Membership constraints

CLEANING DATA IN PYTHON

Adel Nehme
Content Developer @DataCamp

Chapter 2 - Text and categorical data problems

Categories and membership constraints

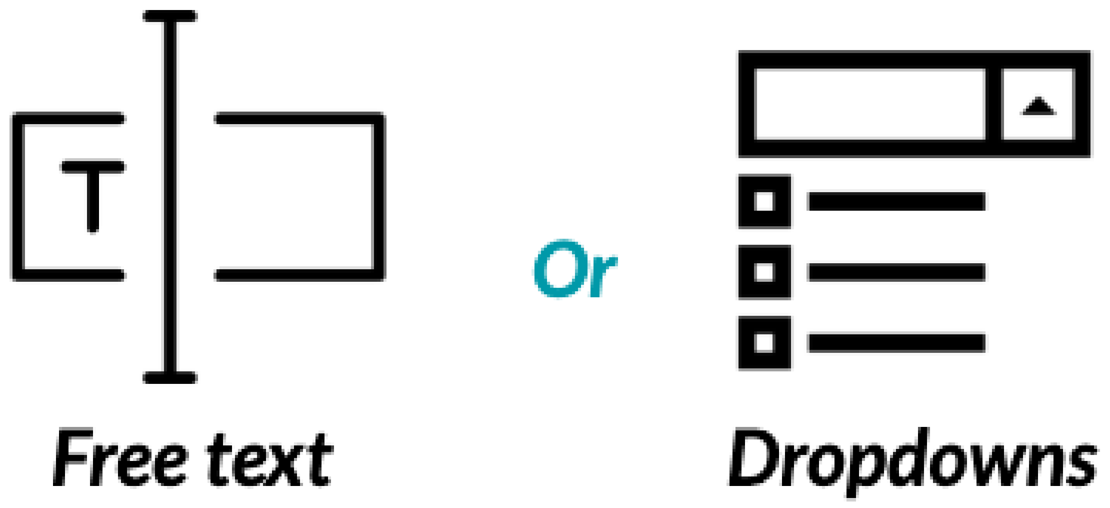
Predefined finite set of categories

Type of data	Example values	Numeric representation
Marriage Status	`unmarried` , `married`	`0` , `1`
Household Income Category	`0-20K` , `20-40K` , ...	`0` , `1` , ..
Loan Status	`default` , `payed` , `no_loan`	`0` , `1` , `2`

*Marriage status can **only** be* `unmarried` _or_ `married`

3. Categories and membership constraints
In this lesson, we'll focus on categorical variables. As discussed early in chapter 1, categorical data represent variables that represent predefined finite set of categories. Examples of this range from marriage status, household income categories, loan status and others. To run machine learning models on categorical data, they are often coded as numbers. Since categorical data represent a predefined set of categories, they can't have values that go beyond these predefined categories.

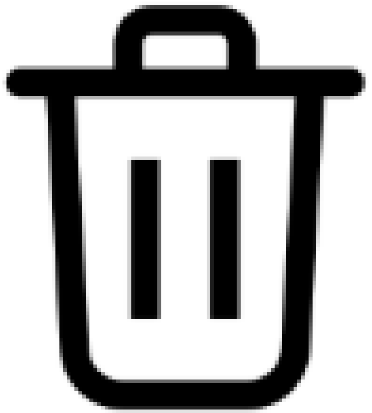
Why could we have these problems?



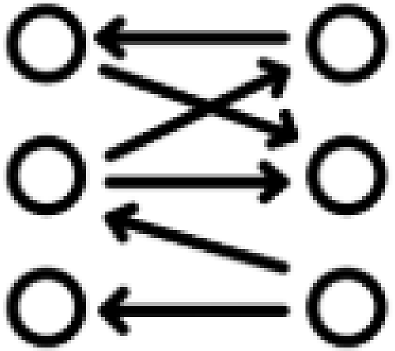
Data Entry Errors **Parsing Errors**

4. Why could we have these problems?
We can have inconsistencies in our categorical data for a variety of reasons. This could be due to data entry issues with free text vs dropdown fields, data parsing errors and other types of errors.

How do we treat these problems?



Dropping Data



Remapping Categories

Inferring Categories

5. How do we treat these problems?
There's a variety of ways we can treat these, with increasingly specific solutions for different types of inconsistencies. Most simply, we can drop the rows with incorrect categories. We can attempt remapping incorrect categories to correct ones, and more. We'll see a variety of ways of dealing with this throughout the chapter and the course, but for now we'll just focus on dropping data.

An example

```
# Read study data and print it
study_data = pd.read_csv('study.csv')
study_data
```

```
       name   birthday blood_type
1      Beth 2019-10-20         B-
2 Ignatius 2020-07-08          A-
3      Paul 2019-08-12         O+
4     Helen 2019-03-17         O-
5 Jennifer 2019-12-17          Z+
6   Kennedy 2020-04-27         A+
7     Keith 2019-04-19        AB+
```

```
# Correct possible blood types
categories
```

```
  blood_type
1         O-
2         O+
3         A-
4         A+
5         B+
6         B-
7        AB+
8        AB-
```

An example

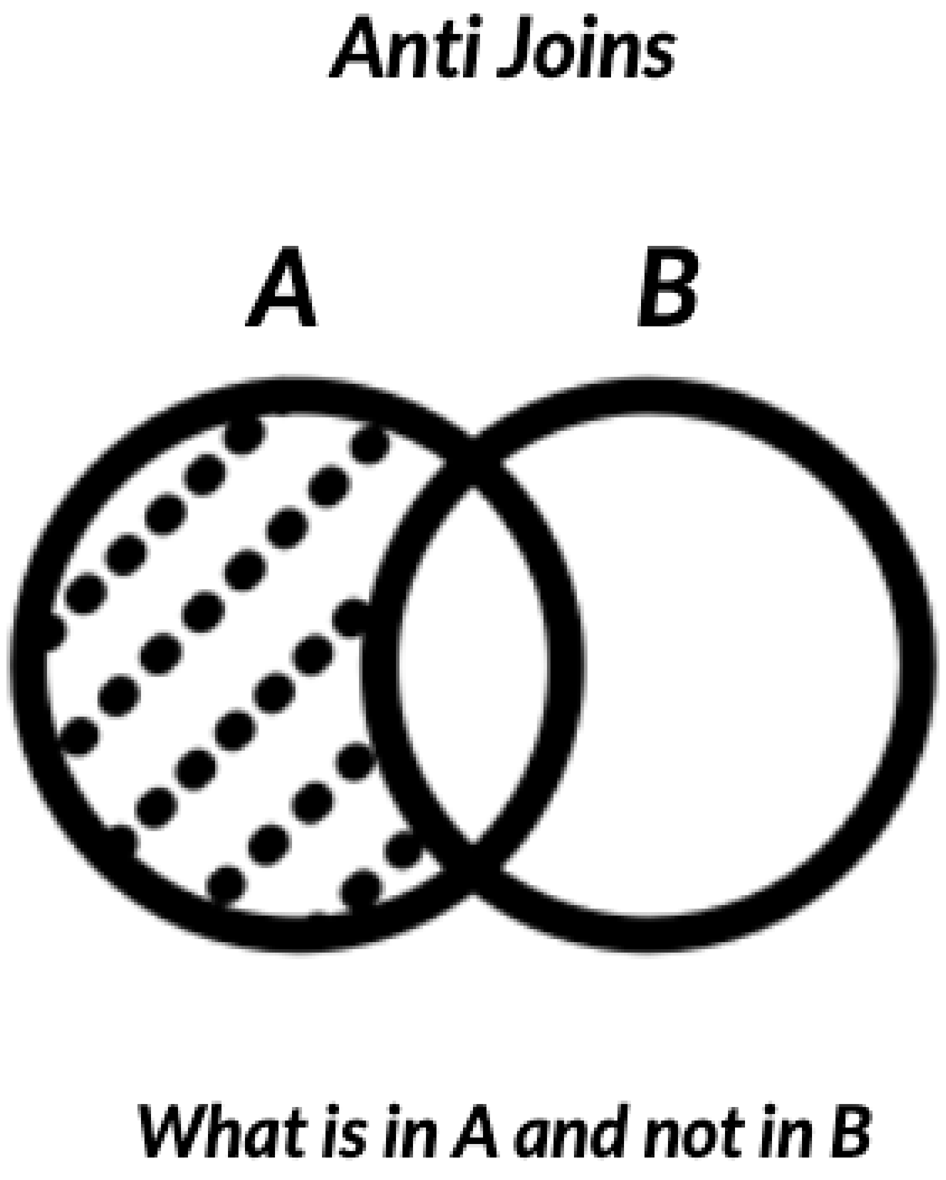
```python
# Read study data and print it
study_data = pd.read_csv('study.csv')
study_data
```

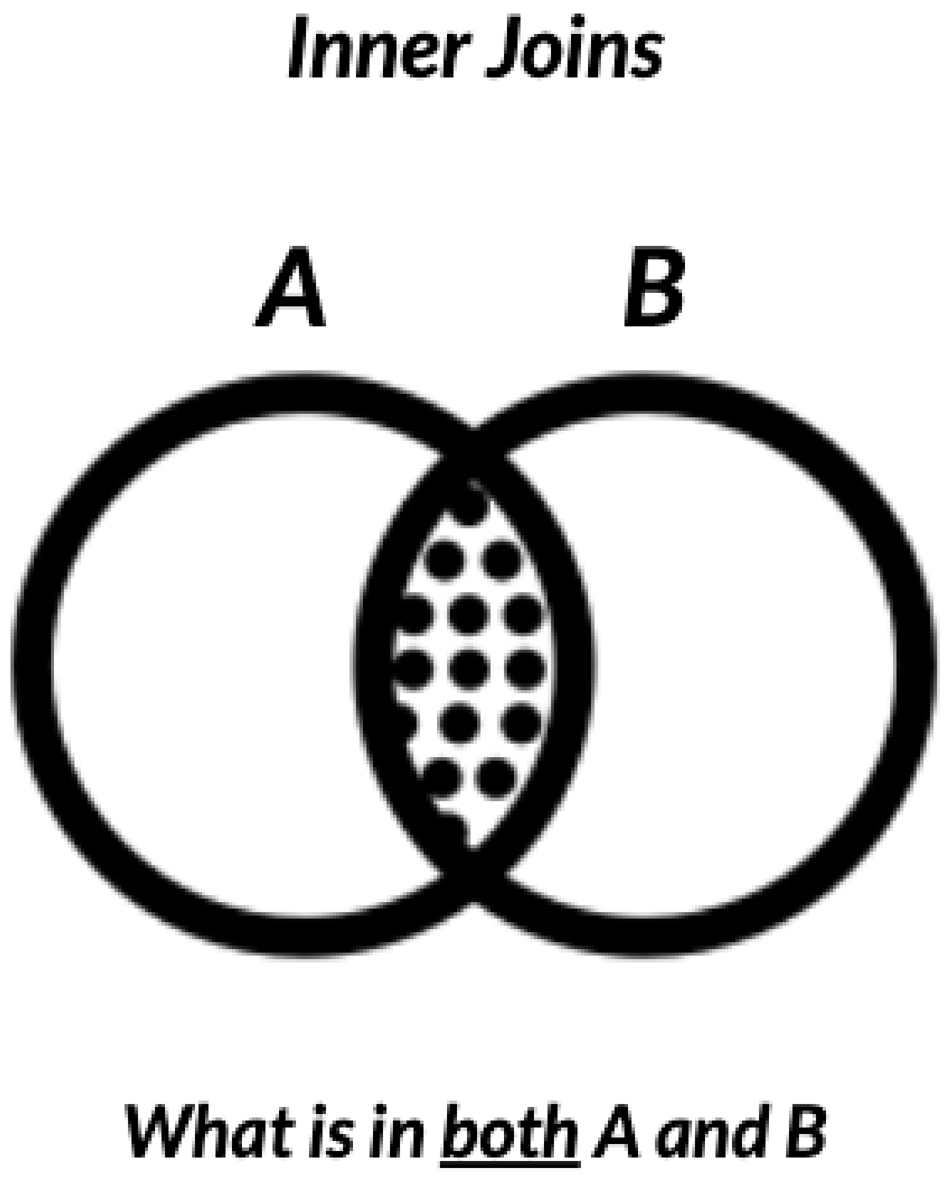
```
       name   birthday blood_type
1      Beth 2019-10-20         B-
2 Ignatius 2020-07-08         A-
3      Paul 2019-08-12         O+
4     Helen 2019-03-17         O-
5 Jennifer 2019-12-17         Z+   <--
6   Kennedy 2020-04-27         A+
7     Keith 2019-04-19        AB+
```

```python
# Correct possible blood types
categories
```

```
  blood_type
1         O-
2         O+
3         A-
4         A+
5         B+
6         B-
7        AB+
8        AB-
```

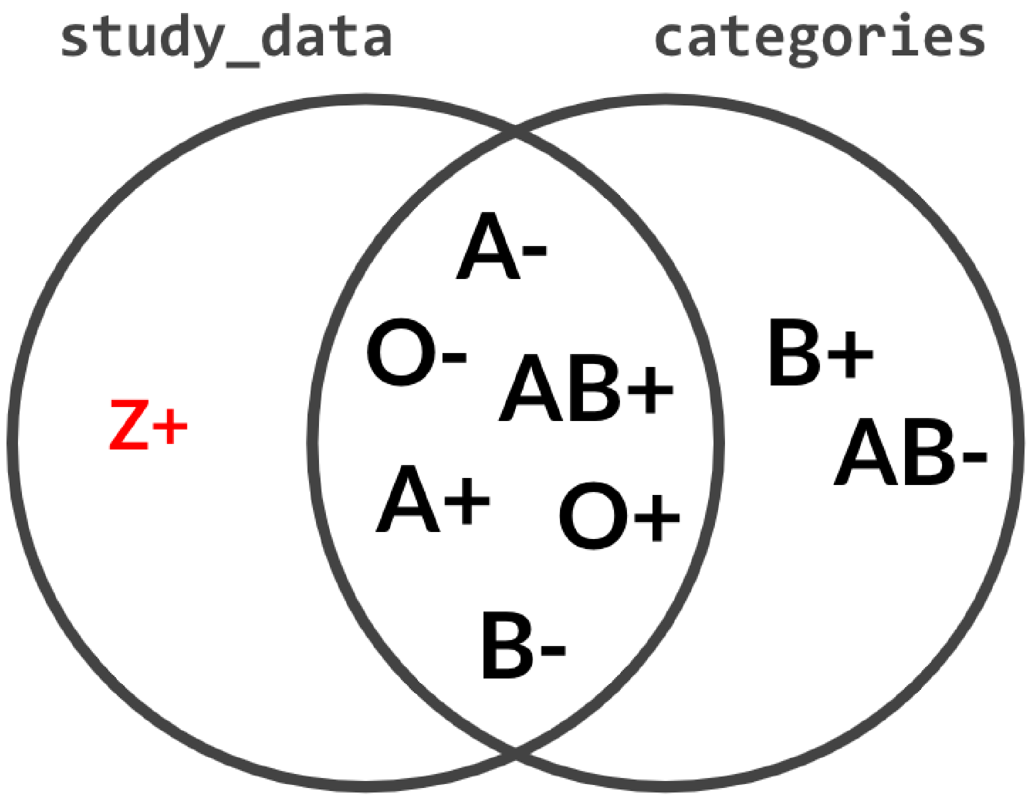
A note on joins





A left anti join on blood types

9. A left anti join on blood types
In our example, an left anti join essentially returns all the data in study data with inconsistent blood types,

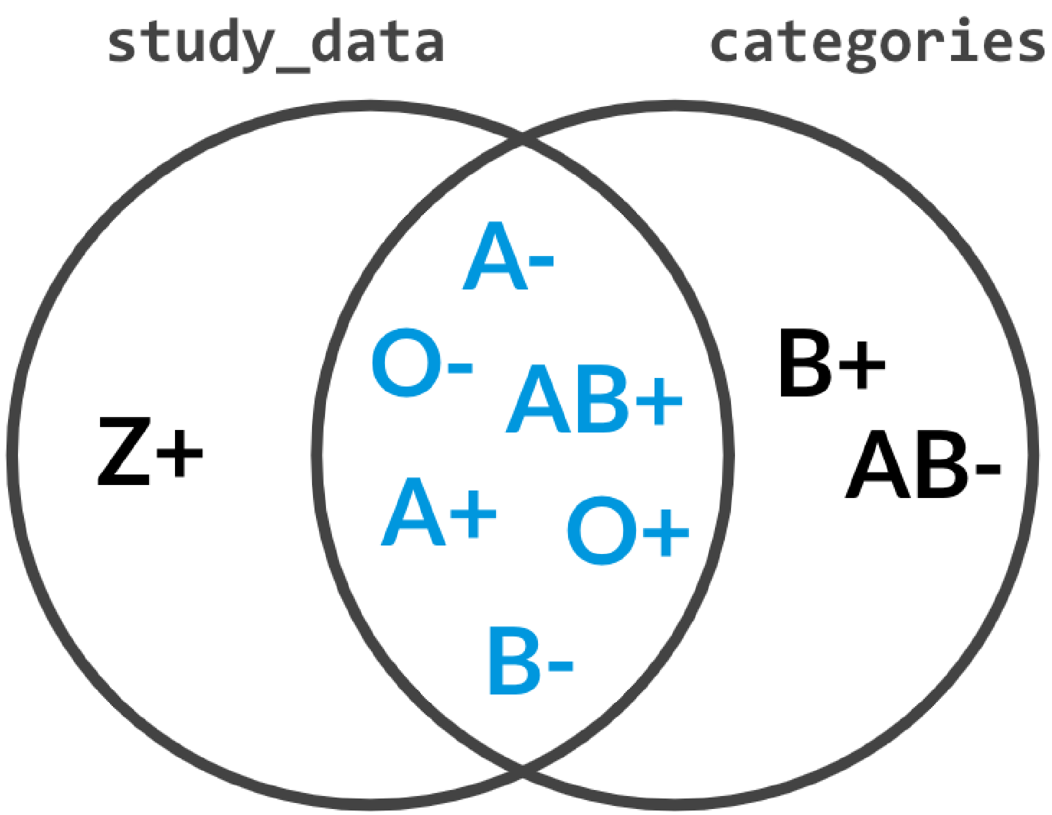


What is in `study_data` *only*

Returns only rows containing Z+

An inner join on blood types

10. An inner join on blood typesÚnd an inner join returns all the rows containing consistent blood types signs.



What is in `study_data` *and* `categories` *only*

Returns all the rows except those containing Z+ , B+ *and* AB-

Finding inconsistent categories

```
inconsistent_categories = set(study_data['blood_type']).difference(categories['blood_type'])
print(inconsistent_categories)
```

```
{'Z+'}
```

```
# Get and print rows with inconsistent categories
inconsistent_rows = study_data['blood_type'].isin(inconsistent_categories)
study_data[inconsistent_rows]
```

```
      name   birthday blood_type
5 Jennifer 2019-12-17         Z+
```

11. Finding inconsistent categories

We first get all inconsistent categories in the blood_type column of the study_data DataFrame. We do that by creating a set out of the blood_type column which stores its unique values, and use the difference method which takes in as argument the blood_type column from the categories DataFrame. This returns all the categories in blood_type that are not in categories. We then find the inconsistent rows by finding all the rows of the blood_type columns that are equal to inconsistent categories by using the isin method, this returns a series of boolean values that are True for inconsistent rows and False for consistent ones. We then subset the study_data DataFrame based on these boolean values, and voila we have our inconsistent data.

Dropping inconsistent categories

```
inconsistent_categories = set(study_data['blood_type']).difference(categories['blood_type'])
inconsistent_rows = study_data['blood_type'].isin(inconsistent_categories)
inconsistent_data = study_data[inconsistent_rows]
# Drop inconsistent categories and get consistent data only
consistent_data = study_data[~inconsistent_rows]
```

```
       name    birthday blood_type
1      Beth  2019-10-20         B-
2  Ignatius  2020-07-08         A-
3      Paul  2019-08-12         O+
4     Helen  2019-03-17         O-
...     ...         ...        ...
```

12. Dropping inconsistent categories
To drop inconsistent rows and keep ones that are only consistent. We just use the tilde symbol while subsetting which returns everything except inconsistent rows.

Let's practice!

CLEANING DATA IN PYTHON

Categorical variables

CLEANING DATA IN PYTHON

Adel Nehme
Content Developer @DataCamp

What type of errors could we have?

I) Value inconsistency

- *Inconsistent fields:* `'married'` , `'Maried'` , `'UNMARRIED'` , `'not married'` ..
- _Trailing white spaces: _ `'married '` , `' married '` ..

II) Collapsing too many categories to few

- *Creating new groups:* `0-20K` , `20-40K` categories ... from continuous household income data
- *Mapping groups to new ones:* Mapping household income categories to 2 `'rich'` , `'poor'`

III) Making sure data is of type `category` (seen in Chapter 1)

Value consistency

Capitalization: `'married'` , `'Married'` , `'UNMARRIED'` , `'unmarried'` ..

```
# Get marriage status column
marriage_status = demographics['marriage_status']
marriage_status.value_counts()
```

```
unmarried    352
married      268
MARRIED      204
UNMARRIED    176
dtype: int64
```

Value consistency

```
# Get value counts on DataFrame
marriage_status.groupby('marriage_status').count()
```

```
                 household_income  gender
marriage_status
MARRIED                       204     204
UNMARRIED                     176     176
married                       268     268
unmarried                     352     352
```

4. Value consistency
For a DataFrame, we can groupby the column and use the dot-count() method.

Value consistency

```python
# Capitalize
marriage_status['marriage_status'] = marriage_status['marriage_status'].str.upper()
marriage_status['marriage_status'].value_counts()
```

```
UNMARRIED    528
MARRIED      472
```

```python
# Lowercase
marriage_status['marriage_status'] = marriage_status['marriage_status'].str.lower()
marriage_status['marriage_status'].value_counts()
```

```
unmarried    528
married      472
```

5. Value consistency
To deal with this, we can either capitalize or lowercase the marriage_status column. This can be done with the str-dot-upper() or dot-lower() functions respectively.

Value consistency

Trailing spaces: `'married '`, `'married'`, `'unmarried'`, `' unmarried'` ..

```
# Get marriage status column
marriage_status = demographics['marriage_status']
marriage_status.value_counts()
```

```
 unmarried    352
unmarried     268
married       204
married       176
dtype: int64
```

6. Value consistency
Another common problem with categorical values are leading or trailing spaces. For example, imagine the same demographics DataFrame containing values with leading spaces. Here's what the counts of married vs unmarried people would look like. Note that there is a married category with a trailing space on the right, which makes it hard to spot on the output, as opposed to unmarried.

Value consistency

```
# Strip all spaces
demographics = demographics['marriage_status'].str.strip()
demographics['marriage_status'].value_counts()
```

```
unmarried     528
married       472
```

7. Value consistency
To remove leading spaces, we can use the str-dot-strip() method which when given no input, strips all leading and trailing white spaces.

Collapsing data into categories

Create categories out of data: `income_group` column from `income` column.

```
# Using qcut()
import pandas as pd
group_names = ['0-200K', '200K-500K', '500K+']
demographics['income_group'] = pd.qcut(demographics['household_income'], q = 3,
                                       labels = group_names)
# Print income_group column
demographics[['income_group', 'household_income']]
```

```
    category  household_income
0   200K-500K  189243
1       500K+  778533
..
```

8. Collapsing data into categories
Sometimes, we may want to create categories out of our data, such as creating household income groups from income data. To create categories out of data, let's use the example of creating an income group column in the demographics DataFrame. We can do this in 2 ways. The first method utilizes the qcut function from pandas, which automatically divides our data based on its distribution into the number of categories we set in the q argument, we created the category names in the group_names list and fed it to the labels argument, returning the following. Notice that the first row actually misrepresents the actual income of the income group, as we didn't instruct qcut where our ranges actually lie.

Collapsing data into categories

Create categories out of data: `income_group` column from `income` column.

```
# Using cut() - create category ranges and names
ranges = [0,200000,500000,np.inf]
group_names = ['0-200K', '200K-500K', '500K+']
# Create income group column
demographics['income_group'] = pd.cut(demographics['household_income'], bins=ranges,
                                      labels=group_names)
demographics[['income_group', 'household_income']]
```

```
      category  Income
0       0-200K  189243
1        500K+  778533
```

9. Collapsing data into categories

We can do this with the cut function instead, which lets us define category cutoff ranges with the bins argument. It takes in a list of cutoff points for each category, with the final one being infinity represented with np-dot-inf(). From the output, we can see this is much more correct.

Collapsing data into categories

Map categories to fewer ones: reducing categories in categorical column.

`operating_system` column is: `'Microsoft', 'MacOS', 'IOS', 'Android', 'Linux'`

`operating_system` column should become: `'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()
```

```
array(['DesktopOS', 'MobileOS'], dtype=object)
```

10. Collapsing data into categories
Sometimes, we may want to reduce the amount of categories we have in our data. Let's move on to mapping categories to fewer ones. For example, assume we have a column containing the operating system of different devices, and contains these unique values. Say we want to collapse these categories into 2, DesktopOS, and MobileOS. We can do this using the replace method. It takes in a dictionary that maps each existing category to the category name you desire. In this case, this is the mapping dictionary. A quick print of the unique values of operating system shows the mapping has been complete.

Let's practice!

CLEANING DATA IN PYTHON

Cleaning text data

CLEANING DATA IN PYTHON

Adel Nehme
Content Developer @ DataCamp

What is text data?

Type of data	Example values
Names	`Alex` , `Sara` ...
Phone numbers	`+96171679912` ...
Emails	`adel@datacamp.com`..
Passwords	...

Common text data problems

1) Data inconsistency:

`+96171679912` or `0096171679912` or ..?

2) Fixed length violations:

Passwords needs to be at least 8 characters

3) Typos:

`+961.71.679912`

Example

```
phones = pd.read_csv('phones.csv')
print(phones)
```

```
           Full name      Phone number
0      Noelani A. Gray  001-702-397-5143
1       Myles Z. Gomez  001-329-485-0540
2         Gil B. Silva  001-195-492-2338
3   Prescott D. Hardin   +1-297-996-4904
4   Benedict G. Valdez  001-969-820-3536
5     Reece M. Andrews              4138
6       Hayfa E. Keith  001-536-175-8444
7      Hedley I. Logan  001-681-552-1823
8     Jack W. Carrillo  001-910-323-5265
9      Lionel M. Davis  001-143-119-9210
```

Example

```
phones = pd.read_csv('phones.csv')
print(phones)
```

```
          Full name      Phone number
0       Noelani A. Gray  001-702-397-5143
1        Myles Z. Gomez  001-329-485-0540
2          Gil B. Silva  001-195-492-2338
3    Prescott D. Hardin   +1-297-996-4904   <-- Inconsistent data format
4    Benedict G. Valdez  001-969-820-3536
5      Reece M. Andrews              4138   <-- Length violation
6        Hayfa E. Keith  001-536-175-8444
7       Hedley I. Logan  001-681-552-1823
8      Jack W. Carrillo  001-910-323-5265
9       Lionel M. Davis  001-143-119-9210
```

Example

```
phones = pd.read_csv('phones.csv')
print(phones)
```

```
              Full name   Phone number
0        Noelani A. Gray  001702397514З
1         Myles Z. Gomez  001329485054О
2           Gil B. Silva  0011954922338
3     Prescott D. Hardin  0012979964904
4     Benedict G. Valdez  0019698203536
5       Reece M. Andrews            NaN
6         Hayfa E. Keith  0015361758444
7        Hedley I. Logan  0016815521823
8       Jack W. Carrillo  0019103235265
9         Lionel M. Davis  0011431199210
```

Fixing the phone number column

```
# Replace "+" with "00"
phones["Phone number"] = phones["Phone number"].str.replace("+", "00")
phones
```

```
           Full name      Phone number
0     Noelani A. Gray  001-702-397-5143
1      Myles Z. Gomez  001-329-485-0540
2        Gil B. Silva  001-195-492-2338
3  Prescott D. Hardin  001-297-996-4904
4  Benedict G. Valdez  001-969-820-3536
5    Reece M. Andrews              4138
6      Hayfa E. Keith  001-536-175-8444
7     Hedley I. Logan  001-681-552-1823
8    Jack W. Carrillo  001-910-323-5265
9     Lionel M. Davis  001-143-119-9210
```

Fixing the phone number column

```python
# Replace "-" with nothing
phones["Phone number"] = phones["Phone number"].str.replace("-", "")
phones
```

```
             Full name   Phone number
0      Noelani A. Gray  0017023975143
1       Myles Z. Gomez  0013294850540
2         Gil B. Silva  0011954922338
3   Prescott D. Hardin  0012979964904
4   Benedict G. Valdez  0019698203536
5     Reece M. Andrews           4138
6       Hayfa E. Keith  0015361758444
7      Hedley I. Logan  0016815521823
8     Jack W. Carrillo  0019103235265
9      Lionel M. Davis  0011431199210
```

Fixing the phone number column

```
# Replace phone numbers with lower than 10 digits to NaN
digits = phones['Phone number'].str.len()
phones.loc[digits < 10, "Phone number"] = np.nan
phones
```

```
              Full name   Phone number
0       Noelani A. Gray  0017023975143
1        Myles Z. Gomez  0013294850540
2          Gil B. Silva  0011954922338
3    Prescott D. Hardin  0012979964904
4    Benedict G. Valdez  0019698203536
5      Reece M. Andrews            NaN
6        Hayfa E. Keith  0015361758444
7       Hedley I. Logan  0016815521823
8      Jack W. Carrillo  0019103235265
9       Lionel M. Davis  0011431199210
```

Fixing the phone number column

```
# Find length of each row in Phone number column
sanity_check = phone['Phone number'].str.len()
```

```
# Assert minmum 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
```

Remember, `assert` ***returns nothing if the condition passes***

But what about more complicated examples?

```
phones.head()
```

```
          Full name       Phone number
0     Olga Robinson  +(01706)-25891
1       Justina Kim    +0500-571437
2    Tamekah Henson      +0800-1111
3     Miranda Solis  +07058-879063
4  Caldwell Gilliam  +(016977)-8424
```

Supercharged control + F

Regular expressions in action

```
# Replace letters with nothing
phones['Phone number'] = phones['Phone number'].str.replace(r'\D+', '')
phones.head()
```

```
          Full name Phone number
0     Olga Robinson   0170625891
1       Justina Kim   0500571437
2    Tamekah Henson     08001111
3     Miranda Solis  07058879063
4  Caldwell Gilliam   0169778424
```

Let's practice!

CLEANING DATA IN PYTHON