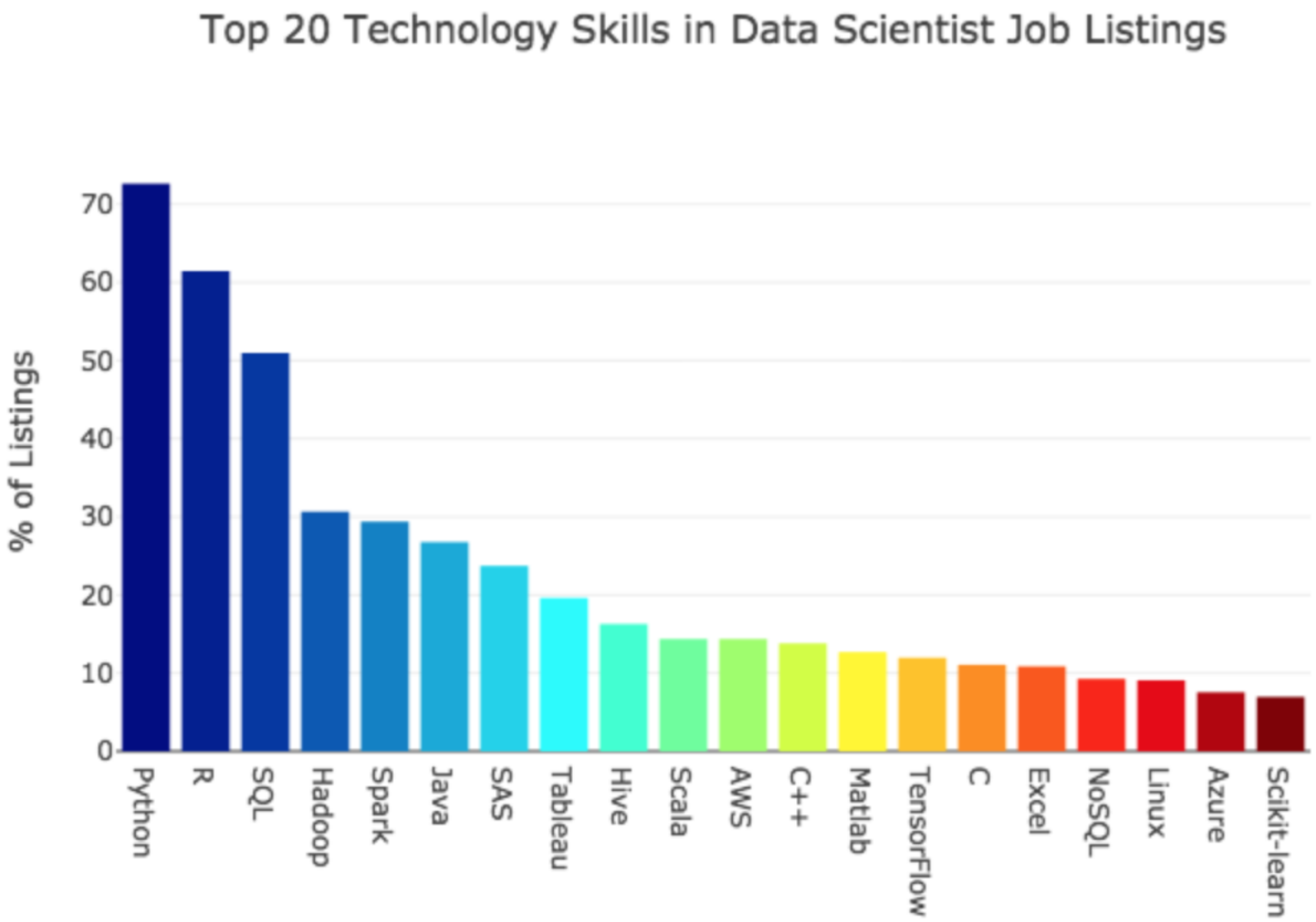
Pandas – Part 1

Pandas

We will use pandas to:

- Read in data from Excel.
- Manipulate data in spreadsheet.
- Visualize data (we will also use another Python package called ggplot to do this).
- Filter and aggregate data from spreadsheet using SQL

Motivation


Top 20 Technology Skills in Data Scientist Job Listings
% of Listings
0
10
20
30
40
50
60
70
Python
R
SQL
Hadoop
Spark
Java
SAS
Tableau
Hive
Scala
AWS
C++
Matlab
TensorFlow
C
Excel
NoSQL
Linux
Azure
Scikit-learn

Reading in Data From Excel

I have the following data saved in the file “Grades_Short.csv”:

F30

	A	B	C	D	E	F	G	H	I
1	Name	Previous_Par	Participation	Mini_Exam1	Mini_Exam2	Participation	Mini_Exam3	Final	Grade
2	Jake	32	1	19.5	20	1	10	33	A
3	Joe	32	1	20	16	1	14	32	A
4	Susan	30	1	19	19	1	10.5	33	A-
5	Sol	31	1	22	13	1	13	34	A
6	Chris	30	1	19	17	1	12.5	33.5	A
7	Tarik	31	1	19	19	1	8	24	B
8	Malik	31.5	1	20	21	1	9	36	A
9									
10									

Let’s see how we read this data into pandas:

Reading in Data From Excel

I have the following data saved in the file “Grades_Short.csv”:

F30

	A	B	C	D	E	F	G	H	I
1	Name	Previous_Par	Participation	Mini_Exam1	Mini_Exam2	Participation	Mini_Exam3	Final	Grade
2	Jake	32	1	19.5	20	1	10	33	A
3	Joe	32	1	20	16	1	14	32	A
4	Susan	30	1	19	19	1	10.5	33	A-
5	Sol	31	1	22	13	1	13	34	A
6	Chris	30	1	19	17	1	12.5	33.5	A
7	Tarik	31	1	19	19	1	8	24	B
8	Malik	31.5	1	20	21	1	9	36	A
9									
10									

Let’s see how we read this data into pandas:

Before you use pandas you must import it. Anytime you use pandas put this line as the top of your code.

```
import pandas as pd

df_grades = pd.read_csv("Grades_Short.csv")
```

Reading in Data From Excel

I have the following data saved in the file “Grades_Short.csv”:

F30

	A	B	C	D	E	F	G	H	I
1	Name	Previous_Par	Participation	Mini_Exam1	Mini_Exam2	Participation	Mini_Exam3	Final	Grade
2	Jake	32	1	19.5	20	1	10	33	A
3	Joe	32	1	20	16	1	14	32	A
4	Susan	30	1	19	19	1	10.5	33	A-
5	Sol	31	1	22	13	1	13	34	A
6	Chris	30	1	19	17	1	12.5	33.5	A
7	Tarik	31	1	19	19	1	8	24	B
8	Malik	31.5	1	20	21	1	9	36	A
9									
10									

Let’s see how we read this data into pandas:

Reading the data into a variable called df_grades.

```
import pandas as pd

df_grades = pd.read_csv("Grades_Short.csv")
```

Built in read_csv method

Path to file

Pandas – read_csv

```
pd.read_csv?
```

```
Signature: pd.read_csv(filepath_or_buffer, sep=',', delimiter=None, header='infer', names=None, index_col=None, usecols=None, squeeze=
False, prefix=None, mangle_dupe_cols=True, dtype=None, engine=None, converters=None, true_values=None, false_values=None, skipinitials
pace=False, skiprows=None, nrows=None, na_values=None, keep_default_na=True, na_filter=True, verbose=False, skip_blank_lines=True, par
se_dates=False, infer_datetime_format=False, keep_date_col=False, date_parser=None, dayfirst=False, iterator=False, chunksize=None, co
mpression='infer', thousands=None, decimal=b'.', lineterminator=None, quotechar='"', quoting=0, escapechar=None, comment=None, encodin
g=None, dialect=None, tupleize_cols=None, error_bad_lines=True, warn_bad_lines=True, skipfooter=0, skip_footer=0, doublequote=True, de
lim_whitespace=False, as_recarray=None, compact_ints=None, use_unsigned=None, low_memory=True, buffer_lines=None, memory_map=False, fl
oat_precision=None)
Docstring:
Read CSV (comma-separated) file into DataFrame

Also supports optionally iterating or breaking of the file
into chunks.

Additional help can be found in the `online docs for IO Tools
<http://pandas.pydata.org/pandas-docs/stable/io.html>`_.

Parameters
----------
filepath_or_buffer : str, pathlib.Path, py._path.local.LocalPath or any object with a read() method (such as a file handle or StringI
O)
    The string could be a URL. Valid URL schemes include http, ftp, s3, and
    file. For file URLs, a host is expected. For instance, a local file could
    be file ://localhost/path/to/table.csv
sep : str, default ','
    Delimiter to use. If sep is None, the C engine cannot automatically detect
    the separator, but the Python parsing engine can, meaning the latter will
    be used and automatically detect the separator by Python's builtin sniffer
    tool, ``csv.Sniffer``. In addition, separators longer than 1 character and
    different from ``'\s+'`` will be interpreted as regular expressions and
    will also force the use of the Python parsing engine. Note that regex
    delimiters are prone to ignoring quoted data. Regex example: ``'\r\t'``
delimiter : str, default ``None``
```

Always specify the input name (order of inputs only matters if you don’t)

Reading in Data From Excel

So, what is df_grades and how does it store the data?

```
import pandas as pd

df_grades = pd.read_csv("Grades_Short.csv")
df_grades
```

Typing the name of any variable at the end of a code cell will display the contents of the variable.

Reading in Data From Excel

So, what is df_grades and how does it store the data?

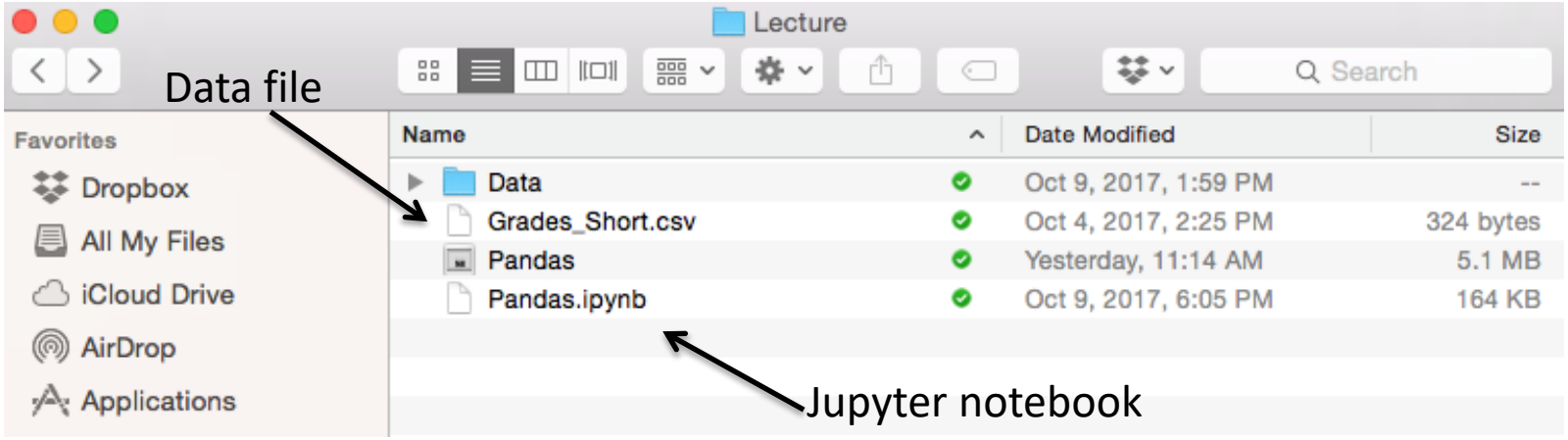
```
import pandas as pd

df_grades = pd.read_csv("Grades_Short.csv")
df_grades
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

- df_grades is a pandas **dataframe.**
- The data is stored in a tabular format very similar to excel.

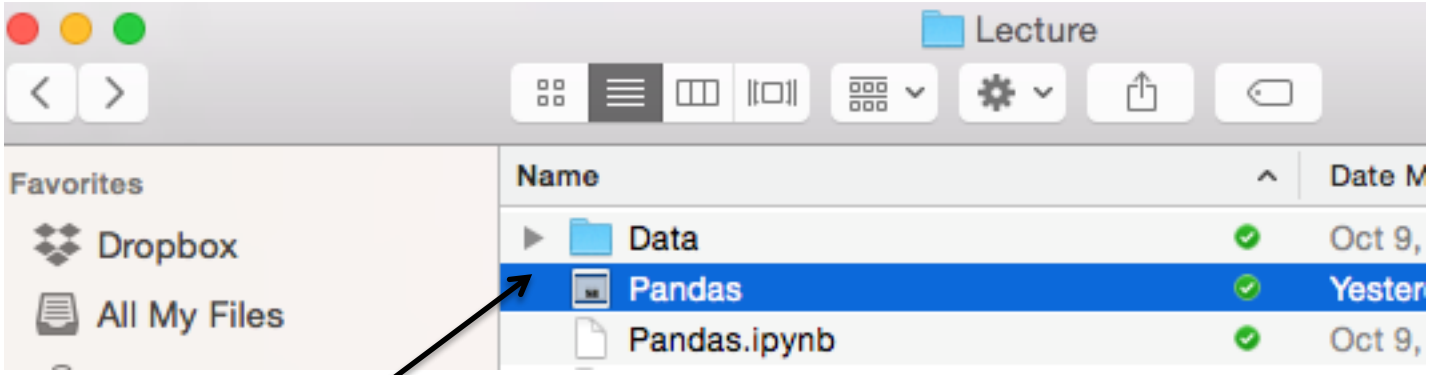
Reading in Data From Excel




```
#Relative file path
df_grades = pd.read_csv("Grades_Short.csv")
df_grades.head()
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	ID
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A	90743
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A	7284
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-	7625
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A	1237
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A	62

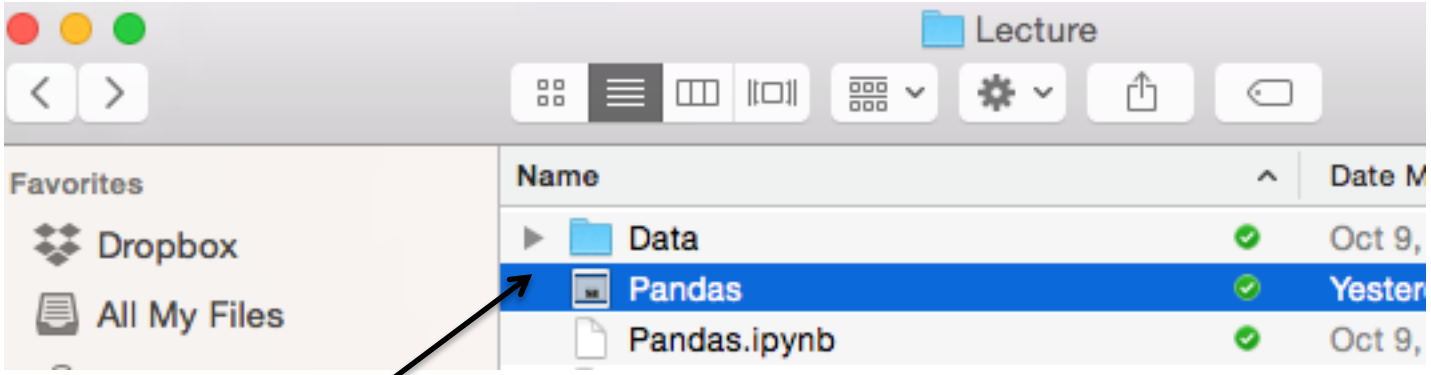
Reading in Data From Excel



Now Grades_Short.csv is in Data Folder

Jupyter notebook

Reading in Data From Excel



Now Grades_Short.csv is in Data Folder

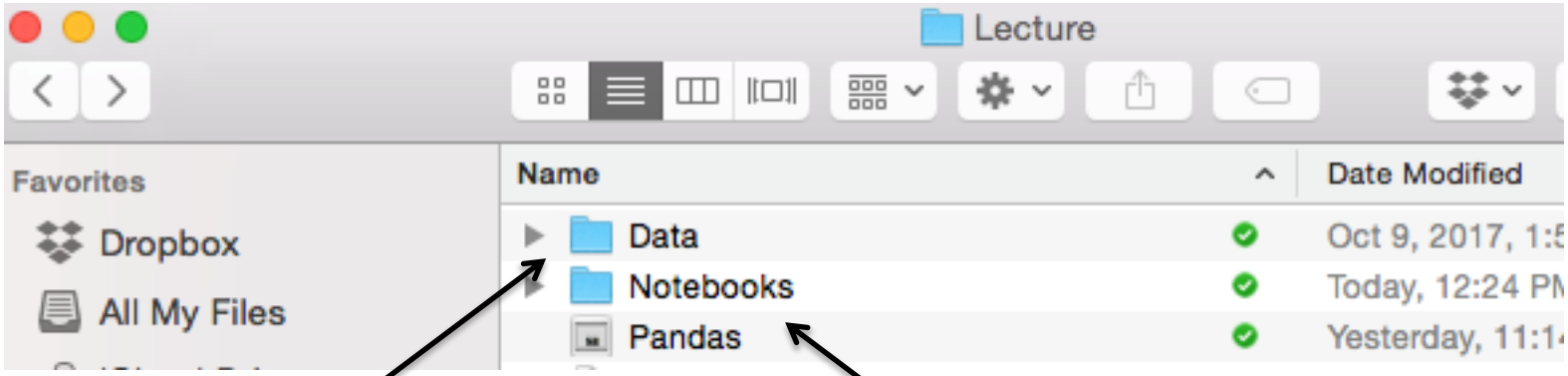
Jupyter Notebook

"/" separates directories

```
#Relative file path
df_grades = pd.read_csv("Data/Grades_Short.csv")
df_grades.head()
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	ID
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A	90743
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A	7284
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-	7625
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A	1237
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A	62

Reading in Data From Excel



Now Grades_Short.csv is in Data Folder

Jupyter notebook in folder Notebooks

".." = go back one directory

```
#Relative file path
df_grades = pd.read_csv("../Data/Grades_Short.csv")
df_grades.head()
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	ID
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A	90743
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A	7284
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-	7625
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A	1237
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A	62

The head() Method

Using the **head()** method

```
import pandas as pd

df_grades = pd.read_csv("Grades_Short.csv")
df_grades.head(3)
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-

- If the data is really large you don't want to print out the entire dataframe to your output.
- The **head(n)** method outputs the first n rows of the data frame. If n is not supplied, the default is the first 5 rows.
- I like to run the head() method after I read in the dataframe to check that everything got read in correctly.
- There is also a **tail(n)** method that returns the last n rows of the dataframe

Basic Features

```
import pandas as pd

df_grades = pd.read_csv("Grades_Short.csv")
df_grades.head(3)
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-

```
#dimension of df
df_grades.shape
```

```
(7, 9)
```

Think of this as a list

```
#How each column is stored
df_grades.dtypes
```

```
Name              object
Previous_Part    float64
Participation1     int64
Mini_Exam1       float64
Mini_Exam2         int64
Participation2     int64
Mini_Exam3       float64
Final            float64
Grade             object
dtype: object
```

object = string

float64 = decimal

int64 = integer

Basic Features

column names

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-

row names = index

Basic Features

column names

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-

row names = index

```
#Get column names
df_grades.columns
```

```
Index(['Name', 'Previous_Part', 'Participation1', 'Mini_Exam1', 'Mini_Exam2',
       'Participation2', 'Mini_Exam3', 'Final', 'Grade'],
      dtype='object')
```

```
#Get row names
df_grades.index
```

```
RangeIndex(start=0, stop=7, step=1)
```

Basic Features

column names

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-

row names = index

- Pandas defaults to have the index be the row number and it will automatically recognize that the first row is the column names.
    - We will discuss later why you would want the index to be something other than the row numbers.
- Next we discuss how to pick out various pieces of the dataframe.

Selecting a Single Column

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

```
#Get Name column
df_grades['Name']
```

```
0      Jake
1       Joe
2     Susan
3       Sol
4     Chris
5     Tarik
6     Malik
Name: Name, dtype: object
```

- Between square brackets, the column must be given as a string
- Outputs column as a series
  - A series is a one dimensional dataframe..more on this in the slicing section

Selecting a Single Column

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

```
#Get Name column
df_grades.Name
```

```
0      Jake
1       Joe
2     Susan
3       Sol
4     Chris
5     Tarik
6     Malik
Name: Name, dtype: object
```

- Exactly equivalent way to get Name column
  - + : don't have to type brackets or quotes
  - -: won't generalize to selecting multiple columns,, won't work if column names have spaces, can't create new columns this way

Selecting Multiple Columns

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

```
#Select multiple columns
df_grades[["Name", "Grade"]]
```

	Name	Grade
0	Jake	A
1	Joe	A
2	Susan	A-
3	Sol	A
4	Chris	A
5	Tarik	B
6	Malik	A

- List of strings, which correspond to column names.
- You can select as many column as you want.
- Column don't have to be contiguous.

Storing Result

```
#Print the column
df_grades["Name"]
```

```
0     Jake
1      Joe
2    Susan
3      Sol
4    Chris
5    Tarik
6    Malik
Name: Name, dtype: object
```

```
#Store the column
names= df_grades["Name"]
names
```

```
0     Jake
1      Joe
2    Susan
3      Sol
4    Chris
5    Tarik
6    Malik
Name: Name, dtype: object
```

The variable name stores a series

Why store a slice?

- We might want/have to do our analysis is steps.
  - Less error prone
  - More readable

Slicing a Series

Slice/index through the index, which is usually numbers

```
names= df_grades["Name"]
names
```

```
0     Jake
1      Joe
2    Susan
3      Sol
4    Chris
5    Tarik
6    Malik
Name: Name, dtype: object
```

Slicing a Series

Slice/index through the index, which is usually numbers

```
names= df_grades["Name"]
names
```

```
0     Jake
1      Joe
2    Susan
3      Sol
4    Chris
5    Tarik
6    Malik
Name: Name, dtype: object
```

Picking out single element

```
names[0]
```

```
'Jake'
```

Slicing a Series

```
names= df_grades["Name"]
names
```

```
0     Jake
1      Joe
2    Susan
3      Sol
4    Chris
5    Tarik
6    Malik
Name: Name, dtype: object
```

Slice/index through the index, which is usually numbers

Picking out single element

```
names[0]
```

```
'Jake'
```

Contiguous slice

non_inclusive

```
names[1:4]
```

```
1      Joe
2    Susan
3      Sol
Name: Name, dtype: object
```

Slicing a Series

Slice/index through the index, which is usually numbers

```
names= df_grades["Name"]
names
```

```
0      Jake
1       Joe
2     Susan
3       Sol
4     Chris
5     Tarik
6     Malik
Name: Name, dtype: object
```

Picking out single element

```
names[0]
```

```
'Jake'
```

Contiguous slice

```
names[1:4]
```

```
1       Joe
2     Susan
3       Sol
Name: Name, dtype: object
```

Arbitrary slice

```
names[[1,2,4]]
```

```
1       Joe
2     Susan
4     Chris
Name: Name, dtype: object
```

Slicing a Data Frame

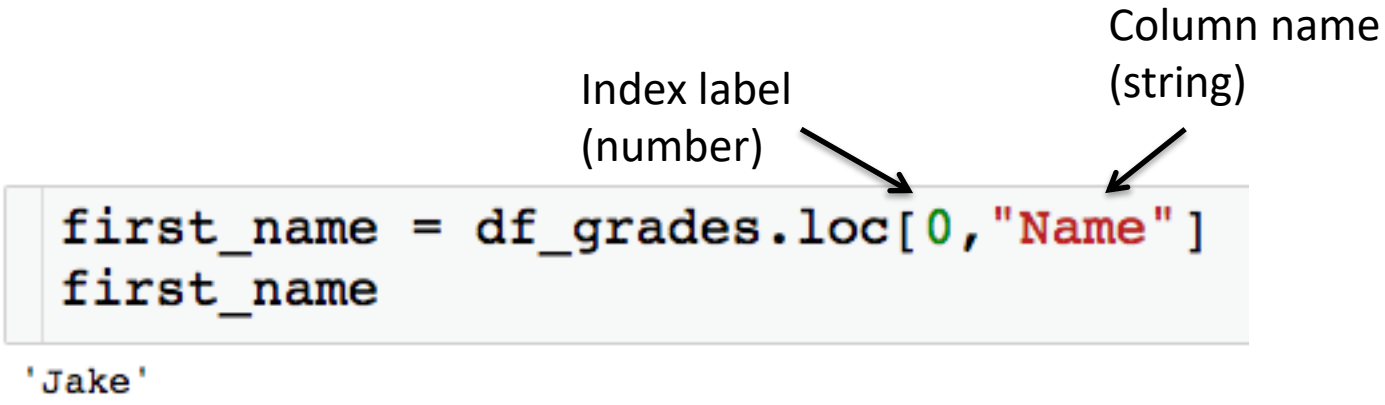
	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

- There are a few ways to pick slice a data frame, we will use the .loc method.
- Access elements through the index labels column names
    - We will see how to change both of these labels later on

Slicing a Data Frame

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

- Pick a single value out.



Slicing a Data Frame

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

- Pick out entire row:

```
first_row = df_grades.loc[0,:]
first_row
```

```
Name                Jake
Previous_Part         32
Participation1         1
Mini_Exam1          19.5
Mini_Exam2            20
Participation2         1
Mini_Exam3            10
Final                 33
Grade                  A
Name: 0, dtype: object
```

"pick out all columns"

first_row is a series

Slicing a Data Frame

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

- Pick out contiguous chunk:

Endpoints are inclusive!

```
slice_one = df_grades.loc[0:2,"Name":"Mini_Exam2"]
slice_one
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2
0	Jake	32.0	1	19.5	20
1	Joe	32.0	1	20.0	16
2	Susan	30.0	1	19.0	19

Slicing a Data Frame

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

- Pick out arbitrary chunk:

```
slice_two = df_grades.loc[[0,2,3], ["Name", "Grade"]]
slice_two
```

	Name	Grade
0	Jake	A
2	Susan	A-
3	Sol	A

Pandas – Part 2

Built in Functions

```
import pandas as pd

df_grades = pd.read_csv("Data/Grades_Short.csv")
df_grades
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

How do I compute the average score on the final?

Built in Functions

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

How do I compute the average score on the final?

```
#Print out
df_grades.Final.mean()
```

```
32.214285714285715
```

```
#Store
avg_final = df_grades.Final.mean()
avg_final
```

```
32.214285714285715
```

Built in mean() method

Built in Functions

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

How do I compute the highest Mini Exam 1 score?

```
max_mini_1 = df_grades["Mini_Exam1"].max()
max_mini_1
```

```
22.0
```

Built in Functions

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

I can actually get all key stats for *numeric* columns at once with the describe() method:

```
summary_df = df_grades.describe()
summary_df
```

summary_df is a dataframe!

	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final
count	7.000000	7.0	7.000000	7.000000	7.0	7.000000	7.000000
mean	31.071429	1.0	19.785714	17.857143	1.0	11.000000	32.214286
std	0.838082	0.0	1.074598	2.734262	0.0	2.217356	3.828154
min	30.000000	1.0	19.000000	13.000000	1.0	8.000000	24.000000
25%	30.500000	1.0	19.000000	16.500000	1.0	9.500000	32.500000
50%	31.000000	1.0	19.500000	19.000000	1.0	10.500000	33.000000
75%	31.750000	1.0	20.000000	19.500000	1.0	12.750000	33.750000
max	32.000000	1.0	22.000000	21.000000	1.0	14.000000	36.000000

Built in Functions

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

I can actually get all key stats for *numeric* columns at once with the describe() method:

```
summary_df = df_grades.describe()
summary_df[["Final", "Mini_Exam3"]]
```

	Final	Mini_Exam3
count	7.000000	7.000000
mean	32.214286	11.000000
std	3.828154	2.217356
min	24.000000	8.000000
25%	32.500000	9.500000
50%	33.000000	10.500000
75%	33.750000	12.750000
max	36.000000	14.000000

Built in Functions

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

I can actually get all key stats for *numeric* columns at once with the describe() method:

```
summary_df = df_grades.describe()
summary_df[["Final", "Mini_Exam3"]]
```

	Final	Mini_Exam3
count	7.000000	7.000000
mean	32.214286	11.000000
std	3.828154	2.217356
min	24.000000	8.000000
25%	32.500000	9.500000
50%	33.000000	10.500000
75%	33.750000	12.750000
max	36.000000	14.000000

Notice here the index is *not* row numbers…

Built in Functions

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

Other useful built in methods:

```
df_grades["Grade"].value_counts()
```

```
A     5
A-    1
B     1
Name: Grade, dtype: int64
```

value_count(): Gives a count of the number of times each unique value apears in the column. Returns a series where indices are the unique column values.

Built in Functions

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

Other useful built in methods:

```
counts = df_grades["Grade"].value_counts()
counts
```

```
A     5
A-    1
B     1
Name: Grade, dtype: int64
```

```
counts["A"]
```

```
5
```

value_count(): Gives a count of the number of times each unique value appears in the column. Returns a series where indices are the unique column values.

Built in Functions

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

Other useful built in methods:

```
df_grades["Grade"].unique()
```

```
array(['A', 'A-', 'B'], dtype=object)
```

```
unique_values = df_grades["Grade"].unique()
unique_values[0]
```

```
'A'
```

```
len(unique_values)
```

```
3
```

unique(): Returns an array of all of the unique values.

Attributes vs. Methods

When do I a put a ()?

```
#Get dimensions
df_grades.shape
```

```
(7, 9)
```

```
#Get column types
df_grades.dtypes
```

```
Name              object
Previous_Part    float64
Participation1     int64
Mini_Exam1       float64
Mini_Exam2         int64
Participation2     int64
Mini_Exam3       float64
Final            float64
Grade             object
dtype: object
```

```
#Get first 5 rows
df_grades.head()
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A

```
#Compute mean of finals column
df_grades["Final"].mean()
```

```
32.214285714285715
```

Attributes vs. Methods

When do I a put a ()?

dataframe attributes

```
#Get dimensions
df_grades.shape
```

```
(7, 9)
```

```
#Get column types
df_grades.dtypes
```

```
Name              object
Previous_Part    float64
Participation1     int64
Mini_Exam1       float64
Mini_Exam2         int64
Participation2     int64
Mini_Exam3       float64
Final            float64
Grade             object
dtype: object
```

dataframe methods

```
#Get first 5 rows
df_grades.head()
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A

```
#Compute mean of finals column
df_grades["Final"].mean()
```

```
32.214285714285715
```

Attributes vs. Methods

When do I a put a ()?

dataframe attributes

```
#Get dimensions
df_grades.shape
```

```
(7, 9)
```

```
#Get column types
df_grades.dtypes
```

```
Name                object
Previous_Part      float64
Participation1       int64
Mini_Exam1         float64
Mini_Exam2           int64
Participation2       int64
Mini_Exam3         float64
Final              float64
Grade               object
dtype: object
```

Features of dataframe

dataframe methods

```
#Get first 5 rows
df_grades.head()
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A

```
#Compute mean of finals column
df_grades["Final"].mean()
```

```
32.214285714285715
```

Require computation for output

Creating New Columns

```
import pandas as pd

df_grades = pd.read_csv("Data/Grades_Short.csv")
df_grades.head()
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A

Let's create a useless new column of all 1s:

```
df_grades["new_column"]=1
df_grades.head()
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	new_column
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A	1
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A	1
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-	1
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A	1
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A	1

Creating New Columns

```
import pandas as pd

df_grades = pd.read_csv("Data/Grades_Short.csv")
df_grades.head()
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A

$33/36$

$32/36$

We can also create column as function of other column. The Final was worth 36 points, let's create a column for each student's percentage.

```
df_grades["Final_Percentage"]=df_grades["Final"]/36
df_grades.head()
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	new_column	Final_Percentage
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A	1	0.916667
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A	1	0.888889
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-	1	0.916667
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A	1	0.944444
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A	1	0.930556

Deleting Columns

```
df_grades.head()
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	new_column	Final_Percentage	Part_perc
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A	1	0.916667	1.0
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A	1	0.888889	1.0
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-	1	0.916667	1.0
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A	1	0.944444	1.0
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A	1	0.930556	1.0

```
#Delete single column
del df_grades["Final_Percentage"]
df_grades.head()
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	new_column	Part_perc
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A	1	1.0
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A	1	1.0
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-	1	1.0
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A	1	1.0
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A	1	1.0

Deleting Columns

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	new_column	Part_perc
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A	1	1.0
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A	1	1.0
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-	1	1.0
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A	1	1.0
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A	1	1.0

```
#Delete multiple columns
df_grades.drop(["new_column","Part_perc"], axis=1, inplace = True)
df_grades.head()
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A

Deleting Columns

```
df_grades.drop?
```

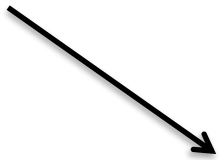
```
Signature: df_grades.drop(labels=None, axis=0, index=None, columns=None, level=None, inplace=False, errors='raise')
Docstring:
Return new object with labels in requested axis removed.

Parameters
----------
labels : single label or list-like
    Index or column labels to drop.
axis : int or axis name
    Whether to drop labels from the index (0 / 'index') or
    columns (1 / 'columns').
index, columns : single label or list-like
    Alternative to specifying `axis` (``labels, axis=1`` is
    equivalent to ``columns=labels``).

    .. versionadded:: 0.21.0
level : int or level name, default None
    For MultiIndex
inplace : bool, default False
    If True, do operation inplace and return None.
errors : {'ignore', 'raise'}, default 'raise'
    If 'ignore', suppress error and existing labels are dropped.
```

The Drop Method

- List of column of index label



```
df_grades.drop(["new_column","Part_perc"], axis=1, inplace = True)
df_grades.head()
```

The Drop Method

- List of column of index label

```
df_grades.drop(["new_column","Part_perc"], axis=1, inplace = True)
df_grades.head()
```

- axis = 1 – delete specified columns
- axis = 0 – delete specified rows

The Drop Method

- List of column of index label

- inplace = True– change df_grades
- inplace = False – return dataframe with specified columns deleted, do not change df_grades

```
df_grades.drop(["new_column","Part_perc"], axis=1, inplace = True)
df_grades.head()
```

- axis = 1 – delete specified columns
- axis = 0 – delete specified rows

The Drop Method - inplace

```
name_grade
```

	Name	Grade
0	Jake	A
1	Joe	A
2	Susan	A-
3	Sol	A
4	Chris	A

```
#Deleting in place
name_grade.drop("Grade", axis=1, inplace=True)
name_grade
```

	Name
0	Jake
1	Joe
2	Susan
3	Sol
4	Chris

The column “Grade” is removed from name_grade

The Drop Method - inplace

```
name_grade
```

	Name	Grade
0	Jake	A
1	Joe	A
2	Susan	A-
3	Sol	A
4	Chris	A

```
#Deleting in place
name_grade.drop("Grade", axis=1, inplace=False)
```

	Name
0	Jake
1	Joe
2	Susan
3	Sol
4	Chris

Just returns a dataframe with “Grade” deleted.

```
name_grade
```

	Name	Grade
0	Jake	A
1	Joe	A
2	Susan	A-
3	Sol	A
4	Chris	A

The dataframe name_grade is unchanged.

The Drop Method - inplace

```
name_grade
```

	Name	Grade
0	Jake	A
1	Joe	A
2	Susan	A-
3	Sol	A
4	Chris	A

```
#Deleting in place
just_name = name_grade.drop("Grade", axis=1, inplace=False)
just_name
```

	Name
0	Jake
1	Joe
2	Susan
3	Sol
4	Chris

Have to store the result in a variable to use the dataframe with “Grade” deleted.

The Drop Method - axis

```
name_grade
```

	Name	Grade
0	Jake	A
1	Joe	A
2	Susan	A-
3	Sol	A
4	Chris	A

```
#Deleting rows in place
name_grade.drop([0,2], axis=0, inplace=True)
name_grade
```

	Name	Grade
1	Joe	A
3	Sol	A
4	Chris	A

List of index labels to be deleted

Sorting Dataframes

```
#Sort dataframe by Final
df_grades.sort_values(by = "Final", inplace=True, ascending=True)

df_grades.head()
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
18	Sarah	22.0	1	18.0	13.0	1	9.0	21.0	C+
15	Josh	23.5	1	17.0	12.0	1	8.5	23.0	C+
5	Tarik	31.0	1	19.0	19.0	1	8.0	24.0	B
10	Michael	29.0	1	20.0	20.0	1	14.0	30.0	A
16	Jackson	28.0	1	18.0	15.5	1	7.0	31.0	B

Sorting Dataframes

```
df_grades.sort_values?
```

```
Signature: df_grades.sort_values(by, axis=0, ascending=True, inplace=False, kind='quicksort', na_position='last')
Docstring:
Sort by the values along either axis

.. versionadded:: 0.17.0

Parameters
----------
by : str or list of str
    Name or list of names which refer to the axis items.
axis : {0 or 'index', 1 or 'columns'}, default 0
    Axis to direct sorting
ascending : bool or list of bool, default True
     Sort ascending vs. descending. Specify list for multiple sort
     orders.  If this is a list of bools, must match the length of
     the by.
inplace : bool, default False
     if True, perform operation in-place
kind : {'quicksort', 'mergesort', 'heapsort'}, default 'quicksort'
     Choice of sorting algorithm. See also ndarray.np.sort for more
     information.  `mergesort` is the only stable algorithm. For
     DataFrames, this option is only applied when sorting on a single
     column or label.
na_position : {'first', 'last'}, default 'last'
     `first` puts NaNs at the beginning, `last` puts NaNs at the end
```

Sorting Dataframes

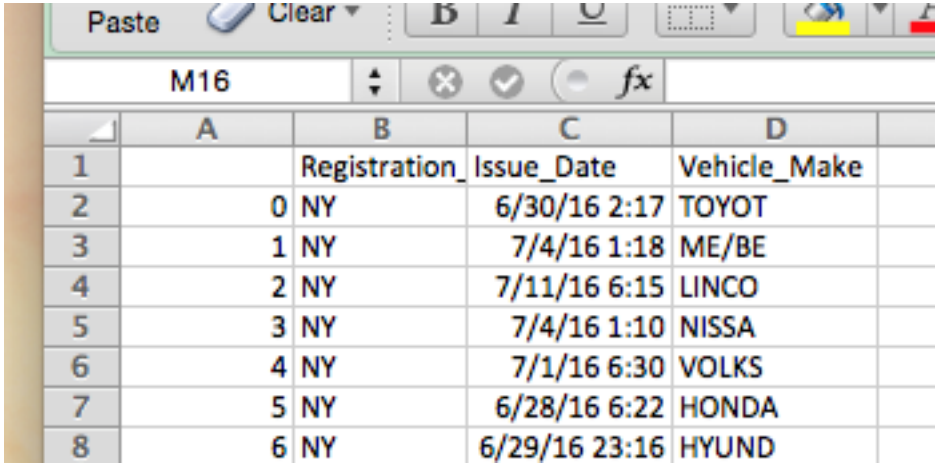
```
#Sort dataframe by Final tiebreak with Previous Part
df_grades.sort_values(by = ["Final", "Previous_Part"]\
                          , inplace=True, ascending=[True,True])

df_grades.head(6)
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
18	Sarah	22.0	1	18.0	13.0	1	9.0	21.0	C+
15	Josh	23.5	1	17.0	12.0	1	8.5	23.0	C+
5	Tarik	31.0	1	19.0	19.0	1	8.0	24.0	B
10	Michael	29.0	1	20.0	20.0	1	14.0	30.0	A
11	Jimmy	27.5	0	7.0	13.0	1	5.5	31.0	B-
16	Jackson	28.0	1	18.0	15.5	1	7.0	31.0	B

Parking Ticket Data

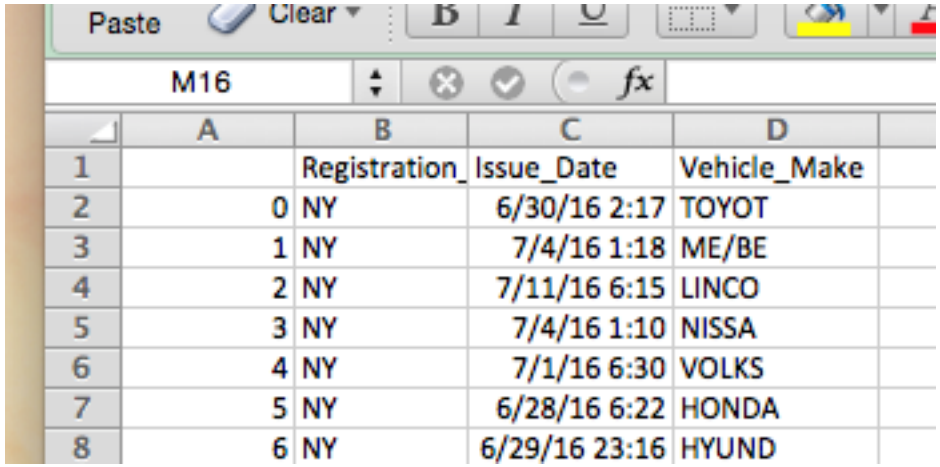
I have the following data saved in the file “Parking.csv”:



	A	B	C	D
1		Registration_	Issue_Date	Vehicle_Make
2	0	NY	6/30/16 2:17	TOYOT
3	1	NY	7/4/16 1:18	ME/BE
4	2	NY	7/11/16 6:15	LINCO
5	3	NY	7/4/16 1:10	NISSA
6	4	NY	7/1/16 6:30	VOLKS
7	5	NY	6/28/16 6:22	HONDA
8	6	NY	6/29/16 23:16	HYUND

Parking Ticket Data

I have the following data saved in the file “Parking.csv”:



	A	B	C	D
1		Registration_	Issue_Date	Vehicle_Make
2	0	NY	6/30/16 2:17	TOYOT
3	1	NY	7/4/16 1:18	ME/BE
4	2	NY	7/11/16 6:15	LINCO
5	3	NY	7/4/16 1:10	NISSA
6	4	NY	7/1/16 6:30	VOLKS
7	5	NY	6/28/16 6:22	HONDA
8	6	NY	6/29/16 23:16	HYUND

Let’s see what happens when we read in the data

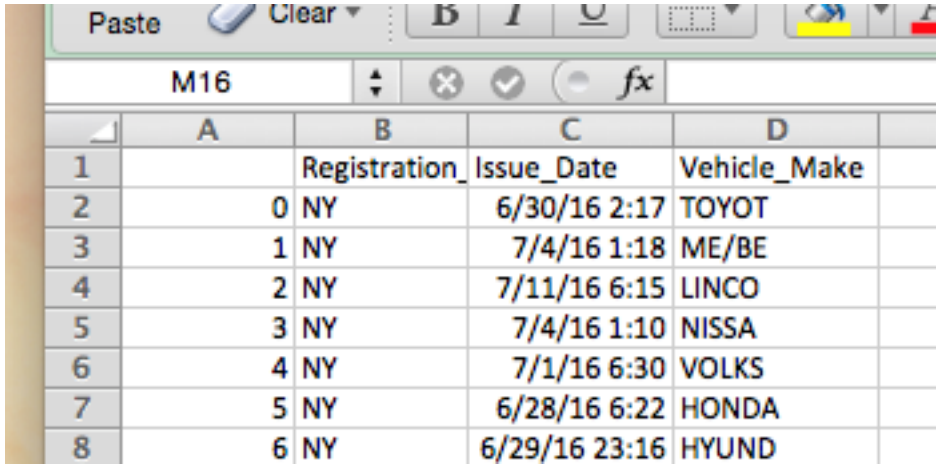
```
df_parking = pd.read_csv("Data/Parking.csv")
df_parking.head()
```

Unwanted column

	Unnamed: 0	Registration_State	Issue_Date	Vehicle_Make
0	0	NY	6/30/16 2:17	TOYOT
1	1	NY	7/4/16 1:18	ME/BE
2	2	NY	7/11/16 6:15	LINCO
3	3	NY	7/4/16 1:10	NISSA
4	4	NY	7/1/16 6:30	VOLKS

Parking Ticket Data

I have the following data saved in the file “Parking.csv”:



	A	B	C	D
1		Registration_	Issue_Date	Vehicle_Make
2	0	NY	6/30/16 2:17	TOYOT
3	1	NY	7/4/16 1:18	ME/BE
4	2	NY	7/11/16 6:15	LINCO
5	3	NY	7/4/16 1:10	NISSA
6	4	NY	7/1/16 6:30	VOLKS
7	5	NY	6/28/16 6:22	HONDA
8	6	NY	6/29/16 23:16	HYUND

Let’s see what happens when we read in the data

```
df_parking = pd.read_csv("Data/Parking.csv")
df_parking.head()
```

Unwanted column

	Unnamed: 0	Registration_State	Issue_Date	Vehicle_Make
0	0	NY	6/30/16 2:17	TOYOT
1	1	NY	7/4/16 1:18	ME/BE
2	2	NY	7/11/16 6:15	LINCO
3	3	NY	7/4/16 1:10	NISSA
4	4	NY	7/1/16 6:30	VOLKS

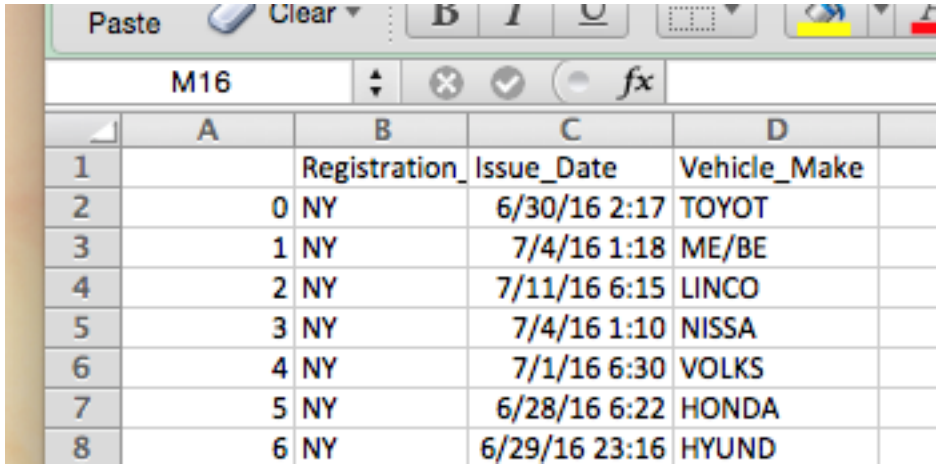
Pandas – read_csv

```
pd.read_csv?
```

```
    parameter ignores commented lines and empty lines if
    ``skip_blank_lines=True``, so header=0 denotes the first line of
    data rather than the first line of the file.
names : array-like, default None
    List of column names to use. If file contains no header row, then you
    should explicitly pass header=None. Duplicates in this list will cause
    a ``UserWarning`` to be issued.
index_col : int or sequence or False, default None
    Column to use as the row labels of the DataFrame. If a sequence is given, a
    MultiIndex is used. If you have a malformed file with delimiters at the end
    of each line, you might consider index_col=False to force pandas to _not_
    use the first column as the index (row names)
usecols : array-like or callable, default None
    Return a subset of the columns. If array-like, all elements must either
    be positional (i.e. integer indices into the document columns) or strings
    that correspond to column names provided either by the user in `names` or
    inferred from the document header row(s). For example, a valid array-like
    `usecols` parameter would be [0, 1, 2] or ['foo', 'bar', 'baz'].
```

Parking Ticket Data

I have the following data saved in the file “Parking.csv”:



	A	B	C	D
1		Registration_	Issue_Date	Vehicle_Make
2	0	NY	6/30/16 2:17	TOYOT
3	1	NY	7/4/16 1:18	ME/BE
4	2	NY	7/11/16 6:15	LINCO
5	3	NY	7/4/16 1:10	NISSA
6	4	NY	7/1/16 6:30	VOLKS
7	5	NY	6/28/16 6:22	HONDA
8	6	NY	6/29/16 23:16	HYUND

Let’s see what happens when we read in the data

```
import pandas as pd

df_parking = pd.read_csv("Data/Parking.csv", index_col=0)
df_parking.head()
```

	Registration_State	Issue_Date	Vehicle_Make
0	NY	6/30/16 2:17	TOYOT
1	NY	7/4/16 1:18	ME/BE
2	NY	7/11/16 6:15	LINCO
3	NY	7/4/16 1:10	NISSA
4	NY	7/1/16 6:30	VOLKS

Specifies which column in the file should be the index col.

Parking Ticket Data

```
import pandas as pd

df_parking = pd.read_csv("Data/Parking.csv", index_col=0)
df_parking.head()
```

	Registration_State	Issue_Date	Vehicle_Make
0	NY	6/30/16 2:17	TOYOT
1	NY	7/4/16 1:18	ME/BE
2	NY	7/11/16 6:15	LINCO
3	NY	7/4/16 1:10	NISSA
4	NY	7/1/16 6:30	VOLKS

```
df_parking.dtypes
```

```
Registration_State    object
Issue_Date            object
Vehicle_Make          object
dtype: object
```

All columns are stored as strings!

Parking Ticket Data

```
import pandas as pd

df_parking = pd.read_csv("Data/Parking.csv", index_col=0)
df_parking.head()
```

	Registration_State	Issue_Date	Vehicle_Make
0	NY	6/30/16 2:17	TOYOT
1	NY	7/4/16 1:18	ME/BE
2	NY	7/11/16 6:15	LINCO
3	NY	7/4/16 1:10	NISSA
4	NY	7/1/16 6:30	VOLKS

```
df_parking.dtypes
```

```
Registration_State    object
Issue_Date            object
Vehicle_Make          object
dtype: object
```

All columns are stored as strings!

Two options to convert Issue_Date column to datetime:

1. Tell pandas Issue_Date is a datetime when reading in the data.
2. Convert the column after having read in the data.

Pandas – read_csv

```
pd.read_csv?
```

```
skip_blank_lines : boolean, default True
    If True, skip over blank lines rather than interpreting as NaN values
parse_dates : boolean or list of ints or names or list of lists or dict, default False

    * boolean. If True -> try parsing the index.
    * list of ints or names. e.g. If [1, 2, 3] -> try parsing columns 1, 2, 3
      each as a separate date column.
    * list of lists. e.g.  If [[1, 3]] -> combine columns 1 and 3 and parse as
      a single date column.
    * dict, e.g. {'foo' : [1, 3]} -> parse columns 1, 3 as date and call result
      'foo'

    If a column or index contains an unparseable date, the entire column or
    index will be returned unaltered as an object data type. For non-standard
    datetime parsing, use ``pd.to_datetime`` after ``pd.read_csv``

    Note: A fast-path exists for iso8601-formatted dates.
infer_datetime_format : boolean, default False
    If True and `parse_dates` is enabled, pandas will attempt to infer the
    format of the datetime strings in the columns, and if it can be inferred,
    switch to a faster method of parsing them. In some cases this can increase
```

Converting to Datetime

Option 1:

```
import pandas as pd

df_parking = pd.read_csv("Data/Parking.csv", index_col=0,\
                         parse_dates = ["Issue_Date"])
df_parking.head()
```

	Registration_State	Issue_Date	Vehicle_Make
0	NY	2016-06-30 02:17:00	TOYOT
1	NY	2016-07-04 01:18:00	ME/BE
2	NY	2016-07-11 06:15:00	LINCO
3	NY	2016-07-04 01:10:00	NISSA
4	NY	2016-07-01 06:30:00	VOLKS

```
df_parking.dtypes
```

```
Registration_State            object
Issue_Date            datetime64[ns]
Vehicle_Make                  object
dtype: object
```

Now Issue_Date is stored as a datetime!

Converting to Datetime

Option 1:

```
import pandas as pd

df_parking = pd.read_csv("Data/Parking.csv", index_col=0,\
                         parse_dates = ["Issue_Date"])
df_parking.head()
```

	Registration_State	Issue_Date	Vehicle_Make
0	NY	2016-06-30 02:17:00	TOYOT
1	NY	2016-07-04 01:18:00	ME/BE
2	NY	2016-07-11 06:15:00	LINCO
3	NY	2016-07-04 01:10:00	NISSA
4	NY	2016-07-01 06:30:00	VOLKS

- List of column names that you want read in as datetimes.
- Pandas usually just figures out format.

```
df_parking.dtypes
```

```
Registration_State              object
Issue_Date              datetime64[ns]
Vehicle_Make                    object
dtype: object
```

Now Issue_Date is stored as a datetime!

Converting to Datetime

Option 2:

```
df_parking.head()
```

	Registration_State	Issue_Date	Vehicle_Make
0	NY	6/30/16 2:17	TOYOT
1	NY	7/4/16 1:18	ME/BE
2	NY	7/11/16 6:15	LINCO
3	NY	7/4/16 1:10	NISSA
4	NY	7/1/16 6:30	VOLKS

Wrap column in to_datetime() built in function

```
#Reassign Issue Date
df_parking["Issue_Date"] = pd.to_datetime(df_parking["Issue_Date"])
df_parking.head()
```

	Registration_State	Issue_Date	Vehicle_Make
0	NY	2016-06-30 02:17:00	TOYOT
1	NY	2016-07-04 01:18:00	ME/BE
2	NY	2016-07-11 06:15:00	LINCO
3	NY	2016-07-04 01:10:00	NISSA
4	NY	2016-07-01 06:30:00	VOLKS

Timestamp Attributes

	Registration_State	Issue_Date	Vehicle_Make
0	NY	2016-06-30 02:17:00	TOYOT
1	NY	2016-07-04 01:18:00	ME/BE
2	NY	2016-07-11 06:15:00	LINCO
3	NY	2016-07-04 01:10:00	NISSA
4	NY	2016-07-01 06:30:00	VOLKS

```
#Get hour
first_datetime = df_parking.loc[0,"Issue_Date"]
first_datetime
```

```
Timestamp('2016-06-30 02:17:00')
```

Timestamp Attributes

```
df_parking.head()
```

	Registration_State	Issue_Date	Vehicle_Make
0	NY	2016-06-30 02:17:00	TOYOT
1	NY	2016-07-04 01:18:00	ME/BE
2	NY	2016-07-11 06:15:00	LINCO
3	NY	2016-07-04 01:10:00	NISSA
4	NY	2016-07-01 06:30:00	VOLKS

```
first_datetime = df_parking.loc[0,"Issue_Date"]
first_datetime
```

```
Timestamp('2016-06-30 02:17:00')
```

```
first_datetime.hour
```

```
2
```

```
first_datetime.dayofweek
```

```
3
```

```
first_datetime.is_leap_year
```

```
True
```

Creating New Columns from Datetime Column

	Registration_State	Issue_Date	Vehicle_Make
0	NY	2016-06-30 02:17:00	TOYOT
1	NY	2016-07-04 01:18:00	ME/BE
2	NY	2016-07-11 06:15:00	LINCO
3	NY	2016-07-04 01:10:00	NISSA
4	NY	2016-07-01 06:30:00	VOLKS

```
df_parking["DOW"] = df_parking["Issue_Date"].dt.weekday_name
df_parking["Month"] = df_parking["Issue_Date"].dt.month
df_parking.head()
```

	Registration_State	Issue_Date	Vehicle_Make	DOW	Month
0	NY	2016-06-30 02:17:00	TOYOT	Thursday	6
1	NY	2016-07-04 01:18:00	ME/BE	Monday	7
2	NY	2016-07-11 06:15:00	LINCO	Monday	7
3	NY	2016-07-04 01:10:00	NISSA	Monday	7
4	NY	2016-07-01 06:30:00	VOLKS	Friday	7

Need .dt if we are applying datetime attributes to a series.

Pandas – Part 3

Changing Column Names

```
import pandas as pd

df_grades = pd.read_csv("Data/Grades_Short.csv")
df_grades
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

```
#Get column names
df_grades.columns
```

```
Index(['Name', 'Previous_Part', 'Participation1', 'Mini_Exam1', 'Mini_Exam2',
       'Participation2', 'Mini_Exam3', 'Final', 'Grade'],
      dtype='object')
```

Changing Column Names

```
#Get column names
df_grades.columns
```

```
Index(['Name', 'Previous_Part', 'Participation1', 'Mini_Exam1', 'Mini_Exam2',
       'Participation2', 'Mini_Exam3', 'Final', 'Grade'],
      dtype='object')
```

```
#Changing column names
df_grades.rename(columns={"Mini_Exam1": "Mini_Exam_1","Mini_Exam2": "Mini_Exam_2"}, \
                          inplace = False)
```

Changing Column Names

```
df_grades.rename?
```

```
Signature: df_grades.rename(mapper=None, index=None, columns=None, axis=None, copy=True, inplace=False, level=None)
Docstring:
Alter axes labels.

Function / dict values must be unique (1-to-1). Labels not contained in
a dict / Series will be left as-is. Extra labels listed don't throw an
error.

See the :ref:`user guide <basics.rename>` for more.

Parameters
----------
mapper, index, columns : dict-like or function, optional
    dict-like or functions transformations to apply to
    that axis' values. Use either ``mapper`` and ``axis`` to
    specify the axis to target with ``mapper``, or ``index`` and
    ``columns``.
axis : int or str, optional
    Axis to target with ``mapper``. Can be either the axis name
    ('index', 'columns') or number (0, 1). The default is 'index'.
copy : boolean, default True
    Also copy underlying data
inplace : boolean, default False
    Whether to return a new %(klass)s. If True then value of copy is
    ignored.
level : int or level name, default None
    In case of a MultiIndex, only rename labels in the specified
    level.
```

Changing Column Names

```
#Get column names
df_grades.columns
```

```
Index(['Name', 'Previous_Part', 'Participation1', 'Mini_Exam1', 'Mini_Exam2',
       'Participation2', 'Mini_Exam3', 'Final', 'Grade'],
      dtype='object')
```

```
#Changing column names
df_grades.rename(columns={"Mini_Exam1": "Mini_Exam_1","Mini_Exam2": "Mini_Exam_2"}, \
                          inplace = False)
```

"curly" brackets around new column name assignments

Changing Column Names

```
#Get column names
df_grades.columns
```

```
Index(['Name', 'Previous_Part', 'Participation1', 'Mini_Exam1', 'Mini_Exam2',
       'Participation2', 'Mini_Exam3', 'Final', 'Grade'],
      dtype='object')
```

```
#Changing column names
df_grades.rename(columns={"Mini_Exam1": "Mini_Exam_1","Mini_Exam2": "Mini_Exam_2"}, \
                          inplace = False)
```

“old_column_name”: “new_column_name”

Changing Column Names

```
#Get column names
df_grades.columns
```

```
Index(['Name', 'Previous_Part', 'Participation1', 'Mini_Exam1', 'Mini_Exam2',
       'Participation2', 'Mini_Exam3', 'Final', 'Grade'],
      dtype='object')
```

```
#Changing column names
df_grades.rename(columns={"Mini_Exam1": "Mini_Exam_1","Mini_Exam2": "Mini_Exam_2"}, \
                          inplace = False)
```

inplace = False (default) returns a new dataframe
(df_grades is unaltered) with updated column names.

Changing Column Names

```
#Get column names
df_grades.columns
```

```
Index(['Name', 'Previous_Part', 'Participation1', 'Mini_Exam1', 'Mini_Exam2',
       'Participation2', 'Mini_Exam3', 'Final', 'Grade'],
      dtype='object')
```

```
#Changing column names
df_grades.rename(columns={"Mini_Exam1": "Mini_Exam_1","Mini_Exam2": "Mini_Exam_2"}, \
                          inplace = False)
```

	Name	Previous_Part	Participation1	Mini_Exam_1	Mini_Exam_2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

Concatenating DataFrames - Stacked

```
df_grades_A
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Sol	31.0	1	22.0	13	1	13.0	34.0	A
3	Chris	30.0	1	19.0	17	1	12.5	33.5	A
4	Malik	31.5	1	20.0	21	1	9.0	36.0	A

```
df_grades_other
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
1	Tarik	31.0	1	19.0	19	1	8.0	24.0	B

Let's say you had separate csv files with the info for the students who got an A and everyone else, but you want to analyze everything together.

Concatenating DataFrames - Stacked

```
df_grades = pd.concat([df_grades_A,df_grades_other],\
                      axis = 0)
```

List of dataframes to concatenate

Concatenating DataFrames - Stacked

```
pd.concat?
```

```
Signature: pd.concat(objs, axis=0, join='outer', join_axes=None, ignore_index=False, keys=None, levels=None, names=None, verify_integr
ity=False, copy=True)
Docstring:
Concatenate pandas objects along a particular axis with optional set logic
along the other axes.

Can also add a layer of hierarchical indexing on the concatenation axis,
which may be useful if the labels are the same (or overlapping) on
the passed axis number.

Parameters
----------
objs : a sequence or mapping of Series, DataFrame, or Panel objects
    If a dict is passed, the sorted keys will be used as the `keys`
    argument, unless it is passed, in which case the values will be
    selected (see below). Any None objects will be dropped silently unless
    they are all None in which case a ValueError will be raised
axis : {0/'index', 1/'columns'}, default 0
    The axis to concatenate along
join : {'inner', 'outer'}, default 'outer'
    How to handle indexes on other axis(es)
join_axes : list of Index objects
    Specific indexes to use for the other n - 1 axes instead of performing
    inner/outer set logic
ignore_index : boolean, default False
    If True, do not use the index values along the concatenation axis. The
    resulting axis will be labeled 0, ..., n - 1. This is useful if you are
    concatenating objects where the concatenation axis does not have
    meaningful indexing information. Note the index values on the other
    axes are still respected in the join.
```

Concatenating DataFrames - Stacked

```
df_grades = pd.concat([df_grades_A,df_grades_other],\
                      axis = 0)
```

axis = 0 (default) – combine the two dataframes by stacking them on top of each other. Set axis =1 to stack side by side.

Concatenating DataFrames - Stacked

```
df_grades_A
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Sol	31.0	1	22.0	13	1	13.0	34.0	A
3	Chris	30.0	1	19.0	17	1	12.5	33.5	A
4	Malik	31.5	1	20.0	21	1	9.0	36.0	A

```
df_grades_other
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
1	Tarik	31.0	1	19.0	19	1	8.0	24.0	B

```
df_grades = pd.concat([df_grades_A,df_grades_other],\
                      axis = 0)
```

- # of columns has to match
- What is going to happen to index?

Concatenating DataFrames - Stacked

```
df_grades = pd.concat([df_grades_A,df_grades_other],\
                      axis = 0)
```

```
df_grades
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Sol	31.0	1	22.0	13	1	13.0	34.0	A
3	Chris	30.0	1	19.0	17	1	12.5	33.5	A
4	Malik	31.5	1	20.0	21	1	9.0	36.0	A
0	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
1	Tarik	31.0	1	19.0	19	1	8.0	24.0	B

- We got a repeated index, which is not good!
- We will eventually see how to “reset” the index.
- We can also correct this when we concatenate.

Concatenating DataFrames - Stacked

```
pd.concat?
```

```
Signature: pd.concat(objs, axis=0, join='outer', join_axes=None, ignore_index=False, keys=None, levels=None, names=None, verify_integr
ity=False, copy=True)
Docstring:
Concatenate pandas objects along a particular axis with optional set logic
along the other axes.

Can also add a layer of hierarchical indexing on the concatenation axis,
which may be useful if the labels are the same (or overlapping) on
the passed axis number.

Parameters
----------
objs : a sequence or mapping of Series, DataFrame, or Panel objects
    If a dict is passed, the sorted keys will be used as the `keys`
    argument, unless it is passed, in which case the values will be
    selected (see below). Any None objects will be dropped silently unless
    they are all None in which case a ValueError will be raised
axis : {0/'index', 1/'columns'}, default 0
    The axis to concatenate along
join : {'inner', 'outer'}, default 'outer'
    How to handle indexes on other axis(es)
join_axes : list of Index objects
    Specific indexes to use for the other n - 1 axes instead of performing
    inner/outer set logic
ignore_index : boolean, default False
    If True, do not use the index values along the concatenation axis. The
    resulting axis will be labeled 0, ..., n - 1. This is useful if you are
    concatenating objects where the concatenation axis does not have
    meaningful indexing information. Note the index values on the other
    axes are still respected in the join.
```

Concatenating DataFrames - Stacked

```
df_grades = pd.concat([df_grades_A,df_grades_other],\
                      axis = 0, ignore_index= True)
```

```
df_grades
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Sol	31.0	1	22.0	13	1	13.0	34.0	A
3	Chris	30.0	1	19.0	17	1	12.5	33.5	A
4	Malik	31.5	1	20.0	21	1	9.0	36.0	A
5	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
6	Tarik	31.0	1	19.0	19	1	8.0	24.0	B

Notice the ignore_index input!

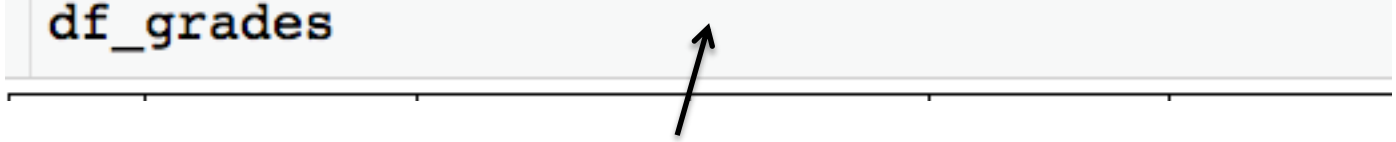
Using the Index

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

- The index in this case is row numbers.
- What if I want to quickly see Joe's row?
  - I have to look up what row Joe is in.
  - Instead, I can make the index the column name.

Using the Index

```
df_grades.set_index("Name", inplace=True)
df_grades
```



Column that will become index (make sure this is unique).

Set_index

```
df_grades.set_index?
```

```
Signature: df_grades.set_index(keys, drop=True, append=False, inplace=False, verify_integrity=False)
Docstring:
Set the DataFrame index (row labels) using one or more existing
columns. By default yields a new object.

Parameters
----------
keys : column label or list of column labels / arrays
drop : boolean, default True
    Delete columns to be used as the new index
append : boolean, default False
    Whether to append columns to existing index
inplace : boolean, default False
    Modify the DataFrame in place (do not create a new object)
verify_integrity : boolean, default False
    Check the new index for duplicates. Otherwise defer the check until
    necessary. Setting to False will improve the performance of this
    method
```

Using the Index

```
df_grades.set_index("Name", inplace=True)
df_grades
```

Modify df_grades

Using the Index

```
df_grades.set_index("Name", inplace=True)
df_grades
```

	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
Name								
Jake	32.0	1	19.5	20	1	10.0	33.0	A
Joe	32.0	1	20.0	16	1	14.0	32.0	A
Susan	30.0	1	19.0	19	1	10.5	33.0	A-
Sol	31.0	1	22.0	13	1	13.0	34.0	A
Chris	30.0	1	19.0	17	1	12.5	33.5	A
Tarik	31.0	1	19.0	19	1	8.0	24.0	B
Malik	31.5	1	20.0	21	1	9.0	36.0	A

The index is now the name column!

Using the Index

```
df_grades.set_index("Name", inplace=True)
df_grades
```

	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
Name								
Jake	32.0	1	19.5	20	1	10.0	33.0	A
Joe	32.0	1	20.0	16	1	14.0	32.0	A
Susan	30.0	1	19.0	19	1	10.5	33.0	A-
Sol	31.0	1	22.0	13	1	13.0	34.0	A
Chris	30.0	1	19.0	17	1	12.5	33.5	A
Tarik	31.0	1	19.0	19	1	8.0	24.0	B
Malik	31.5	1	20.0	21	1	9.0	36.0	A

```
#Easy to find Joe's grade
df_grades.loc["Joe", "Grade"]
```

```
'A'
```

Using the Index

	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	ID
Name									
Jake	32.0	1	19.5	20	1	10.0	33.0	A	90743
Joe	32.0	1	20.0	16	1	14.0	32.0	A	7284
Susan	30.0	1	19.0	19	1	10.5	33.0	A-	7625
Sol	31.0	1	22.0	13	1	13.0	34.0	A	1237
Chris	30.0	1	19.0	17	1	12.5	33.5	A	62
Tarik	31.0	1	19.0	19	1	8.0	24.0	B	87452
Malik	31.5	1	20.0	21	1	9.0	36.0	A	9374

Want to make new ID column the index

Using the Index

	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	ID
Name									
Jake	32.0	1	19.5	20	1	10.0	33.0	A	90743
Joe	32.0	1	20.0	16	1	14.0	32.0	A	7284
Susan	30.0	1	19.0	19	1	10.5	33.0	A-	7625
Sol	31.0	1	22.0	13	1	13.0	34.0	A	1237
Chris	30.0	1	19.0	17	1	12.5	33.5	A	62
Tarik	31.0	1	19.0	19	1	8.0	24.0	B	87452
Malik	31.5	1	20.0	21	1	9.0	36.0	A	9374

```
df_grades.set_index("ID", inplace=False)
```

	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
ID								
90743	32.0	1	19.5	20	1	10.0	33.0	A
7284	32.0	1	20.0	16	1	14.0	32.0	A
7625	30.0	1	19.0	19	1	10.5	33.0	A-
1237	31.0	1	22.0	13	1	13.0	34.0	A
62	30.0	1	19.0	17	1	12.5	33.5	A
87452	31.0	1	19.0	19	1	8.0	24.0	B
9374	31.5	1	20.0	21	1	9.0	36.0	A

This accomplishes the task but we lose the name column.

Using the Index

	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	ID
Name									
Jake	32.0	1	19.5	20	1	10.0	33.0	A	90743
Joe	32.0	1	20.0	16	1	14.0	32.0	A	7284
Susan	30.0	1	19.0	19	1	10.5	33.0	A-	7625
Sol	31.0	1	22.0	13	1	13.0	34.0	A	1237
Chris	30.0	1	19.0	17	1	12.5	33.5	A	62
Tarik	31.0	1	19.0	19	1	8.0	24.0	B	87452
Malik	31.5	1	20.0	21	1	9.0	36.0	A	9374

```
#Add names column
df_grades["Name"] = df_grades.index
#Get new ID indes
df_grades.set_index("ID", inplace=True)
df_grades
```

	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Name
ID									
90743	32.0	1	19.5	20	1	10.0	33.0	A	Jake
7284	32.0	1	20.0	16	1	14.0	32.0	A	Joe
7625	30.0	1	19.0	19	1	10.5	33.0	A-	Susan
1237	31.0	1	22.0	13	1	13.0	34.0	A	Sol
62	30.0	1	19.0	17	1	12.5	33.5	A	Chris
87452	31.0	1	19.0	19	1	8.0	24.0	B	Tarik
9374	31.5	1	20.0	21	1	9.0	36.0	A	Malik

Resetting Index

```
df_grades
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
ID									
90743	Jake	32.0	1	19.5	20	1	10.0	33.0	A
7284	Joe	32.0	1	20.0	16	1	14.0	32.0	A
7625	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
1237	Sol	31.0	1	22.0	13	1	13.0	34.0	A
62	Chris	30.0	1	19.0	17	1	12.5	33.5	A
87452	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
9374	Malik	31.5	1	20.0	21	1	9.0	36.0	A

How do I go back to having row numbers?

Resetting Index

```
df_grades.reset_index?
```

```
Signature: df_grades.reset_index(level=None, drop=False, inplace=False, col_level=0, col_fill='')
Docstring:
For DataFrame with multi-level index, return new DataFrame with
labeling information in the columns under the index names, defaulting
to 'level_0', 'level_1', etc. if any are None. For a standard index,
the index name will be used (if set), otherwise a default 'index' or
'level_0' (if 'index' is already taken) will be used.

Parameters
----------
level : int, str, tuple, or list, default None
    Only remove the given levels from the index. Removes all levels by
    default
drop : boolean, default False
    Do not try to insert index into dataframe columns. This resets
    the index to the default integer index.
inplace : boolean, default False
    Modify the DataFrame in place (do not create a new object)
col_level : int or str, default 0
    If the columns have multiple levels, determines which level the
    labels are inserted into. By default it is inserted into the first
    level.
col_fill : object, default ''
    If the columns have multiple levels, determines how the other
    levels are named. If None then the index name is repeated.
```

Using the Index

```
#First reset the index
df_grades.reset_index(drop=False, inplace=True)
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	ID
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A	90743
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A	7284
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-	7625
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A	1237
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A	62
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B	87452
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A	9374

- reset_index() will make your index row numbers again.
- Useful when manipulating dataframes and index can get messed up

Missing Data

	A	B	C	D	E	F	G	H	I	J
	Name	Previous_Par	Participation	Mini_Exam1	Mini_Exam2	Participation	Mini_Exam3	Final	Grade	Temp
	Jake	32	1	19.5	20	1	10	33	A	-1
	Joe	NA	1	20	16	1	14	32	A	23
	Sol	31	1	22	13	1	13	34	A	34
	Chris	30	-1	19	not available	1	12.5	33.5	A	72

```
df_missing = pd.read_csv("Data/Missing_Data.csv")
df_missing
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A	-1
1	Joe	NaN	1	20.0	16	1	14.0	32.0	A	23
2	Sol	31.0	1	22.0	13	1	13.0	34.0	A	34
3	Chris	30.0	-1	19.0	not available	1	12.5	33.5	A	72

Not that different columns have different indicators for missing data.

Missing Data

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A	-1
1	Joe	NaN	1	20.0	16	1	14.0	32.0	A	23
2	Sol	31.0	1	22.0	13	1	13.0	34.0	A	34
3	Chris	30.0	-1	19.0	not available	1	12.5	33.5	A	72

```
df_missing.dtypes
```

```
Name              object
Previous_Part    float64
Participation1     int64
Mini_Exam1       float64
Mini_Exam2        object
Participation2     int64
Mini_Exam3       float64
Final            float64
Grade             object
Temp               int64
dtype: object
```

We can replace the missing data with a true NaN (right now everything is just a string).

Missing Data

```
df_missing = pd.read_csv("Data/Missing_Data.csv", \
                         na_values=["NaN", "not available"])
df_missing
```

List of strings specifying which values are missing.

Missing Data

```
pd.read_csv?
```

```
nrows : int, default None
    Number of rows of file to read. Useful for reading pieces of large files
na_values : scalar, str, list-like, or dict, default None
    Additional strings to recognize as NA/NaN. If dict passed, specific
    per-column NA values.  By default the following values are interpreted as
    NaN: '', '#N/A', '#N/A N/A', '#NA', '-1.#IND', '-1.#QNAN', '-NaN', '-nan',
    '1.#IND', '1.#QNAN', 'N/A', 'NA', 'NULL', 'NaN', 'n/a', 'nan',
    'null'.
keep_default_na : bool, default True
    If na_values are specified and keep_default_na is False the default NaN
    values are overridden, otherwise they're appended to.
```

Missing Data

```
df_missing = pd.read_csv("Data/Missing_Data.csv", \
                         na_values=["NaN", "not available"])
df_missing
```

List of strings specifying which values are missing.

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	Jake	32.0	1	19.5	20.0	1	10.0	33.0	A	-1
1	Joe	NaN	1	20.0	16.0	1	14.0	32.0	A	23
2	Sol	31.0	1	22.0	13.0	1	13.0	34.0	A	34
3	Chris	30.0	-1	19.0	NaN	1	12.5	33.5	A	72

Special NaN value (from numpy package), which is not a string.

Missing Data

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A	-1
1	Joe	NaN	1	20.0	16	1	14.0	32.0	A	23
2	Sol	31.0	1	22.0	13	1	13.0	34.0	A	34
3	Chris	30.0	-1	19.0	not available	1	12.5	33.5	A	72

We know “NaN” and “not available” are missing data points, but what about -1?

Missing Data

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A	-1
1	Joe	NaN	1	20.0	16	1	14.0	32.0	A	23
2	Sol	31.0	1	22.0	13	1	13.0	34.0	A	34
3	Chris	30.0	-1	19.0	not available	1	12.5	33.5	A	72

We know “NaN” and “not available” are missing data points, but what about -1?

- For the Participation1 column the -1 is probably missing data.
- For the Temp column, the -1 is likely not missing data, since -1 is a valid temperature.

For each column, we can specify exactly which values correspond to missing data.

Missing Data

```
df_missing = pd.read_csv("Data/Missing_Data.csv", na_values={"Mini_Exam2" : "not available",\
                                                             "Participation1": -1})

df_missing
```

Curly brackets

Missing Data

```
df_missing = pd.read_csv("Data/Missing_Data.csv", na_values={"Mini_Exam2" : "not available",\
                                                             "Participation1": -1})
df_missing
```

Column name as string

NaN value

Missing Data

```
df_missing = pd.read_csv("Data/Missing_Data.csv", na_values={"Mini_Exam2" : "not available",\
                                                             "Participation1": -1})
df_missing
```

Column name as string

NaN value

“For column Participation1, replace all -1s with a NaN.”

Missing Data

```
df_missing = pd.read_csv("Data/Missing_Data.csv", na_values={"Mini_Exam2" : "not available",\
                                                             "Participation1": -1})
df_missing
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	Jake	32.0	1.0	19.5	20.0	1	10.0	33.0	A	-1
1	Joe	NaN	1.0	20.0	16.0	1	14.0	32.0	A	23
2	Sol	31.0	1.0	22.0	13.0	1	13.0	34.0	A	34
3	Chris	30.0	NaN	19.0	NaN	1	12.5	33.5	A	72

Notice that the -1 was replaced only in Participation1 column

Comparing Approaches

Approach 1:

- Does a global search and replace in all columns.

```
df_missing = pd.read_csv("Data/Missing_Data.csv", \
                         na_values=["NaN", "not available"])
df_missing
```

Approach 2:

- Allows you to specify column by column the values that should be replaced with NaN.

```
df_missing = pd.read_csv("Data/Missing_Data.csv", na_values={"Mini_Exam2" : "not available",\
                                                             "Participation1": -1})
df_missing
```

Benefiting of Having NaNs

- Have common symbol for where there is missing data
    - Good for you and good for others looking at your code/data
    - These entries will be ignored if you try to compute means of columns with NaNs.
- We can easily get rid of column/rows with missing data
- We can easily replace the missing values with the mean of the column, for example.

Isnull() Method

- The isnull() method lets you check where the NaNs are:

```
df = pd.read_csv("Data/Missing_Data.csv", na_values=["NaN", -1, "not available"])
df
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	Jake	32.0	1.0	19.5	20.0	1	10.0	33.0	A	NaN
1	Joe	NaN	1.0	20.0	16.0	1	14.0	32.0	A	23.0
2	Sol	31.0	1.0	22.0	13.0	1	13.0	34.0	A	34.0
3	Chris	30.0	NaN	19.0	NaN	1	12.5	33.5	A	72.0

```
#Using isnull()
df.isnull()
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	False	False	False	False	False	False	False	False	False	True
1	False	True	False	False	False	False	False	False	False	False
2	False	False	False	False	False	False	False	False	False	False
3	False	False	True	False	True	False	False	False	False	False

Isnull() Method

- The isnull() method lets you check where the NaNs are:

```
#Using isnull()
df.isnull()
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	False	False	False	False	False	False	False	False	False	True
1	False	True	False	False	False	False	False	False	False	False
2	False	False	False	False	False	False	False	False	False	False
3	False	False	True	False	True	False	False	False	False	False

```
#Remember Booleans are just 0s and 1s.
#Check how many NaNs are in each column
df.isnull().sum()
```

```
Name              0
Previous_Part     1
Participation1    1
Mini_Exam1        0
Mini_Exam2        1
Participation2    0
Mini_Exam3        0
Final             0
Grade             0
Temp              1
dtype: int64
```

Dropna() Method

```
df_missing = pd.read_csv("Data/Missing_Data.csv", na_values=["NaN", "not available",\
                                                             -1])
df_missing
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	Jake	32.0	1.0	19.5	20.0	1	10.0	33.0	A	NaN
1	Joe	NaN	1.0	20.0	16.0	1	14.0	32.0	A	23.0
2	Sol	31.0	1.0	22.0	13.0	1	13.0	34.0	A	34.0
3	Chris	30.0	NaN	19.0	NaN	1	12.5	33.5	A	72.0

How do I get rid of all rows with NaN?

Dropna() Method

```
df_missing = pd.read_csv("Data/Missing_Data.csv", na_values=["NaN", "not available",\
                                                             -1])
df_missing
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	Jake	32.0	1.0	19.5	20.0	1	10.0	33.0	A	NaN
1	Joe	NaN	1.0	20.0	16.0	1	14.0	32.0	A	23.0
2	Sol	31.0	1.0	22.0	13.0	1	13.0	34.0	A	34.0
3	Chris	30.0	NaN	19.0	NaN	1	12.5	33.5	A	72.0

How do I get rid of all rows with NaN?

```
df_missing.dropna(axis = 0, inplace=False)
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
2	Sol	31.0	1.0	22.0	13.0	1	13.0	34.0	A	34.0

- Setting axis = 1 would drop all columns with an NaN

Dropna() Method

```
df_missing.dropna?
```

```
Signature: df_missing.dropna(axis=0, how='any', thresh=None, subset=None, inplace=False)
Docstring:
Return object with labels on given axis omitted where alternately any
or all of the data are missing

Parameters
----------
axis : {0 or 'index', 1 or 'columns'}, or tuple/list thereof
    Pass tuple or list to drop on multiple axes
how : {'any', 'all'}
    * any : if any NA values are present, drop that label
    * all : if all values are NA, drop that label
thresh : int, default None
    int value : require that many non-NA values
subset : array-like
    Labels along other axis to consider, e.g. if you are dropping rows
    these would be a list of columns to include
inplace : boolean, default False
    If True, do operation inplace and return None.
```

Fillna() Method

```
df_missing = pd.read_csv("Data/Missing_Data.csv", na_values=["NaN", "not available",\
                                                             -1])
df_missing
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	Jake	32.0	1.0	19.5	20.0	1	10.0	33.0	A	NaN
1	Joe	NaN	1.0	20.0	16.0	1	14.0	32.0	A	23.0
2	Sol	31.0	1.0	22.0	13.0	1	13.0	34.0	A	34.0
3	Chris	30.0	NaN	19.0	NaN	1	12.5	33.5	A	72.0

Rather than getting rid of rows/columns, we fill the “holes” in a number of ways.

```
#Replace with specific value
df_missing.fillna(0, inplace=False)
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	Jake	32.0	1.0	19.5	20.0	1	10.0	33.0	A	0.0
1	Joe	0.0	1.0	20.0	16.0	1	14.0	32.0	A	23.0
2	Sol	31.0	1.0	22.0	13.0	1	13.0	34.0	A	34.0
3	Chris	30.0	0.0	19.0	0.0	1	12.5	33.5	A	72.0

Fillna() Method

```
df_missing.fillna?
```

```
Signature: df_missing.fillna(value=None, method=None, axis=None, inplace=False, limit=None, downcast=None, **kwargs)
Docstring:
Fill NA/NaN values using the specified method

Parameters
----------
value : scalar, dict, Series, or DataFrame
    Value to use to fill holes (e.g. 0), alternately a
    dict/Series/DataFrame of values specifying which value to use for
    each index (for a Series) or column (for a DataFrame). (values not
    in the dict/Series/DataFrame will not be filled). This value cannot
    be a list.
method : {'backfill', 'bfill', 'pad', 'ffill', None}, default None
    Method to use for filling holes in reindexed Series
    pad / ffill: propagate last valid observation forward to next valid
    backfill / bfill: use NEXT valid observation to fill gap
axis : {0 or 'index', 1 or 'columns'}
inplace : boolean, default False
    If True, fill in place. Note: this will modify any
    other views on this object, (e.g. a no-copy slice for a column in a
    DataFrame).
limit : int, default None
    If method is specified, this is the maximum number of consecutive
    NaN values to forward/backward fill. In other words, if there is
    a gap with more than this number of consecutive NaNs, it will only
    be partially filled. If method is not specified, this is the
    maximum number of entries along the entire axis where NaNs will be
    filled. Must be greater than 0 if not None.
downcast : dict, default is None
    a dict of item->dtype of what to downcast if possible,
    or the string 'infer' which will try to downcast to an appropriate
    equal type (e.g. float64 to int64 if possible)
```

Fillna() Method

```
df_missing = pd.read_csv("Data/Missing_Data.csv", na_values=["NaN", "not available",\
                                                             -1])
df_missing
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	Jake	32.0	1.0	19.5	20.0	1	10.0	33.0	A	NaN
1	Joe	NaN	1.0	20.0	16.0	1	14.0	32.0	A	23.0
2	Sol	31.0	1.0	22.0	13.0	1	13.0	34.0	A	34.0
3	Chris	30.0	NaN	19.0	NaN	1	12.5	33.5	A	72.0

Rather than getting rid of rows/columns, we fill the “holes” in a number of ways.

```
#Replace with specific value in specific column
mean_temp = df_missing.Temp.mean()
df_missing.fillna({'Temp': mean_temp}, inplace=False)
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	Jake	32.0	1.0	19.5	20.0	1	10.0	33.0	A	43.0
1	Joe	NaN	1.0	20.0	16.0	1	14.0	32.0	A	23.0
2	Sol	31.0	1.0	22.0	13.0	1	13.0	34.0	A	34.0
3	Chris	30.0	NaN	19.0	NaN	1	12.5	33.5	A	72.0