COMP41680

Data Preparation and Manipulation with Pandas

Slides by Derek Greene

UCD School of Computer Science

Spring 2018

Overview

- Data Preparation
- Clean v Noisy Data
- Pandas Package for Python
- Pandas Series
- Pandas Data Frames
- Loading Data
- Working with Data Frames
- Normalising Data
- Aggregating Data
- Handling Missing Values

Data Preparation

- 80% of a typical data science project is cleaning and preparing the data, while the remaining 20% is actual data analysis
  - New York Times, 2014 (http://nyti.ms/1p3Zoql)
- Typically involves one or more of the following tasks:
  - Data cleaning: Fix erroneous feature values in the raw data.
  - Data selection: Filter the full dataset to find a useful subset to work with, removing noisy cases.
  - Duplicate elimination: Remove duplicate cases.
  - Normalisation: Scale numeric values to conform to minimum and maximum values.
  - Handling missing values: Many real datasets have missing values, either because it exists and was not collected or it never existed.
  - Data integration: Match up data for related cases from multiple different raw data sources.

Noisy Data v Clean Data

- **Example:** Raw dataset of metadata for election candidates, versus cleaned version of the same data.

Lastname	Firstname	Gender	Party	Constituency	DOB
Ryan	Noel	M	FF	Dun Laoghaire	1965-11-2
Lisa Lynch		Female		Rathmines	3 Feb 1981
Mark Ward			Fianna Fail	Carlow, Ireland	18/12/1972
Grealish	Mary	F	Labour	24 Main St, Carlow	
Lynch	Lisa	F	FG		3/2/1981

Lastname	Firstname	Gender	Party	Constituency	DOB
Ryan	Noel	M	Fianna Fail	Dun Laoghaire	02/11/1965
Lynch	Lisa	F	Fine Gael	Dublin South–East	03/02/1981
Ward	Mark	M	Fianna Fail	Carlow-Kilkenny	18/12/1972
Grealish	Mary	F	Labour	Carlow-Kilkenny	NaN

Pandas Package for Python

- Pandas offers two new data structures that are optimised for data analysis and manipulation.
  1. A Data Frame is a flexible two-dimensional, potentially heterogeneous tabular data structure.
  2. A Series is a data structure for a single column of a Data Frame.

Lastname
Ryan
Lynch
Ward
Grealish

DOB
02/11/1965
03/02/1981
18/12/1972
NaN

- Key distinction of these data structures over basic Python data structures is that they make it easy to associate an index with data - i.e. row and column names.

Pandas Series

- Series: a one-dimensional array capable of holding any data type.
- The key difference between a Series and a standard Python list is that, as well as having a numeric position, the elements in the array can have a custom label or index of any type.

Series of 4 values

Position	Index	Values
0	Ireland	4613000
1	Belgium	11190845
2	France	66627000
3	Spain	46439000

Series of 5 values

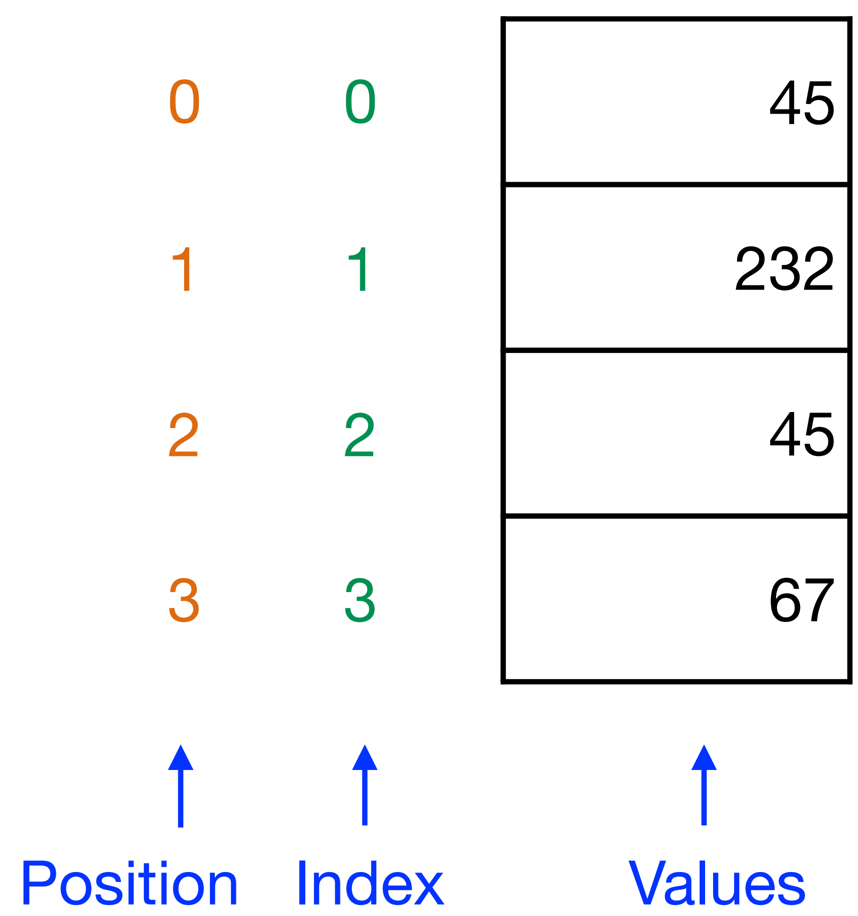
Position	Index	Values
0	982424	Mark
1	992343	Alice
2	961011	Lisa
3	998714	Bob
4	940067	Emma

Creating Pandas Series

- To create a new series, we use the `pd.Series()` method. The simplest approach is to pass in a Python list and use a numeric index.
- The axis labels for the data as referred to as the index. The length of index must be the same as the length of data.
- Since we have not passed any index in the code above, the default index will be created with values [0, 1, 2, 3, ...]
- But in most useful cases, the index will be different from the position, representing some unique identifier for each value in the series.

```
s = pd.Series([45,232,45,67])
print(s)
```

```
0      45
1     232
2      45
3      67
```



Creating Pandas Series

- We can also explicitly pass a list of index names to the `Series()` method to provide a more useful index. The length of the index should be the same as the number of values.

```
populations = pd.Series([1357000000,
1252000000, 321068000, 249900000],
["China", "India", "USA", "Indonesia"])
```

Create a new series with 4 values and an index

```
print(populations)
```

```
China            1357000000
India            1252000000
USA               321068000
Indonesia         249900000
```

Index

Values

- The use of an index is similar to a Python dictionary. In fact we can create a pandas series directly from a Python dictionary:

```
dpop = {"China":1357000000, "India":1252000000, "USA":321068000, "Indonesia":
249900000}
populations = pd.Series(dpop)
```

```
print(populations.index)
```

```
Index(['China', 'India', 'Indonesia', 'USA'], dtype='object')
```

Accessing Pandas Series

- A Series offers a number of different ways to access elements. We can use simple position numbers like with lists:

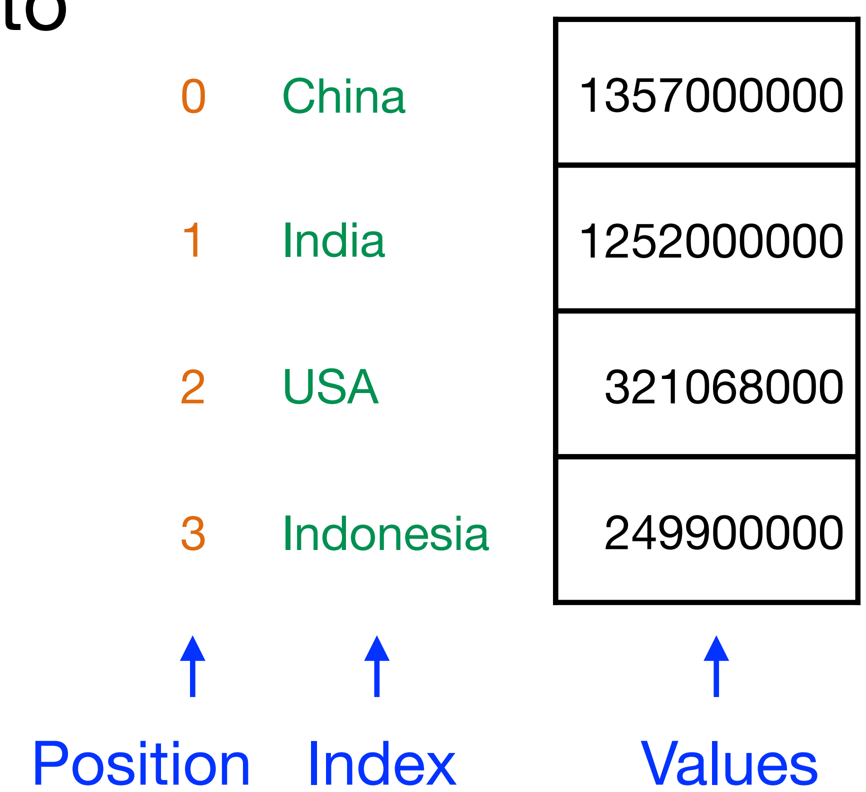
```
populations = pd.Series([1357000000,
1252000000, 321068000, 249900000], ["China",
"India", "USA", "Indonesia"])
```

```
populations[0]
```

```
1357000000
```

```
populations[2]
```

```
321068000
```



- We can use slicing via the `i:j` operator. Remember this includes the elements from position *i* up to but not including *j*:

```
populations[0:2]
```

```
China      1357000000
India      1252000000
```

```
populations[1:]
```

```
India          1252000000
USA             321068000
Indonesia       249900000
```

```
populations[:3]
```

```
China      1357000000
India      1252000000
USA         321068000
```

Accessing Pandas Series

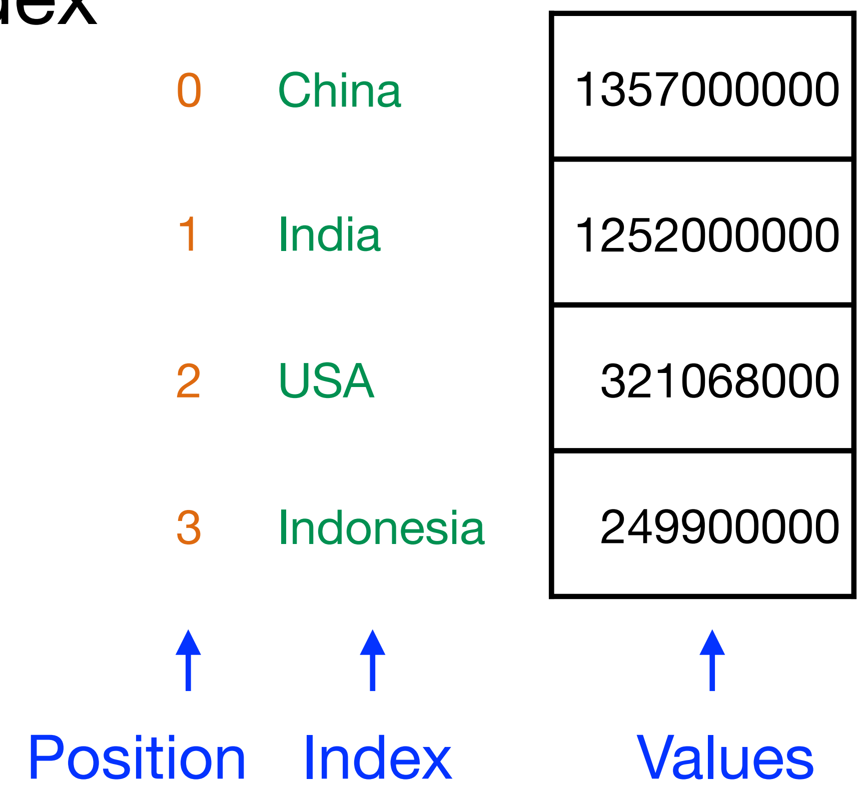
- We can also access elements using the index defined at creation, similar to a dictionary:

```
populations["USA"]
```

```
321068000
```

```
populations["China"]
```

```
1357000000
```



- Using the index is also the easiest way to change the values of elements in a Series, although we can also use numeric positions to change values:

```
populations["China"] = 1374730000
populations["USA"] = 329001000
print(populations)
```

```
China        1374730000
India        1252000000
USA           329001000
Indonesia     249900000
```

```
populations[0] = 1374730000
populations[2] = 329001000
print(populations)
```

```
China        1374730000
India        1252000000
USA           329001000
Indonesia     249900000
```

Accessing Pandas Series

- In many cases we might want to filter the values in a Pandas Series, to reduce it to a subset of the original elements.
- We can do this by indexing with a Boolean expression:

```
populations > 1000000000
```

```
China         True
India         True
USA          False
Indonesia    False
```

Check which values are > 1 billion

```
populations[populations > 1000000000]
```

```
China     1357000000
India     1252000000
```

Create a new series, with only values > 1 billion

```
populations < 1000000000
```

```
China        False
India        False
USA           True
Indonesia     True
```

Check which values are < 1 billion

```
populations[populations < 1000000000]
```

```
USA          321068000
Indonesia    249900000
```

Create a new series, with only values < 1 billion

Series Statistics

- A Series has associated functions for many simple analyses of numeric series - e.g. range, mean, standard deviation...

```
populations.min()
```
```
249900000
```

```
populations.median()
```
```
786534000.0
```

```
populations.std()
```
```
590603801.3107152
```

```
populations.max()
```
```
1357000000
```

```
populations.mean()
```
```
794992000.0
```

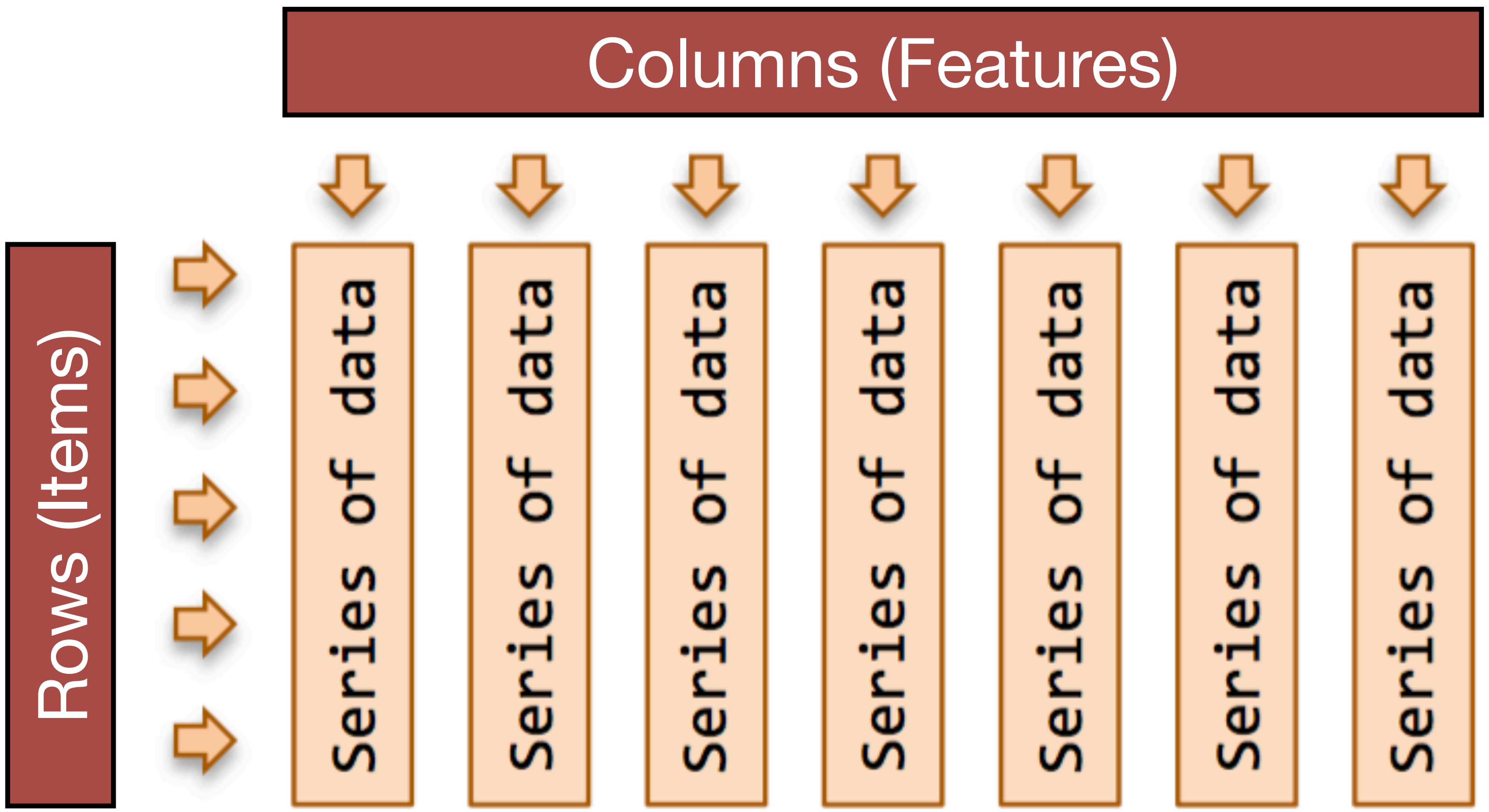
```
populations.sum()
```
```
3179968000
```

- The `describe()` function gives a useful statistical summary of a Series.

```
populations.describe()
```
```
count    4.000000e+00
mean     7.949920e+08
std      5.906038e+08
min      2.499000e+08
25%      3.032760e+08
50%      7.865340e+08
75%      1.278250e+09
max      1.357000e+09
```

Data Frames

- Recall the Analytics Base Table (ABT) idea from the CRISP-DM model, where cases are represented by descriptive features.
- Equivalent in Pandas is a Data Frame: a 2-dimensional labelled data structure with columns of data that can be of different types.
- Every column in a Data Frame is itself a Pandas Series.



Data Frames

- The number, type, and meaning of the values stored in each column of a Data Frame depends on the data being analysed.
- **Example:** Data Frame of size 4 rows x 3 columns, with both a row and column index. The column index indicates the feature name, the row index indicates the country name. Both are unique.

Column position →
Column index →

		0	1	2
		Capital	Population	GDP-BN
0	Ireland	Dublin	4613000	250.8
1	Belgium	Brussels	11190845	524.4
2	France	Paris	66627000	2833.0
3	Spain	Madrid	46439000	1619.0

↑ Row position ↑ Row index

Data Frames

- By default, IPython Notebooks will render a Data Frame in an easily readable tabular format.

	First Name	Last Name	Country
0	Malcom	Jones	England
1	Felix	Brown	USA
2	Alex	Cooper	Poland
3	Tod	Campbell	USA
4	Derek	Ward	Switzerland
5	Mark	Shaw	Male

	first	last	gender	age	city	married
email						
rays@lolezpod.rs	Raymond	Stewart	Male	21	Cork	FALSE
rowe@fehos.cr	Ivan	Rowe	Male	40	Dublin	TRUE
tbowen@lo.me	Tom	Bowen	Male	34	Galway	TRUE
rosie97@uja.as	Rosie	Wood	Female	56	London	TRUE
lisae@gmail.com	Lisa	Estrada	Female	24	Cardiff	FALSE
markshaw@vazaw.sn	Mark	Shaw	Male	63	Dublin	TRUE
kath99@gmail.com	Katharine	Walsh	Female	27	Paris	TRUE
alice@hipipu.va	Alice	Cox	Female	40	London	TRUE

	ProductName	PaymentType	CustomerName	Quantity	UnitPrice
0	Product-A	Visa	Alice	1	44.99
1	Product-C	Mastercard	Bob	2	7.49
2	Product-D	Amex	Charlie	5	11.99
3	Product-E	Visa	Alice	3	6.99
4	Product-A	Amex	Charlie	2	39.99

	GEOID	2005	2006	2007	2008	2009	2010	2011	2012	2013
State										
Alabama	04000US01	37150	37952	42212	44476	39980	40933	42590	43464	41381
Alaska	04000US02	55891	56418	62993	63989	61604	57848	57431	63648	61137
Arizona	04000US04	45245	46657	47215	46914	45739	46896	48621	47044	50602
Arkansas	04000US05	36658	37057	40795	39586	36538	38587	41302	39018	39919
California	04000US06	51755	55319	55734	57014	56134	54283	53367	57020	57528

Data Frames

- **Example:** Data Frame of size 8 rows x 6 columns. Each row is identified by a unique index (an email address).

Column index

		first	last	gender	age	city	married
	email						
0	**rays@lolezpod.rs**	Raymond	Stewart	Male	21	Cork	FALSE
1	**rowe@fehos.cr**	Ivan	Rowe	Male	40	Dublin	TRUE
2	**tbowen@lo.me**	Tom	Bowen	Male	34	Galway	TRUE
3	**rosie97@uja.as**	Rosie	Wood	Female	56	London	TRUE
4	**lisae@gmail.com**	Lisa	Estrada	Female	24	Cardiff	FALSE
5	**markshaw@vazaw.sn**	Mark	Shaw	Male	63	Dublin	TRUE
6	**kath99@gmail.com**	Katharine	Walsh	Female	27	Paris	TRUE
7	**alice@hipipu.va**	Alice	Cox	Female	40	London	TRUE

Row position

Row index

Column Values (Each a series)

Creating Data Frames

- The easiest way to create a DataFrame is to pass the `DataFrame()` method a dictionary of lists, where each list will be a column:

```
countries = pd.DataFrame({"Country":["China", "India", "USA", "Indonesia"],
             "Population":[1357000000, 1252000000, 321068000, 249900000],
             "GDP":[11384760, 2182580, 17968200, 888648],
             "Life Expectancy":[75.41, 68.13, 79.68, 72.45]})
countries
```

By default, the row index will be numeric, counting from 0

	Country	GDP	Life Expectancy	Population
0	China	11384760	75.41	1357000000
1	India	2182580	68.13	1252000000
2	USA	17968200	79.68	321068000
3	Indonesia	888648	72.45	249900000

Column index

- The `shape` variable tells us that the data frame has 4 rows, each with 4 columns.

```
countries.shape
```

```
(4, 4)
```

Loading CSV Data

- The CSV ("Comma Separated Values") file format is often used to exchange tabular data between different applications (e.g. Excel).
- Essentially a plain text file where values are split by a comma separator. Alternatively can be tab or space separated.
- Often the first line is a header, explaining the meaning of each value.

```
Player,Team,Total Goals,Penalties,Home,Away
J Vardy,Leicester City,19,4,11,8
H Kane,Tottenham,16,4,7,9
R Lukaku,Everton,16,1,8,8
O Ighalo,Watford,15,0,8,7
S Aguero,Manchester City,14,1,10,4
R Mahrez,Leicester City,14,4,4,10
O Giroud,Arsenal,12,0,4,8
D Costa,Chelsea,10,0,7,3
J Defoe,Sunderland,10,0,3,7
G Wijnaldum,Newcastle Utd,9,0,9,0
```

Player	Team	Total Goals	Penalties	Home	Away
J Vardy	Leicester City	19	4	11	8
H Kane	Tottenham	16	4	7	9
R Lukaku	Everton	16	1	8	8
O Ighalo	Watford	15	0	8	7
S Aguero	Manchester City	14	1	10	4
R Mahrez	Leicester City	14	4	4	10
O Giroud	Arsenal	12	0	4	8
D Costa	Chelsea	10	0	7	3
J Defoe	Sunderland	10	0	3	7
G Wijnaldum	Newcastle Utd	9	0	9	0

- A simple way to work with data in CSV files is to use Pandas to load the data into a new Data Frame.

Loading CSV Data

- We read a Data Frame from a CSV file via the `read_csv()` function.
- The first line contains the column index names, each subsequent line will be a row in the frame.
- By default, the function assumes the values are comma-separated.

```
df = pd.read_csv("countries.csv")
```

Column index

By default, the row index will be numeric, same as the position

	Country ID	Life Exp.	Top-10 Income	Infant Mort.	Mil. Spend	School Years	CPI
0	Afghanistan	59.61	23.21	74.3	4.44	0.4	1.5171
1	Haiti	45.00	47.67	73.1	0.09	3.4	1.7999
2	Nigeria	51.30	38.23	82.6	1.07	4.1	2.4493
3	Egypt	70.48	26.58	19.6	1.86	5.3	2.8622
4	Argentina	75.77	32.30	13.3	0.76	10.1	2.9961
5	China	74.87	29.98	13.7	1.95	6.4	3.6356
6	Brazil	73.12	42.93	14.5	1.43	7.2	3.7741
7	Israel	81.30	28.80	3.6	6.77	12.5	5.8069
8	U.S.A	78.51	29.85	6.3	4.72	13.7	7.1357
9	Ireland	80.15	27.23	3.5	0.60	11.5	7.5360
10	U.K.	80.09	28.49	4.4	2.59	13.0	7.7751
11	Germany	80.24	22.07	3.5	1.31	12.0	8.0461
12	Canada	80.99	24.79	4.9	1.42	14.2	8.6725
13	Australia	82.09	25.40	4.2	1.86	11.5	8.8442
14	Sweden	81.43	22.18	2.4	1.27	12.8	9.2985
15	New Zealand	80.67	27.81	4.9	1.13	12.3	9.4627

Loading CSV Data

- We can also tell the `read_csv()` function to use one of the columns in the CSV file as the index for the rows in our data.

```
df = pd.read_csv("countries.csv",index_col="Country ID")
```

Column index

Row index

	Life Exp.	Top-10 Income	Infant Mort.	Mil. Spend	School Years	CPI
Country ID						
Afghanistan	59.61	23.21	74.3	4.44	0.4	1.5171
Haiti	45.00	47.67	73.1	0.09	3.4	1.7999
Nigeria	51.30	38.23	82.6	1.07	4.1	2.4493
Egypt	70.48	26.58	19.6	1.86	5.3	2.8622
Argentina	75.77	32.30	13.3	0.76	10.1	2.9961
China	74.87	29.98	13.7	1.95	6.4	3.6356
Brazil	73.12	42.93	14.5	1.43	7.2	3.7741
Israel	81.30	28.80	3.6	6.77	12.5	5.8069
U.S.A	78.51	29.85	6.3	4.72	13.7	7.1357
Ireland	80.15	27.23	3.5	0.60	11.5	7.5360
U.K.	80.09	28.49	4.4	2.59	13.0	7.7751
Germany	80.24	22.07	3.5	1.31	12.0	8.0461
Canada	80.99	24.79	4.9	1.42	14.2	8.6725
Australia	82.09	25.40	4.2	1.86	11.5	8.8442
Sweden	81.43	22.18	2.4	1.27	12.8	9.2985
New Zealand	80.67	27.81	4.9	1.13	12.3	9.4627

Data Frame Statistics

- Once we have loaded a Data Frame, the `describe()` function gives a useful summary table with statistics for each column.

```
df.describe()
```

	Life Exp.	Top-10 Income	Infant Mort.	Mil. Spend	School Years	CPI
count	16.000000	16.000000	16.000000	16.000000	16.00000	16.000000
mean	73.476250	29.845000	20.550000	2.079375	9.40000	5.725750
std	11.481893	7.295689	28.351296	1.766950	4.28859	2.917551
min	45.000000	22.070000	2.400000	0.090000	0.40000	1.517100
25%	72.460000	25.247500	4.050000	1.115000	6.12500	2.962625
50%	79.300000	28.150000	5.600000	1.425000	11.50000	6.471300
75%	80.750000	30.560000	15.775000	2.110000	12.57500	8.202700
max	82.090000	47.670000	82.600000	6.770000	14.20000	9.462700

- We can also get individual statistics for each column:

```
df.mean()
```

```
Life Exp.         73.476250
Top-10 Income     29.845000
Infant Mort.      20.550000
Mil. Spend         2.079375
School Years       9.400000
CPI                5.725750
```

Mean for columns

```
df.std()
```

```
Life Exp.         11.481893
Top-10 Income      7.295689
Infant Mort.      28.351296
Mil. Spend         1.766950
School Years       4.288590
CPI                2.917551
```

Standard deviation

```
df.sum()
```

```
Life Exp.         1175.620
Top-10 Income      477.520
Infant Mort.       328.800
Mil. Spend          33.270
School Years       150.400
CPI                 91.612
```

Sum all columns

Accessing Values in Data Frames

- Columns in a Data Frame can be accessed using the index name of the column to give a single Series.

```
df["School Years"]
```

```
Country ID
Afghanistan      0.4
Haiti            3.4
Nigeria          4.1
Egypt            5.3
Argentina       10.1
China            6.4
Brazil           7.2
Israel          12.5
U.S.A           13.7
Ireland         11.5
U.K.            13.0
Germany         12.0
Canada          14.2
Australia       11.5
Sweden          12.8
New Zealand     12.3
```

Result is a new Series, where the row index is also copied.

```
df[["CPI","School Years"]]
```

	CPI	School Years
Country ID		
Afghanistan	1.5171	0.4
Haiti	1.7999	3.4
Nigeria	2.4493	4.1
Egypt	2.8622	5.3
Argentina	2.9961	10.1
China	3.6356	6.4
Brazil	3.7741	7.2
Israel	5.8069	12.5
U.S.A	7.1357	13.7
Ireland	7.5360	11.5
U.K.	7.7751	13.0
Germany	8.0461	12.0
Canada	8.6725	14.2
Australia	8.8442	11.5
Sweden	9.2985	12.8
New Zealand	9.4627	12.3

Multiple columns can be selected by passing a list of column names.

Result is a new Data Frame, where the row index is also copied.

Accessing Values in Data Frames

- We can access rows of a Data Frame in different ways.

- We can access a single row by numeric position using `iloc[]`:

```
df.iloc[0]
```

```
Life Exp.          59.6100
Top-10 Income      23.2100
Infant Mort.       74.3000
Mil. Spend          4.4400
School Years        0.4000
CPI                 1.5171
Name: Afghanistan, dtype: float64
```

Returns a Series corresponding to first row of `df` (i.e position 0)

- We can access a single row by index name using `loc[]`:

```
df.loc["Sweden"]
```

```
Life Exp.          81.4300
Top-10 Income      22.1800
Infant Mort.        2.4000
Mil. Spend          1.2700
School Years       12.8000
CPI                 9.2985
Name: Sweden, dtype: float64
```

Returns a Series corresponding to row of `df` with index "Sweden"

- Both methods can be used to specify multiple rows to access:

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

Use slicing to specify multiple positions

```
df.loc[["Sweden","Ireland"]]
```

Use a list to specify multiple index names

Accessing Values in Data Frames

- There are a variety of different ways to access and change the individual values in a Data Frame:

	Life Exp.	Top-10 Income	Infant Mort.	Mil. Spend	School Years	CPI
Country ID						
Afghanistan	59.61	23.21	74.3	4.44	0.4	1.5171
Haiti	45.00	47.67	73.1	0.09	3.4	1.7999
Nigeria	51.30	38.23	82.6	1.07	4.1	2.4493
Egypt	70.48	26.58	19.6	1.86	5.3	2.8622
Argentina	75.77	32.30	13.3	0.76	10.1	2.9961
China	74.87	29.98	13.7	1.95	6.4	3.6356
Brazil	73.12	42.93	14.5	1.43	7.2	3.7741
Israel	81.30	28.80	3.6	6.77	12.5	5.8069
U.S.A	78.51	29.85	6.3	4.72	13.7	7.1357
Ireland	80.15	27.23	3.5	0.60	11.5	7.5360
U.K.	80.09	28.49	4.4	2.59	13.0	7.7751
Germany	80.24	22.07	3.5	1.31	12.0	8.0461
Canada	80.99	24.79	4.9	1.42	14.2	8.6725
Australia	82.09	25.40	4.2	1.86	11.5	8.8442
Sweden	81.43	22.18	2.4	1.27	12.8	9.2985
New Zealand	80.67	27.81	4.9	1.13	12.3	9.4627

Access by column index, then row index

```
df["Mil. Spend"]["China"]
```
```
1.95
```
```
df["School Years"]["Ireland"]
```
```
11.5
```

Access by row index, then column index

```
df.loc["Canada"]["School Years"]
```
```
14.2
```

Access by row position, then column index

```
df.iloc[15]["Life Exp."]
```
```
80.67
```

Access by row position, then column position

```
df.iloc[15][4]
```
```
12.3
```

Data Normalisation

- We can apply arithmetic operations to scale numeric Data Frames:

```
df = pd.DataFrame(
{"home":[0,3,2,4,2],
"away":[0,1,0,2,0]})
```

	away	home
0	0	0
1	1	3
2	0	2
3	2	4
4	0	2

```
df / 10
```

	away	home
0	0.0	0.0
1	0.1	0.3
2	0.0	0.2
3	0.2	0.4
4	0.0	0.2

```
df * 100
```

	away	home
0	0	0
1	100	300
2	0	200
3	200	400
4	0	200

- This allows us to easily normalise our data in different ways:

```
df - df.mean()
```

Subtract the mean column values from each row in `df`

	away	home
0	-0.6	-2.2
1	0.4	0.8
2	-0.6	-0.2
3	1.4	1.8
4	-0.6	-0.2

```
df / df.max()
```

Divide each row by maximum value for each column

	away	home
0	0.0	0.00
1	0.5	0.75
2	0.0	0.50
3	1.0	1.00
4	0.0	0.50

- If we have made any modifications to a Data Frame, we can export the data as a new CSV file using the `to_csv()` function.

```
df = df - df.max()
df.to_csv("modified.csv")
```

The default value separator is a comma

Aggregating Data

- We can group rows in Data Frames based on some categorical value in each row. First we use `groupby()` to create the groups, and then can apply some other function to combine the results:

	Name	Party	Tweets
0	E. Kenny	Fine Gael	399
1	M. Martin	Fianna Fail	938
2	L. Varadkar	Fine Gael	1830
3	N. Collins	Fianna Fail	1946
4	J. Burton	Labour	907

```
groups = df.groupby("Party")
```

We will group rows using the "Party" column

```
groups.sum()
```

	Tweets
Party	
Fianna Fail	2884
Fine Gael	2229
Labour	907

We can now apply the `sum()` function to get the total sum of tweets for each party

```
groups.mean()
```

	Tweets
Party	
Fianna Fail	1442.0
Fine Gael	1114.5
Labour	907.0

We could also apply the `mean()` function to get the mean number of tweets for each party

Handling Missing Data

- Many real datasets have missing values, either because it existed but was not collected or because it never existed.
- In the titanic.csv dataset, 86 out of 418 of the values in the "Age" column are missing, as indicated by the null/empty value `NaN`

	PassengerId	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	892	3	Kelly, Mr. James	male	34.5	0	0	330911	7.8292	NaN	Q
1	893	3	Wilkes, Mrs. James (Ellen Needs)	female	47.0	1	0	363272	7.0000	NaN	S
2	894	2	Myles, Mr. Thomas Francis	male	62.0	0	0	240276	9.6875	NaN	Q
3	895	3	Wirz, Mr. Albert	male	27.0	0	0	315154	8.6625	NaN	S
4	896	3	Hirvonen, Mrs. Alexander (Helga E Lindqvist)	female	22.0	1	1	3101298	12.2875	NaN	S
5	897	3	Svensson, Mr. Johan Cervin	male	14.0	0	0	7538	9.2250	NaN	S
6	898	3	Connolly, Miss. Kate	female	30.0	0	0	330972	7.6292	NaN	Q
7	899	2	Caldwell, Mr. Albert Francis	male	26.0	1	1	248738	29.0000	NaN	S
8	900	3	Abrahim, Mrs. Joseph (Sophie Halaut Easu)	female	18.0	0	0	2657	7.2292	NaN	C
9	901	3	Davies, Mr. John Samuel	male	21.0	2	0	A/4 48871	24.1500	NaN	S
10	902	3	Ilieff, Mr. Ylio	male	NaN	0	0	349220	7.8958	NaN	S
11	903	1	Jones, Mr. Charles Cresson	male	46.0	0	0	694	26.0000	NaN	S
12	904	1	Snyder, Mrs. John Pillsbury (Nelle Stevenson)	female	23.0	1	0	21228	82.2667	B45	S
13	905	2	Howard, Mr. Benjamin	male	63.0	1	0	24065	26.0000	NaN	S
14	906	1	Chaffee, Mrs. Herbert Fuller (Carrie Constance...	female	47.0	1	0	W.E.P. 5734	61.1750	E31	S
15	907	2	del Carlo, Mrs. Sebastiano (Argenia Genovesi)	female	24.0	1	0	SC/PARIS 2167	27.7208	NaN	C
16	908	2	Keane, Mr. Daniel	male	35.0	0	0	233734	12.3500	NaN	Q
17	909	3	Assaf, Mr. Gerios	male	21.0	0	0	2692	7.2250	NaN	C
18	910	3	Ilmakangas, Miss. Ida Livija	female	27.0	1	0	STON/O2. 3101270	7.9250	NaN	S
19	911	3	Assaf Khalil, Mrs. Mariana (Miriam")"	female	45.0	0	0	2696	7.2250	NaN	C
20	912	1	Rothschild, Mr. Martin	male	55.0	1	0	PC 17603	59.4000	NaN	C
21	913	3	Olsen, Master. Artur Karl	male	9.0	0	1	C 17368	3.1708	NaN	S
22	914	1	Flegenheim, Mrs. Alfred (Antoinette)	female	NaN	0	0	PC 17598	31.6833	NaN	S

```
df.isnull().sum()
```

```
PassengerId      0
Pclass           0
Name             0
Sex              0
Age             86
SibSp            0
Parch            0
Ticket           0
Fare             1
Cabin          327
Embarked         0
```

Sum number of null (`NaN`) values in each column

Handling Missing Data

- One option is to simply drop a feature with many missing values:

```
df = df.drop(["Age"], axis=1)
```

`Axis=1` means a column

- But if we expect age to play an important role, then we want to keep the column and estimate the missing values in some way.
- Simple approach is to fill in missing values using the mean value.

```
mean_age = df["Age"].mean()
df["Age"] = df["Age"].fillna(mean_age)
```

Fill in every `NaN` value with mean value 30.27

	PassengerId	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	892	3	Kelly, Mr. James	male	34.50000	0	0	330911	7.8292	NaN	Q
1	893	3	Wilkes, Mrs. James (Ellen Needs)	female	47.00000	1	0	363272	7.0000	NaN	S
2	894	2	Myles, Mr. Thomas Francis	male	62.00000	0	0	240276	9.6875	NaN	Q
3	895	3	Wirz, Mr. Albert	male	27.00000	0	0	315154	8.6625	NaN	S
4	896	3	Hirvonen, Mrs. Alexander (Helga E Lindqvist)	female	22.00000	1	1	3101298	12.2875	NaN	S
5	897	3	Svensson, Mr. Johan Cervin	male	14.00000	0	0	7538	9.2250	NaN	S
6	898	3	Connolly, Miss. Kate	female	30.00000	0	0	330972	7.6292	NaN	Q
7	899	2	Caldwell, Mr. Albert Francis	male	26.00000	1	1	248738	29.0000	NaN	S
8	900	3	Abrahim, Mrs. Joseph (Sophie Halaut Easu)	female	18.00000	0	0	2657	7.2292	NaN	C
9	901	3	Davies, Mr. John Samuel	male	21.00000	2	0	A/4 48871	24.1500	NaN	S
10	902	3	Ilieff, Mr. Ylio	male	30.27259	0	0	349220	7.8958	NaN	S

`NaN` has been replaced with 30.27

Overview

- Data Preparation
- Clean v Noisy Data
- Pandas Package for Python
- Pandas Series
- Pandas Data Frames
- Loading Data
- Working with Data Frames
- Normalising Data
- Aggregating Data
- Handling Missing Values