Cleaning Data in Python
by datacamp

reminder of workflow
access data> explore and process data> visualize and extract insights> report insights

.sum() on a string column will return a concatenation of all the strings
pandas see strings as dtype object

example - we have sales['Revenue'] which outputs numbers followed by $ (ie 232324$); we want to get the sum of the column
to remove $ from Revenue column
sales['Revenue'] = sales['Revenue'].str.strip('$')
we then want to change the column dtype to integer
sales['Revenue'] = sales['Revenue'].astype('int')
**if column had decimals we would change it to dtype 'float'
verify that sales['Revenue'] is an integer
assert sales ['Revenue'].dytpe == 'int'

How to deal with out of range data
-dropping data
-setting custom minimums and maximums
-treat as missing and impute
-setting custom value depending on business assumptions

example - often things will be rated on scales
this example scale is 1-5
we can ensure that no inputs are outside of this range
movies[movies['avg_rating'] > 5]
several movies come up with a rating of 6, we know that this isn't possible
we can drop these values using filtering
movies = moves[movies['avg_rating'] <=5]
or we can drop these values using .drop()
movies.drop(movies[movies['avg_rating'] > 5].index, inplace=True)
'inplace' argument to True, values are dropped in place and we don't have to create a new column
use 'assert' to ensure change
assert movies['avg_rating'].max() <= 5
can also convert avg_rating > 5 to 5
movies.loc[movie['avg_rating'] > 5, 'avg_rating'] = 5

again, can make sure that this change was done using an ‘assert’ statement
assert movies[‘avg_rating’].max() <= 5
**remember no output means that it passed

Date range example
eval to see if columns are datetime
a way to convert to date
user_signups[‘subscription_date’] =
pd.to_datetime(user_signups[‘subscription_date’]).dt.date

today_date = dt.date.today()
drop values using filtering
user_signups = user_signups[user_signups [‘subscription_date’] < today_date]
drop values using .drop()
user_signups.drop(user_signups[user_signups[‘subscription_date’] >
today_date].index, inplace = True)

Or hardcode dates with upper limit
user_signups.loc[user_signups[‘subscription_date’] > today_date,
‘subscription_date’] = today_date
assert is true
assert user_signups.subscription_date.max().date() <= today_date

What are duplicate values?
-all columns have the same values
why do they happen > data entry/human error, bugs and design errors, or most
commonly d/t join or merge errors
how to find them
get duplicates across all columns
duplicates = df.duplicated()
print(duplicates)
output > boolean for each entry
get duplicate rows
duplicates = df.duplicated()
df[duplicates]
to help properly calibrate, we can use the ‘subset’ and ‘keep’ argument
subset gives a list of column names to check for duplication
keep states whether to keep first (‘first’), last (‘last’), or all (False) duplicate values
example - checking column names for duplication
column_names = [‘first_name’, ‘last_name’, ‘address’]
duplicates = height_weight.duplicated(subset = column_names, keep = False)
then sort the duplicate values
height_weight[duplicates].sort_values(by = ‘first_name’)

**there can be complete duplicates where every field is the same or incomplete where one or more of the fields is off
next is to drop duplicates with .drop_duplicates; has same arguments of 'subset' and 'keep'
in addition has argument 'inplace', which drops duplicated rows directly inside DataFrame without creating new object (True)
height_weight.drop_duplicates(inplace=True)

How to treat duplicate values?
with the combo .groupby() and .agg() methods
example
column_names = ['first_name', 'last_name', 'address']
summaries = {'height': 'max', 'weight': 'mean'}
height_weight = height_weight.groupby(by = column_names).agg(summaries).reset_index()

Text and categorical data problems
categorical data represent variables that represent predefined finite set of categories
needs to be coded into numbers in order to run machine learning models on them
causes of errors with categorical data > data entry issues, data parsing errors

How do we treat these problems
can drop the rows with incorrect categories
can attempt to remap incorrect categories to correct ones
**good practice is to keep a log of all possible values of your categorical data

Refresher on joins
for categorical data we focus on anti and inner joins
DFs are joined on common columns between them
anti-joins take in two DFs and return on DF that is not contained in another
example - left anti-join of DF A and B and returning columns of A and B for values only found in A of the common column between them being joined on
inner joins return only the data that is contained in both DFs
in this example inner join of A and B would return columns from both DFs for values only found in A and B of the common column between them being joined on
example
inconsistent_categories = set(study_data['blood_type']).difference(categories['blood_type'])
print(inconsistent_categories)
'set' method stores the designated unique values
'difference' method takes in the designated column and returns all categories that are not in categories

get all the rows with inconsistent categories
inconsistent_rows = study_data['blood_type'].isin(inconsistent_categories)
this finds all rows within the blood_type column that are equal to inconsistent categories
this returns a series of boolean values that are True for inconsistent rows and False for consistent ones
subset the DF based on these boolean values
study_data[inconsistent_rows]
we can drop inconsistent categories and get consistent data only by using the ~ (tilde/approx) symbol within our subset
consistent_data = study_data[~inconsistent_rows]

Different common problems with data
-data type constraints
-data range constraints
-uniqueness constraints
-membership constraints (for categorical values)

Types of errors with categorical variables
1. value inconsistency (ie spelling mistakes or trailing whitespaces)
2. collapsing too many categories to few (example mapping multiple income ranges to either 'rich' or 'poor')
3. making sure data is of type category

Checking value consistency
df.value_counts() to get frequencies
**reminder value_counts() only works on Series
to get value counts on DF
df.groupby('col').count()

Dealing with capitalization
df['col'] = df['col'].str.upper() or str.lower()

Dealing with white spaces
df['col'] = df['col'].str.strip()
strips all pre and post white space

Collapsing data into categories
example - creating categories out of data income_group column and income column
import pandas as pd
group_names = ['0-200K', '200K-500K', '500K+']
demographics['income_group'] = pd. qcut(demographics['household_income'],

q=3, labels=group_names)
to print
demographics[['income_group', 'household_income']]
'qcut' method automatically divides our data based on its distribution into the number of categories we set in the q argument

***Better way - use pd.cut and create category ranges and names
ranges = [0, 200000, 500000, np.inf]
group_names = ['0-200K', '200-500K', '500K+']
create income group column
demographics['income_group'] = pd.cut(demographics['household_income'], bins=ranges, labels=group_names)
print
demographics[['income_group', 'household_income']]

How to map categories to fewer ones (ie reducing categories in categorical column)
example
operating_system column has values 'Microsoft', 'MacOS', 'IOS', 'Android', 'Linux'
we want to change it to 'DesktopOS', 'MobileOS'
create mapping dictionary and replace
mapping = {'Microsoft':'DesktopOS', 'MacOS':'DesktopOS', 'Linux':'DesktopOS', 'IOS':'MobileOS', 'Android':'MobileOS'}
devices['operating_system'] = devices['operating_system'].replace(mapping)
devices['operating_system'].unique()

Cleaning text data
types of text data > phone numbers, emails, addresses, and more
common text data problems include:
handling inconsistencies
making sure text data is of a certain length
typos
others

example - making phone numbers uniform
initial review shows length inconsistencies, dashes, and other symbols
want to make it all numbers so easier to feed into automated models
phones['Phone number'] = phones['Phone number'].str.replace('+', '00')
we replaced any '+' that started area codes with '00'
now replace dashes
phones['Phone number'] = phones['Phone number'].str.replace('-', '')
now we are going to replace all phone numbers with lower than 10 digits to NaN
digits = phones['Phone number'].str.len()

phones.loc[digits < 10, ‘Phone number’] = np.nan
use ‘assert’ to ensure changes are a success
find the length of each row in Phone number column
sanity_check = phone[‘Phone number’].str.len()
assert minimum phone number length is 10
assert sanity_check.min() >= 10
assert all numbers do not have ‘+’ or ‘-‘
assert phone[‘Phone number’].str.contains(‘=|-‘).any() == False
pipe here works as an ‘or’ statement
‘any’ method returns True if any values within str.contains + or -

What if it is more complicated?
can use regular expressions
regular expressions give us the ability to search for any pattern in text data
works like control + f(find) in your browser
example - only extract digits from the phone number column
replace letters with nothing
phones[‘Phone number’] = phones [‘Phone number’].str.replace(r’\D+’, ‘’)

Uniformity
verifying unit uniformity is imperative to having accurate analysis
example - dataset where some temperatures are accidentally in Fahrenheit instead of Celsius
temp_fah = temperatures.loc[temperatures[‘Temperature’] > 40, ‘Temperature’]
change to celsius
temp_cels = (temp_fah - 32) * (5/9)
temperatures.loc[temperatures[‘Temperature’] > 40, ‘Temperature’] = temp_cels
assert conversion is correct
assert temperatures[’Temperature’].max() < 40

Treating date data
example
birthdays[‘Birthday’] = pd.to_datetime(birthdays[‘Birthday’], infer_datetime_format=True, errors=‘coerce’)
‘errors’ argument returns NA for rows where conversion failed
**in pandas NAT is datetime equivalent to NaN for missing values

We can also convert date time into a style that we prefer
birthdays[‘Birthday’] = birthdays[‘Birthday’].dt.strftime(‘%d-%m-%Y’)

Complex example that stumped me
# Find values of acct_cur that are equal to 'euro'
acct_eu = banking['acct_cur'] == 'euro'

Convert acct_amount where it is in euro to dollars
banking.loc[acct_eu, 'acct_amount'] = banking.loc[acct_eu, 'acct_amount'] * 1.1

Unify acct_cur column by changing 'euro' values to 'dollar'
banking.loc[acct_eu, 'acct_cur'] = 'dollar'

Assert that only dollar currency remains
assert banking['acct_cur'].unique() == 'dollar'

Cross field validation to diagnose dirty data
is the use of multiple fields in your dataset to sanity check the integrity of your data
example - summing economy, business, and first class values to ensure equal to total passengers on plane
sum_classes = flight[['economy_class', 'business_class', 'first_class']].sum(axis = 1)
find instances where the total passengers column is equal to the sum of the classes
passenger_equ = sum_classes == flights['total_passengers']
find and filter out rows with inconsistent passenger totals
inconsistent_pass = flights[~passenger_equ]
consistent_pass = flights[passenger_equ]

Another example - making sure that the age and birthday columns are correct
import pandas as pd
import datetime as dt
#convert to datetime and get today's date
users['Birthday'] = pd.to_datetime(users['Birthday'])
today = dt.date.today()
#for each row in the Birthday column, calculate year difference
age_manual = today.year - users['Birthday'].dt.year
#find instances where ages match
age_equ = age_manual == users['Age']
#find and filter out rows with inconsistent age
inconsitent_age = users[~age_equ]
consistent_age = users[age_equ]

What to do with inconsistencies found in cross field validation
drop data
set to missing and impute
apply rules from domain knowledgedat

Example

# Store today's date and find ages
today = dt.date.today()
ages_manual = today.year - banking['birth_date'].dt.year

# Find rows where age column == ages_manual
age_equ = ages_manual == banking['age']

# Store consistent and inconsistent data
consistent_ages = banking[age_equ]
inconsistent_ages = banking[~age_equ]

# Store consistent and inconsistent data
print("Number of inconsistent ages: ", inconsistent_ages.shape[0])

Completeness
Finding missing values
df.isna() returns boolean True for missing values
chain with .sum() to get the summary of missing values

Example - using the Missingno package
package for visualizing and understanding missing data
import missing as msno
import matplotlib.pyplot as pat
#visualize missing ness
msno.matrix(airquality)
plt.show()
#isolate missing and complete values aside
missing = airquality[airquality['CO2'].isna()]
complete = airquality[~airquality['CO2'].isna()]
#use .describe() on each variable
complete.describe()
missing.describe()
#**in this example we note that the missing values happen at really low temperatures
#we can confirm this visually
sorted_airquality = airquality.sort_values(by = 'Temperature')
msno.matrix(sorted_airquality)
plt.show()
#values are sorted from smallest to largest by default
#the visualization confirms our CO2 measurements are lost for really low temperatures
#leads us to question, ?is this a sensor failure

Missingness types
MCAR - missing completely at random
-no systematic relationship between missing data and other values
-data entry errors when inputting data
MAR - missing at random
-confusing name
-systematic relationship between missing data and other observed values
-example missing ozone data for high temperatures
MNAR - missing not at random
-systematic relationship between missing data unobserved values
-example missing temperature values for high temperatures

How to deal with missing data
drop
impute with statistical measures or ML models

df.dropna(subset =[‘col’])
or
x = df[‘col’].mean()
df.fillna({‘col’:’x’})

Comparing strings
Minimum edit distance
a systematic way to identify how close two strings are
example
intention
execution
What are the least possible amount of steps needed to transition from one string to another
using insertion, deletion, substitution, transposition
delete i, substitute e,x, and e, and add c
minimum edit distance is 5
**many different algorithms and packages available to do this
this example will use the Levenshtein distance and the fuzz package

from thefuzz import fuzz
fuzz.WRatio(‘Reeding’, ‘Reading’)
this computes the similarity with our output being 86
0 is not similar at all, 100 is an exact match
**similarity can still be strong with partial string comparisons
example
fuzz.WRatio(‘Houston Rockets’, ‘Rockets’)
output 90

Can also compare a string with an array of strings by using the extract function from the process module
from thefuzz import process
define the string and array of possible matches
string = ‘Houston Rockets vs Los Angeles Lakers’
choices = pd.Series ([‘Rockets vs Lakers’, ‘Lakers vs Rockets’, ‘Houston vs Los Angeles’, ‘Heat vs Bulls’])
process.extract(string, choices, limit = 2)
**output is a list of tuples with 3 elements (matching string being returned, similarity score, its index in the array)

What happens when there are many inconsistencies and manual replacement is simply not feasible?
example
we have a survey with free text responses from all 50 states, with many labels for each state
we can use string similarity
define unique values
print(survey[‘state’].unique())
create a category DF that contains the correct categories for each state
#for each correct category
for state in categories[‘state’]:
#find potential matches in states with types
#set ‘limit’ argument to the length of the DF with DF.shape[0]
    matches = process.extract(state, survey[‘state’], limit=survey.shape[0])
    for potential_match in matches:
    if potential_match[1] >= 80:
#replace typo with correct category
    survey.loc[survey[‘state’] == potential_match[0], ‘state’] = state

Complex example
# Iterate through categories
for cuisine in categories:
  # Create a list of matches, comparing cuisine with the cuisine_type column
  matches = process.extract(cuisine, restaurants['cuisine_type'], limit=len(restaurants.cuisine_type))

  # Iterate through the list of matches
  for match in matches:
     # Check whether the similarity score is greater than or equal to 80
    if match[1] >= 80:
      # If it is, select all rows where the cuisine_type is spelled this way, and set

them to the correct cuisine
    restaurants.loc[restaurants['cuisine_type'] == match[0], 'cuisine_type'] = cuisine

# Inspect the final result
print(restaurants['cuisine_type'].unique())

Record linkage
is the act of linking data from different sources regarding the same entity
workflow
clean df A and df B > generate pairs > compare pairs > score pairs > link data
can do all of this with the record linkage package

**ideally we want to generate all possible pairs between df A and df B
however this can lead to billions of pairs
we can apply ‘blocking’ to avoid this
blocking is creating pairs on a matching column

example - combining two census
import recordlinkage
#create indexing object, essentially an object we can use to generate pairs from our DataFrames
indexer = recordlinkage.Index()
#generate pairs on common column, in this case ‘state’
indexer.block(‘state’)
pairs = indexer.index(census_A, census_B)
this takes in the two DataFrames
print(pairs)
output is a pandas multi index object ie an array containing possible pairs of indices
*this will make it much easier to subset DataFrames on

Comparing the DataFrames
generate the pairs
pairs = indexer.index(census_A, census_B)
create a compare object > similar to above generation of the pairs, but this one is responsible for assigning different comparison procedures for pairs
compare_cl = recordlinkage.Compare()
we want to eval columns for which we want exact matches between the pairs
in this example find exact matches for pairs of date_of_birth and state
compare_cl.exact(‘date_of_birth’, ‘date_of_birth’, label=‘date_of_birth’)
compare_cl.exact(‘state’, ‘state’, label=‘state’)
‘label’ argument allows us set the column name in the resulting DataFrame

next in this example we want to find similar matches for pairs of surname and address_1 using string similarity
compare_cl.string(‘surname’, ‘surname’, threshold=0.85, label=‘surname’)
compare_cl.string(‘address_1’, ‘address_1’, threshold=0.85, label=‘address_1’)
‘threshold’ argument sets the similarity cutoff point, again between 0 and 1
compute the matches
potential_matches = compare_cl.compute(pairs, census_A, census_B)
**always keep the came order of DataFrames when entering them in through this process as arguments
output is a multi index DataFrame
first index is the row index from census_A
second index is a list of all row indices in census_B
columns are the columns being compared with values being 1 for a match and 0 for not a match

another example
# Create a comparison object
comp_cl = recordlinkage.Compare()

# Find exact matches on city, cuisine_types -
comp_cl.exact('city', 'city', label='city')
comp_cl.exact('cuisine_type', 'cuisine_type', label='cuisine_type')

# Find similar matches of rest_name
comp_cl.string('rest_name', 'rest_name', label='name', threshold = 0.8)

# Get potential matches and print
potential_matches = comp_cl.compute(pairs, restaurants, restaurants_new)
print(potential_matches)

Probable matches
matches = potential_matches[potential_matches.sum(axis =1) >= 3]
print(matches)
output is for census example is row indices between census A and census B that are most likely duplicates
next step is to extract one of the index columns and subset its associated DataFrame to filter for duplicates
in this example we choose the second index column, which represents row indices of census B
we want to extract those indices, and subset census B on them to remove duplicates with census A before appending them together
we access a DF’s index using the index attribute
matches.index

in this example it returns a multi index object containing pairs of row indices from census A and census B
we want to extract all census B indices, so we chain it with the get_level-values method
this takes in which column index we want to extract its values
we can either input the index column’s name or its order
in this example it is 1
duplicate_rows = matches.index.get_level_values(1)
print(census_B_index)
finding the duplicates in census_B
simply subset on all indices of census_B with the ones found through record linkage
census_B_duplicates = census_B[census_B.index.isin(duplicate_rows)]
find non duplicates
census_B_new = census_B[~census_B.index.isin(duplicate_rows)]
link the DataFrames with the append method
full_census = census_A.append(census_B_new)

another example
```
# Isolate potential matches with row sum >=3
matches = potential_matches[potential_matches.sum(axis=1) >= 3]

# Get values of second column index of matches
matching_indices = matches.index.get_level_values(1)

# Subset restaurants_new based on non-duplicate values
non_dup = restaurants_new[~restaurants_new.index.isin(matching_indices)]

# Append non_dup to restaurants
full_restaurants = restaurants.append(non_dup)
print(full_restaurants)
```