# Pandas

→ For 1-D data, we use pandas.Series created through the pd.Series constructor
data → elements of series
dtype → np.float32 (use np types)

Each element has an index and each element is referred as the label.

custom element index can be set
index = ['a', 8, 0.3]

Another way to set index is to pass a python dictionary in the data argument.

→ 2-D data
we use the pandas.DataFrame object
essentially the same arguments as series
also takes columns argument which are similar to index but for columns
columns = ['c1', 'c2']

When dictionary is passed as data, keys are used as column keys.

Upcasting occurs on a per column basis

→ new_df = df1.append(df2)
append returns a new df and does NOT modify the original.

Scanned by CamScanner

when appending a series to df, we need to pass the name of the series or use ignore_indxe = True.

df2 = df1.append(ser, ignore_index = True)

Dropping

using labels to specify labels of the rows/columns we want to drop.
We use this alongside the axis keyword which has default value 0

Second method is to directly use index or columns keywords arguments without having to use axis.

```
df_drop = df.drop(labels = 'r1'
          df.drop(labels = ['c1', 'c2'], axis = 1)
          df.drop(index = 'r2)
          df.drop(columns = 'c2')
          df.drop(index = 'r2', columns = 'c2')
```

Drop will always return a new df and NOT modify the original.

When using labels and axis, we can't drop both rows and columns from the df.

Scanned by CamScanner

## Concat

pd.concat([df1, df2], axis=0)

pd.concat takes in a list of pandas objects (normally list of df) to concat.

axis argument specifies whether we concat rows (axis=0) or columns (axis=1).

If row labels do not match, NaN is padded

## Merging

pd.merge takes two required arguments - the two df to merge.

without using arguments, merge joins two df using all their column common labels.

The rows that contain the exact values for the common column labels will be merged.

pd.merge(df1, df2)

## Indexing

```
df = pd.DataFrame({'c1': [1,2],
                   'c2': [3,4],
                   'c3': [5,6]},
                   index = ['r1','r2'])
```

col1 = df['c1'] → returns series

| | |
|---|---|
| r1 | 1 |
| r2 | 2 |

Scanned by CamScanner

col1_df = df[['c1']] → return df

| | c1 |
|---|---|
| r1 | 1 |
| r2 | 2 |

col23 = df[['c2','c3']] → return df

| | c2 | c3 |
|---|---|---|
| r1 | 3 | 5 |
| r2 | 4 | 6 |

can also use df[0:2] ← exclusive
df['r2':'r3'] ← inclusive
for slicing.

df['r1'] → will result in error as column name is expected.

iloc to access rows based on their integer index.
df.iloc[1]
df.iloc[[0,2]]
df.iloc[bool_list]

loc uses row labels rather than indexes
df.loc['r2']
→ to get single value → df.loc['r1','c2']
df.loc[['r1','r2'],'c2']

Scanned by CamScanner

## Reading data

CSV

pd.readcsv('filename.csv')

we can set the index_col key word arg to specify which column we want to use as index. By default integer indexes are used as row labels

Excel

pd.read_excel('filname.xlsx')

when we dont use any keyword argument, it only returns the first sheet. When we set the sheet_name keyword, we obtain specific spreadsheet using its integer index or name. Sheet_name can also take an array. (sheet_name = None) not sure what this does

JSON

pd.read_json('filename.json')

When we dont use any keyword argument, it treats each outer key of json data as column label and each inner key as row label.

When we set orient='index' outer keys are treated as row labels and the inner keys are treated as column labels.

Scanned by CamScanner

## Writing to files

CSV

```
to_csv('filename.csv')
```

When we dont specify keyword arguments, it will write row labels as first column. If they are not useful, do index = False

- Excel

to_excel will only write a single df to a spreadsheet. To write multiple,

```
with pd.ExcelWriter('data.xlsx') as writer:
    df1.to_excel(writer, index=False, sheet_name='mlb')
```

JSON

```
df.to_json('data.json', orient='index')
```

same usage as in reading.

## Grouping

groupby

```
groups = df.groupby('year')
for name, group in groups:
    print('year : {y}'.format(name))
    print('{y}'.format(group))

groups.get_group(2016)
groups.sum()
groups.mean()
groups.filter(lambda x: x.name > 2015)
```

Scanned by CamScanner

Also possible to group by multiple columns

df.groupby(['yearid', 'teamid'])

Features

Columns of the df are referred as the features of the dataset

Quantitative feature - height or weight is a feature that can be measured ~~no~~ numerically. These are features we could calculate sum, mean or other metrics for.

Categorical feature - gender or birthplace is one where the values are categories that could be used to group the data~~base~~set.

→ df.sum()
→ df.sum(axis=1)
→ df.mean()
→ df.mean(axis=1)

Weighted features -
default axis is 1.
df.multiply(2)
df.multiply([1000, 1], axis=0)

Scanned by CamScanner

## Filtering

Using relation operations as filter conditions

cruz = df['playerId'] == 'cruz'
>
!=

for strings
df2 = df['playerId'].str.startswith('c')
.str.endswith
.str.contains
= ~df['playerId'].str.contains
not contains

df2 = df['year'].isin([2016, 2017])
.isna
.notna } for NaN

for applying filters on colums

df2 = df[df['HR'] > 40]

Scanned by CamScanner

## Sorting

```
sort1 = df.sort_values('yearId')
sort2 = df.sort_values('playerId', ascending=False)
```

default is True

for multiple colums

```
sort3 = df.sort_values(['yearId', 'playerId'],
                       ascending=[True, False])
```

each additional label here is used to break ties. i-1 gets more preference than i

## Metrics

for Quantitative

| Metric | Description |
|---|---|
| count | Number of rows in dataframe |
| mean | Mean value for a feature |
| std | Standard deviation " " " |
| min | minimum value in a feature |
| 25% | 25th percentile of a feature |
| 50% | 50th " . Identical to median |
| 75% | 75th percentile of a feature |
| max | Maximum value in a feature |

```
metrics = df.describe()
metrics = df.describe(percentiles=[0.2, 0.8])
```

for specific percentiles

Scanned by CamScanner

For categorical
we don't calculate mean, median, etc. Instead we use frequency counts.

df.value_counts()

setting normalize = True returns frequency proportions rather than count. We can also set ascending

df.value_counts(normalize = True)
df.value_counts(ascending = True)

If we just want unique values
df['playerId'].unique()

Plotting
import matplotlib.pyplot as plt.
df.plot()
plt.show()
plt.savefig('df.png') → save to png or pdf.

to set details
plt.title('HR vs Year')
plt.xlabel('Year')
plt.ylabel('HR count')

df.plot(kind='hist')
kind='box'

Scanned by CamScanner

2.5GHz with Turbo Boost up to 3.1GHz
1000 GB HDD

## toNumpy

Most machine learning frameworks use NumPy directly. So df has to be converted to np.

Even the categorical features like gender and birthplace have to be converted.

The easiest way to do this is to convert each categorical feature into set of indicator features

→ converted = pd.get_dummies(df)
print(converted.columns)

After converting, we can then convert to a NumPy matrix
n-matrix = converted.values

Scanned by CamScanner