Data Analyst in Python

**Course: Introduction to Data Science in Python**

**Chapter 1**

**Section 1 - Dive Into Python**

Remember to import the appropriate modules

Alias modules – *import pandas as pd*

**Section 2 - Creating Variables**

Must start with a letter (usually lowercase), then letters, numbers, underscores, but no special characters.

Case-sensitive

**Section 3 - Functions**

Section of code (action) that converts some input to a given output

plt.plot(x-value, y-value, label=’label’)

Common errors: missing commas (syntax), missing parenthesis (syntax)

**Chapter 2**

**Section 1 - pandas**

pandas is a module for working with tabular data

load from multiple sources

search for particular rows or columns

calculate aggregate data

combine data from different sources

DataFrame loading is the first step

Easiest way to create DataFrame is using a .csv file

Always add import pandas as pd

head() function prints first 5 rows

info() method gives a summary of the data you’ve imported

**Section 2 - Selecting Columns**

Calculate data from column… (sum of numbers, etc)

Plot data from columns…

**Section 3 - Selecting Rows with Logic**

Return true or false when checking = = or > or < or >= or !=

= = checks for equality while = sets a value

credit\_records[credit\_records.price > 20.00] >>> selecting rows where the column price is greater than 20.00

Booleans (only two, true and false)

**Chapter 3 - Plotting Data with matplotlib**

**Section 1 - Creating line plots**

Import matplotlib\

from matplotlib import pyplot as plt

plt.plot(dataframe\_x\_values.column\_name, dataframe\_y\_values.column\_name)  
plt.show()

multiple lines use a 2nd, or 3rd, etc. plt.plot statement before plt.show()

**Section 2 - Adding text to plots**

plt.xlabel(“Label for X”)

plt.ylabel(“Label for Y”)

plt.title(“Plot Title”, fontsize=xx) #fontsize is optional

plt.plot(dataframe\_x\_values.column\_name, dataframe\_y\_values.column\_name, label=”Label for Legend”)

plt.legend(color=”green”) #color is optional

plt.text(x\_coord, y\_coord, “Text Message”)

plt.show()

**Section 3 - Styling graphs**

**Chapter 4**

Section 1

Section 2 - Bar Charts

plt.bar(arg1, arg2, yerr=dataframe.column\_for\_error)  
plt.ylabel(“Label”)  
plt.show()

plt.barh() # plot horizontal bars

Stacked Bar chart

plt.bar(x\_column, y\_column1, label=’Label’)  
plt.bar(x\_column, y\_column2, bottom=y\_column1, label=’Label’)  
plt.legend()  
plt.show()

Section 3 - Histograms

plt.hist(dataframe.column, bins=[number of bins], range=(xmin, xmax))

Normalizing

Compare differing sample sizes on portion of total sample (normalized to 1)

Add keyword: density=true

Section 4

**Course: Intermediate Python**

Chapter 1

Section 1

Section 2

Section 3

Section 4

Chapter 2

Section 1

Section 2

Section 3

Section 4

Chapter 3

Section 1

Section 2

Section 3

Section 4

Chapter 4

Section 1

Section 2

Section 3

Section 4

**Assessment: Python Programming**

**Course: Data Manipulation with pandas**

Chapter 1

Section 1

Section 2

**Section 3 - New Columns**

Add new columns when existing data doesn’t meet your needs

Mutate, transform, feature engineering

dataframe[“new\_column”] = dataframe[“existing\_column”] {math}

dataframe[“new\_column”] = dataframe[“column1”] \* dataframe[“column2”]

**Chapter 2 - Aggregating Data**

**Section 1 - Summary Statistics**

df[“column”].mean()

.median(), .mode(), min(), max(), var(), std()

.agg() allows for the use of custom calculations for analysis

df[“column”].agg(function\_name) for single column

df[[“column1”, “column2”]].agg(function\_name) for multiple columns

use .agg() to get multiple summaries: df[“column”].agg([function1\_name, function2\_name])

.cumsum() summation of data moving down the row also: .cummax(), .cummin(),.cumprod()

**Section 2 - Counting**

Dropping duplicates: df.drop\_duplicates(subset=”column”) for a single column

Multiple columns: df.drop\_duplicates(subset=[“column1”, “column2”])

Get counts of unique items: df[“column”].value\_counts()

Sort: df[“column”].value\_counts(sort=True)

Normalize to distribution: df[“column”].value\_counts(normalize=True)

**Section 3 - Grouped Summary Statistics**

df[df[“column”] == “criteria”][“column2”].mean() find average of column2

Group: df.groupby(“grouping\_column”)[“investigative\_column”].mean()

Using agg() for multiple statistics: df.groupby(“group\_column”)[“investigative\_column”].agg([min, max, sum])

Group by multiple columns: df.groupby([“column1”, “column2”])[“investigate\_column”].mean()

Group & aggregate multiples: df. groupby([“column1”, “column2”])[[“investigate\_column”, “i\_column2”]].mean()

**Section 4 - Pivot Tables**

Pivot tables help summarize and group data: df.pivot\_table(values=”column”, index=”column”)

Group by index, values are columns to be summarized

Multiple stats: df.pivot\_table(values=”column”, index=”column”, aggfunc=[np.mean, np.median])

Multiple columns: df.pivot\_table(values=”column”, index=”column”, columns=”column2”)

Replace null/NaN with value: df.pivot\_table(values=”column”, index=”column”, columns=”column2”, fill\_value=0)

Mean: Multiple columns: df.pivot\_table(values=”column”, index=”column”, columns=”column2”, margins=True)

**Chapter 3**

**Section 1 - Explicit Indexes**

df\_ind = df.set\_index(“name”) >>> can accept a list as well > [“index1”, “index2”]

df\_ind.reset\_index() >>> can accept drop=True argument to remove the index altogether

indexes make sub-setting with .loc easier

If multiple indexes, subset outer level using .loc >>> df\_ind.loc[[“index1”, “index2”]]

If multiple levels, pass list of tuples to .loc >>> df\_ind.loc[[(“index1”, “index2”), (“index3”, “index4”)]]

Sort by index using df\_ind.sort\_index()

Control order and level by passing lists >>> df\_ind.sort\_index(level=[“index1”, “index2”], ascending=[True, False])

**Section 2 - Slicing and subsetting with .loc and .iloc**

Slice >>> list[x:y] includes x but not y

Slice dataset: first sort by index, then use .loc >>> use index values >>> list\_sort.loc[“index1”:“index2”]

Slicing inner index levels: list\_sort.loc[(“index1”, “index2”):(“index3”, “index4”)]

Slicing dataset columns: list\_sort[:, “column1”:”column2”]

Slice on rows and columns >>> list\_sort.loc[(“index1”, “index2”):(“index3”, “index4”), “column1”:”column2”]

Slicing dates, pass dates as strings, but can pass partial dates like “2014”

Slicing with iloc uses row/column numbers >>> list.iloc[row\_x:row\_y, column\_a:column\_b]

**Section 3 - Working with pivot tables**

Can use .loc and slicing on pivot tables

Calculate mean by axis: pivot\_table.mean(axis=”index”)

Calculate across columns using pivot\_table.mean(axis=”columns”) >>> every column has the same data type

**Chapter 4 - Creating & Visualizing DataFrames**

**Section 1 - Visualizing Data**

Use pyplot from matplotlib >>> import as plt is standard…

Create histogram >>> data\_frame[“column”].hist() then plt.show()

Change bins >>> data\_frame[“column”].hist(bins=\_\_) then plt.show()

Plot group stats >>> avg = data\_frame.groupby(“column\_or\_index”)[“column2”].mean() then…

Graph using avg.plot(kind=”bar”) then plt.show()

Line plots show variables over time well

Pass in multiple arguments: data.plot(x=”column\_or\_index”, y=”column”, kind=”line”) then plt.show()

Pass angle of rotation for x-axis labels using rot argument with a number

Scatter plot to show correlation >>> data\_frame.plot(x=”column1”, y=”column2”, kind=”scatter”) then plt.show()

Histogram with multiple data:

data[data[“column”] == \_\_\_][“column2”].hist(alpha = 0.7) #alpha changes transparency  
data[data[“column”] == \_\_\_][“column2”].hist(alpha = 0.7)  
plt.legend([“label1”, “label2”])  
plt.show()

**Section 2 - Missing Data**

Check for missing values >>> dataframe.isna()

Chain >>> dataframe.isna().any()

Sum Booleans >>> dataframe.isna().sum()

Can chart the missing value data as well…

Remove missing rows?

Drop rows >>> dataframe.dropna()

Replace missing values >>> dataframe.fillna(\_\_value\_\_)

**Section 3 - Creating Dataframes**

Dictionary >>> key/value pairs held >>> dictionary = {“key”: value}

From list of dictionaries: row-by-row

>>> first dictionary, second dictionary, etc… keys become column name, values become values

>>> pd.DataFrame(list\_of\_dictionaries)

From dictionary of lists: column by column

>>> column names come from keys, column values come from list values for each key

**Section 4 - Reading and writing to CSVs**

CSV files are great ways to store and transmit data and are easily convertible and usable by multiple systems

Import: dataframe = pd.read\_csv(“file\_name”)

Export: dataframe.to\_csv(“file\_name”)

**Course: Joining Data with pandas**

**Chapter 1 - Data Merging Basics**

**Section 1 - Inner Joins**

To merge dataframes >>> dataframe\_merged = df1.merge(df2, on=’common\_column’)

Use df.head() and df.shape() to gauge where commonalities may exist…

Inner joins will exclude data where no commonality exists

Columns with like names will automatically be re-named or can be manually suffixed

To override auto-naming >>> dataframe\_merged = df1.merge(df2, on=’common\_column’, suffixes=(‘df1’, df2’))

**Section 2 - One to Many Relationships**

Relationships between tables can vary (one-to-one or one-to-many)

One-to-one has only one relation between ‘left’ and ‘right’ tables

One-to-many has one record in ‘left’ table, but can have many records in the ‘right’ table

Merging one-to-many can result in duplicate data in rows coming from the ‘left’ table

**Section 3**

Multiple tables merges are often necessary to fully evaluate your data

To merge multiple tables >>> df\_merged = df1.merge(df2, on=[‘column1’, ‘column2’]).merge(df3, on=’column’, suffixes=(‘suffix1’, ‘suffix2’)) # can use backslash to run on multiple lines in python file

Merging on two columns require that both columns match in both tables

Can keep going for many more tables >>> df1.merge(df2, on=”column1”) \

.merge(df3, on=”column2”) \

.merge(df3, on=”column3”) #etc

**Chapter 2 – Merging Data With Different Join Types**

**Section 1 - Left Join**

Return all tables from left columns from ‘left’ table and only rows from ‘right’ table where there is a match

To left join, add how=’left’ >>> df\_merged = df1.merge(df2, on=’column’, how=’left’)

**Section 2 - Other Joins**

Reverse of left joins, keeps all rows and data from ‘right’ table and only joins where data on ‘left’ matches

If you have different column names >>> df\_merged = df1.merge(df2, how=’right’, left\_on=’column1’, right\_on=’column2’)

Outer join preserves all rows from ‘left’ and ‘right’ table whether there is a match or not

To outer join >>> df\_merged = df1.merge(df2, how=’outer’, suffixes=(‘\_df1’, ‘\_df2’))

**Section 3 - Merging a table to itself**

Left and right tables will be the same >>> df\_merged = df1.merge(df1, left\_on=’column’, right\_on=’column’)

Inner join is performed, so some records may drop out depending on the data structure

LEFT or RIGHT joins can also be performed using the “how” argument

Can be used with hierarchical or sequential data

**Section 4 - Merging on Indexes**

To set index from csv data >>> df = pd.read\_csv(‘file.csv’, index\_col=[‘column’])

Method automatically adjusts to use index names or column names, so you pass the index name to the ‘on’ argument.

Can also use multi-index merges >>> df\_merged = df1.merge(df2, on=[‘index1’, ‘index2’])

Use left\_on and right\_on arguments if index names are different

Also specify true/false for index >>> df\_merged = df1.merge(df2, left\_on=’id1’, left\_index=True, right\_on=’id2’, right\_index=True)

**Chapter 3 – Advanced Merging and Concatenating**

**Section 1**

**Section 2**

**Section 3**

**Section 4**

**Chapter 4 – Merging Ordered and Time-Series Data**

**Section 1**

**Section 2**

**Section 3**

**Section 4**

**Assessment: Data Manipulation with Python**

**Course: Introduction to Data Visualization with Matplotlib**