

# Toolbox

Bhaswar Chakma

2021-07-10



# Contents

	<b>5</b>
<b>1 dplyr vs pandas</b>	<b>7</b>
1.1 Create Dataframe . . . . .	7
<b>2 Python</b>	<b>9</b>
2.1 Pandas I: Basics . . . . .	9
2.2 Pandas II: Indexing, Arithmetic, Missing Values . . . . .	26
<b>3 R</b>	<b>31</b>
<b>4 SQL</b>	<b>33</b>
4.1 CREATE . . . . .	33
4.2 DROP . . . . .	34
4.3 ALTER . . . . .	34
4.4 TRUNCATE . . . . .	35
4.5 Guided Exercise: Create table and insert data . . . . .	35
4.6 Guided Exercise: Use the <b>ALTER</b> statement to add, delete, or modify columns in two of the existing tables created in the pre- vious exercise. . . . .	36
4.7 Guided Exercise: TRUNCATE . . . . .	37
4.8 Guided Exercise: DROP . . . . .	37







# Chapter 1

## dplyr vs pandas

```
library(dplyr)
```

```
import pandas as pd
```

### 1.1 Create Dataframe

pandas

dplyr

```
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

```
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
```



# Chapter 2

# Python

## 2.1 Pandas I: Basics

NumPy creates ndarrays that must contain values that are of the same data type. Pandas creates dataframes. Each column in a dataframe is an ndarray. This allows us to have traditional tables of data where each column can be a different data type.

Important References:

- Series: <https://pandas.pydata.org/pandas-docs/stable/reference/series.html>
- DataFrame: <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html>

```
import numpy as np
import pandas as pd
```

### 2.1.1 Series

The basic data structure in pandas is the series. You can construct it in a similar fashion to making a numpy array. The command to make a Series object is `pd.Series(data, index=index)`. Note that the `index` argument is optional.

```
data = pd.Series([0.25, 0.5, 0.75, 1.0])
print(data)
```

```
## 0    0.25
## 1    0.50
## 2    0.75
## 3    1.00
## dtype: float64
```

```
print(type(data)) # data type
```

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

```
print(data.values) # data values
```

```
## [0.25 0.5  0.75 1.  ]
```

```
print(type(data.values)) # The values attribute of the series is a numpy array.
```

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

```
print(data.index)
```

```
## RangeIndex(start=0, stop=4, step=1)
```

```
print(type(data.index)) # the row names are known as the index
```

```
## <class 'pandas.core.indexes.range.RangeIndex'>
```

You can subset a pandas series like other python objects.

```
print(data) # example data
```

```
## 0    0.25
## 1    0.50
## 2    0.75
## 3    1.00
## dtype: float64
```

```
print(data[1]) # select the 2nd value
```

```
## 0.5
```

```
print(type(data[1])) # when you select only one value, it simplifies the object
```

```
## <class 'numpy.float64'>
```

```
print(data[1:3])
```

```
## 1    0.50
## 2    0.75
## dtype: float64
```

```
print(type(data[1:3])) # slicing / selecting multiple values returns a series
```

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

You can also do fancy indexing by subsetting w/a numpy array e.g. repeat observations.

```
print(data[np.array([1, 0, 1, 2])])
```

```
## 1    0.50
## 0    0.25
## 1    0.50
## 2    0.75
## dtype: float64
```

Pandas uses a 0-based index by default. You may also specify the index values.

```
data = pd.Series([0.25, 0.5, 0.75, 1.0], index = ['a', 'b', 'c', 'd'])
print(data)
```

```
## a    0.25
## b    0.50
## c    0.75
## d    1.00
## dtype: float64
```

```
data.values
```

```
## array([0.25, 0.5 , 0.75, 1.  ])
```

```
data.index
```

```
## Index(['a', 'b', 'c', 'd'], dtype='object')
```

### Subset with index position or name

- subset with index position

```
data[1]
```

```
## 0.5
```

- subset with index name

```
data["a"]
```

```
## 0.25
```

### Slicing with :

```
data[0:2] # slicing behavior is unchanged
```

```
## a    0.25
## b    0.50
## dtype: float64
```

```
data["a":"c"] # slicing using index names includes the last value
```

```
## a    0.25
## b    0.50
## c    0.75
## dtype: float64
```

### Create a series from a python dictionary

```
# remember, dictionary construction uses curly braces {}
samp_dict = {'Tony Stark': "Robert Downey Jr.",
             'Steve Rogers': "Chris Evans",
             'Natasha Romanoff': "Scarlett Johansson",
             'Bruce Banner': "Mark Ruffalo",
             'Thor': "Chris Hemsworth",
             'Clint Barton': "Jeremy Renner"}
samp_series = pd.Series(samp_dict)
samp_series
```

```
## Tony Stark           Robert Downey Jr.
## Steve Rogers         Chris Evans
## Natasha Romanoff     Scarlett Johansson
## Bruce Banner         Mark Ruffalo
## Thor                 Chris Hemsworth
## Clint Barton         Jeremy Renner
## dtype: object
```

```
print(samp_series.index) # dtype = object is for strings but allows mixed data types.
```

```
## Index(['Tony Stark', 'Steve Rogers', 'Natasha Romanoff', 'Bruce Banner',
##        'Thor', 'Clint Barton'],
##        dtype='object')
```

```
samp_series.values
```

```
## array(['Robert Downey Jr.', 'Chris Evans', 'Scarlett Johansson',
##        'Mark Ruffalo', 'Chris Hemsworth', 'Jeremy Renner'], dtype=object)
```

Another example:

```
# ages during the First Avengers film (2012)
age_dict = {'Thor': 1493,
            'Steve Rogers': 104,
            'Natasha Romanoff': 28,
            'Clint Barton': 41,
            'Tony Stark': 42,
            'Bruce Banner': 42} # note that the dictionary order is not same here
ages = pd.Series(age_dict)
print(ages)
```

```
## Thor           1493
## Steve Rogers   104
## Natasha Romanoff  28
## Clint Barton   41
## Tony Stark     42
## Bruce Banner   42
## dtype: int64
```

Use `np.NaN` to specify missing values.

```
# ages during the First Avengers film (2012)
hero_dict = {'Thor': np.NaN,
             'Steve Rogers': 'Captain America',
             'Natasha Romanoff': 'Black Widow',
             'Clint Barton': 'Hawkeye',
             'Tony Stark': 'Iron Man',
             'Bruce Banner': 'Hulk'}
hero_names = pd.Series(hero_dict)
print(hero_names)
```

```
## Thor                NaN
## Steve Rogers        Captain America
## Natasha Romanoff    Black Widow
## Clint Barton        Hawkeye
## Tony Stark           Iron Man
## Bruce Banner        Hulk
## dtype: object
```

### 2.1.2 DataFrame

There are multiple ways of creating a DataFrame in Pandas:

**Create a dataframe by providing a dictionary of series objects.**

- The dictionary key becomes the column name. The dictionary values become values.
- The keys within the dictionaries become the index.

```
# we previously created the following series
type(samp_series)
```

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

```
type(hero_names)
```

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

```
type(ages)
```

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

```
# Now create data frame using those series
avengers = pd.DataFrame({'actor': samp_series, 'hero name': hero_names, 'age': ages})
# the DataFrame will match the indices and sort them
print(avengers)
```

```
##           actor      hero name  age
## Bruce Banner    Mark Ruffalo    Hulk    42
## Clint Barton      Jeremy Renner  Hawkeye   41
## Natasha Romanoff Scarlett Johansson Black Widow  28
## Steve Rogers      Chris Evans  Captain America  104
## Thor              Chris Hemsworth    NaN  1493
## Tony Stark        Robert Downey Jr.   Iron Man    42
```

```
print(type(avengers)) # this is a DataFrame object
```

```
## <class 'pandas.core.frame.DataFrame'>
```

The data is a list of dictionaries. Each dictionary needs to have the same set of keys, otherwise, NaNs will appear.

**Data is a list of dictionaries**

```
data = [{'a': 0, 'b': 0},
        {'a': 1, 'b': 2},
        {'a': 2, 'b': 5}]
data
```

```
## [{'a': 0, 'b': 0}, {'a': 1, 'b': 2}, {'a': 2, 'b': 5}]
```

```
print(pd.DataFrame(data, index = [1, 2, 3]))
```

```
##    a  b
## 1  0  0
## 2  1  2
## 3  2  5
```

Mismatch of keys produces NaN

```
data2 = [{'a': 0, 'b': 0},
         {'a': 1, 'b': 2},
         {'a': 2, 'c': 5}] # mismatch of keys. NAs will appear
data2
```

```
## [{'a': 0, 'b': 0}, {'a': 1, 'b': 2}, {'a': 2, 'c': 5}]
```

```
pd.DataFrame(data2)## if the index argument is not supplied, it defaults to integer i
```

```
##      a      b      c
## 0  0  0.0  NaN
## 1  1  2.0  NaN
## 2  2  NaN  5.0
```

### Convert a dictionary to a DataFrame.

- The keys form column names, and the values are lists/arrays of values.
- The arrays need to be of the same length.

```
data3 = {'a': [1, 2, 3], 'b': ['x', 'y', 'z']}
data3
```

```
## {'a': [1, 2, 3], 'b': ['x', 'y', 'z']}
```

```
pd.DataFrame(data3)
```

```
##      a      b
## 0  1      x
## 1  2      y
## 2  3      z
```

```
data4 = {'a': [1, 2, 3, 4], 'b': ['x', 'y', 'z']} # arrays are not of the same length
pd.DataFrame(data4)
```

The code above will get the following error

```
ValueError: arrays must all be same length
```

### Turn a 2D Numpy array (matrix) into a DataFrame by adding column names and optionally index values.

```
data = np.random.randint(10, size = 10).reshape((5,2))
print(data)
```



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

```
print(pd.DataFrame(data, columns = ["x","y"], index = ['a','b','c','d','e']))
```

```
##    x  y
## a  2  0
## b  1  8
## c  2  3
## d  4  6
## e  8  9
```

### 2.1.3 Subsetting the DataFrame

In a DataFrame, the `.columns` attribute show the column names and the `.index` attribute show the row names.

```
print(avengers)
```

```
##              actor      hero name  age
## Bruce Banner    Mark Ruffalo      Hulk    42
## Clint Barton    Jeremy Renner    Hawkeye    41
## Natasha Romanoff  Scarlett Johansson  Black Widow    28
## Steve Rogers      Chris Evans  Captain America    104
## Thor             Chris Hemsworth      NaN    1493
## Tony Stark       Robert Downey Jr.    Iron Man     42
```

```
print(avengers.columns)
```

```
## Index(['actor', 'hero name', 'age'], dtype='object')
```

```
print(avengers.index)
```

```
## Index(['Bruce Banner', 'Clint Barton', 'Natasha Romanoff', 'Steve Rogers',
##       'Thor', 'Tony Stark'],
##       dtype='object')
```

You can select a column using:

- dot notation

```
avengers.actor # extracting the column
```

```
## Bruce Banner           Mark Ruffalo
## Clint Barton           Jeremy Renner
## Natasha Romanoff      Scarlett Johansson
## Steve Rogers           Chris Evans
## Thor                   Chris Hemsworth
## Tony Stark             Robert Downey Jr.
## Name: actor, dtype: object
```

- single square brackets.

```
avengers["hero name"] # if there's a space in the column name, you'll need to use square brackets
```

```
## Bruce Banner           Hulk
## Clint Barton           Hawkeye
## Natasha Romanoff      Black Widow
## Steve Rogers           Captain America
## Thor                   NaN
## Tony Stark             Iron Man
## Name: hero name, dtype: object
```

Single column is returned as series. For example, `avengers.actor` is a Pandas Series.

```
type(avengers.actor)
```

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

### Subset

```
print(avengers) # just for ease of inspection
```

```
##              actor      hero name  age
## Bruce Banner   Mark Ruffalo    Hulk   42
## Clint Barton   Jeremy Renner   Hawkeye  41
## Natasha Romanoff Scarlett Johansson Black Widow  28
## Steve Rogers     Chris Evans  Captain America  104
## Thor            Chris Hemsworth      NaN  1493
## Tony Stark      Robert Downey Jr.    Iron Man   42
```

```
avengers.actor[1] # 0 based indexing
```

```
## 'Jeremy Renner'
```

```
avengers.actor[avengers.age == 42]
```

```
## Bruce Banner      Mark Ruffalo
## Tony Stark        Robert Downey Jr.
## Name: actor, dtype: object
```

```
avengers["hero name"]['Steve Rogers']
```

```
## 'Captain America'
```

```
avengers["hero name"]['Steve Rogers':'Tony Stark']
```

```
## Steve Rogers      Captain America
## Thor              NaN
## Tony Stark        Iron Man
## Name: hero name, dtype: object
```

### 2.1.4 .loc

The `.loc` attribute can be used to subset the DataFrame using the index names.

```
avengers.loc['Thor'] # subset based on location to get a row
```

```
## actor      Chris Hemsworth
## hero name      NaN
## age          1493
## Name: Thor, dtype: object
```

```
print(type(avengers.loc['Thor']))
```

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

```
print(type(avengers.loc['Thor'].values)) # the values are of mixed type but is still a numpy array
# this is possible because it is a structured numpy array. (covered in "Python for Data Science"
```

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

```
print(avengers.loc[ : , 'age']) # subset based on location to get a column
```

```
## Bruce Banner          42
## Clint Barton          41
## Natasha Romanoff      28
## Steve Rogers          104
## Thor                  1493
## Tony Stark            42
## Name: age, dtype: int64
```

```
print(type(avengers.loc[:, 'age'])) #the object is a pandas series
```

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

```
print(type(avengers.loc[:, 'age'].values))
```

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

```
avengers.loc['Steve Rogers', 'age'] # you can provide a pair of 'coordinates' to get a
```

```
## 104
```

### 2.1.5 .iloc

The `.iloc` attribute can be used to subset the DataFrame using the index position (zero-indexed).

```
print(avengers) # just for ease of inspection
```

```
##              actor      hero name  age
## Bruce Banner    Mark Ruffalo    Hulk    42
## Clint Barton    Jeremy Renner   Hawkeye  41
## Natasha Romanoff Scarlett Johansson Black Widow  28
## Steve Rogers      Chris Evans  Captain America  104
## Thor             Chris Hemsworth      NaN  1493
## Tony Stark       Robert Downey Jr.   Iron Man    42
```

```
avengers.iloc[3,] # subset based on index location
```

```
## actor          Chris Evans
## hero name      Captain America
## age            104
## Name: Steve Rogers, dtype: object
```

```
avengers.iloc[0, 1] # pair of coordinates
```

```
## 'Hulk'
```

### 2.1.6 Assignment with .loc and .iloc

The .loc and .iloc attributes can be used in conjunction with assignment.

```
# set values individually
avengers.loc['Thor', 'age'] = 1500
avengers.loc['Thor', 'hero name'] = 'Thor'
avengers
```

```
##              actor          hero name  age
## Bruce Banner      Mark Ruffalo      Hulk   42
## Clint Barton      Jeremy Renner     Hawkeye  41
## Natasha Romanoff  Scarlett Johansson Black Widow  28
## Steve Rogers      Chris Evans    Captain America  104
## Thor              Chris Hemsworth    Thor   1500
## Tony Stark        Robert Downey Jr.   Iron Man   42
```

```
# assign multiple values at once
avengers.loc['Thor', ['hero name', 'age']] = [np.NaN, 1493]
avengers
```

```
##              actor          hero name  age
## Bruce Banner      Mark Ruffalo      Hulk   42
## Clint Barton      Jeremy Renner     Hawkeye  41
## Natasha Romanoff  Scarlett Johansson Black Widow  28
## Steve Rogers      Chris Evans    Captain America  104
## Thor              Chris Hemsworth    NaN   1493
## Tony Stark        Robert Downey Jr.   Iron Man   42
```

### 2.1.7 .loc vs .iloc with numeric index

The following DataFrame has a numeric index, but it starts at 1 instead of 0.

```
data = [{'a': 11, 'b': 2},
        {'a': 12, 'b': 4},
        {'a': 13, 'b': 6}]
df = pd.DataFrame(data, index = [1, 2, 3])
df
```

```
##      a  b
## 1  11  2
## 2  12  4
## 3  13  6
```

`.loc` always uses the actual index..

```
df.loc[1, :]
```

```
## a      11
## b        2
## Name: 1, dtype: int64
```

`.iloc` always uses the position using a 0-based index..

```
df.iloc[1, :]
```

```
## a      12
## b        4
## Name: 2, dtype: int64
```

```
df.iloc[3, :] # using a position that doesn't exist results in an exception.
```

IndexError: single positional indexer is out-of-bounds

### 2.1.8 Boolean subsetting examples with `.loc`

```
print(avengers) # just for ease of inspection
```

```
##              actor      hero name  age
## Bruce Banner    Mark Ruffalo      Hulk   42
## Clint Barton    Jeremy Renner    Hawkeye   41
## Natasha Romanoff  Scarlett Johansson  Black Widow   28
## Steve Rogers      Chris Evans  Captain America  104
## Thor              Chris Hemsworth      NaN  1493
## Tony Stark        Robert Downey Jr.    Iron Man    42
```

```
# select avengers whose age is less than 50 and greater than 40
# select the columns 'hero name' and 'age'
avengers.loc[ (avengers.age < 50) & (avengers.age > 40), ['hero name', 'age']]
```

```
##           hero name  age
## Bruce Banner      Hulk  42
## Clint Barton    Hawkeye  41
## Tony Stark      Iron Man  42
```

```
# Use the index of the DataFrame, treat it as a string, and select rows that start with B
avengers.loc[ avengers.index.str.startswith('B'), : ]
```

```
##           actor hero name  age
## Bruce Banner Mark Ruffalo   Hulk  42
```

```
# Use the index of the DataFrame, treat it as a string,
# find the character capital R. Find returns -1 if it does not find the letter
# We select rows that did not result in -1, which means it does contain a capital R
avengers.loc[ avengers.index.str.find('R') != -1, : ]
```

```
##           actor           hero name  age
## Natasha Romanoff Scarlett Johansson Black Widow  28
## Steve Rogers      Chris Evans  Captain America  104
```

```
python avengers.loc[ avengers.index.str.find('X') != -1, : ] gets
the message
```

```
Error: unexpected ':' in "avengers.loc[ avengers.index.str.find('X')
!= -1, :]"
```

### 2.1.9 Other commonly used DataFrame attributes

```
avengers.T # the transpose
```

```
##           Bruce Banner  Clint Barton  ...      Thor      Tony Stark
## actor      Mark Ruffalo  Jeremy Renner  ...  Chris Hemsworth  Robert Downey Jr.
## hero name      Hulk      Hawkeye      ...      NaN      Iron Man
## age            42            41      ...      1493            42
##
## [3 rows x 6 columns]
```

```
avengers.dtypes # the data types contained in the DataFrame
```

```
## actor      object
## hero name   object
## age        int64
## dtype: object
```

```
avengers.shape # shape
```

```
## (6, 3)
```

### 2.1.10 Importing Data with pd.read\_csv()

```
# Titanic Dataset
url = 'https://assets.datacamp.com/production/course_1607/datasets/titanic_sub.csv'
titanic = pd.read_csv(url)
```

```
titanic
```

```
##      PassengerId  Survived  Pclass  ...    Fare  Cabin  Embarked
## 0              1         0        3  ...    7.2500   NaN        S
## 1              2         1        1  ...   71.2833   C85        C
## 2              3         1        3  ...    7.9250   NaN        S
## 3              4         1        1  ...   53.1000  C123        S
## 4              5         0        3  ...    8.0500   NaN        S
## ..          ...         ...      ...  ...    ...    ...        ...
## 886            887         0        2  ...   13.0000   NaN        S
## 887            888         1        1  ...   30.0000   B42        S
## 888            889         0        3  ...   23.4500   NaN        S
## 889            890         1        1  ...   30.0000  C148        C
## 890            891         0        3  ...    7.7500   NaN        Q
##
## [891 rows x 11 columns]
```

```
titanic.shape
```

```
## (891, 11)
```

```
titanic.columns
```



```
## Index(['PassengerId', 'Survived', 'Pclass', 'Sex', 'Age', 'SibSp', 'Parch',
##       'Ticket', 'Fare', 'Cabin', 'Embarked'],
##       dtype='object')
```

```
titanic.index
```

```
## RangeIndex(start=0, stop=891, step=1)
```

```
titanic.info()
```

```
## <class 'pandas.core.frame.DataFrame'>
## RangeIndex: 891 entries, 0 to 890
## Data columns (total 11 columns):
## PassengerId      891 non-null int64
## Survived         891 non-null int64
## Pclass           891 non-null int64
## Sex              891 non-null object
## Age              714 non-null float64
## SibSp            891 non-null int64
## Parch            891 non-null int64
## Ticket           891 non-null object
## Fare             891 non-null float64
## Cabin            204 non-null object
## Embarked         889 non-null object
## dtypes: float64(2), int64(5), object(4)
## memory usage: 76.7+ KB
```

```
titanic.describe() # displays summary statistics of the numeric variables
```

```
##      PassengerId  Survived  Pclass  ...  SibSp  Parch    Fare
## count  891.000000  891.000000  891.000000  ...  891.000000  891.000000  891.000000
## mean    446.000000    0.383838    2.308642  ...    0.523008    0.381594   32.204208
## std     257.353842    0.486592    0.836071  ...    1.102743    0.806057   49.693429
## min       1.000000    0.000000    1.000000  ...    0.000000    0.000000    0.000000
## 25%     223.500000    0.000000    2.000000  ...    0.000000    0.000000    7.910400
## 50%     446.000000    0.000000    3.000000  ...    0.000000    0.000000   14.454200
## 75%     668.500000    1.000000    3.000000  ...    1.000000    0.000000   31.000000
## max     891.000000    1.000000    3.000000  ...    8.000000    6.000000  512.329200
##
## [8 rows x 7 columns]
```

## 2.2 Pandas II: Indexing, Arithmetic, Missing Values

### 2.2.1 Indexing

Series that we will use as examples

```
# note that the value after the decimal place corresponds to the letter position.
# i.e. 1.4 corresponds to d, the fourth letter.
original1 = pd.Series([1.4, 2.3, 3.1, 4.2], index = ['d','c','a','b'])
original2 = pd.Series([2.2, 3.1, 1.3, 4.4], index = ['b','a','c','d'])
```

When you create a series, the original order of the index is preserved..

```
original1
```

```
## d    1.4
## c    2.3
## a    3.1
## b    4.2
## dtype: float64
```

```
original2
```

```
## b    2.2
## a    3.1
## c    1.3
## d    4.4
## dtype: float64
```

Making a DataFrame with multiple series with the same index preserves the index order..

```
pd.DataFrame({"x":original1, "x2": original1 * 2})
```

```
##      x  x2
## d  1.4  2.8
## c  2.3  4.6
## a  3.1  6.2
## b  4.2  8.4
```

Note that `original1` and `original2` have different index orders. Because `original1` and `original2` have index in different order, Pandas will sort the index before putting them together.

```
df = pd.DataFrame({"x":original1, "y": original2})
df
```

```
##      x    y
## a  3.1  3.1
## b  4.2  2.2
## c  2.3  1.3
## d  1.4  4.4
```

```
original1.index # the index of original1 is the letters d, c, a, b in a tuple-like object
```

```
## Index(['d', 'c', 'a', 'b'], dtype='object')
```

```
original1['d':'a'] # when slicing pandas uses the index order or original1
```

```
## d    1.4
## c    2.3
## a    3.1
## dtype: float64
```

When slicing Pandas uses the index order of the DataFrame, which has been sorted.

```
df.index
```

```
## Index(['a', 'b', 'c', 'd'], dtype='object')
```

```
df['a':'c']
```

```
##      x    y
## a  3.1  3.1
## b  4.2  2.2
## c  2.3  1.3
```

### Rearranging value

Both Series and DataFrames have the `.sort_index()` and `.sort_values()` methods which can be used to rearrange the value.

```
original2
```

```
## b    2.2
## a    3.1
## c    1.3
## d    4.4
## dtype: float64
```

```
original2.sort_index()
```

```
## a    3.1
## b    2.2
## c    1.3
## d    4.4
## dtype: float64
```

```
original2.sort_values()
```

```
## c    1.3
## b    2.2
## a    3.1
## d    4.4
## dtype: float64
```

```
df
```

```
##      x    y
## a  3.1  3.1
## b  4.2  2.2
## c  2.3  1.3
## d  1.4  4.4
```

```
df.sort_values(by = "x", ascending = False)
```

```
##      x    y
## b  4.2  2.2
## a  3.1  3.1
## c  2.3  1.3
## d  1.4  4.4
```

### Changing the Index

The index of a Pandas Series or Pandas DataFrame is immutable and cannot be modified. However, if you want to change the index of a series or dataframe, you can define a new index and replace the existing index of the series/DataFrame.

```
original1
```

```
## d    1.4
## c    2.3
## a    3.1
## b    4.2
## dtype: float64
```

```
original1.index = range(4) # I replace the index of the series with this range object.
original1
```

```
## 0    1.4
## 1    2.3
## 2    3.1
## 3    4.2
## dtype: float64
```

```
original1.index # We can see this has automatically become a RangeIndex object
```

```
## RangeIndex(start=0, stop=4, step=1)
```

```
original1[1]
```

```
## 2.3
```

```
original1.loc[1] # behaves the same as above
```

```
## 2.3
```

```
original1.iloc[1] # behaves the same as above because the range index starts at 0
```

```
## 2.3
```

```
original1.index = range(1,5)
original1
```

```
## 1    1.4
## 2    2.3
## 3    3.1
## 4    4.2
## dtype: float64
```

```
original1[1]
```

```
## 1.4
```

```
original1.loc[1]
```

```
## 1.4
```

```
original1.iloc[1] # behavior is different because range index starts at 1
```

```
## 2.3
```

```
original1['a'] # throws an error because 'a' is no longer part of the index and cannot
```

KeyError: 'a'

You can change the index of a DataFrame by defining a new object and assigning it to the index.

```
df
```

```
##      x    y
## a  3.1  3.1
## b  4.2  2.2
## c  2.3  1.3
## d  1.4  4.4
```

```
df.index = ['j', 'k', 'l', 'm']
df
```

```
##      x    y
## j  3.1  3.1
## k  4.2  2.2
## l  2.3  1.3
## m  1.4  4.4
```

## Chapter 3

# R





# Chapter 4

## SQL

### 4.1 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.

## 4.2 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;
```

## 4.3 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;
```

## 4.4 TRUNCATE

```
TRUNCATE TABLE table_name;
```

## 4.5 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;
```

#### 4.6 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;
```

## 4.7 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;
```

## 4.8 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;
```