INTRODUCTION TO DATA SCIENCE
Data Manipulation
Pandas

Jakapun Tachaiya (Ph.D.)
Department of Computer Science
Faculty of Science, KMITL

Outline

01 – What is Pandas? How is it useful for data analytic?

02 – Pandas data type.

03 – Indexing, slicing and operations.

04 – Aggregation and grouping

05 – Working on multiple dataframes.

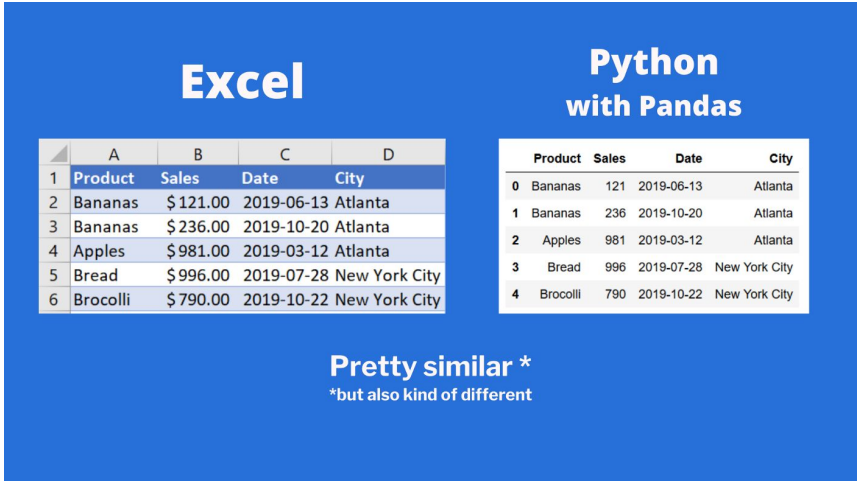
06 – Pivot Table

01

What is Pandas?
How is it useful for data analytic?

What is Pandas?

- All-in-one tools for doing data analysis in Python
  - Inspecting, cleaning, transforming, visualizing and analyzing data.
- Similar to Excel format (row-col, 2D format)
- “Pandas is excel on steroid”
- Built on top of **NumPy**

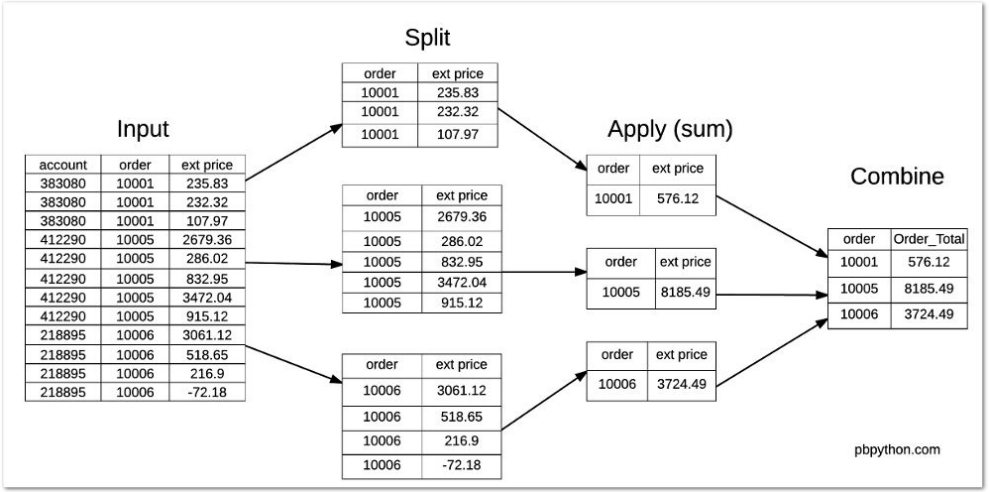


	A	B	C	D
1	Product	Sales	Date	City
2	Bananas	$121.00	2019-06-13	Atlanta
3	Bananas	$236.00	2019-10-20	Atlanta
4	Apples	$981.00	2019-03-12	Atlanta
5	Bread	$996.00	2019-07-28	New York City
6	Brocolli	$790.00	2019-10-22	New York City

	Product	Sales	Date	City
0	Bananas	121	2019-06-13	Atlanta
1	Bananas	236	2019-10-20	Atlanta
2	Apples	981	2019-03-12	Atlanta
3	Bread	996	2019-07-28	New York City
4	Brocolli	790	2019-10-22	New York City

Why Pandas is better than Excel?

1. Limitation by size (excel is limited ~ 1 M rows)
2. Complex data transformation
3. Automation
4. Crossed-platform capabilities



What is Numpy?

- Fast **vectorized array operations** for data munging and cleaning, subsetting and filtering, transformation, and any other kinds of computations
- Efficient descriptive statistics and **aggregating/summarizing** data
- Data alignment and relational data manipulations for **merging and joining** together heterogeneous data sets

import numpy as np

02

Pandas data type

Data

- What is this data?

R011	42ND STREET & 8TH AVENUE	00228985	00008471	00000441	00001455	00000134	00033341	00071255
R170	14TH STREET-UNION SQUARE	00224603	00011051	00000827	00003026	00000660	00089367	00199841
R046	42ND STREET & GRAND CENTRAL	00207758	00007908	00000323	00001183	00003001	00040759	00096613

- Semantics: real-world meaning of the data

Data Terminology

Items

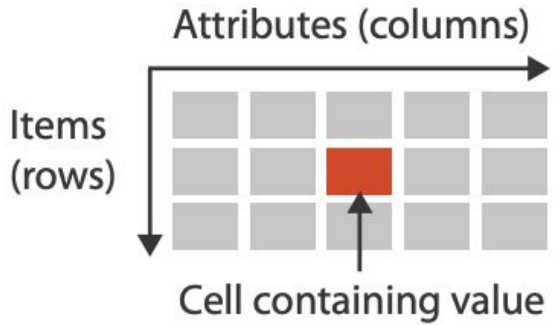
- An item is an individual discrete entity
- **a row in a table**

Attributes

- An **attribute** is some specific property that can be measured, observed, or logged
- variable, (data) dimension
- **a column in a table**

	REMOTE	STATION	FF	SEN/DIS	7-D AFAS UNL
1	R011	42ND STREET & 8TH AVENUE	00228985	00008471	00000441
2	R170	14TH STREET-UNION SQUARE	00224603	00011051	00000827
3	R046	42ND STREET & GRAND CENTRAL	00207758	00007908	00000323
4	R012	34TH STREET & 8TH AVENUE	00188311	00006490	00000498
5	R293	34TH STREET - PENN STATION	00168768	00006155	00000523
6	R033	42ND STREET/TIMES SQUARE	00159382	00005945	00000378
7	R022	34TH STREET & 6TH AVENUE	00156008	00006276	00000487

➔ Tables



Attribute/column Types

May be further specified for computational storage/processing

- **Categorical**: string, boolean, blood type
- **Ordered**: enumeration, t-shirt size, temperature (Hot, warm & cold)
- **Quantitative** (Numeric): integer, float, fixed decimal, datetime

Sometimes, types can be inferred from the data - e.g. numbers and none have decimal points → integer - could be incorrect (data doesn't have floats, but could be)

Categorial, Ordinal, and Quantitative (Numeric) data

A	B	C	S	T	U
Order ID	Order Date	Order Priority	Product Container	Product Base Margin	Ship Date
3	10/14/06	5-Low	Large Box	0.8	10/21/06
6	2/21/08	4-Not Specified	Small Pack	0.55	2/22/08
32	7/16/07	2-High	Small Pack	0.79	7/17/07
32	7/16/07	2-High	Jumbo Box	0.72	7/17/07
32	7/16/07	2-High	Medium Box	0.6	7/18/07
32	7/16/07	2-High	Medium Box	0.65	7/18/07
35	10/23/07	4-Not Specified	Wrap Bag	0.52	10/24/07
35	10/23/07	4-Not Specified	Small Box	0.58	10/25/07
36	11/3/07	1-Urgent	Small Box	0.55	11/3/07
65	3/18/07	1-Urgent	Small Pack	0.49	3/19/07
66	1/20/05	5-Low	Wrap Bag	0.56	1/20/05
69	6/4/05	4-Not Specified	Small Pack	0.44	6/6/05
69	6/4/05	4-Not Spec		0.6	6/6/05
70	12/18/06	5-Low		0.59	12/23/06
70	12/18/06	5-Low		0.82	12/23/06
96	4/17/05	2-High		0.55	4/19/05
97	1/29/06	3-Medium		0.38	1/30/06
129	11/19/08	5-Low		0.37	11/28/08

quantitative
ordinal
categorical

Categorial, Ordinal, and Quantitative (Numeric) data

A	B	C	S	T	U
Order ID	Order Date	Order Priority	Product Container	Product Base Margin	Ship Date
3	10/14/06	5-Low	Large Box	0.8	10/21/06
6	2/21/08	4-Not Specified	Small Pack	0.55	2/22/08
32	7/16/07	2-High	Small Pack	0.79	7/17/07
32	7/16/07	2-High	Jumbo Box	0.72	7/17/07
32	7/16/07	2-High	Medium Box	0.6	7/18/07
32	7/16/07	2-High	Medium Box	0.65	7/18/07
35	10/23/07	4-Not Specified	Wrap Bag	0.52	10/24/07
35	10/23/07	4-Not Specified	Small Box	0.58	10/25/07
36	11/3/07	1-Urgent	Small Box	0.55	11/3/07
65	3/18/07	1-Urgent	Small Pack	0.49	3/19/07
66	1/20/05	5-Low	Wrap Bag	0.56	1/20/05
69	6/4/05	4-Not Specified	Small Pack	0.44	6/6/05
69	6/4/05	4-Not Spec		0.6	6/6/05
70	12/18/06	5-Low		0.59	12/23/06
70	12/18/06	5-Low		0.82	12/23/06
96	4/17/05	2-High		0.55	4/19/05
97	1/29/06	3-Medium		0.38	1/30/06
129	11/19/08	5-Low		0.37	11/28/08

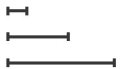
quantitative
ordinal
categorical

➔ Categorical

➔ Ordered

➔ *Ordinal*

➔ *Quantitative*



Pandas datatype

- Pandas **Series** object - 1 dimensional data & Homogenous (each series must have the same datatype on members)
- Pandas **Dataframe** object - 2 dimension data, a combined series.

Series

	apples
0	3
1	2
2	0
3	1

Series

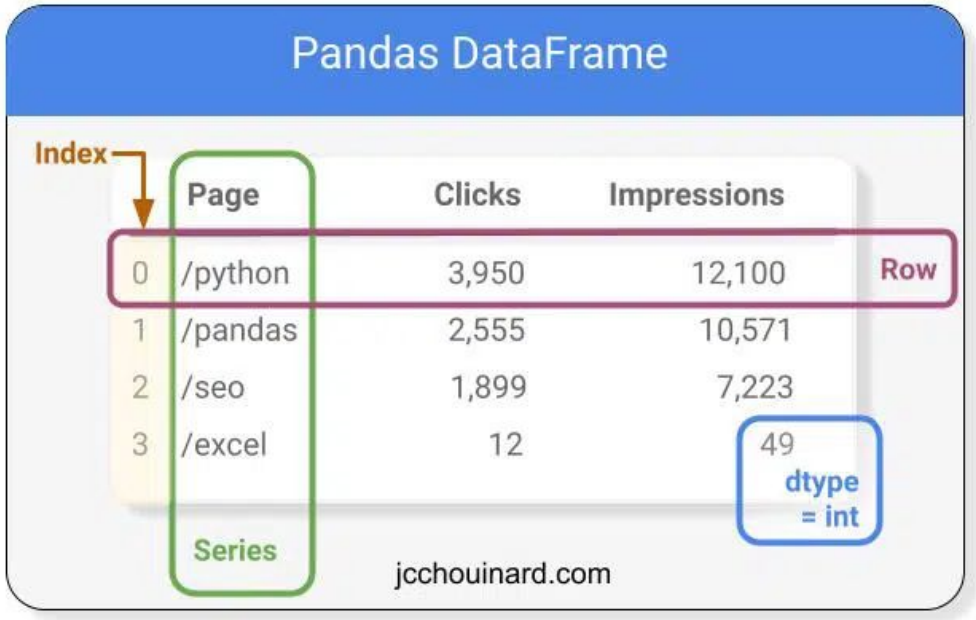
	oranges
0	0
1	3
2	7
3	2

DataFrame

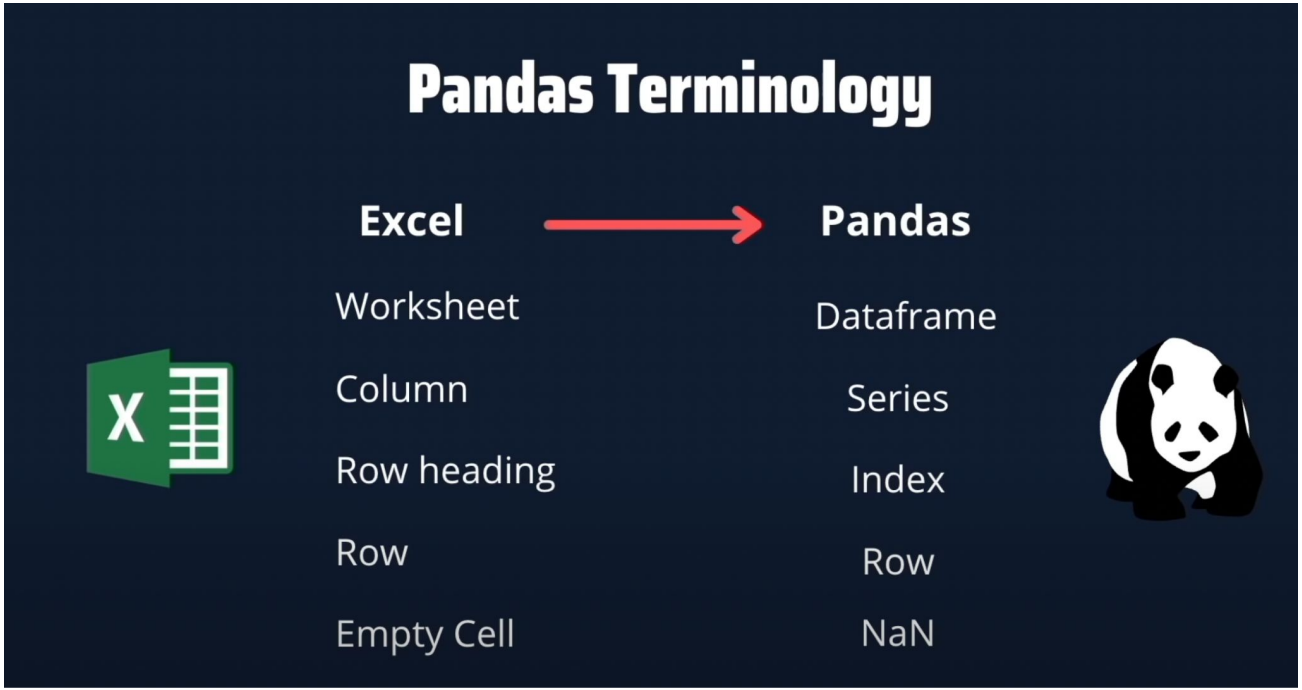
	apples	oranges
0	3	0
1	2	3
2	0	7
3	1	2

Basic Structure of a Pandas Dataframe

Terminology: Rows - Indexes, Columns - Series, and data types of the values(dtype).



Comparing Terminology



Series

- A one-dimensional array (with a type) with an **index**
- Index defaults to numbers but can also be text (like a dictionary)
- Allows easier reference to specific items
- `obj = pd.Series([7,14,-2,1])`
- Basically two arrays: `obj.values` and `obj.index`
- Can specify the index explicitly and use strings
-
```
obj2 = pd.Series([4, 7, -5, 3],
                 index=['d', 'b', 'a', 'c'])
```
- Kind of like fixed-length, ordered dictionary + can create from a dictionary
-
```
obj3 = pd.Series({'Ohio': 35000, 'Texas': 71000,
                  'Oregon': 16000, 'Utah': 5000})
```

Series

```
s2 = pd.Series([1,1,1,1], index=[0,1,2,3])
```

```
0    1
1    1
2    1
3    1
dtype: int64
```

```
import pandas as pd
```

```
s = pd.Series([1,1,1,1], index=[4,3,2,1])
```

```
4    1
3    1
2    1
1    1
dtype: int64
```

Data Frame

- A dictionary of Series (labels for each series)
- A spreadsheet with column headers
- Has an index shared with each series
- Allows easy reference to any cell
-
```
df = DataFrame({'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada'],
                'year': [2000, 2001, 2002, 2001],
                'pop': [1.5, 1.7, 3.6, 2.4]})
```
- Index is automatically assigned just as with a series but can be passed in as well via index kwarg
- Can reassign column names by passing columns kwarg

Data Frame

```
df = pd.read_csv('penguins_lter.csv')
```

Column Names

Index

	studyName	Sample Number	Species	Region	Island	Stage	Individual ID	Clutch Completion	Date Egg	Culmen Length (mm)
0	PAL0708	1	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg Stage	N1A1	Yes	11/11/07	39.1
1	PAL0708	2	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg Stage	N1A2	Yes	11/11/07	39.5
2	PAL0708	3	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg Stage	N2A1	Yes	11/16/07	40.3
3	PAL0708	4	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg Stage	N2A2	Yes	11/16/07	NaN
4	PAL0708	5	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg Stage	N3A1	Yes	11/16/07	36.7
...	...	...	...	...	...	...	...	...	...	...
339	PAL0910	120	Gentoo penguin (Pygoscelis papua)	Anvers	Biscoe	Adult, 1 Egg Stage	N38A2	No	12/1/09	NaN
340	PAL0910	121	Gentoo penguin (Pygoscelis papua)	Anvers	Biscoe	Adult, 1 Egg Stage	N39A1	Yes	11/22/09	46.8
341	PAL0910	122	Gentoo penguin (Pygoscelis papua)	Anvers	Biscoe	Adult, 1 Egg Stage	N39A2	Yes	11/22/09	50.4
342	PAL0910	123	Gentoo penguin (Pygoscelis papua)	Anvers	Biscoe	Adult, 1 Egg Stage	N43A1	Yes	11/22/09	45.2
343	PAL0910	124	Gentoo penguin (Pygoscelis papua)	Anvers	Biscoe	Adult, 1 Egg Stage	N43A2	Yes	11/22/09	49.9

344 rows × 17 columns

How to create Dataframe

1. Arrays (List)

Numpy Array

```
np.array([[1, 4], [2, 5], [3, 6]])
```

List Arrays

```
data = [[1, 4], [2, 5], [3, 6]]
```

	col1	col2
0	1	4
1	2	5
2	3	6

2. Dictionary

```
my_dict = {'key1': value1, 'key2': value2}
```

Column Name — Data

Series (1D array)

3. CSV file.

persons.csv - Notepad

File Edit Format View Help

```
Family Name,Given Name,VIAF ID
Ackersdijck,Willem Cornelis,17959345
Adelung,Friedrich von,22963658
Afzelius,Arvid August,49972119
Amerling,Karel,13331054
Anton,Karl Gottlob von,183632821
Arwidsson,Adolf Ivar,8184878
Asbjørnsen,Peter Christen,116587918
Attems,Heinrich,37665468
```

03

Indexing, slicing and operations

Selecting Column and Row

1. Selecting column - select column/columns from the original dataframe.
2. Selecting row (Indexing) - select desired rows based on conditions.
    a. .loc OR [] is **primarily label based (Boolean Indexing)**.
    b. .iloc is primarily integer **position** OR **index based**.

Selecting Column

```
df = pd.read_csv('penguins_lter.csv')
```

Column Names

	studyName	Sample Number	Species	Region	Island	Stage	Individual ID	Clutch Completion	Date Egg	Culmen Length (mm)
0	PAL0708	1	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg Stage	N1A1	Yes	11/11/07	39.1
1	PAL0708	2	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg Stage	N1A2	Yes	11/11/07	39.5
2	PAL0708	3	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg Stage	N2A1	Yes	11/16/07	40.3
3	PAL0708	4	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg Stage	N2A2	Yes	11/16/07	NaN
4	PAL0708	5	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg Stage	N3A1	Yes	11/16/07	36.7
...	...	...	...	...	...	...	...	...	...	...
339	PAL0910	120	Gentoo penguin (Pygoscelis papua)	Anvers	Biscoe	Adult, 1 Egg Stage	N38A2	No	12/1/09	NaN
340	PAL0910	121	Gentoo penguin (Pygoscelis papua)	Anvers	Biscoe	Adult, 1 Egg Stage	N39A1	Yes	11/22/09	46.8
341	PAL0910	122	Gentoo penguin (Pygoscelis papua)	Anvers	Biscoe	Adult, 1 Egg Stage	N39A2	Yes	11/22/09	50.4
342	PAL0910	123	Gentoo penguin (Pygoscelis papua)	Anvers	Biscoe	Adult, 1 Egg Stage	N43A1	Yes	11/22/09	45.2
343	PAL0910	124	Gentoo penguin (Pygoscelis papua)	Anvers	Biscoe	Adult, 1 Egg Stage	N43A2	Yes	11/22/09	49.9

Index

344 rows × 17 columns

Column: `df['Island']`

Selecting Row (index)

```
df = pd.read_csv('penguins_lter.csv')
```

Column Names

Row: `df.loc[2]`

Index

	studyName	Sample Number	Species	Region	Island	Stage	Individual ID	Clutch Completion	Date Egg	Culmen Length (mm)
0	PAL0708	1	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg Stage	N1A1	Yes	11/11/07	39.1
1	PAL0708	2	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg Stage	N1A2	Yes	11/11/07	39.5
2	PAL0708	3	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg Stage	N2A1	Yes	11/16/07	40.3
3	PAL0708	4	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg Stage	N2A2	Yes	11/16/07	NaN
4	PAL0708	5	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg Stage	N3A1	Yes	11/16/07	36.7
...	...	...	...	...	...	...	...	...	...	...
339	PAL0910	120	Gentoo penguin (Pygoscelis papua)	Anvers	Biscoe	Adult, 1 Egg Stage	N38A2	No	12/1/09	NaN
340	PAL0910	121	Gentoo penguin (Pygoscelis papua)	Anvers	Biscoe	Adult, 1 Egg Stage	N39A1	Yes	11/22/09	46.8
341	PAL0910	122	Gentoo penguin (Pygoscelis papua)	Anvers	Biscoe	Adult, 1 Egg Stage	N39A2	Yes	11/22/09	50.4
342	PAL0910	123	Gentoo penguin (Pygoscelis papua)	Anvers	Biscoe	Adult, 1 Egg Stage	N43A1	Yes	11/22/09	45.2
343	PAL0910	124	Gentoo penguin (Pygoscelis papua)	Anvers	Biscoe	Adult, 1 Egg Stage	N43A2	Yes	11/22/09	49.9

344 rows × 17 columns

Column: `df['Island']`

Selecting Cell

```
df = pd.read_csv('penguins_lter.csv')
```

Column Names

	studyName	Sample Number	Species	Region	Island	Stage	Individual ID	Clutch Completion	Date Egg	Culmen Length (mm)
0	PAL0708	1	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg Stage	N1A1	Yes	11/11/07	39.1
1	PAL0708	2	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg Stage	N1A2	Yes	11/11/07	39.5
2	PAL0708	3	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg Stage	N2A1	Yes	11/16/07	40.3
3	PAL0708	4	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg Stage	N2A2	Yes	11/16/07	NaN
4	PAL0708	5	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg Stage	N3A1	Yes	11/16/07	36.7
...	...	...	...	...	...	...	...	...	...	...
339	PAL0910	120	Gentoo penguin (Pygoscelis papua)	Anvers	Biscoe	Adult, 1 Egg Stage	N38A2	No	12/1/09	NaN
340	PAL0910	121	Gentoo penguin (Pygoscelis papua)	Anvers	Biscoe	Adult, 1 Egg Stage	N39A1	Yes	11/22/09	46.8
341	PAL0910	122	Gentoo penguin (Pygoscelis papua)	Anvers	Biscoe	Adult, 1 Egg Stage	N39A2	Yes	11/22/09	50.4
342	PAL0910	123	Gentoo penguin (Pygoscelis papua)	Anvers	Biscoe	Adult, 1 Egg Stage	N43A1	Yes	11/22/09	45.2
343	PAL0910	124	Gentoo penguin (Pygoscelis papua)	Anvers	Biscoe	Adult, 1 Egg Stage	N43A2	Yes	11/22/09	49.9

344 rows × 17 columns

Row: `df.loc[2]`

Index

Cell: `df.loc[341,'Species']`

Column: `df['Island']`

Pandas Difference Between loc[] vs iloc[]

Operatation	`[]` method	`loc` method	`iloc` method
Select a single column by label	`df['A']`	`df.loc[:, 'A']`	-
Select list of columns by label	`df[['A', 'C']]`	`df.loc[:, ['A', 'C']]`	-
Slice columns by label	-	`df.loc[:, 'A':'C']` *	
Select a single column by position	-	-	`df.iloc[:, 1]`
Select list of columns by position	-	-	`df.iloc[:, [0, 2]]`
Slice columns by position	-	-	`df.iloc[:, 0:2]`
Select a single row by label	-	`df.loc['b']`	-
Select a list of rows by label	-	`df.loc[['b', 'd']]`	-
Slice rows by label	`df['b':'d']` *	`df.loc['b':'d']` *	-
Select a single row by position	-	-	`df.iloc[1]`
Select a list of rows by position	-	-	`df.iloc[[1, 3]]`
Slice rows by position	`df[1:4]`	-	`df.iloc[1:4]`
Select list of rows & columns by label	-	`df.loc[['b', 'd'], ['A', 'C']]`	-
Select list of rows & columns by position	-	-	`df.iloc[[1, 3], [2, 1]]`
Slice rows & columns by label	-	`df.loc['b':'c', 'A':'C']` *	-
Slice rows & columns by position	-	-	`df.iloc[1:3, 0:2]`

* slicing expects an inclusive end of the selection

In [1]:
```
import pandas as pd
# create a DataFrame
index= ['a', 'b', 'c', 'd']
d = {'A': pd.Series([1., 2., 3., 4.], index=index),
     'B': pd.Series([5., 6., 7., 8.], index=index),
     'C': pd.Series([9., 10., 11., 12.], index=index)}
df = pd.DataFrame(d)
df
```

Out[1]:

	A	B	C
a	1.0	5.0	9.0
b	2.0	6.0	10.0
c	3.0	7.0	11.0
d	4.0	8.0	12.0

In [2]:
```
# slicing rows and columns using labels (inclusive ends)
df.loc['b':'c', 'A':'C']
```

Out[2]:

	A	B	C
b	2.0	6.0	10.0
c	3.0	7.0	11.0

In [3]:
```
# slicing rows and columns using positions (exclusive ends)
df.iloc[1:3, 0:3]
```

Out[3]:

	A	B	C
b	2.0	6.0	10.0
c	3.0	7.0	11.0

Example on operations on dataframe

```
# calculating the average score and assigning the result to a new column
df_exams['average'] = (df_exams['math score'] + df_exams['reading score'] + df_exams['writing score'])/3
```

```
# showing the dataframe
df_exams
```

	gender	race/ethnicity	parental level of education	lunch	test preparation course	math score	reading score	writing score	average
0	female	group B	bachelor's degree	standard	none	72	72	74	72.666667
1	female	group C	some college	standard	completed	69	90	88	82.333333
2	female	group B	master's degree	standard	none	90	95	93	92.666667
3	male	group A	associate's degree	free/reduced	none	47	57	44	49.333333
4	male	group C	some college	standard	none	76	78	75	76.333333
...	...	...	...	...	...	...	...	...	...
995	female	group E	master's degree	standard	completed	88	99	95	94.000000
996	male	group C	high school	free/reduced	none	62	55	55	57.333333
997	female	group C	high school	free/reduced	completed	59	71	65	65.000000
998	female	group D	some college	standard	completed	68	78	77	74.333333
999	female	group D	some college	free/reduced	none	77	86	86	83.000000

Operations on Dataframe

1. **Dropping Column/row**
2. **DataFrame Access and Manipulation**
3. **Basic statistical functions** - within col operation. sum(), mean(), std(), max(), min(), ptile(), count(), value_counts()
4. **Replace Nan Value**
5. **Sort Dataframe** by col / cols
6. **Multiple columns' operation** - average values from multiple cols.
7. **Custom function** - create customized function in python.

1. Dropping Column/row

pandas.DataFrame.drop

```
DataFrame.drop(labels=None, *, axis=0, index=None, columns=None,
level=None, inplace=False, errors='raise')
```

- Can drop one or more entries
- Series:
  - `new_obj = obj.drop('c')`
  - `new_obj = obj.drop(['d', 'c'])`
- Data Frames:
  - `axis` keyword defines which axis to drop (default 0)
  - `axis=0` → rows, `axis=1` → columns
  - `axis = 'columns'`

	Courses	Fee	Duration	Discount
0	Spark	20000	30day	1000
1	PySpark	25000	40days	2300
2	Hadoop	26000	NaN	1500
3	Python	22000	None	1200

```
df2 = df.drop("Fee",axis = 1)
print(df2)
```

	Courses	Duration	Discount
0	Spark	30day	1000
1	PySpark	40days	2300
2	Hadoop	NaN	1500
3	Python	None	1200

```
df2 = df.drop(0,axis = 0)
print(df2)
```

	Courses	Fee	Duration	Discount
1	PySpark	25000	40days	2300
2	Hadoop	26000	NaN	1500
3	Python	22000	None	1200

2. DataFrame Access and Manipulation

DataFrame Access and Manipulation

- `df.values → 2D NumPy array`
- Accessing a column:
  - `df["<column>"]`
  - `df.<column>`
  - Both return Series
  - Dot syntax only works when the column is a valid identifier
- Assigning to a column:

```
- df["<column>"] = <scalar> # all cells set to same value
- df["<column>"] = <array>  # values set in order
- df["<column>"] = <series> # values set according to match
                            # between df and series indexes
```

	Courses	Fee	Duration	Discount
0	Spark	20000	30day	1000
1	PySpark	25000	40days	2300
2	Hadoop	26000	NaN	1500
3	Python	22000	None	1200

```
df["Fee"] = 1000
print(df)
```

	Courses	Fee	Duration	Discount
0	Spark	1000	30day	1000
1	PySpark	1000	40days	2300
2	Hadoop	1000	NaN	1500
3	Python	1000	None	1200

```
df["Fee"] = [10,20,30,40]
print(df)
```

	Courses	Fee	Duration	Discount
0	Spark	10	30day	1000
1	PySpark	20	40days	2300
2	Hadoop	30	NaN	1500
3	Python	40	None	1200

3. Basic statistical functions

within col operation. sum(), mean(), std(), max(), min(), ptile(), count(), value_counts()

	Courses	Fee	Duration	Discount
0	Spark	20000	30day	1000
1	PySpark	25000	40days	2300
2	Hadoop	26000	NaN	1500
3	Python	22000	None	1200

```
print(df["Fee"].mean())
```

```
23250.0
```

```
print(df["Courses"].value_counts())
```

```
Spark      1
PySpark    1
Hadoop     1
Python     1
Name: Courses, dtype: int64
```

4. Replace Nan

```
df2 = df.fillna(0)
print(df2)
```

	Courses	Fee	Duration	Discount
0	Spark	20000.0	30day	1000
1	PySpark	0.0	40days	2300
2	Hadoop	26000.0	0	1500
3	Python	22000.0	0	1200

	Courses	Fee	Duration	Discount
0	Spark	20000	30day	1000
1	PySpark	25000	40days	2300
2	Hadoop	26000	NaN	1500
3	Python	22000	None	1200

```
df['Fee'] = df['Fee'].fillna(0)
print(df)
```

	Courses	Fee	Duration	Discount
0	Spark	20000.0	30day	1000
1	PySpark	0.0	40days	2300
2	Hadoop	26000.0	NaN	1500
3	Python	22000.0	None	1200

5. Sort Dataframe

Sorting by Value (sort_values)

- `sort_values` method on series
  - `obj.sort_values()`
- Missing values (`NaN`) are at the end by default (`na_position` controls, can be first)
- `sort_values` on DataFrame:
  - `df.sort_values(<list-of-columns>)`
  - `df.sort_values(by=['a', 'b'])`
  - Can also use `axis=1` to sort by index labels

	Courses	Fee	Duration	Discount
0	Spark	20000.0	30day	1000
1	PySpark	NaN	40days	2300
2	Hadoop	26000.0	NaN	1000
3	Python	22000.0	None	1200

```
df = df.sort_values(['Fee'], ascending=False)
print(df)
```

	Courses	Fee	Duration	Discount
2	Hadoop	26000.0	NaN	1000
3	Python	22000.0	None	1200
0	Spark	20000.0	30day	1000
1	PySpark	NaN	40days	2300

```
df = df.sort_values(["Discount",'Fee'], ascending=False)
print(df)
```

	Courses	Fee	Duration	Discount
1	PySpark	NaN	40days	2300
3	Python	22000.0	None	1200
2	Hadoop	26000.0	NaN	1000
0	Spark	20000.0	30day	1000

6. Multiple columns' operation

	Courses	Fee	Duration	Discount
0	Spark	20000.0	30day	1000
1	PySpark	NaN	40days	2300
2	Hadoop	26000.0	NaN	1000
3	Python	22000.0	None	1200

```
df["total"] = df["Fee"] - df["Discount"]
print(df)
```

	Courses	Fee	Duration	Discount	total
0	Spark	20000.0	30day	1000	19000.0
1	PySpark	NaN	40days	2300	NaN
2	Hadoop	26000.0	NaN	1000	25000.0
3	Python	22000.0	None	1200	20800.0

	Courses	Fee	Duration	Discount
0	Spark	20000.0	30day	1000
1	PySpark	NaN	40days	2300
2	Hadoop	26000.0	NaN	1000
3	Python	22000.0	None	1200

7. Custom function

Custom function with lambda

```
def discountPercent(fee,discount):
    percent = (discount*100)/fee
    return percent

df["discount_Percent"] = df.apply(lambda x: discountPercent(x["Fee"],x["Discount"],),axis = 1)
print(df)
```

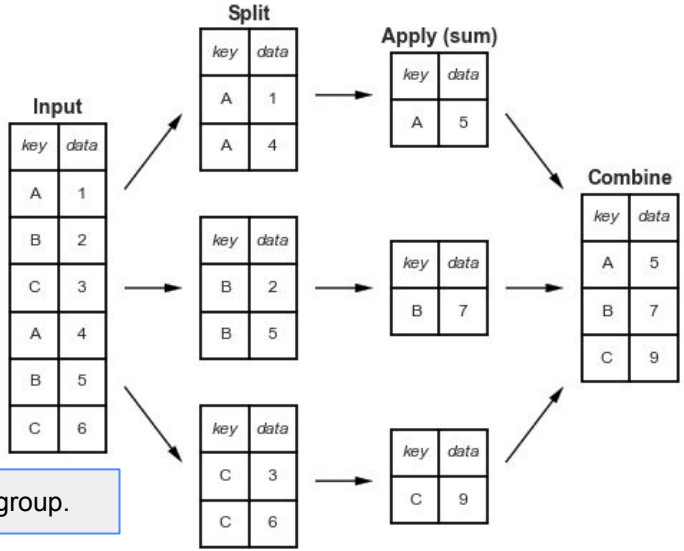
	Courses	Fee	Duration	Discount	discount_Percent
0	Spark	20000.0	30day	1000	5.000000
1	PySpark	NaN	40days	2300	NaN
2	Hadoop	26000.0	NaN	1000	3.846154
3	Python	22000.0	None	1200	5.454545

04
Aggregation and Grouping

Group by

GroupBy : A way to aggregate value in the table with a group with the same value.

- 3 Steps:
  1. Split - breaking up and grouping a DataFrame on the value of the specified key.
  2. Apply - some function, aggregation, transformation, or filtering, within the individual groups.
  3. Combine - merges the results from each groups.



Find the sum value in each key group.

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4
..	...	...	...	...	...	...	...
239	29.03	5.92	Male	No	Sat	Dinner	3
240	27.18	2.00	Female	Yes	Sat	Dinner	2
241	22.67	2.00	Male	Yes	Sat	Dinner	2
242	17.82	1.75	Male	No	Sat	Dinner	2
243	18.78	3.00	Female	No	Thur	Dinner	2

Tips table

```sql
SELECT sex, count(*)
FROM tips
GROUP BY sex;
/*
Female     87
Male      157
*/
```

```
In [19]: tips.groupby("sex").size()
Out[19]:
sex
Female     87
Male      157
dtype: int64
```

SQL

Pandas

Aggregation and Grouping

Aggregations - sum(), mean(), median(), min(), and max()

- gives insight into the nature of a potentially large dataset.

Aggregation	Description
`count()`	Total number of items
`first()`, `last()`	First and last item
`mean()`, `median()`	Mean and median
`min()`, `max()`	Minimum and maximum
`std()`, `var()`	Standard deviation and variance
`mad()`	Mean absolute deviation
`prod()`	Product of all items
`sum()`	Sum of all items

dataframe

	method	number	orbital_period	mass	distance	year
0	Radial Velocity	1	269.300	7.10	77.40	2006
1	Radial Velocity	1	874.774	2.21	56.95	2008
2	Radial Velocity	1	763.000	2.60	19.84	2011
3	Radial Velocity	1	326.030	19.40	110.62	2007
4	Radial Velocity	1	516.220	10.50	119.47	2009

Function ".describe()" on dataframe

	number	orbital_period	mass	distance	year
count	498.00000	498.000000	498.000000	498.000000	498.000000
mean	1.73494	835.778671	2.509320	52.068213	2007.377510
std	1.17572	1469.128259	3.636274	46.596041	4.167284
min	1.00000	1.328300	0.003600	1.350000	1989.000000
25%	1.00000	38.272250	0.212500	24.497500	2005.000000
50%	1.00000	357.000000	1.245000	39.940000	2009.000000
75%	2.00000	999.600000	2.867500	59.332500	2011.000000
max	6.00000	17337.500000	25.000000	354.000000	2014.000000

Examples on Groupby (1)

For each group of **method**, we want to find the median of **orbital_period** value.

dataframe

	method	number	orbital_period	mass	distance	year
0	Radial Velocity	1	269.300	7.10	77.40	2006
1	Radial Velocity	1	874.774	2.21	56.95	2008
2	Radial Velocity	1	763.000	2.60	19.84	2011
3	Radial Velocity	1	326.030	19.40	110.62	2007
4	Radial Velocity	1	516.220	10.50	119.47	2009

```
1 planets.groupby('method')['orbital_period'].median()
```

```
method
Astrometry                         631.180000
Eclipse Timing Variations         4343.500000
Imaging                          27500.000000
Microlensing                      3300.000000
Orbital Brightness Modulation        0.342887
Pulsar Timing                       66.541900
Pulsation Timing Variations       1170.000000
Radial Velocity                    360.200000
Transit                              5.714932
Transit Timing Variations           57.011000
Name: orbital_period, dtype: float64
```

Examples on Groupby (2)

Get the first score of GRE test for each person.

Name	Attempt	GRE Score
Jim	First	298
Jim	Second	321
Jim	Third	314
Pam	First	318
Pam	Second	330

Jim	298
Pam	318

```
# the first GRE score for each student
df.groupby('Name')['GRE Score'].first()
```

First value in each Group

05
Working on multiple dataframes

Concatenation And Append.

- Data come from combining different data sources and time.
- Concat - function that can combine 2 dataframe into one in either direction.

```
df1
```

	A	B
1	A1	B1
2	A2	B2

```
df2
```

	A	B
3	A3	B3
4	A4	B4

```
pd.concat([df1, df2])
```

	A	B
1	A1	B1
2	A2	B2
3	A3	B3
4	A4	B4

```
df3
```

	A	B
0	A0	B0
1	A1	B1

```
df4
```

	C	D
0	C0	D0
1	C1	D1

```
pd.concat([df3, df4], axis='col')
```

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1

Combining Datasets: Merge and Join

Pandas has in-memory **join** operations, very similar to relational databases like SQL.

1. Categories of Joins - *one-to-one*, *many-to-one*, and *many-to-many* joins.
2. How to join/merge - 4 different kinds of joins.

`df1`

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

`df2`

	employee	hire_date
0	Lisa	2004
1	Bob	2008
2	Jake	2012
3	Sue	2014

How do you combine these 2 dataframe?

Combining Datasets: Merge and Join

Pandas has in-memory **join** operations, very similar to relational databases like SQL.

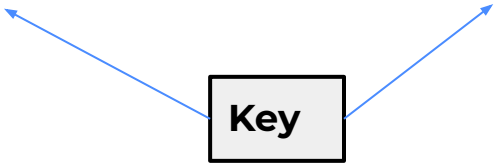
1. Categories of Joins - *one-to-one*, *many-to-one*, and *many-to-many* joins.
2. How to join/merge - 4 different kinds of joins.

df1

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df2

	employee	hire_date
0	Lisa	2004
1	Bob	2008
2	Jake	2012
3	Sue	2014

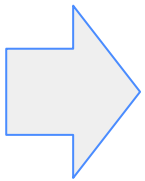


df1

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df2

	employee	hire_date
0	Lisa	2004
1	Bob	2008
2	Jake	2012
3	Sue	2014



	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

Categories of Joins(1)

1. One-to-one join - values in key columns are **unique** in the columns on **both** tables.
2. Many-to-one join - values in one of key column are **unique** in one table and **duplicated** in the other table**.**
3. Many-to-many join - values in key columns are **duplicated** in the columns on **both** tables.

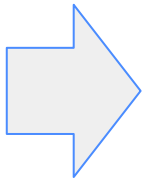
One-to-one join

df1

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df2

	employee	hire_date
0	Lisa	2004
1	Bob	2008
2	Jake	2012
3	Sue	2014



	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

To identify category of joins: 1) locate key and 2) check if key rows contain dup/unique value in both table.

Categories of Joins(2)

Many-to-one join

`df3`

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

`df4`

	group	supervisor
0	Accounting	Carly
1	Engineering	Guido
2	HR	Steve

`pd.merge(df3, df4)`

	employee	group	hire_date	supervisor
0	Bob	Accounting	2008	Carly
1	Jake	Engineering	2012	Guido
2	Lisa	Engineering	2004	Guido
3	Sue	HR	2014	Steve

Merge by 2 key columns [“group”]

Categories of Joins(3)

Many-to-Many join

`df1`

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

`df5`

	group	skills
0	Accounting	math
1	Accounting	spreadsheets
2	Engineering	coding
3	Engineering	linux
4	HR	spreadsheets
5	HR	organization

`pd.merge(df1, df5)`

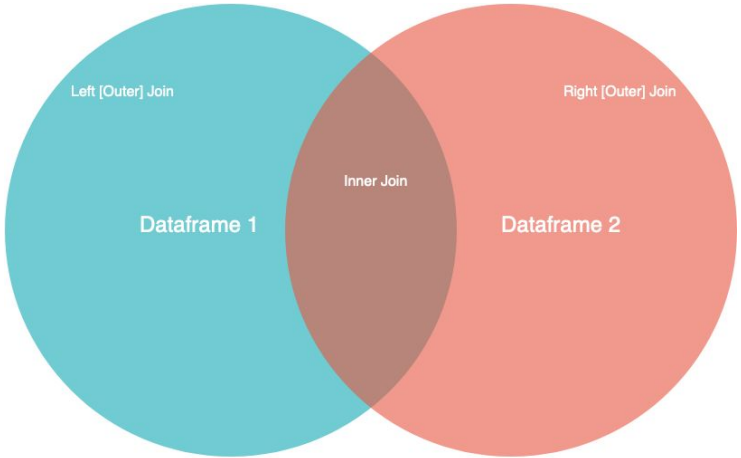
	employee	group	skills
0	Bob	Accounting	math
1	Bob	Accounting	spreadsheets
2	Jake	Engineering	coding
3	Jake	Engineering	linux
4	Lisa	Engineering	coding
5	Lisa	Engineering	linux
6	Sue	HR	spreadsheets
7	Sue	HR	organization

Merge by 1 key column [“group”]

How to join/merge

4 different kinds of joins

1. Inner - Use **intersection** of keys from both frames [default].
2. Outer - Use **union** of keys from both frames.
3. Left - Use **keys from left** frame only.
4. Right - Use **keys from right** frame only.



4 different kinds of joins (1)

Inner - Use **intersection** of keys from both dataframes [default].

`df6`

	name	food
0	Peter	fish
1	Paul	beans
2	Mary	bread

`df7`

	name	drink
0	Mary	wine
1	Joseph	beer

`pd.merge(df6, df7)`

	name	food	drink
0	Mary	bread	wine

Outer - Use **union** of keys from both dataframes.

`df6`

	name	food
0	Peter	fish
1	Paul	beans
2	Mary	bread

`df7`

	name	drink
0	Mary	wine
1	Joseph	beer

What does the outer join of df6 and df7 look like?

4 different kinds of joins (2)

Outer - Use **union** of keys from both dataframes.

`df6`

	name	food
0	Peter	fish
1	Paul	beans
2	Mary	bread

`df7`

	name	drink
0	Mary	wine
1	Joseph	beer

`pd.merge(df6, df7, how='outer')`

	name	food	drink
0	Peter	fish	NaN
1	Paul	beans	NaN
2	Mary	bread	wine
3	Joseph	NaN	beer

4 different kinds of joins (3)

Left - Use **keys from left** dataframe only.

`df6`

	name	food
0	Peter	fish
1	Paul	beans
2	Mary	bread

`df7`

	name	drink
0	Mary	wine
1	Joseph	beer

`pd.merge(df6, df7, how='left')`

	name	food	drink
0	Peter	fish	NaN
1	Paul	beans	NaN
2	Mary	bread	wine

What will a df look like with pd.merge(df6,df7, how = right)?

06

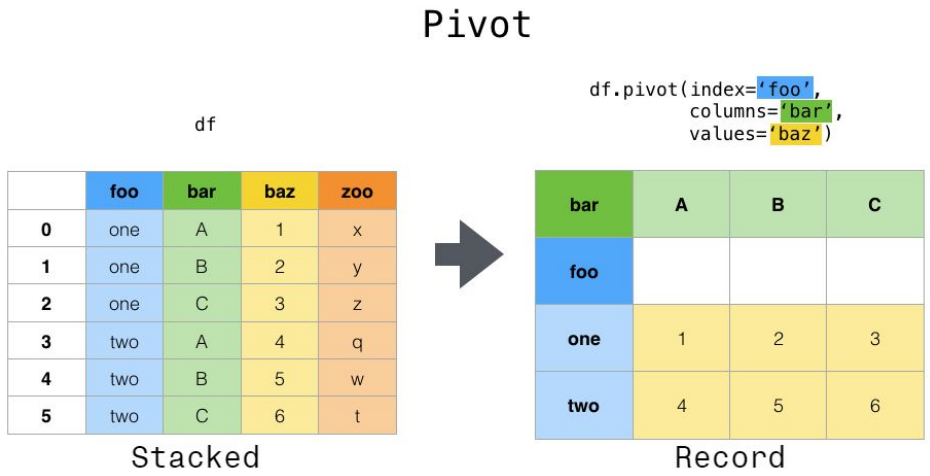
Pivot Tables

What is Pivot table?

- A data manipulation tool that **rearranges a table** and sometimes **aggregates the values** for easy analysis.
- **Group the data by one or more columns** and then **summarize the values** using various statistics such as mean, sum, and count.

Pivot_table syntax

```
pandas.pivot_table(data,
                   values=None,
                   index=None,
                   columns=None,
                   aggfunc='mean',
                   fill_value=None,
                   margins=False,
                   dropna=True,
                   margins_name='All',
                   observed=False,
                   sort=True)
```

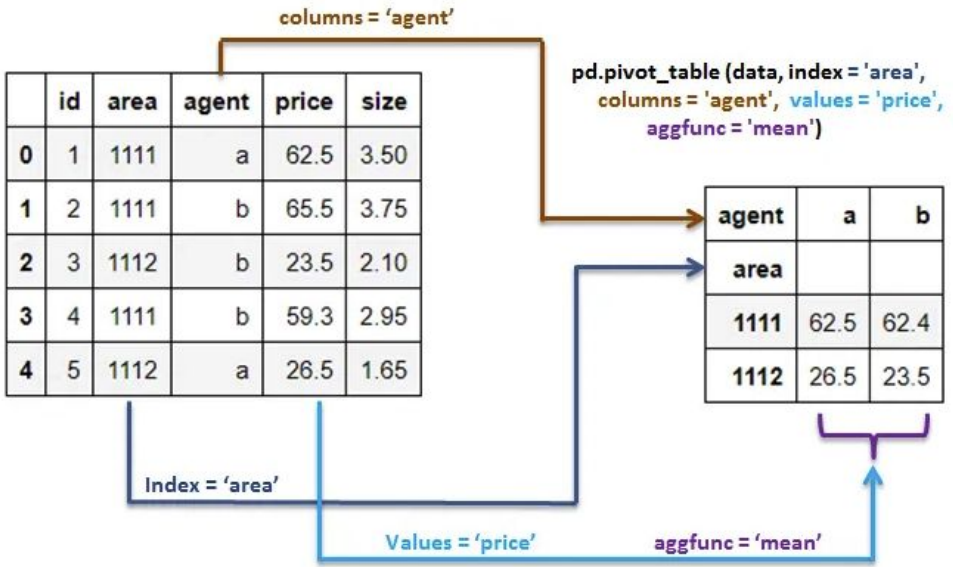


	foo	bar	baz	zoo
0	one	A	1	x
1	one	B	2	y
2	one	C	3	z
3	two	A	4	q
4	two	B	5	w
5	two	C	6	t

bar	A	B	C
foo			
one	1	2	3
two	4	5	6

Pivot table example (1)

An easy-to-use analogy — In short, the pivot table syntax says that for every `'index'`, return the `'aggfunc'` of the `'values'` column(s), segregated (further grouped) by `'columns'`.



	id	area	agent	price	size
0	1	1111	a	62.5	3.50
1	2	1111	b	65.5	3.75
2	3	1112	b	23.5	2.10
3	4	1111	b	59.3	2.95
4	5	1112	a	26.5	1.65

agent	a	b
area		
1111	62.5	62.4
1112	26.5	23.5

Pivot table example (2)

	guests	nights	meal_plan	room_type	lead_time	year	month	market	room_price	status
5305	2	2	Meal Plan 1	Room_Type 1	127	2018	7	Online	114.30	Not Canceled
31271	2	3	Meal Plan 1	Room_Type 1	279	2018	10	Offline	110.00	Canceled
4440	3	4	Meal Plan 1	Room_Type 4	126	2018	5	Online	106.03	Not Canceled
972	2	3	Meal Plan 1	Room_Type 1	2	2018	3	Online	101.00	Not Canceled
2889	2	5	Meal Plan 1	Room_Type 2	5	2017	12	Complementary	1.60	Not Canceled
22349	2	4	Meal Plan 1	Room_Type 1	89	2018	5	Online	119.85	Not Canceled
30842	2	1	Not Selected	Room_Type 1	153	2018	7	Online	94.50	Not Canceled
23665	2	1	Meal Plan 1	Room_Type 1	1	2018	9	Corporate	184.00	Not Canceled
23994	2	4	Meal Plan 1	Room_Type 1	36	2018	1	Online	60.29	Canceled
27390	2	3	Meal Plan 2	Room_Type 1	105	2017	10	Offline	110.00	Canceled

the Hotel Reservations Dataset from Kaggle available under the Attribution 4.0 International (CC BY 4.0)

Pivot table example (2)

```
pd.pivot_table(data,
               index='market',
               values = 'room_type',
              aggfunc = 'nunique')
```

In the code above, for every `'market'` group, return the number of `unique` `'room_types'`.

	room_type
market	
Aviation	2
Complementary	7
Corporate	7
Offline	7
Online	7

Aggregate function

'Aggfunc' parameter (Default: aggfunc = 'mean')

This is the function to aggregate the values (or choose the value to return) per group.

The table below contains the commonly used `aggfunc` options and whether they apply to only numeric functions or both numeric and categorical functions.

Numeric columns:

'mean' — average of each group
'sum' — total sum per group
'median' — median value per group
'std' — standard deviation
'var' — variance for each group
'mad' — mean absolute deviation
'prod' — product of values per group

Both numeric and categorical columns

'count' — number of rows per group
'min' — minimum value per group
'max' — maximum value
'first' — first row value per group
'last' — last row value
'nunique'- number of unique values per group

Choosing Aggregate functions

`aggfunc` dictionary — we can also have `{'values':'aggfunc'}` dictionary pairs where we pass different functions for different `values` columns.

```
pd.pivot_table(data,
               index = 'market',
               aggfunc = {'room_type': 'count',
                          'lead_time':'max',
                          'room_price': 'mean'})
```

Here, for every `'market'` group, return the `count` of `'room_types'`, the `maximum` `'lead_time'`, and the `average` `'price'`.

	lead_time	room_price	room_type
market			
Aviation	23	100.704000	125
Complementary	386	3.141765	391
Corporate	297	82.911740	2017
Offline	443	91.632679	10528
Online	443	112.256855	23214

Reset Index

`df.reset_index()` — The results produced in the examples above are grouped by the `index` parameter, making it the new index. You can use `df.reset_index()` to reset this column to a regular column, which also adds a RangeIndex with integers from 0.

	lead_time	room_price	room_type
market			
Aviation	23	100.704000	125
Complementary	386	3.141765	391
Corporate	297	82.911740	2017
Offline	443	91.632679	10528
Online	443	112.256855	23214

```
df = pd.pivot_table(data,
                    index = 'market',
                    aggfunc = {'room_type': 'count',
                               'lead_time':'max',
                               'room_price': 'mean'})
df.reset_index()
```

	market	lead_time	room_price	room_type
0	Aviation	23	100.704000	125
1	Complementary	386	3.141765	391
2	Corporate	297	82.911740	2017
3	Offline	443	91.632679	10528
4	Online	443	112.256855	23214

Image by author

Multi-index column headers

		id				price				size			
	agent	a		b		a		b		a		b	
	sold	no	yes	no	yes	no	yes	no	yes	no	yes	no	yes
area	year												
1111	2014	NaN	NaN	NaN	2.0	NaN	NaN	NaN	65.5	NaN	NaN	NaN	3.75
	2015	1.0	NaN	NaN	4.0	62.5	NaN	NaN	59.3	3.5	NaN	NaN	2.95
1112	2014	NaN	5.0	3.0	NaN	NaN	26.5	23.5	NaN	NaN	1.65	2.1	NaN

Multi-index example

```
pd.pivot_table(data,
               index = 'market',
               columns = ['status','year'],
               values = 'room_price',
               aggfunc = 'mean',
               fill_value=0).head()
```

In the code above, we group the data by `'market'` and then return the `average` `'room price'` further grouped by `status` and `year`.

status (index 0 - level 0)	Canceled		Not Canceled	
year (index 1 - level 1)	2017	2018	2017	2018
market				
Aviation	0.000000	102.243243	0.000000	100.056818
Complementary	0.000000	0.000000	2.452734	3.521825
Corporate	84.265116	98.229322	76.480747	83.028502
Offline	97.670602	101.350321	86.852632	88.115008
Online	91.057495	116.123477	99.369477	113.183294

columns = ['status','year']