Working with Categorical Data in Python
by datacamp

What does it mean to be ‘categorical’
- finite number of groups (or categories)
- categories are usually fixed or known (ie eye color)
- known as qualitative data

Numerical data
- know as quantitative data
- expressed using a numerical value
- usually a measurement (ie height or weight)

Further breakdown of categorical data
two types ordinal or nominal
Ordinal
-categorical variables that have a natural order (ie a survey)
-a natural rank order
Nominal
-cannot be placed into a natural order (ie choosing your favorite color)

Can use the describe or value_counts method on a pandas Series
example
adult[‘Marital Status’].describe()
adult[‘Marital Status’].value_counts(normalize=True)
*remembe normalize argument gives output as proportions (or relative frequencies) instead of as raw count

Running .dtype on a pandas Series
if dtype is object
output will be dtyple(‘O’)
example
adult[‘Marital Status’].dtype
can turn dtype objects or strings into a more specific dtype of ‘category’
adult[‘Marital Status’} = adult[‘Marital Status’].astype(‘category’)
now if .dtype is ran
output is a list of all the categories found within ‘Marital Status’

How to create a categorical Series
my_data = [‘A’, ‘A’, ‘C’, ‘B’, ‘C’, ‘A’]
my_series1 = pd.Series(my_data, dtype=‘category’)

print(my_series1)
another way which tells pandas that the categories have a logical order by setting the argument ‘ordered=True’
my_series2 = pd.Categorical(my_data, categories=[‘C’, ‘B’, ‘A’], ordered=True)
if this is set to true than how the categories are ordered will be how they are ordered

Why do we use dtype categorical
- huge memory saver
    - .nbytes tells us how many bytes are being used for ‘Marital Status’ as object 260k vs as categorical 32k

Since pandas loads the entire dataframe into your working memory, it can be helpful to know the .dtypes of some of the columns before reading the data in this way you can create a dictionary and dictate some of those types
example
adult_dytpes = {‘Marital Status’: ‘category’}
adult = pd.read_csv(‘data/adult.csv’, dtype=adult_dtypes)

The .groupby() is essential for exploring categorical variables in pandas
splits data across the unique values of the column specified
allows us to split data across groups and to perform analysis on each group
**essentially equal to creating multiple DataFrames, one for each value in the specified variable
example
adult1 = adult[adult[‘Above/Below 50k’] == ‘ <=50k’]
adult2 = adult[adult[‘Above/Below 50k’] == ‘ >=50k’]
these two filtered variables created to represent two separate DataFrames can be replaced by or essentially this exact action can be completed by using .groupby
groupby_object = adult.groupby(by=[‘Above/Below 50k’])
the ‘by’ argument is used to specify the variable or variables to split the data by
**for future, does not need to be a list

Can apply functions to .groupby()
example
groupby_object.mean()

For datasets with multiple columns very important to specify columns of interest
adult.groupby(by=[‘Above/Below 50k’])[‘Age’, ‘Education Num’].sum()
or (same result)
adult.groupby(by=[‘Above/Below 50k’]).sum()[[‘Age, ‘Education Num’]]
option 1 is better - *runs faster

Can groupby() multiple columns
adult.grouby(by=[‘Above/Below 50k’, ‘Marital Status’]).size()
**when groupbying multiple columns, only combinations that exist are created

Setting category variables
example
dogs[‘coat’] = dogs[‘coat’].astype(‘category’)
dogs[‘coat’].value_counts(dropna=False)

**Very important feature the .cat accessor object
object let’s us access and manipulate the categories of a categorical Series
common parameters:
 - new_categories: a list of categories
 - inplace: boolean value - whether or not the update should overwrite the Series
 - ordered: boolean value - whether or not the categorical is treated as an ordered categorical

Setting Series categories cat.set_categories()
-can be used to set the order of categories
-all values not specified in this method are dropped
dogs[‘coat’] = dogs[‘coat’].cat.set_categories(new_categories=[‘short’, ‘medium’, ‘long’])
**any values not listed in the new-categories list will be dropped
**if to re-eval dogs[‘coat’] with value_counts we would see that the old category ‘wirehaired’ has been set to NaN
this is because the wirehaired category is not listed in the new-categories parameter and is no longer recognized
‘ordered’ parameter set to True will order the categories in the order in which they have been set in the last

Can add categories to help clarify what a missing value actually means
cat.add_categories()
-does not change the value of any data in the DataFrame
-categories not listed in this method are left alone
in this example the dogs[‘likes_people’] column has 938 rows without a response
?did the dog shelter not check or maybe they couldn’t tell
Add categories to help determine this
dogs[‘likes_people’] = dogs[‘likes_people’].astype(‘category’) #changing this to a categorical variable
dogs[‘likes_people’] =
dogs[‘likes_people’].cat.add_categories(new_categories=[‘did not check’, ‘could not tell’])
#now lets check this column

dogs[‘likes_people’].cat.categories
output > Index is now no, yes, did not check, could not tell
**key to remember this didn’t update any of the values there are still 938 rows without a response

Removing categories
dogs[‘coat’] = dogs[‘coat’].astype(‘category’)
dogs[‘coat’].cat.remove_categories(removals=[‘wire-haired’])
removes the ‘wire-haired’ category all together
*means that all wire-haired values will be set to NaN values now

Changing values
example - we want to know if adopted dog is safe around children, to be safe we want to change response ‘maybe’ to ‘no’
dogs.loc[dogs[‘likes_children’] == ‘maybe’, ‘likes_children’] = ‘no’

Renaming categories
-key point must use new category names, cannot rename a category to the name of another category
-also cannot use this method to collapse two or more categories into one (ie take three separate categories and label them all the same thing)
in this example we want to rename ‘Unknown Mix’ to just ‘Unknown’
use the rename_categories method
Series.cat.rename_categories(new_categories=dict)
**just as heads up this method does not require a dictionary but it can be a friendly tool
#make a dictionary
my_changes = {‘Unknown Mix’: ‘Unknown’}
then
dogs[‘breed’] = dogs[‘breed’].cat.rename_categories(my_changes)
**can also use lambda functions
example
dogs[‘sex’] = dogs[‘sex’].cat.rename_categories(lambda c: c.title())
*what we did here was take the categories in ‘sex’ and capitalize them

Collapsing categories
here we will use the example of collapsing dogs colors
particularly we want to all two colors with one of those colors being black and collapse them into a category called just ‘black’
dictionary is helpful here
update_colors = {‘black and brown’: ‘black’, ‘black and tan’: ‘black’, ‘black and white’: ‘black’}
**key this method does not preserve the categorical data type or the .accessor

object
dogs[‘main_color’] = dogs[‘color’].replace(update_colors)
if we were to check dogs[‘main_color’].dtype we would see that it is not a ‘category’ but instead an ‘object’
so we need to turn the new column into a categorical column
dogs[‘main_color’] = dogs[‘main_color’].astype(‘category’)

Reordering categories
Why would you reorder?

1. creating a ordinal variable
2. to set the order that variable are displayed in analysis
3. memory savings

example
dogs[‘coat’] = dogs[‘coat’].cat.reorder_categories(new_categories = [‘short’, ‘medium’, ‘wire-haired’, ‘long’], ordered=True)
**can also add parameter ‘inplace’ to True; this updates variable without needing to set the variable equal to an updated version of itself
several functions and methods in pandas have those as a parameter and it is generally used as a way to reduce the amount of typed code
**inplace is a memory saver
**by setting ordered to False you are stating that the categories do not have to be ordinal

Cleaning and accessing data
possible issues with categorical data

1. inconsistent values - example ‘Ham’ vs ‘ham’ vs ‘ Ham’
2. misspelled values - example ‘Ham’, ‘Ham’
3. wrong dtype - df[‘Our Column’].dtype #need to make sure we’ve kept the ‘category’ type and isn’t the ‘object’ type

Identifying issues
Series.cat.categories
Series.value_counts()

Fixing issues: whitespace
removing whitespace: .strip()
dogs[‘get_along_cats’]= dogs[‘get_along_cats’].str.strip() #this removes leading and trailing whitespace

Fixing issues: capitalization
can use .title(), .upper(), or .lower() to fit our needs
dogs[‘get_along_cats’] = dogs[‘get_along_cats’].str.title()

Fixing issues: misspelled words
fixing a typo with .replace()
replace_map = {'Noo':'No'}
dogs{'get_along_cats'].replace(replace_map, inplace=True)

***remember that all of this is likely to change dtype to object
lastly convert back to a category with .astype('category')

Using the str accessor object
searching for a string
dogs['breed'].str.contains('Shepherd', regex=False)
'regex' parameter to false ensures that we use string matching and not a regular expression

Can use loc and iloc with columns that are categories
example - pulling dogs that get along with cats by their size
dogs.loc[dogs['get_along_cats'] == 'Yes', 'size'].value_counts(sort=False)
value_counts does not automatically use the categorical order when printing results
'sort' to False so that the output will be ordered by the order of the category

Categorical plots using Seaborn
'kind' argument - 'strip', 'swarm', 'box', 'violin', 'boxes', 'point', 'bar' or 'count'
example - look at the review score across the categorical variable Pool using a boxplot
sns.catplot(x='Pool', y='Score', data=reviews, kind='box')
*grid can help to view outliers sometimes
sns.set_style('whitegrid'"
to change font size
sns.set(font_scale=1.4)

Seaborn bar charts goal is to summarize a numerical variable across the different levels of a categorical variable
the height of the bar is a point estimate for the mean of the data
black lines represent confidence interval
CIs in seaborn represent 95% confidence that the true mean of the data will fall within
hue variable allows us to split the data a second time

Point plot
may help users focus on the different values across the categories by adding a connecting line across the points
the y-axis is changed to better focus on the points

‘dodge’ argument to True offsets the lines to make it easier for users to see the CIs and mean
‘join’ argument default is True and draws a line to compare between points, set to False to remove

The catplot(kind=‘count’)
uses the count plot to display frequencies
simply counts the number of occurrences of the categorical variables specified in the x, y, or hue parameters
catplot count plot is just a typical bar graph
**while all the other kinds aggregate a numerical variable across a categorical variable (ie mean estimate)

Additional catplot() options for better visualization
in this example we wanted to push users to see the count within continents vs across continents
we can create a facetgrid
sns.catplot(x=‘Traveler type’, kind=‘count’, col=‘User continent’, col_wrap=3, palette=sns.color_palette(’Set1’), data=reviews)
‘col’ argument tells seaborn to create a catplot of each category in this case ‘User continent’
‘col_wrap’ argument tells seaborn to lay x amount of graphs before moving down a vertical position

Updating plots
setup: save your graphic as an object called ‘ax’
plot title: ax.fig.suptitle(‘My title’)
axis labels: ax.set_axis_labels(‘x-axis-label’, ‘y-axis-label’
title height: plt.subplots_adjust(top=.9) #the top of the actual plot to be at 90% of the full graphic

Categorical pitfalls
changing an object column to category column can have huge memory savings up to 90%
changing an int column to category will have less up to 60%
more specifically the number of bytes needed for a categorical column is proportional to the number of categories

Using categories can by frustrating
using the .str accessor object to manipulate data converts the Series to an object
using the .apply() method outputs a new Series as an object
*common methods of adding, removing, replacing, or setting categories do not all handle missing categories the same way

**NumPy functions generally do not work with categorical Series

.value_counts(dropna=False) is always a good way to see if changes worked as you intended
can change dtype when category is an integer in order to use a NumPy array
example
used_cars['number_of_photos'].astype(int).sum()

Label encoding
basics:

1. codes each category as an integer from 0 through n-1, where n is the number of categories
2. A -1 code is reserved for any missing values
3. can save on memory
4. often used in surveys

**this is not the best encoding method for machine learning

Creating codes
can get a lobel encoding by using .cat.codes
example - say we want to convert the values of 'manufacturer_name' from categorical to integers
used_cars['manufacturer_code'] = used_cars['manufacturer_name'].cat.codes

**this is often used in surveys
the codes are often kept in a code book or a data dictionary
creating a code book
create a label encoding and save the new dataset, you will want to create a map from the new codes to the old values
example
start by creating an object from the codes and an object for the categories
codes = used_cars['manufacturer_name'].cat.codes
categories = used_cars['manufacturer_name']
then use the zip function to iterate through the entries of codes and categories one at a time
name_map = dict(zip(codes, categories))
putting the zip function with the dictionary call, the unique combinations of codes and categories will be added as key-value pairs
then print(name_map) in order to reveal which code maps to which category
to revert to previous values
used_cars['manufacturer_code'].map(name_map
.map is similar to .replace
.map replaces the Series values based on the keys of the name_map and their corresponding values

we can use .map in this context because we have a complete mapping (ie every single value has a key)

Boolean coding
example find all body types that have ‘van’ in them
used_cars[‘body_type’].str.contains(‘van’, regex=False)
then create the boolean coding
used_cars[‘van_code’] = np.where(used_cars[‘body_type’].str.contains(‘van’, regex=False), 1, 0)

One-hot encoding with pandas
pd.get_dummies() #making dummy variables
arguments data - a df, columns - a list-like object of a column names, pre-fix - a string to add to the beginning of each category
example
used_cars_onehot = pd.get_dummies(used_cars[[‘odometer_value’, ‘color’]])
used_cars_onehot will now have all object and categorical column one-hot encoded
meaning any numeric columns will remain the same
colors had 12 unique values > 12 columns have been created, one column per unique value (in this case for each color)
a ‘0’ in that new column means not that color
a ‘1’ means that color
this example now has 13 columns 1 for odometer and 12 for colors

Can specify which columns to use
used_cars_onehot = pd.get_dummies(used)cars, columns=‘[‘color’], prefix=‘’)
**argument ‘prefix’ is set to blank since we are only using one column
in this example instead of getting prefix “color” before _eachcolor we will just get _eachcolor
ie instead of color_black as the column name, we will get _black as the column name

**key points
if there are hundreds of unique values then hundreds of new columns will be made
be aware in certain machine learning models this can cause overfitting
**be aware NaN values do not get their own column
 - no need to have a column for missing values