COMP41680

Data Preparation and Manipulation

Derek Greene

UCD School of Computer Science

Spring 2023

Data Preparation

- Data in real world rarely arrives in a clean and homogeneous form.
- Typically, datasets tends to be incomplete, noisy, and inconsistent.
- Often need to invest considerable time and effort to address these issues before we can perform any detailed analysis or modelling.
- Up to 80% of a typical data science project is cleaning and preparing the data, while the remaining 20% is actual data analysis.

  http://nyti.ms/1p3ZoqI

Data Preparation

- Data preparation 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 contain missing values for various reasons. They are often encoded as null values, blanks or using some other placeholder.
  - Data integration: Match up data for related cases across multiple different raw data sources.
  - Feature engineering: Creating more useful features from the raw data to describe the items of interest, often taking into account domain knowledge.

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

- Manipulating and cleaning data can be slow and tedious. Python can (partially) simplify and automate this process.
- Pandas is an open source package providing high-performance data structures and analysis tools for Python.
- It provides a Python equivalent of the data analysis and manipulation functionality available in the R programming language. Full details at:

  http://pandas.pydata.org

- After installing Anaconda, you will have access to Pandas without needing to install anything else.
- Once the package is installed, we can import it as pd for shorthand.

```
import pandas as pd
```

Pandas Package for Python

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

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.
- Key differences between a Series and a standard Python list:
  - All the values in a Series have the same type.
  - As well as having a numeric position, the elements in the array can have a custom "name" or index.

Series of 4 values

Position	Index	Values
0	Argentina	75.77
1	Australia	82.09
2	Brazil	73.12
3	Canada	80.99

Series of 5 values

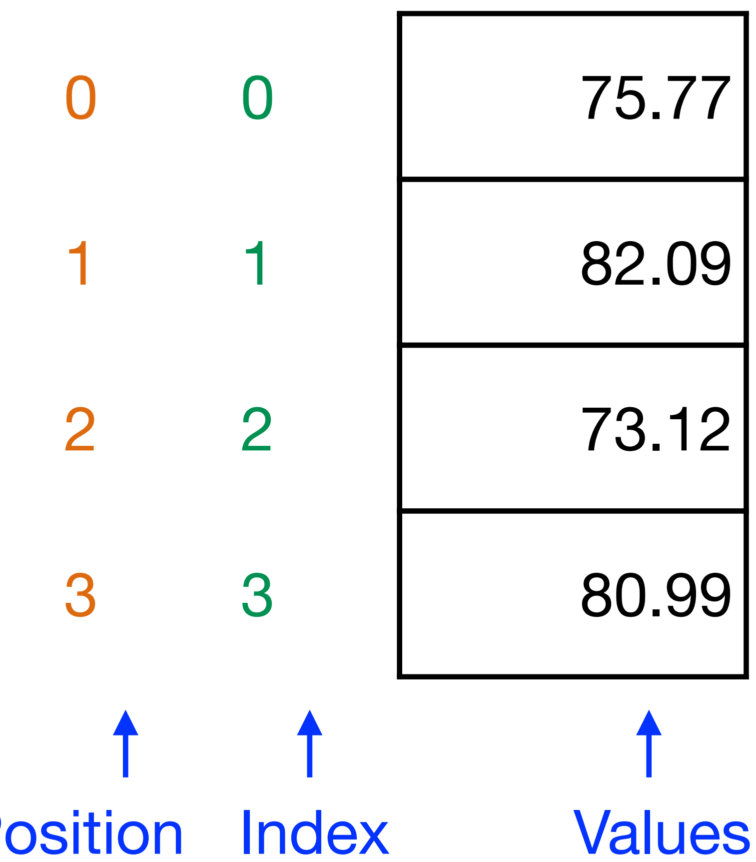
Position	Index	Values
0	982424	Mark
1	992343	Alice
2	961011	Lisa
3	998714	Bob
4	940067	Emma

Creating Pandas Series

- To create a new series, we use the `pd.Series()` function. The simplest approach is to pass in a Python list and use a numeric index.
- Axis labels for the data are referred to as the index. The length of index must be the same as the length of data.
- Since we have not specified any index in the code above, the default index will be created as [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.

```
y = [75.77, 82.09, 73.12, 80.99]
s = pd.Series(y)
print(s)
```

```
0    75.77
1    82.09
2    73.12
3    80.99
dtype: float64
```



Creating Pandas Series

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

```
values = [75.77, 82.09, 73.12, 80.99]
labels = ["Argentina", "Australia", "Brazil", "Canada"]
life_exp = pd.Series(values, labels)
```

Create a new series with 4 values and 4 index labels

Values

```
print(life_exp)
```

```
Argentina     75.77
Australia     82.09
Brazil        73.12
Canada        80.99
```

Index

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

```
d_life_exp = {"Argentina": 75.77, "Australia": 82.09, "Brazil": 73.12,
"Canada": 80.99}
life_exp = pd.Series(d_life_exp)
```

```
print(life_exp.index)
```

```
Index(['Argentina', 'Australia', 'Brazil', 'Canada'], dtype='object')
```

Accessing Pandas Series

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

```
values = [75.77, 82.09, 73.12, 80.99]
labels = ["Argentina", "Australia",
"Brazil", "Canada"]
life_exp = pd.Series(values, labels)
```

```
life_exp[0]
```

```
75.77
```

```
life_exp[2]
```

```
73.12
```

Position	Index	Values
0	Argentina	75.77
1	Australia	82.09
2	Brazil	73.12
3	Canada	80.99

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

```
life_exp[0:2]
```

```
Argentina     75.77
Australia     82.09
```

```
life_exp[1:]
```

```
Australia     82.09
Brazil        73.12
Canada        80.99
```

```
life_exp[:3]
```

```
Argentina     75.77
Australia     82.09
Brazil        73.12
```

Accessing Pandas Series

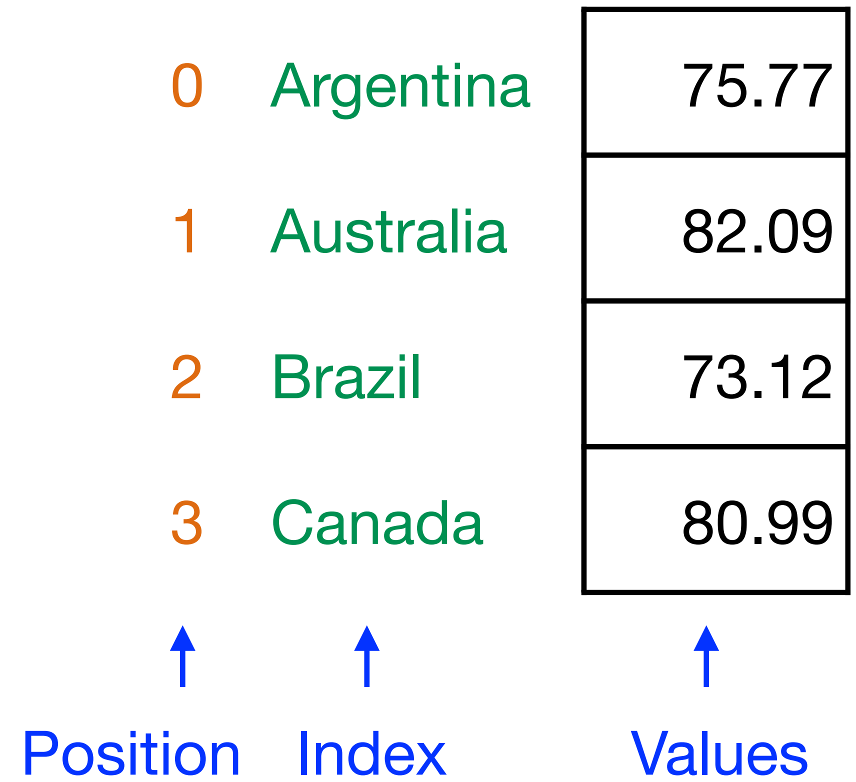
- We can also access values using the index defined at creation, similar to a dictionary:

```
life_exp["Argentina"]
```

```
75.77
```

```
life_exp["Brazil"]
```

```
73.12
```



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

```
life_exp["Argentina"] = 80
life_exp["Brazil"] = 75.2
print(life_exp)
```

```
Argentina     80.00
Australia     82.09
Brazil        75.20
Canada        80.99
```

```
life_exp[0] = 80
life_exp[2] = 75.2
print(life_exp)
```

```
Argentina     80.00
Australia     82.09
Brazil        75.20
Canada        80.99
```

Applying Conditions to Series

- We might want to filter the values in a Pandas Series, to reduce it to a subset of the original values based on a condition.
- We can do this by indexing with a boolean expression:

```
life_exp > 80
```

```
Argentina     False
Australia      True
Brazil        False
Canada         True
```

Check which values are > 80

```
life_exp[life_exp > 80]
```

```
Australia     82.09
Canada        80.99
```

Filter the Series, keeping only values > 80

```
life_exp < 80
```

```
Argentina      True
Australia     False
Brazil         True
Canada        False
```

Check which values are < 80

```
life_exp[life_exp < 80]
```

```
Argentina     75.77
Brazil        73.12
```

Filter the Series, keeping only values < 80

Series Statistics

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

```
life_exp.min()
```
```
73.12
```

```
life_exp.median()
```
```
78.38
```

```
life_exp.std()
```
```
4.2604880393369635
```

```
life_exp.max()
```
```
82.09
```

```
life_exp.mean()
```
```
77.9925
```

```
life_exp.sum()
```
```
311.97
```

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

```
life_exp.describe()
```
```
count     4.000000
mean     77.992500
std       4.260488
min      73.120000
25%      75.107500
50%      78.380000
75%      81.265000
max      82.090000
```

Sorting Series

- To sort a Series by its values, use the `sort_values()` function. By default the values will be sorted in ascending order (i.e. smallest to largest).

```
life_exp.sort_values()
```

```
Brazil        73.12
Argentina     75.77
Canada        80.99
Australia     82.09
```

- To sort a Series in descending order instead, we specify the argument `ascending=False`.

```
life_exp.sort_values(ascending=False)
```

```
Australia     82.09
Canada        80.99
Argentina     75.77
Brazil        73.12
```

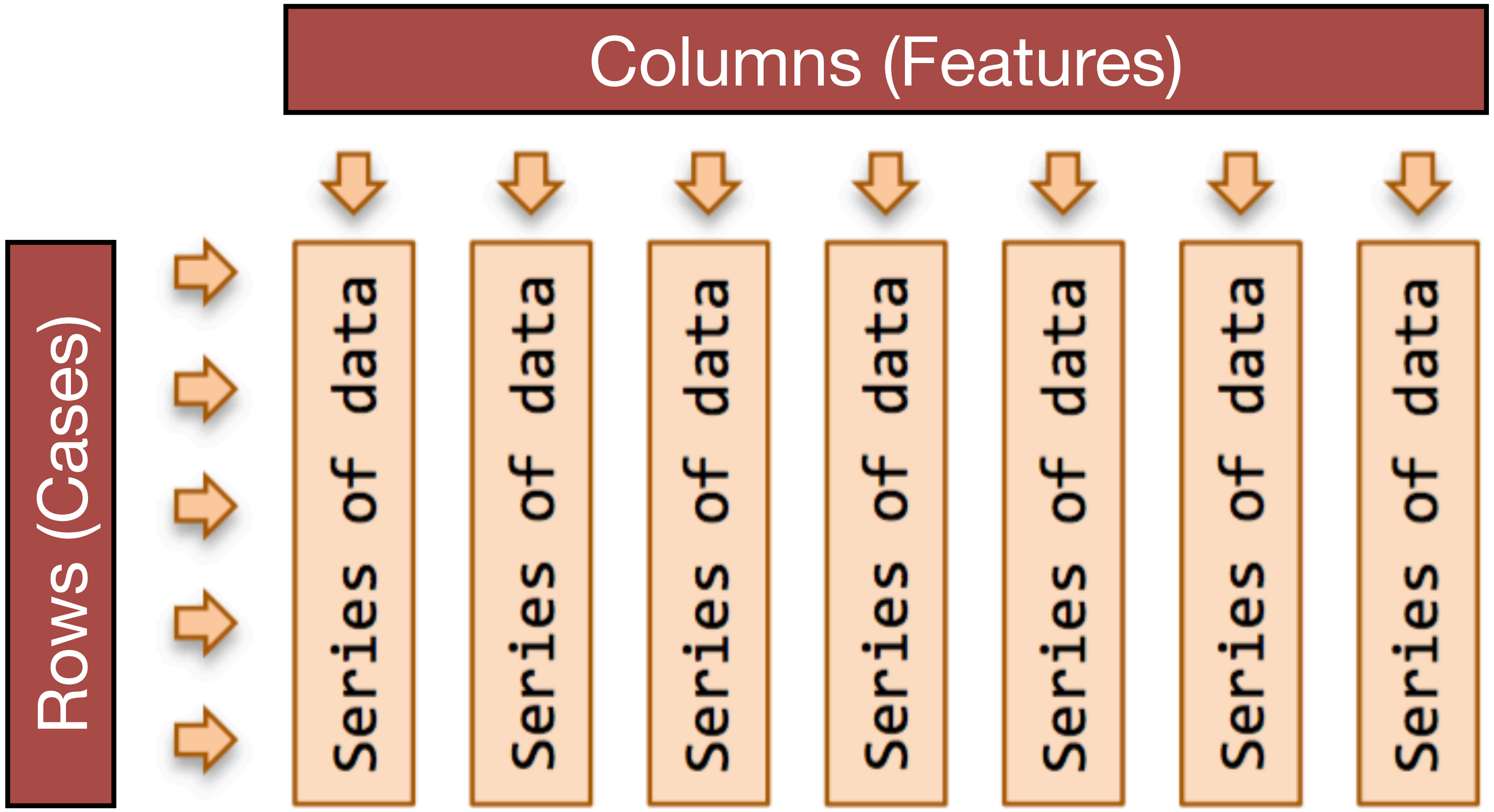
- We can also sort the values in a Series based on their associated index labels by calling the `sort_index()` function.

```
life_exp.sort_index()
```

```
Argentina     75.77
Australia     82.09
Brazil        73.12
Canada        80.99
```

Pandas DataFrames

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



Pandas DataFrames

- The number, type, and meaning of the values stored in each column of a DataFrame depends on the data being analysed.
- **Example:** DataFrame 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 →		0	1	2
Column index →		Region	Population	Life Exp
0	Argentina	South America	43.59	75.77
1	Australia	Oceania	23.99	82.09
2	Brazil	South America	200.4	73.12
3	Canada	North America	35.99	80.99
↑ Row position	↑ Row index			

Pandas DataFrames

- **Example:** DataFrame 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

Columns (Each a series)

Creating DataFrames

- An easy way to create a DataFrame is to pass the `pd.DataFrame()` function a dictionary of lists, where each list will be a column:

```
countries = ["Argentina", "Australia", "Brazil", "Canada"]
regions = ["South America", "Oceania", "South America", "North America"]
pops = [43.59, 23.99, 200.4, 35.99]
life_exp = [75.77, 82.09, 73.12, 80.99]
```

```
d = {"Country":countries, "Region":regions, "Population":pops,
"Life Exp":life_exp}
df = pd.DataFrame(d)
```

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

	Country	Region	Population	Life Exp
0	Argentina	South America	43.59	75.77
1	Australia	Oceania	23.99	82.09
2	Brazil	South America	200.40	73.12
3	Canada	North America	35.99	80.99

Column index

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

```
df.shape
```

```
(4, 4)
```

Loading DataFrames

- 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.

```
Country,Region,Population,Life Exp,Landlocked,Language
Argentina,South America,43.59,75.77,No,Spanish
Australia,Oceania,23.99,82.09,No,English
Brazil,South America,200.4,73.12,No,Portuguese
Canada,North America,35.99,80.99,No,English
Chad,Africa,11.63,49.81,Yes,Arabic
China,Asia,1357,74.87,No,Chinese
Egypt,Africa,90.37,70.48,No,Arabic
Germany,Europe,81.46,80.24,No,German
Ireland,Europe,4.64,80.15,No,English
Japan,Asia,126.26,84.36,No,Japanese
Mexico,North America,127.58,75.05,No,Spanish
New Zealand,Oceania,4.66,80.67,No,English
Niger,Africa,18.05,55.13,Yes,French
Nigeria,Africa,186.99,51.3,No,English
Paraguay,South America,6.78,76.99,Yes,Spanish
Portugal,Europe,10.29,80.68,No,Portuguese
South Korea,Asia,51.71,83.23,No,Korean
Spain,Europe,47.13,83.49,No,Spanish
Switzerland,Europe,8.12,82.5,Yes,German
United Kingdom,Europe,65.1,80.09,No,English
United States,North America,321.07,78.51,No,English
```

Input CSV file: world_data.csv

Country	Region	Population	Life Exp	Landlocked	Language
Argentina	South America	43.59	75.77	No	Spanish
Australia	Oceania	23.99	82.09	No	English
Brazil	South America	200.4	73.12	No	Portuguese
Canada	North America	35.99	80.99	No	English
Chad	Africa	11.63	49.81	Yes	Arabic
China	Asia	1357	74.87	No	Chinese
Egypt	Africa	90.37	70.48	No	Arabic
Germany	Europe	81.46	80.24	No	German
Ireland	Europe	4.64	80.15	No	English
Japan	Asia	126.26	84.36	No	Japanese
Mexico	North America	127.58	75.05	No	Spanish
New Zealand	Oceania	4.66	80.67	No	English
Niger	Africa	18.05	55.13	Yes	French
Nigeria	Africa	186.99	51.3	No	English
Paraguay	South America	6.78	76.99	Yes	Spanish
Portugal	Europe	10.29	80.68	No	Portuguese
South Korea	Asia	51.71	83.23	No	Korean
Spain	Europe	47.13	83.49	No	Spanish
Switzerland	Europe	8.12	82.5	Yes	German
United Kingdom	Europe	65.1	80.09	No	English
United States	North America	321.07	78.51	No	English

Loading DataFrames

- We read a DataFrame 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 values are comma-separated.

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

Column index

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

	Country	Region	Population	Life Exp	Landlocked	Language
0	Argentina	South America	43.59	75.77	No	Spanish
1	Australia	Oceania	23.99	82.09	No	English
2	Brazil	South America	200.40	73.12	No	Portuguese
3	Canada	North America	35.99	80.99	No	English
4	Chad	Africa	11.63	49.81	Yes	Arabic
5	China	Asia	1357.00	74.87	No	Chinese
6	Egypt	Africa	90.37	70.48	No	Arabic
7	Germany	Europe	81.46	80.24	No	German
8	Ireland	Europe	4.64	80.15	No	English
9	Japan	Asia	126.26	84.36	No	Japanese
10	Mexico	North America	127.58	75.05	No	Spanish
11	New Zealand	Oceania	4.66	80.67	No	English
12	Niger	Africa	18.05	55.13	Yes	French
13	Nigeria	Africa	186.99	51.30	No	English
14	Paraguay	South America	6.78	76.99	Yes	Spanish
15	Portugal	Europe	10.29	80.68	No	Portuguese
16	South Korea	Asia	51.71	83.23	No	Korean
17	Spain	Europe	47.13	83.49	No	Spanish
18	Switzerland	Europe	8.12	82.50	Yes	German
19	United Kingdom	Europe	65.10	80.09	No	English
20	United States	North America	321.07	78.51	No	English

Loading DataFrames

- 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("world_data.csv", index_col="Country")
```

Column index

Row index

	Region	Population	Life Exp	Landlocked	Language
Country					
Argentina	South America	43.59	75.77	No	Spanish
Australia	Oceania	23.99	82.09	No	English
Brazil	South America	200.40	73.12	No	Portuguese
Canada	North America	35.99	80.99	No	English
Chad	Africa	11.63	49.81	Yes	Arabic
China	Asia	1357.00	74.87	No	Chinese
Egypt	Africa	90.37	70.48	No	Arabic
Germany	Europe	81.46	80.24	No	German
Ireland	Europe	4.64	80.15	No	English
Japan	Asia	126.26	84.36	No	Japanese
Mexico	North America	127.58	75.05	No	Spanish
New Zealand	Oceania	4.66	80.67	No	English
Niger	Africa	18.05	55.13	Yes	French
Nigeria	Africa	186.99	51.30	No	English
Paraguay	South America	6.78	76.99	Yes	Spanish
Portugal	Europe	10.29	80.68	No	Portuguese
South Korea	Asia	51.71	83.23	No	Korean
Spain	Europe	47.13	83.49	No	Spanish
Switzerland	Europe	8.12	82.50	Yes	German
United Kingdom	Europe	65.10	80.09	No	English
United States	North America	321.07	78.51	No	English

DataFrame Attributes

Country	Region	Population	Life Exp	Landlocked	Language
Argentina	South America	43.59	75.77	No	Spanish
Australia	Oceania	23.99	82.09	No	English
Brazil	South America	200.4	73.12	No	Portuguese
Canada	North America	35.99	80.99	No	English
Chad	Africa	11.63	49.81	Yes	Arabic
China	Asia	1357	74.87	No	Chinese
Egypt	Africa	90.37	70.48	No	Arabic
Germany	Europe	81.46	80.24	No	German
Ireland	Europe	4.64	80.15	No	English
Japan	Asia	126.26	84.36	No	Japanese
Mexico	North America	127.58	75.05	No	Spanish
New Zealand	Oceania	4.66	80.67	No	English
Niger	Africa	18.05	55.13	Yes	French
Nigeria	Africa	186.99	51.3	No	English
Paraguay	South America	6.78	76.99	Yes	Spanish
Portugal	Europe	10.29	80.68	No	Portuguese
South Korea	Asia	51.71	83.23	No	Korean
Spain	Europe	47.13	83.49	No	Spanish
Switzerland	Europe	8.12	82.5	Yes	German
United Kingdom	Europe	65.1	80.09	No	English
United States	North America	321.07	78.51	No	English

```
df.shape
```

```
(21, 5)
```

```
list(df.columns)
```

```
['Region', 'Population',
'Life Exp', 'Landlocked',
'Language']
```

```
list(df.index)
```

```
['Argentina', 'Australia',
 'Brazil', 'Canada',
 'Chad', 'China',
 'Egypt', 'Germany',
 'Ireland', 'Japan',
 'Mexico', 'New Zealand',
 'Niger', 'Nigeria',
 'Paraguay', 'Portugal',
 'South Korea','Spain',
 'Switzerland',
 'United Kingdom',
 'United States']
```

DataFrame Statistics

- Once we have created or loaded a DataFrame, the associated `describe()` function generates a new summary DataFrame with statistics for each column.
- Note only numeric columns are summarised.

```
df.describe()
```

	Population	Life Exp
count	21.000000	21.000000
mean	134.419524	75.215238
std	291.621883	10.356273
min	4.640000	49.810000
25%	11.630000	74.870000
50%	47.130000	80.090000
75%	126.260000	80.990000
max	1357.000000	84.360000

- We can also get individual statistics for the columns:

```
df.mean()
```

```
Population   134.419524
Life Exp      75.215238
```

Mean for each column

```
df.std()
```

```
Population   291.621883
Life Exp      10.356273
```

Standard deviation for each column

```
df.sum()
```

```
Population      2822.81
Life Exp        1579.52
```

Totals for each column

Accessing Columns in DataFrames

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

```
df["Population"]
```

```
Argentina           43.59
Australia           23.99
Brazil             200.40
Canada              35.99
Chad                11.63
China             1357.00
Egypt               90.37
Germany             81.46
Ireland              4.64
Japan              126.26
Mexico             127.58
New Zealand          4.66
Niger               18.05
Nigeria            186.99
Paraguay             6.78
Portugal            10.29
South Korea         51.71
Spain               47.13
Switzerland          8.12
United Kingdom      65.10
United States      321.07
```

Result is a new Series.
Row index is also copied.

```
df[["Region","Population"]]
```

Country	Region	Population
Argentina	South America	43.59
Australia	Oceania	23.99
Brazil	South America	200.40
Canada	North America	35.99
Chad	Africa	11.63
China	Asia	1357.00
Egypt	Africa	90.37
Germany	Europe	81.46
Ireland	Europe	4.64
Japan	Asia	126.26
Mexico	North America	127.58
New Zealand	Oceania	4.66
Niger	Africa	18.05
Nigeria	Africa	186.99
Paraguay	South America	6.78
Portugal	Europe	10.29
South Korea	Asia	51.71
Spain	Europe	47.13
Switzerland	Europe	8.12
United Kingdom	Europe	65.10
United States	North America	321.07

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

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

Accessing Rows in DataFrames

- We can access rows of a DataFrame in several different ways.

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

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

```
Region            South America
Population                43.59
Life Exp                  75.77
Landlocked                   No
Language                Spanish
Name: Argentina, dtype: object
```

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["Spain"]
```

```
Region            Europe
Population         47.13
Life Exp           83.49
Landlocked            No
Language         Spanish
Name: Spain, dtype: object
```

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

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

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

Use slicing to specify multiple positions

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

Use a list to specify multiple index names

Accessing Values in DataFrames

Country	Region	Population	Life Exp	Landlocked	Language
Argentina	South America	43.59	75.77	No	Spanish
Australia	Oceania	23.99	82.09	No	English
Brazil	South America	200.4	73.12	No	Portuguese
Canada	North America	35.99	80.99	No	English
Chad	Africa	11.63	49.81	Yes	Arabic
China	Asia	1357	74.87	No	Chinese
Egypt	Africa	90.37	70.48	No	Arabic
Germany	Europe	81.46	80.24	No	German
Ireland	Europe	4.64	80.15	No	English
Japan	Asia	126.26	84.36	No	Japanese
Mexico	North America	127.58	75.05	No	Spanish
New Zealand	Oceania	4.66	80.67	No	English
Niger	Africa	18.05	55.13	Yes	French
Nigeria	Africa	186.99	51.3	No	English
Paraguay	South America	6.78	76.99	Yes	Spanish
Portugal	Europe	10.29	80.68	No	Portuguese
South Korea	Asia	51.71	83.23	No	Korean
Spain	Europe	47.13	83.49	No	Spanish
Switzerland	Europe	8.12	82.5	Yes	German
United Kingdom	Europe	65.1	80.09	No	English
United States	North America	321.07	78.51	No	English

Access individual value by column index, then row index

```
df["Life Exp"]["Egypt"]
```

```
70.48
```

```
df["Landlocked"]["Mexico"]
```

```
No
```

Access by row index, then column index

```
df.loc["Spain"]["Population"]
```

```
47.13
```

```
df.iloc[18][4]
```

```
German
```

Access by row position, then column position

```
df.iloc[19]["Region"]
```

```
Europe
```

Access by row position, then column index

Applying Conditions to DataFrames

- We can filter a DataFrame to reduce it to a subset of the original values based on a condition applied to one or more columns in the frame. We do this by indexing with a boolean expression.

```
df["Life Exp"] > 80
```

```
Argentina          False
Australia           True
Brazil             False
Canada              True
Chad               False
China              False
Egypt              False
Germany             True
Ireland             True
Japan               True
Mexico             False
New Zealand         True
Niger              False
Nigeria            False
Paraguay           False
Portugal            True
South Korea         True
Spain               True
Switzerland         True
United Kingdom      True
United States      False
```

```
df[df["Life Exp"] > 80]
```

Filter DataFrame, keeping only rows where 'Life Exp' > 80

Country	Region	Population	Life Exp	Landlocked	Language
Australia	Oceania	23.99	82.09	No	English
Canada	North America	35.99	80.99	No	English
Germany	Europe	81.46	80.24	No	German
Ireland	Europe	4.71	80.15	No	English
Japan	Asia	126.26	84.36	No	Japanese
New Zealand	Oceania	4.66	80.67	No	English
Portugal	Europe	10.29	80.68	No	Portuguese
South Korea	Asia	51.71	83.23	No	Korean
Spain	Europe	47.13	83.49	No	Spanish
Switzerland	Europe	8.12	82.50	Yes	German
United Kingdom	Europe	65.10	80.09	No	English

Applying Conditions to DataFrames

- We can combine several different conditions using a boolean operator such as AND (**&**) or OR (**|**). Note each condition is surrounded by parentheses

```
df[(df["Life Exp"] > 80) & (df["Region"] == "Europe")]
```

Country	Region	Population	Life Exp	Landlocked	Language
Germany	Europe	81.46	80.24	No	German
Ireland	Europe	4.71	80.15	No	English
Portugal	Europe	10.29	80.68	No	Portuguese
Spain	Europe	47.13	83.49	No	Spanish
Switzerland	Europe	8.12	82.50	Yes	German
United Kingdom	Europe	65.10	80.09	No	English

Select rows which satisfy 1st condition AND 2nd condition

```
df[(df["Life Exp"] < 50) | (df["Life Exp"] > 83)]
```

Country	Region	Population	Life Exp	Landlocked	Language
Chad	Africa	11.63	49.81	Yes	Arabic
Japan	Asia	126.26	84.36	No	Japanese
South Korea	Asia	51.71	83.23	No	Korean
Spain	Europe	47.13	83.49	No	Spanish

Select rows which satisfy 1st condition OR 2nd condition

Sorting DataFrames

- When sorting DataFrames, we call the `sort_values()` function and specify a single column to sort by. To sort in descending order, pass the argument `ascending=False`

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

Country	Region	Population	Life Exp	Landlocked	Language
China	Asia	1357	74.87	No	Chinese
United States	North America	321.07	78.51	No	English
Brazil	South America	200.4	73.12	No	Portuguese
Nigeria	Africa	186.99	51.3	No	English

...

- We can also specify a list of multiple columns to sort by:

```
df.sort_values(by=["Landlocked","Population"])
```

Country	Region	Population	Life Exp	Landlocked	Language
Ireland	Europe	4.64	80.15	No	English
New Zealand	Oceania	4.66	80.67	No	English
Portugal	Europe	10.29	80.68	No	Portuguese
Australia	Oceania	23.99	82.09	No	English

...