Day 1

Pandas Cheet Sheet :

Pandas_Cheat_Sheet.pdf

Link :

https://python-tricks.com/data-structure-in-pandas/

Pandas

Data Wrangling

with pandas Cheat Sheet
http://pandas.pydata.org

Pandas API Reference Pandas User Guide

Tidy Data – A foundation for wrangling in pandas

In a tidy data set:

&

Each **variable** is saved in its own **column**

Each **observation** is saved in its own **row**

Tidy data complements pandas's **vectorized operations**. pandas will automatically preserve observations as you manipulate variables. No other format works as intuitively with pandas.

Creating DataFrames

	a	b	c
1	4	7	10
2	5	8	11
3	6	9	12

```
df = pd.DataFrame(
          {"a" : [4, 5, 6],
           "b" : [7, 8, 9],
           "c" : [10, 11, 12]},
         index = [1, 2, 3])
```
Specify values for each column.

```
df = pd.DataFrame(
     [[4, 7, 10],
      [5, 8, 11],
      [6, 9, 12]],
     index=[1, 2, 3],
     columns=['a', 'b', 'c'])
```
Specify values for each row.

		a	b	c
N	v			
D	1	4	7	10
	2	5	8	11
e	2	6	9	12

```
df = pd.DataFrame(
          {"a" : [4 ,5, 6],
           "b" : [7, 8, 9],
           "c" : [10, 11, 12]},
index = pd.MultiIndex.from_tuples(
        [('d', 1), ('d', 2),
         ('e', 2)], names=['n', 'v']))
```
Create DataFrame with a MultiIndex

Method Chaining

Most pandas methods return a DataFrame so that another pandas method can be applied to the result. This improves readability of code.

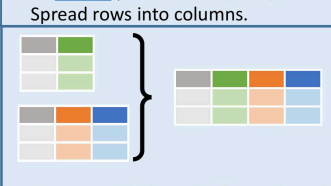
```
df = (pd.melt(df)
        .rename(columns={
                'variable':'var',
                'value':'val'})
        .query('val >= 200')
     )
```

Reshaping Data – Change layout, sorting, reindexing, renaming

pd.melt(df)
Gather columns into rows.

df.pivot(columns='var', values='val')
Spread rows into columns.

pd.concat([df1,df2])
Append rows of DataFrames



pd.concat([df1,df2], axis=1)
Append columns of DataFrames

df.sort_values('mpg')
Order rows by values of a column (low to high).

df.sort_values('mpg', ascending=False)
Order rows by values of a column (high to low).

df.rename(columns = {'y':'year'})
Rename the columns of a DataFrame

df.sort_index()
Sort the index of a DataFrame

df.reset_index()
Reset index of DataFrame to row numbers, moving index to columns.

df.drop(columns=['Length', 'Height'])
Drop columns from DataFrame

Subset Observations - rows

df[df.Length > 7]
Extract rows that meet logical criteria.
df.drop_duplicates()
Remove duplicate rows (only considers columns).
df.sample(frac=0.5)
Randomly select fraction of rows.
df.sample(n=10) Randomly select n rows.
df.nlargest(n, 'value')
Select and order top n entries.
df.nsmallest(n, 'value')
Select and order bottom n entries.
df.head(n)
Select first n rows.
df.tail(n)
Select last n rows.

Subset Variables - columns

df[['width', 'length', 'species']]
Select multiple columns with specific names.
df['width'] *or* **df.width**
Select single column with specific name.
df.filter(regex='*regex*')
Select columns whose name matches regular expression *regex*.

Using query

query() allows Boolean expressions for filtering rows.

```
df.query('Length > 7')
df.query('Length > 7 and Width < 8')
df.query('Name.str.startswith("abc")',
         engine="python")
```

Subsets - rows and columns

Use **df.loc[]** and **df.iloc[]** to select only rows, only columns or both.
Use **df.at[]** and **df.iat[]** to access a single value by row and column.
First index selects rows, second index columns.

df.iloc[10:20]
Select rows 10-20.
df.iloc[:, [1, 2, 5]]
Select columns in positions 1, 2 and 5 (first column is 0).
df.loc[:, 'x2':'x4']
Select all columns between x2 and x4 (inclusive).
df.loc[df['a'] > 10, ['a', 'c']]
Select rows meeting logical condition, and only the specific columns .
df.iat[1, 2] Access single value by index
df.at[4, 'A'] Access single value by label

Logic in Python (and pandas)

<	Less than	!=	Not equal to
>	Greater than	df.column.isin(*values*)	Group membership
==	Equals	pd.isnull(*obj*)	Is NaN
<=	Less than or equals	pd.notnull(*obj*)	Is not NaN
>=	Greater than or equals	&,|,~,^,df.any(),df.all()	Logical and, or, not, xor, any, all

regex (Regular Expressions) Examples

'\.'	Matches strings containing a period '.'
'Length$'	Matches strings ending with word 'Length'
'^Sepal'	Matches strings beginning with the word 'Sepal'
'^x[1-5]$'	Matches strings beginning with 'x' and ending with 1,2,3,4,5
'^(?!Species$).*'	Matches strings except the string 'Species'

Cheatsheet for pandas (http://pandas.pydata.org/ originally written by Irv Lustig, Princeton Consultants, inspired by Rstudio Data Wrangling Cheatsheet

Summarize Data

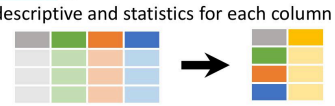
`df['w'].value_counts()`
Count number of rows with each unique value of variable

`len(df)`
# of rows in DataFrame.

`df.shape`
Tuple of # of rows, # of columns in DataFrame.

`df['w'].nunique()`
# of distinct values in a column.

`df.describe()`
Basic descriptive and statistics for each column (or GroupBy).



pandas provides a large set of summary functions that operate on different kinds of pandas objects (DataFrame columns, Series, GroupBy, Expanding and Rolling (see below)) and produce single values for each of the groups. When applied to a DataFrame, the result is returned as a pandas Series for each column. Examples:

`sum()`
Sum values of each object.

`count()`
Count non-NA/null values of each object.

`median()`
Median value of each object.

`quantile([0.25,0.75])`
Quantiles of each object.

`apply(function)`
Apply function to each object.

`min()`
Minimum value in each object.

`max()`
Maximum value in each object.

`mean()`
Mean value of each object.

`var()`
Variance of each object.

`std()`
Standard deviation of each object.

Group Data

`df.groupby(by="col")`
Return a GroupBy object, grouped by values in column named "col".

`df.groupby(level="ind")`
Return a GroupBy object, grouped by values in index level named "ind".

All of the summary functions listed above can be applied to a group.
Additional GroupBy functions:

`size()`
Size of each group.

`agg(function)`
Aggregate group using function.

The examples below can also be applied to groups. In this case, the function is applied on a per-group basis, and the returned vectors are of the length of the original DataFrame.

`shift(1)`
Copy with values shifted by 1.

`rank(method='dense')`
Ranks with no gaps.

`rank(method='min')`
Ranks. Ties get min rank.

`rank(pct=True)`
Ranks rescaled to interval [0, 1].

`rank(method='first')`
Ranks. Ties go to first value.

`shift(-1)`
Copy with values lagged by 1.

`cumsum()`
Cumulative sum.

`cummax()`
Cumulative max.

`cummin()`
Cumulative min.

`cumprod()`
Cumulative product.

Windows

`df.expanding()`
Return an Expanding object allowing summary functions to be applied cumulatively.

`df.rolling(n)`
Return a Rolling object allowing summary functions to be applied to windows of length n.

Handling Missing Data

`df.dropna()`
Drop rows with any column having NA/null data.

`df.fillna(value)`
Replace all NA/null data with value.

Make New Columns

`df.assign(Area=lambda df: df.Length*df.Height)`
Compute and append one or more new columns.

`df['Volume'] = df.Length*df.Height*df.Depth`
Add single column.

`pd.qcut(df.col, n, labels=False)`
Bin column into n buckets.

pandas provides a large set of **vector functions** that operate on all columns of a DataFrame or a single selected column (a pandas Series). These functions produce vectors of values for each of the columns, or a single Series for the individual Series. Examples:

`max(axis=1)`
Element-wise max.

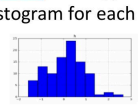
`min(axis=1)`
Element-wise min.

`clip(lower=-10,upper=10)`
Trim values at input thresholds

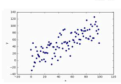
`abs()`
Absolute value.

Plotting

`df.plot.hist()`
Histogram for each column



`df.plot.scatter(x='w',y='h')`
Scatter chart using pairs of points



Combine Data Sets

adf

x1	x2
A	1
B	2
C	3

+

bdf

x1	x3
A	T
B	F
D	T

=

Standard Joins

x1	x2	x3
A	1	T
B	2	F
C	3	NaN

`pd.merge(adf, bdf, how='left', on='x1')`
Join matching rows from bdf to adf.

x1	x2	x3
A	1.0	T
B	2.0	F
D	NaN	T

`pd.merge(adf, bdf, how='right', on='x1')`
Join matching rows from adf to bdf.

x1	x2	x3
A	1	T
B	2	F

`pd.merge(adf, bdf, how='inner', on='x1')`
Join data. Retain only rows in both sets.

x1	x2	x3
A	1	T
B	2	F
C	3	NaN
D	NaN	T

`pd.merge(adf, bdf, how='outer', on='x1')`
Join data. Retain all values, all rows.

Filtering Joins

x1	x2
A	1
B	2

`adf[adf.x1.isin(bdf.x1)]`
All rows in adf that have a match in bdf.

x1	x2
C	3

`adf[~adf.x1.isin(bdf.x1)]`
All rows in adf that do not have a match in bdf.

ydf

x1	x2
A	1
B	2
C	3

+

zdf

x1	x2
B	2
C	3
D	4

=

Set-like Operations

x1	x2
B	2
C	3

`pd.merge(ydf, zdf)`
Rows that appear in both ydf and zdf (Intersection).

x1	x2
A	1
B	2
C	3
D	4

`pd.merge(ydf, zdf, how='outer')`
Rows that appear in either or both ydf and zdf (Union).

x1	x2
A	1

```
pd.merge(ydf, zdf, how='outer',
         indicator=True)
.query('_merge == "left_only"')
.drop(columns=['_merge'])
```
Rows that appear in ydf but not zdf (Setdiff).

Cheatsheet for pandas (http://pandas.pydata.org/) originally written by Irv Lustig, Princeton Consultants, inspired by Rstudio Data Wrangling Cheatsheet

What is Pandas?

Pandas is a powerful and easy-to-use open-source data analysis and data manipulation library for Python. It provides two primary data structures:

- **Series**: 1D labeled array
- **DataFrame**: 2D labeled data structure (like a table)

Pandas is widely used for handling and analyzing structured data, including reading, cleaning, and manipulating datasets.

---

1. Installation

If you haven't installed Pandas yet, you can do so by using pip:

```
pip install pandas
```

---

2. Basic Concepts and Structures in Panda

Series (1D Data)

A Pandas Series is a one-dimensional array with labeled data. You can think of it like a column in a spreadsheet or a SQL database table.

```
import pandas as pd
```

```
# Creating a Series
data = [10, 20, 30, 40, 50]
series = pd.Series(data)

print(series)
```

Output:

```
0    10
1    20
2    30
3    40
4    50
dtype: int64
```

You can also specify custom indices:

```
series = pd.Series(data, index=['a', 'b', 'c', 'd', 'e'])
print(series)
```

Output:

```
a    10
b    20
c    30
d    40
e    50
dtype: int64
```

DataFrame (2D Data)

A DataFrame is a 2-dimensional table, where each column can be of different data types. A DataFrame is made up of a collection of Series objects, all sharing the same index.

```
# Creating a DataFrame from a dictionary
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'City': ['New York', 'Los Angeles', 'Chicago']
}

df = pd.DataFrame(data)
print(df)
```

Output:

```
      Name  Age         City
0    Alice   25     New York
1      Bob   30  Los Angeles
2  Charlie   35      Chicago
```

---

3. Reading and Writing Data

Reading Data

Pandas supports a variety of formats to read data, including CSV, Excel, SQL, and JSON.

- **CSV**:

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

- **Excel**:

```
df = pd.read_excel('path_to_file.xlsx')
```

- **JSON**:

```
df = pd.read_json('path_to_file.json')
```

Writing Data

You can also write data to different formats:

```
# Write DataFrame to CSV
df.to_csv('output.csv', index=False)

# Write DataFrame to Excel
df.to_excel('output.xlsx', index=False)

# Write DataFrame to JSON
df.to_json('output.json')
```

---

4. DataFrame Operations

Selecting Data

You can select columns and rows in various ways.

- **Select a Column**:

```
print(df['Name'])  # Selects the 'Name' column
```

- **Select Multiple Columns**:

```
print(df[['Name', 'Age']])  # Selects 'Name' and 'Age' columns
```

- **Select Rows by Index**:

```
print(df.iloc[0])  # Selects the first row
```

- **Select Rows by Condition**:

```
# Select rows where Age is greater than 30
print(df[df['Age'] > 30])
```

Output:

```
    Name  Age     City
2  Charlie   35  Chicago
```

Sorting Data

- **Sort by Column**:

```
print(df.sort_values(by='Age'))  # Sort by 'Age'
```

Adding/Modifying Columns

You can easily add or modify columns.

```
df['Salary'] = [50000, 60000, 70000]  # Add new column 'Salary'
df['Age'] = df['Age'] + 1  # Modify 'Age' column by adding 1 year
print(df)
```

Output:

```
      Name  Age         City  Salary
0    Alice   26     New York   50000
1      Bob   31  Los Angeles   60000
2  Charlie   36      Chicago   70000
```

Handling Missing Data

Pandas has built-in support for handling missing data (NaN values).

- **Check for Missing Data**:

```
print(df.isnull())  # Returns True for missing data
```

- **Drop Rows with Missing Data**:

```
df.dropna(inplace=True)  # Drop rows with any missing data
```

- **Fill Missing Data**:

```
df.fillna(0, inplace=True)  # Replace NaN values with 0
```

---

5. Advanced DataFrame Operations

Grouping Data

Pandas allows you to group data and apply aggregate functions.

```
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Edward'],
    'Age': [25, 30, 35, 30, 25],
    'City': ['New York', 'Los Angeles', 'Chicago', 'New York', 'Chicago']
}

df = pd.DataFrame(data)

# Group by 'Age' and get count of each age
grouped = df.groupby('Age').size()
print(grouped)
```

Output:

```
Age
25    2
30    2
```

```
35    1
dtype: int64
```

Merging DataFrames

You can merge DataFrames using similar methods to SQL joins.

```
df1 = pd.DataFrame({'ID': [1, 2, 3], 'Name': ['Alice', 'Bob', 'Charlie']})
df2 = pd.DataFrame({'ID': [1, 2, 4], 'Age': [25, 30, 40]})

# Merge on 'ID'
merged_df = pd.merge(df1, df2, on='ID', how='inner')
print(merged_df)
```

Output:

```
   ID     Name  Age
0   1    Alice   25
1   2      Bob   30
```

---

6. Visualization with Pandas

Pandas integrates well with libraries like Matplotlib to visualize data. Here's an example:

```
import matplotlib.pyplot as plt

# Creating a simple bar chart of 'Age' distribution
df['Age'].value_counts().plot(kind='bar')
plt.show()
```

---

7. Common Pandas Functions Summary

- `df.head()`: Displays the first few rows of the DataFrame
- `df.tail()`: Displays the last few rows
- `df.describe()`: Gives summary statistics
- `df.info()`: Displays info about the DataFrame including data types
- `df.columns`: List column names
- `df.shape`: Returns the shape (number of rows, columns)
- `df.dtypes`: Displays data types of each column