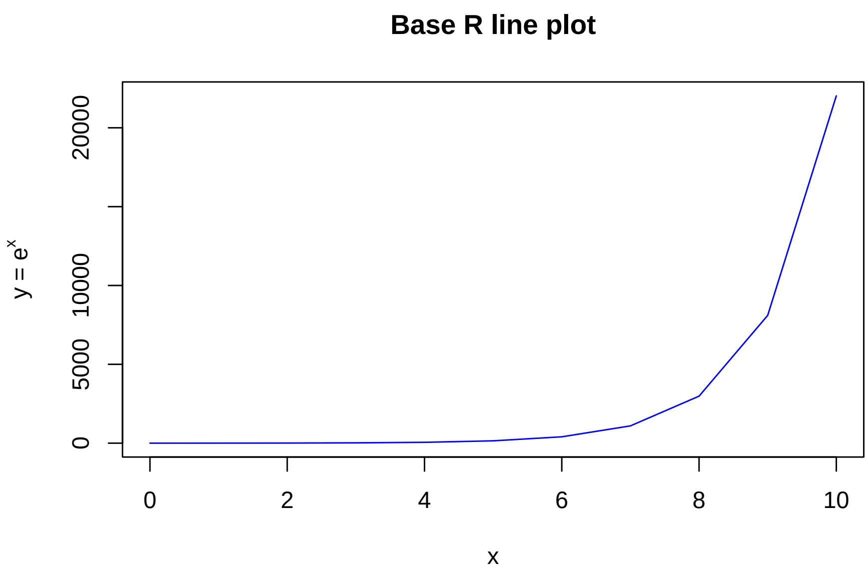
7.1 Base R vs Matplotlib

plot()

plt.plot()

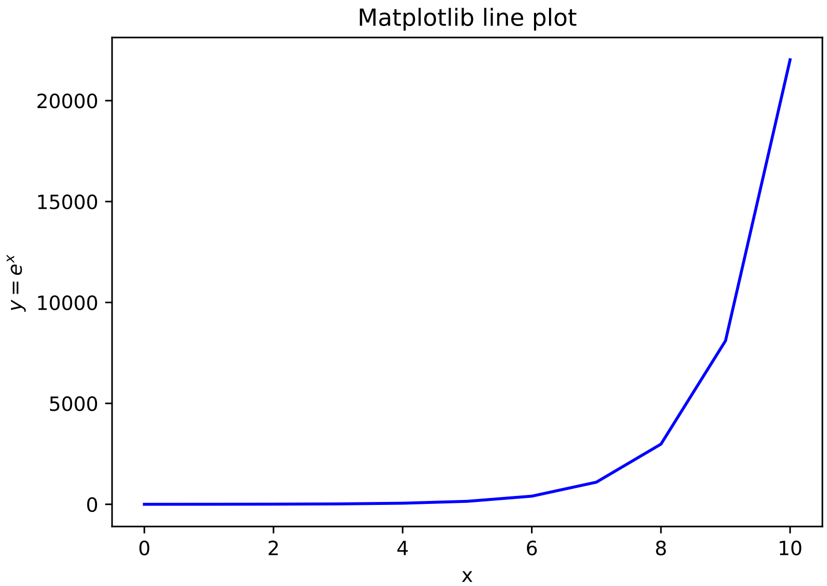
```
# toy data
x <- seq(0, 10)
y <- exp(x)
```

```
# default "type" is "p"
plot(
  x, y,
  type = "l",
  col = "blue",
  # title
  main = "Base R line plot",
  xlab = "x",
  ylab = expression("y = e"^"x")
  # latex2exp package offers more
)
```




```
# toy data
x = np.arange(11)
y = np.exp(np.arange(11))
```

```
# default 'kind' is 'line'
plt.plot(
  x, y,
  color = 'blue'
  # titles are added separately
);
plt.title('Matplotlib line plot');
plt.xlabel('x');
plt.ylabel('$y = e^x$');
plt.show()
```



Matplotlib line plot
20000
15000
10000
5000
0
$y = e^x$
0
2
4
6
8
10
x

Chapter 8

scikit-learn

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

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

```
## '0.24.2'
```

8.1 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()
```

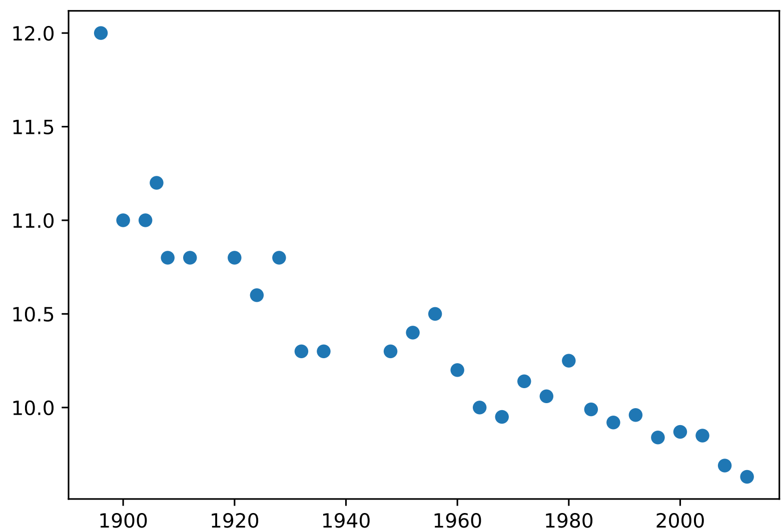
```
##     year  time
```

```
## 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 0x0000000042BE7DD8>
```

```
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.

8.1.1 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)
```

8.1.2 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'>
```

8.1.3 Fit

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

```
## LinearRegression()
```

```
print(model.coef_) # coefficient
```

```
## [-0.01327532]
```

```
print(model.intercept_) # intercept
```

```
## 36.30912040967222
```

8.1.4 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])
```

8.1.5 Scatter Plot: Actual vs Fitted

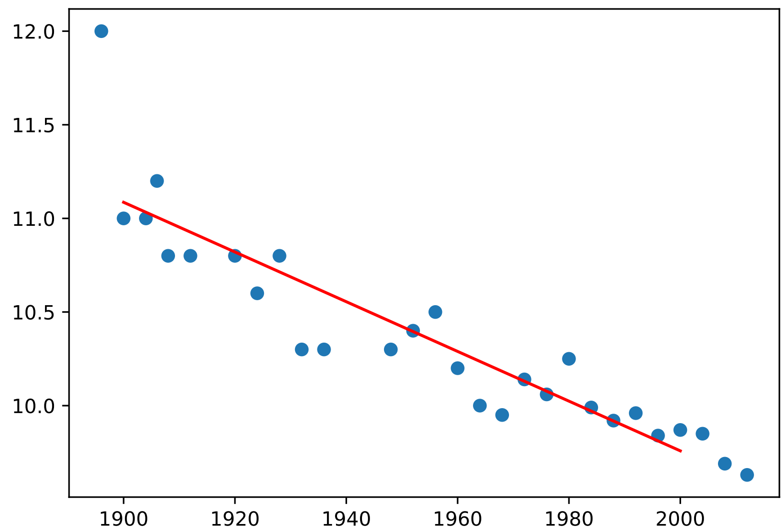
```
plt.scatter(x, y)
```

```
## <matplotlib.collections.PathCollection object at 0x0000000042C51D30>
```

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

```
## [<matplotlib.lines.Line2D object at 0x0000000042C63048>]
```

```
plt.show()
```



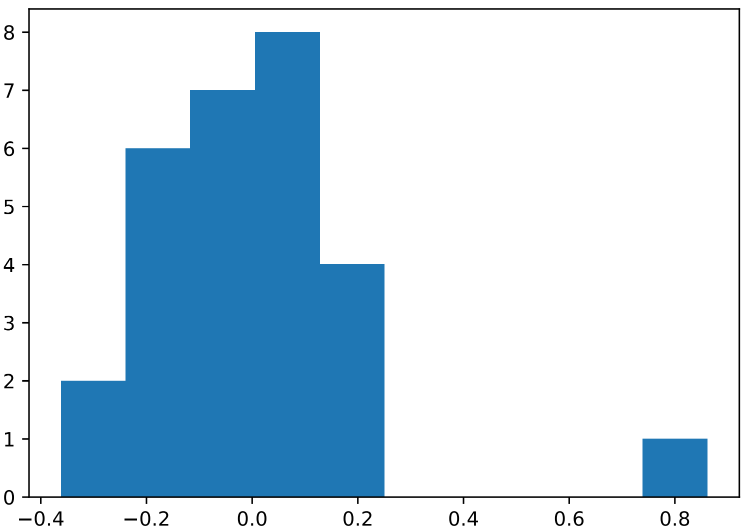

```
# find residuals
residuals = y - model.predict(X)
np.mean(residuals) # check mean
```

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

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

```
## (array([2., 6., 7., 8., 4., 0., 0., 0., 0., 1.]), array([-0.36119479, -0.23898595, -0.11677712
##         0.24984939,  0.37205822,  0.49426705,  0.61647589,  0.73868472,
##         0.86089356]), <a list of 10 Patch objects>)
```

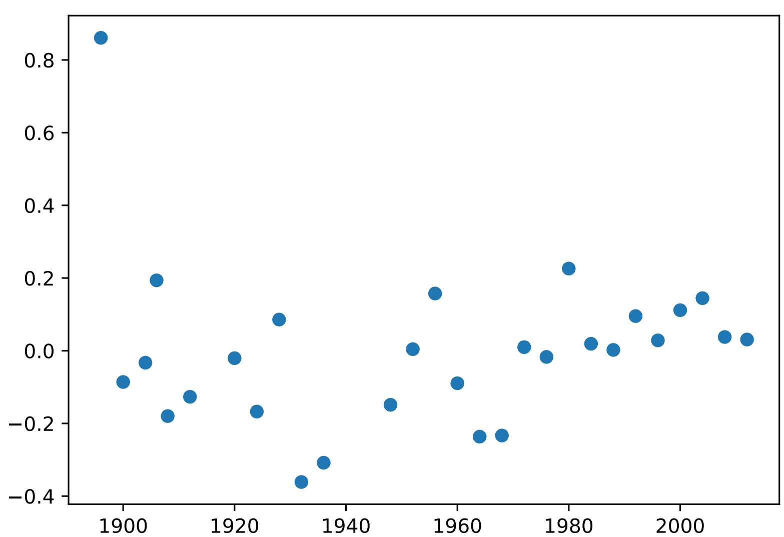
```
plt.show()
```




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

```
## [<matplotlib.lines.Line2D object at 0x0000000042D047F0>]
```

```
plt.show()
```



8.2 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.

8.2.1 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
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]
```

8.2.2 Another Example

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

```
iron.head()
```

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

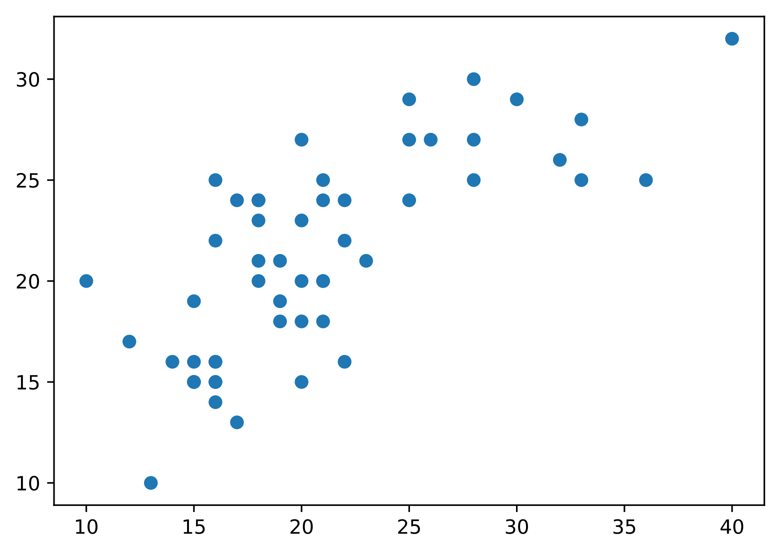
```
iron.shape
```

```
## (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
```

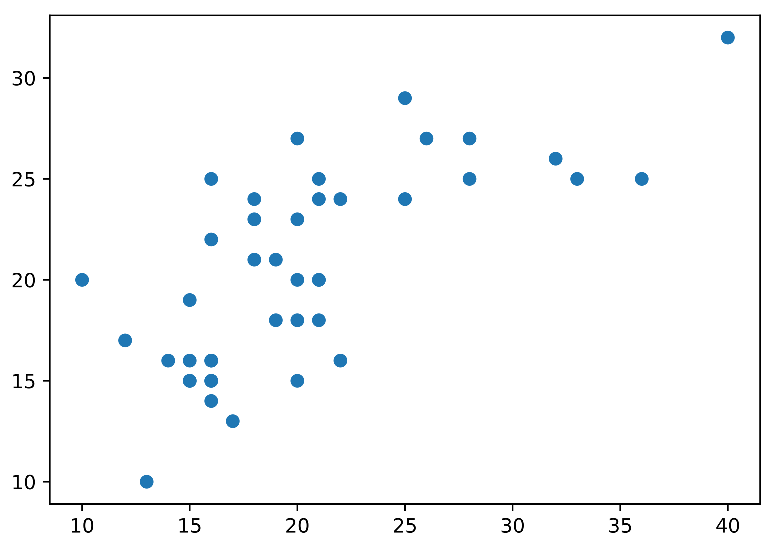
```
test.shape
```

```
## (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)
```

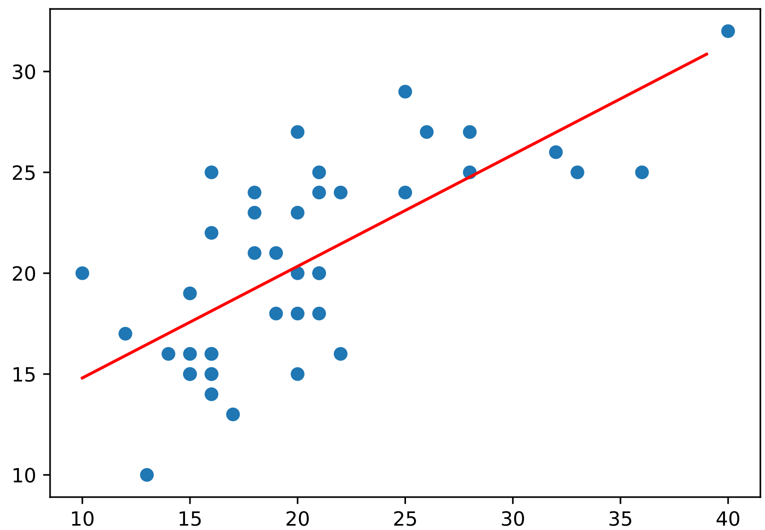
```
## 0.5023548215592256
```

```
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 0x00000000056067F0>
```

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



8.3 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 ~ 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 9

R Strings

9.1 String Manupulation 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" "!"
```

9.2 stringr

```
library(stringr)
```

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

9.3 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.

9.3.1 stringr Functions for Regular Expressions

Function	Job
`str_detect`(str, pattern)	Detects the presence of a pattern and returns TRUE if it is found
`str_locate`(str, pattern)	Locate the 1st position of a pattern and return a matrix with start & end.s

Function	Job
`str_extract`(str, pattern)	Extracts text corresponding to the first match.
`str_match`(str, pattern)	Extracts capture groups formed by () from the first match.
`str_split`(str, pattern)	Splits string into pieces and returns a list of character vectors.

Chapter 10

SQL

10.1 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.

10.2 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;
```

10.3 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;
```

10.4 TRUNCATE

```
TRUNCATE TABLE table_name;
```

10.5 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 PETSALE;
SELECT * FROM PET;
```

10.6 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 `PETSALE` table and show the altered table.

```
ALTER TABLE PETSALE
ADD COLUMN QUANTITY INTEGER;

SELECT * FROM PETSALE;
```

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

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

SELECT * FROM PETSALE;
```

Delete the `PROFIT` column from the `PETSALE` table and show the altered table.

```
ALTER TABLE PETSALE
DROP COLUMN PROFIT;

SELECT * FROM PETSALE;
```

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;
```

10.7 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;
```

10.8 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;
```

10.9 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

10.9.1 *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%';
```

10.9.2 *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%';
```

10.9.3 *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"
```

10.10 Exercise: Sorting

10.10.1 *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;
```

10.10.2 *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;
```

10.10.3 *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	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`.

10.11 Exercise 3: Grouping

10.11.1 *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;
```

10.11.2 *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;
```

10.11.3 *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;
```

10.11.4 *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;
```

10.11.5 *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;
```