**DATA ANALYSIS STEPS WITH PYTHON**

#If it’s the first time working with excel, import the 'pip install xlrd and openpyxl' as well, it reads excel files

#A Personal advice before starting the project is to just look at the excel file. Sometimes when I do, I start formulating hypothesis not related to the study

1. **Import Necessary Labs to use**

import pandas as pd

import numpy as np

import seaborn as sns

import matplotlib.pyplot as plt

from sklearn.impute import SimpleImputer

1. **Import the file and assign a variable name**
2. If its just one sheet to be imported

**df = pd.read\_excel(r'C:\Github\Fullstack-Data-Analyst\data\_projects\Hrsa\_Awards\grant\_data.xlsx')**

1. If its more than one sheet

# df\_sheet0 = df.parse(0)

# df\_sheet1 = df.parse(1)

1. If you want just one specific sheet in the file

data = pd.read\_excel(r'Pathname', sheet\_name='Name of Sheet')

1. If you want to align column name to the right

variable = (path, skipinitialspace=True) or 'skip\_blank\_lines=True

1. **Print the Imported data**
2. Print the data

print(df)

1. Print the information of the data

print(df.info())

1. Print a description of the data

print(df.describe())

**Something to Note**

The print(data) – Just prints the data

The print(info) – You get, number of columns, names of the columns, data types, the memory size & number of rows

The print(describe) – Provides standard deviation, minimum, maximum, 25%, 50%, 75% percentiles

1. **Modify the tiles of the columns(when coding, its advised not to have spaces in the data)**
2. Clear spaces from column names or The above replace goes for anything else you want taken from title

df.columns = df.columns.str.replace(' ', '')

1. Rename one column

df.rename(columns = {'Abstract': 'Abunuwasi'}, inplace=True)

1. Rename All Columns

df.columns = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', '1', '2', '3', '4', '5', '6']

The retrieved data has 32 columns each value above represents a column, Only use **(c)** if you are sure

For my data, I split the zip codes from extensions then removed the old column and the the extension, same way with phone

1. **Handling Data Types**
2. Look through the data to decide correct data types(common data types are – object, int64, datetime64[ns],float64

df.info()

1. Change everything to a desired data type

df = df.astype(str)

1. Change specific columns to specific data type

df = df.astype({'AwardYear': 'int64', 'GrantSerialNumber': 'int64', 'ProjectPeriodStartDate': 'datetime64[ns]', 'GrantProjectPeriodEndDate': 'str', 'DataWarehouseRecordCreateDate': 'datetime64[ns]', 'GrantProgramDescription': 'object', 'DUNSNumber': 'float64', 'UniqueEntityIdentifier' : 'object', 'GeocodingArtifactAddressPrimaryXCoordinate': 'float64', 'GeocodingArtifactAddressPrimaryYCoordinate': 'float64'})

1. Change one column to specific data type. The error is to force python to ignore errors – works per column

df['GrantProjectPeriodEndDate'] = pd.to\_datetime(df['GrantProjectPeriodEndDate'], errors='coerce')

1. Sometimes one may encounter errors e.g. datetime can only be within 584 years, anything above it raises an error. The ‘coerce’ simply overrides the error.
2. When dealing with astype(64) and tu\_meric, astype only works if all values in column are numbers while to\_numeric can work if there are other values in data. If there are non-numeric values, it will create NaN

One column

pd.to\_numeric(df['mix\_col'], errors='coerce')

df['col\_str'] = df['col\_str'].astype('int')

Entire Data

df = df.astype(int) #or (str, ,datetime, float)

1. **Handling Unique values (values that should not exist, i.e., ‘$’ in a numeric column)**
2. Finding Unique Values

for col in df.columns:

    print('{} : {}'.format(col,df[col].unique()))

1. Supposing there are unique values like ‘$’ or ‘,’

df ['col\_name'] = df['col\_name'].str.replace('$','').str.replace(',', '')

df['col\_name'] = pd.to\_numeric(df['col\_name'])

OR For entire Dataset

for col in df.columns:

    df[col].replace({'?':np.nan},inplace=True)

MISSING DATA

1. **Find Missing Values – Less than 6% is generally accepted as standard for missing data**
2. Find total columns with missing values

mis\_num = df.isna().any().sum()

1. Find which columns has the missing values

mis\_num = df.isna().any()

1. Find how many values are missing per column

mis\_num = df.isna().sum()

1. Find which rows has the missing values

mis\_row = df.isna().any()

1. Find Missing Values by Percentage

mis\_col = df.isna().sum() \* 100 / len(df)

1. **Check if there are any zeroes in the number – can then determine if the ‘0’ denotes empty value or a meaningful value**

check\_zero = df.isin([0]).sum()

1. **Find Duplicates**

check\_dups = df.duplicated().sum()

**WHAT TO DO WITH DUPLICATES, ZEROS, MISSING DATA**

1. **Duplicates**
2. Keep the duplicates
3. Drop all duplicates

df.drop\_duplicates()

1. Drop all but first or last duplicates

df.drop\_duplicates(subset=None, keep='first', inplace=False, ignore\_index=False) #==> Subset is column name || keep = 'first', 'last', 'False' (False drops all duplicates)|| inplace == bool to keep changes or return new copy || ignore\_index == bool to return new index

1. Targeting specific subset

df.drop\_duplicates(subset=['colname', 'colname2'], keep='first', inplace=False, ignore\_index=False)

1. h
2. **Zeros**
3. Drop empty values if there is no meaning

df.dropna()

1. Drop specific columns(examples columns with more than 6% missing numbers)

df.drop(columns='colum name') or columns =['col1', 'col2']

1. Drop if NA is found in a column (using either any or all)

df = df.dropna(axis= 0, how='any', thresh=2, subset=None, inplace=False) # ==> axis 0/index --rows, 1--columns || how=>'any' or 'all', || thresh=> if 2 or more non-NA values are found, keep axis || inplace=> bool where True happens to current DataFrame while False return a copy

1. Replace the zeroes with NaN

for column in df:

    df[column].where(df[column] !=0, inplace=True)

1. Run the code to drop NaN as now the zeroes above are now missing values

df.dropna()

MISSING DATA

**When dealing with missing data, you can use a distribution matrix to decide how to deal with the data**

1. **Missing Data**
2. Run a distribution plot or a heatmap to show missing data(Prefer to use mean if normal distribution or Median if there is skewness)

**GeocodingArtifactAddressPrimaryYCoordinate 0.292595**

sns.histplot(aft\_dropdf['GeocodingArtifactAddressPrimaryYCoordinate'], color='blue')

**Chart, histogram

Description automatically generated**

**GeocodingArtifactAddressPrimaryXCoordinate 0.292595**

sns.histplot(aft\_dropdf['GeocodingArtifactAddressPrimaryXCoordinate'], color='blue')

**Chart, bar chart, histogram

Description automatically generated**

**DUNSNumber 0.027009**

sns.histplot(aft\_dropdf['DUNSNumber'], color='blue')

**Chart, bar chart, histogram

Description automatically generated**

**U.S.-MexicoBorderCountyIndicator 0.292595**

sns.histplot(aft\_dropdf['U.S.-MexicoBorderCountyIndicator'], color='blue')

**Chart, bar chart

Description automatically generated**

**GrantProgramDirectorE-mail 0.018006**

sns.histplot(aft\_dropdf['GrantProgramDirectorE-mail'], color='blue')

**Chart, line chart

Description automatically generated**

**GrantProgramDirectorPhoneNumber 0.085528**

sns.histplot(aft\_dropdf['GrantProgramDirectorPhoneNumber'], color='blue')

**GranteeStateAbbreviation 0.288094**

sns.histplot(aft\_dropdf['GranteeStateAbbreviation'], color='blue')

**Chart, histogram

Description automatically generated**

**GranteeZIPCode 0.009003**

sns.histplot(aft\_dropdf['GranteeZIPCode'], color='blue')

Based on the distribution above, I was unable to run graphs on GranteeZipCode and GrantProgramDirectorPhoneNumber because they are categorical data---for now that’s not a big deal.

1. Drop missing column

I am going to ignore the missing values on because they are all addresses, However my analysis will only compare the Grantee City. I can’t backfill or frontfill the DUNSNumber due to its uniqueness. This is for Python Analyses, hower when dealing with SQL, All missing data will come in play as you will see

1. Fill data with a specific Value

df['column name'].fillna(12, inplace = True)

1. Fill a specific index

df.loc[2,'column name'] = 25

1. Filling the data with Mean, Mode, Median

col\_mean = round(df['GeocodingArtifactAddressPrimaryYCoordinate'].mean(),2)

df['GeocodingArtifactAddressPrimaryYCoordinate'].fillna(col\_mean, inplace=True)

col\_med = round(df['GeocodingArtifactAddressPrimaryXCoordinate'].median(),2)

df['GeocodingArtifactAddressPrimaryXCoordinate'].fillna(col\_med, inplace=True)

col\_mod = df['column name'].mode()

df['column name'].fillna(col\_mod, inplace=True)

1. Filling with next Value Specific Column

df = df['column name'].ffill(inplace=True)

1. Filling with Previous Values Specific Column

df = df['column name'].bfill(inplace=True)

1. Filling Front or Back entire dataset

df.fillna(method='ffill', inplace=True)

df.fillna(method='backfill', inplace=True)

**USING Machine Learning to Impute the data (**

1. Import the SimpleImputer
2. Impute Using Mean

imputer = SimpleImputer(missing\_values=np.NaN, strategy='mean')

df[['GeocodingArtifactAddressPrimaryYCoordinate']] = imputer.fit\_transform(df[['GeocodingArtifactAddressPrimaryYCoordinate']])

1. Impute Using Median

imputer\_med = SimpleImputer(missing\_values=np.NaN, strategy='median')

df[['GeocodingArtifactAddressPrimaryXCoordinate']] = imputer\_med.fit\_transform(df[['GeocodingArtifactAddressPrimaryXCoordinate']])

1. Import using Mode

For Categorical Column

imputer\_mode = SimpleImputer(missing\_values=np.NaN, strategy='most\_frequent')

df[['U.S.-MexicoBorderCountyIndicator']] = imputer\_mode.fit\_transform(df[['U.S.-MexicoBorderCountyIndicator']])

For Numerical Column

imputer\_mode = SimpleImputer(missing\_values=np.NaN, strategy='most\_frequent')

df[['DUNSNumber']] = imputer\_mode.fit\_transform(df[['DUNSNumber']])

1. Imputing using a constant value

imputer\_const = SimpleImputer(missing\_values=np.NaN, strategy='constant', fill\_value=80)

df[['column\_name']] = imputer\_const.fit\_transform(df[['column\_name']])

SAVING THE CLEANED DATA

1. **Save Data in File type it came in**
2. Numerical Data

nume\_data = df.select\_dtypes(include= ['float64', 'int64'])

1. Categorical Data

cate\_data = df.select\_dtypes(exclude= ['float64', 'int64'])

1. Save entire data

df.to\_csv('cleaned\_hrsa.csv')

df.to\_excel('cleaned\_hrsa\_data.xlsx')

1. **Saving the Data as Excel**
2. Numerical Data

nume\_data.to\_excel('cleanedexcel\_hrsa\_numerical.xlsx')

1. Categorical Data

cate\_data.to\_excel('cleanedexcel\_hrsa.xlsx')

1. **Saving the Data as CSV**
2. Numerical Data

nume\_data.to\_csv('cleaned\_hrsa\_numerical.csv')

1. Categorical Data

cate\_data.to\_csv('cleaned\_hrsa\_categorical.csv')

1. **G**
2. **G**
3. **G**
4. **G**
5. **G**
6. **G**
7. **G**
8. **G**