**Data Cleaning**

Data cleaning is an essential step when data wrangling; if you don’t clean your data, you’re likely to run into some issues when it comes time to build your models. Putting the time into cleaning your data will result in a seamless transition from wrangling to EDA and modeling.

1

**Cleaning Data in Python**

Save



4 - 6 Hours

91 Points

A vital component of data science involves acquiring raw data and getting it ready for analysis. In fact, data scientists tend to spend more time cleaning and manipulating data than analyzing it. This course will equip you with skills to clean data in Python, from learning how to diagnose data for problems to dealing with missing values and outliers.

2

**Python Data Science Toolbox, Part 2**

Save



4 - 6 Hours

91 Points

In this DataCamp resource, you'll continue to build your Python data science skills. First, you'll enter the wonderful world of iterators, objects that you’ve already encountered in the context of loops. You’ll also learn about list comprehensions, a handy tool that all data scientists should have in their toolboxes. You'll end the course by working through a case study in which you'll apply all of the techniques you’ve learned.

We've included this resource in this subunit because it will teach you the skills you need — including using iterators, enumeration, and list comprehension — to write efficient data cleaning functions instead of having to repeatedly hardcode solutions.

3

**Handling Text Data**

Save



40 Minutes - 1 Hour

15 Points

This resource will walk you through some pandas string operations before demonstrating how you can use them to clean up particularly messy datasets.

To use this resource, open it up and download the associated notebook using the download button on the right side of the screen. It's important that, as you work through the resource, you make a note of the output from the code in each cell. You can either run the code locally or review it on the resource site.

This notebook contains an excerpt from the [*Python Data Science Handbook*](http://shop.oreilly.com/product/0636920034919.do) by Jake VanderPlas; the content is available [*on GitHub*](https://github.com/jakevdp/PythonDataScienceHandbook).

The text is released under the [*CC-BY-NC-ND license*](https://creativecommons.org/licenses/by-nc-nd/3.0/us/legalcode), and code is released under the [*MIT license*](https://opensource.org/licenses/MIT). If you find this content useful, please consider supporting the work by [*buying the book*](http://shop.oreilly.com/product/0636920034919.do)!

< [Pivot Tables](https://github.com/springboard-curriculum/PythonDataScienceHandbook/blob/13290ad4bf36924bbafde3a5c1b37e7858402e4a/notebooks/03.09-Pivot-Tables.ipynb) | [Contents](https://github.com/springboard-curriculum/PythonDataScienceHandbook/blob/13290ad4bf36924bbafde3a5c1b37e7858402e4a/notebooks/Index.ipynb) | [Working with Time Series](https://github.com/springboard-curriculum/PythonDataScienceHandbook/blob/13290ad4bf36924bbafde3a5c1b37e7858402e4a/notebooks/03.11-Working-with-Time-Series.ipynb) >

# Vectorized String Operations

One strength of Python is its relative ease in handling and manipulating string data. Pandas builds on this and provides a comprehensive set of vectorized string operations that become an essential piece of the type of munging required when working with (read: cleaning up) real-world data. In this section, we'll walk through some of the Pandas string operations, and then take a look at using them to partially clean up a very messy dataset of recipes collected from the Internet.

## Introducing Pandas String Operations

We saw in previous sections how tools like NumPy and Pandas generalize arithmetic operations so that we can easily and quickly perform the same operation on many array elements. For example:

import numpy as np

x = np.array([2, 3, 5, 7, 11, 13])

x \* 2

array([ 4, 6, 10, 14, 22, 26])

This vectorization of operations simplifies the syntax of operating on arrays of data: we no longer have to worry about the size or shape of the array, but just about what operation we want done. For arrays of strings, NumPy does not provide such simple access, and thus you're stuck using a more verbose loop syntax:

data = ['peter', 'Paul', 'MARY', 'gUIDO']

[s.capitalize() for s in data]

['Peter', 'Paul', 'Mary', 'Guido']

This is perhaps sufficient to work with some data, but it will break if there are any missing values. For example:

data = ['peter', 'Paul', None, 'MARY', 'gUIDO']

[s.capitalize() for s in data]

---------------------------------------------------------------------------

AttributeError Traceback (most recent call last)

<ipython-input-3-3b0264c38d59> in <module>

1 data = ['peter', 'Paul', None, 'MARY', 'gUIDO']

----> 2 [s.capitalize() for s in data]

<ipython-input-3-3b0264c38d59> in <listcomp>(.0)

1 data = ['peter', 'Paul', None, 'MARY', 'gUIDO']

----> 2 [s.capitalize() for s in data]

AttributeError: 'NoneType' object has no attribute 'capitalize'

Pandas includes features to address both this need for vectorized string operations and for correctly handling missing data via the str attribute of Pandas Series and Index objects containing strings. So, for example, suppose we create a Pandas Series with this data:

import pandas as pd

names = pd.Series(data)

names

0 peter

1 Paul

2 None

3 MARY

4 gUIDO

dtype: object

We can now call a single method that will capitalize all the entries, while skipping over any missing values:

names.str.capitalize()

0 Peter

1 Paul

2 None

3 Mary

4 Guido

dtype: object

Using tab completion on this str attribute will list all the vectorized string methods available to Pandas.

## Tables of Pandas String Methods

If you have a good understanding of string manipulation in Python, most of Pandas string syntax is intuitive enough that it's probably sufficient to just list a table of available methods; we will start with that here, before diving deeper into a few of the subtleties. The examples in this section use the following series of names:

monte = pd.Series(['Graham Chapman', 'John Cleese', 'Terry Gilliam',

'Eric Idle', 'Terry Jones', 'Michael Palin'])

### Methods similar to Python string methods

Nearly all Python's built-in string methods are mirrored by a Pandas vectorized string method. Here is a list of Pandas str methods that mirror Python string methods:

|  |  |  |  |
| --- | --- | --- | --- |
| len() | lower() | translate() | islower() |
| ljust() | upper() | startswith() | isupper() |
| rjust() | find() | endswith() | isnumeric() |
| center() | rfind() | isalnum() | isdecimal() |
| zfill() | index() | isalpha() | split() |
| strip() | rindex() | isdigit() | rsplit() |
| rstrip() | capitalize() | isspace() | partition() |
| lstrip() | swapcase() | istitle() | rpartition() |

Notice that these have various return values. Some, like lower(), return a series of strings:

monte.str.lower()

0 graham chapman

1 john cleese

2 terry gilliam

3 eric idle

4 terry jones

5 michael palin

dtype: object

But some others return numbers:

monte.str.len()

0 14

1 11

2 13

3 9

4 11

5 13

dtype: int64

Or Boolean values:

monte.str.startswith('T')

0 False

1 False

2 True

3 False

4 True

5 False

dtype: bool

Still others return lists or other compound values for each element:

monte.str.split()

0 [Graham, Chapman]

1 [John, Cleese]

2 [Terry, Gilliam]

3 [Eric, Idle]

4 [Terry, Jones]

5 [Michael, Palin]

dtype: object

We'll see further manipulations of this kind of series-of-lists object as we continue our discussion.

### Methods using regular expressions

In addition, there are several methods that accept regular expressions to examine the content of each string element, and follow some of the API conventions of Python's built-in re module:

| **Method** | **Description** |
| --- | --- |
| match() | Call re.match() on each element, returning a boolean. |
| extract() | Call re.match() on each element, returning matched groups as strings. |
| findall() | Call re.findall() on each element |
| replace() | Replace occurrences of pattern with some other string |
| contains() | Call re.search() on each element, returning a boolean |
| count() | Count occurrences of pattern |
| split() | Equivalent to str.split(), but accepts regexps |
| rsplit() | Equivalent to str.rsplit(), but accepts regexps |

With these, you can do a wide range of interesting operations. For example, we can extract the first name from each by asking for a contiguous group of characters at the beginning of each element:

monte.str.extract('([A-Za-z]+)', expand=False)

0 Graham

1 John

2 Terry

3 Eric

4 Terry

5 Michael

dtype: object

Or we can do something more complicated, like finding all names that start and end with a consonant, making use of the start-of-string (^) and end-of-string ($) regular expression characters:

monte.str.findall(r'^[^AEIOU].\*[^aeiou]$')

0 [Graham Chapman]

1 []

2 [Terry Gilliam]

3 []

4 [Terry Jones]

5 [Michael Palin]

dtype: object

The ability to concisely apply regular expressions across Series or Dataframe entries opens up many possibilities for analysis and cleaning of data.

### Miscellaneous methods

Finally, there are some miscellaneous methods that enable other convenient operations:

| **Method** | **Description** |
| --- | --- |
| get() | Index each element |
| slice() | Slice each element |
| slice\_replace() | Replace slice in each element with passed value |
| cat() | Concatenate strings |
| repeat() | Repeat values |
| normalize() | Return Unicode form of string |
| pad() | Add whitespace to left, right, or both sides of strings |
| wrap() | Split long strings into lines with length less than a given width |
| join() | Join strings in each element of the Series with passed separator |
| get\_dummies() | extract dummy variables as a dataframe |

#### Vectorized item access and slicing

The get() and slice() operations, in particular, enable vectorized element access from each array. For example, we can get a slice of the first three characters of each array using str.slice(0, 3). Note that this behavior is also available through Python's normal indexing syntax–for example, df.str.slice(0, 3) is equivalent to df.str[0:3]:

monte.str[0:3]

0 Gra

1 Joh

2 Ter

3 Eri

4 Ter

5 Mic

dtype: object

Indexing via df.str.get(i) and df.str[i] is likewise similar.

These get() and slice() methods also let you access elements of arrays returned by split(). For example, to extract the last name of each entry, we can combine split() and get():

monte.str.split().str.get(-1)

0 Chapman

1 Cleese

2 Gilliam

3 Idle

4 Jones

5 Palin

dtype: object

#### Indicator variables

Another method that requires a bit of extra explanation is the get\_dummies() method. This is useful when your data has a column containing some sort of coded indicator. For example, we might have a dataset that contains information in the form of codes, such as A="born in America," B="born in the United Kingdom," C="likes cheese," D="likes spam":

full\_monte = pd.DataFrame({'name': monte,

'info': ['B|C|D', 'B|D', 'A|C',

'B|D', 'B|C', 'B|C|D']})

full\_monte

|  | **name** | **info** |
| --- | --- | --- |
| **0** | Graham Chapman | B|C|D |
| **1** | John Cleese | B|D |
| **2** | Terry Gilliam | A|C |
| **3** | Eric Idle | B|D |
| **4** | Terry Jones | B|C |
| **5** | Michael Palin | B|C|D |

The get\_dummies() routine lets you quickly split-out these indicator variables into a DataFrame:

full\_monte['info'].str.get\_dummies('|')

|  | **A** | **B** | **C** | **D** |
| --- | --- | --- | --- | --- |
| **0** | 0 | 1 | 1 | 1 |
| **1** | 0 | 1 | 0 | 1 |
| **2** | 1 | 0 | 1 | 0 |
| **3** | 0 | 1 | 0 | 1 |
| **4** | 0 | 1 | 1 | 0 |
| **5** | 0 | 1 | 1 | 1 |

With these operations as building blocks, you can construct an endless range of string processing procedures when cleaning your data.

We won't dive further into these methods here, but I encourage you to read through ["Working with Text Data"](http://pandas.pydata.org/pandas-docs/stable/text.html) in the Pandas online documentation, or to refer to the resources listed in [Further Resources](https://github.com/springboard-curriculum/PythonDataScienceHandbook/blob/13290ad4bf36924bbafde3a5c1b37e7858402e4a/notebooks/03.13-Further-Resources.ipynb).

## Example: Recipe Database

These vectorized string operations become most useful in the process of cleaning up messy, real-world data. Here I'll walk through an example of that, using an open recipe database compiled from various sources on the Web. Our goal will be to parse the recipe data into ingredient lists, so we can quickly find a recipe based on some ingredients we have on hand.

The scripts used to compile this can be found at <https://github.com/fictivekin/openrecipes>, and the link to the current version of the database is found there as well.

As of Summer 2020, this database is about no longer supported so we have placed the last available dump 20170107-061401-recipeitems.json in the data folder.

The database is in JSON format, so we will try pd.read\_json to read it:

try:

recipes = pd.read\_json('./data/20170107-061401-recipeitems\_small\_sample.json')

except ValueError as e:

print("ValueError:", e)

ValueError: Trailing data

Oops! We get a ValueError mentioning that there is "trailing data." Searching for the text of this error on the Internet, it seems that it's due to using a file in which each line is itself a valid JSON, but the full file is not. Let's check if this interpretation is true:

# need use stringio here for latest pandas (1.1.1) for some reason seems to cuase issues with pd.read\_json()

# https://stackoverflow.com/questions/63553845/pandas-read-json-valueerror-protocol-not-known

from io import StringIO, BytesIO

import gzip

with gzip.open('./data/20170107-061401-recipeitems.json.gz', 'rb') as f:

line = f.readline()

pd.read\_json(BytesIO(line)).shape

(2, 12)

Yes, apparently each line is a valid JSON, so we'll need to string them together. One way we can do this is to actually construct a string representation containing all these JSON entries, and then load the whole thing with pd.read\_json:

# read the entire file into a Python array

with gzip.open('./data/20170107-061401-recipeitems.json.gz', 'rb') as f:

# Extract each line

data = (BytesIO(line).getvalue().decode().strip() for line in f)

# Reformat so each line is the element of a list

data\_json = "[{0}]".format(','.join(data))

# read the result as a JSON

recipes = pd.read\_json(StringIO(data\_json))

recipes.shape

(173278, 17)

We see there are nearly 200,000 recipes, and 17 columns. Let's take a look at one row to see what we have:

recipes.iloc[0]

\_id {'$oid': '5160756b96cc62079cc2db15'}

name Drop Biscuits and Sausage Gravy

ingredients Biscuits\n3 cups All-purpose Flour\n2 Tablespo...

url http://thepioneerwoman.com/cooking/2013/03/dro...

image http://static.thepioneerwoman.com/cooking/file...

ts {'$date': 1365276011104}

cookTime PT30M

source thepioneerwoman

recipeYield 12

datePublished 2013-03-11

prepTime PT10M

description Late Saturday afternoon, after Marlboro Man ha...

totalTime NaN

creator NaN

recipeCategory NaN

dateModified NaN

recipeInstructions NaN

Name: 0, dtype: object

There is a lot of information there, but much of it is in a very messy form, as is typical of data scraped from the Web. In particular, the ingredient list is in string format; we're going to have to carefully extract the information we're interested in. Let's start by taking a closer look at the ingredients:

recipes.ingredients.str.len().describe()

count 173278.000000

mean 244.617926

std 146.705285

min 0.000000

25% 147.000000

50% 221.000000

75% 314.000000

max 9067.000000

Name: ingredients, dtype: float64

The ingredient lists average 250 characters long, with a minimum of 0 and a maximum of nearly 10,000 characters!

Just out of curiousity, let's see which recipe has the longest ingredient list:

recipes.name[np.argmax(recipes.ingredients.str.len())]

'Carrot Pineapple Spice &amp; Brownie Layer Cake with Whipped Cream &amp; Cream Cheese Frosting and Marzipan Carrots'

That certainly looks like an involved recipe.

We can do other aggregate explorations; for example, let's see how many of the recipes are for breakfast food:

recipes.description.str.contains('[Bb]reakfast').sum()

3524

Or how many of the recipes list cinnamon as an ingredient:

recipes.ingredients.str.contains('[Cc]innamon').sum()

10526

We could even look to see whether any recipes misspell the ingredient as "cinamon":

recipes.ingredients.str.contains('[Cc]inamon').sum()

11

This is the type of essential data exploration that is possible with Pandas string tools. It is data munging like this that Python really excels at.

### A simple recipe recommender

Let's go a bit further, and start working on a simple recipe recommendation system: given a list of ingredients, find a recipe that uses all those ingredients. While conceptually straightforward, the task is complicated by the heterogeneity of the data: there is no easy operation, for example, to extract a clean list of ingredients from each row. So we will cheat a bit: we'll start with a list of common ingredients, and simply search to see whether they are in each recipe's ingredient list. For simplicity, let's just stick with herbs and spices for the time being:

spice\_list = ['salt', 'pepper', 'oregano', 'sage', 'parsley',

'rosemary', 'tarragon', 'thyme', 'paprika', 'cumin']

We can then build a Boolean DataFrame consisting of True and False values, indicating whether this ingredient appears in the list:

import re

spice\_df = pd.DataFrame(dict((spice, recipes.ingredients.str.contains(spice, re.IGNORECASE))

for spice in spice\_list))

spice\_df.head()

|  | **salt** | **pepper** | **oregano** | **sage** | **parsley** | **rosemary** | **tarragon** | **thyme** | **paprika** | **cumin** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | False | False | False | True | False | False | False | False | False | False |
| **1** | False | False | False | False | False | False | False | False | False | False |
| **2** | True | True | False | False | False | False | False | False | False | True |
| **3** | False | False | False | False | False | False | False | False | False | False |
| **4** | False | False | False | False | False | False | False | False | False | False |

Now, as an example, let's say we'd like to find a recipe that uses parsley, paprika, and tarragon. We can compute this very quickly using the query() method of DataFrames, discussed in [High-Performance Pandas: eval() and query()](https://github.com/springboard-curriculum/PythonDataScienceHandbook/blob/13290ad4bf36924bbafde3a5c1b37e7858402e4a/notebooks/03.12-Performance-Eval-and-Query.ipynb):

selection = spice\_df.query('parsley & paprika & tarragon')

len(selection)

10

We find only 10 recipes with this combination; let's use the index returned by this selection to discover the names of the recipes that have this combination:

recipes.name[selection.index]

2069 All cremat with a Little Gem, dandelion and wa...

74964 Lobster with Thermidor butter

93768 Burton's Southern Fried Chicken with White Gravy

113926 Mijo's Slow Cooker Shredded Beef

137686 Asparagus Soup with Poached Eggs

140530 Fried Oyster Po’boys

158475 Lamb shank tagine with herb tabbouleh

158486 Southern fried chicken in buttermilk

163175 Fried Chicken Sliders with Pickles + Slaw

165243 Bar Tartine Cauliflower Salad

Name: name, dtype: object

Now that we have narrowed down our recipe selection by a factor of almost 20,000, we are in a position to make a more informed decision about what we'd like to cook for dinner.

### Going further with recipes

Hopefully this example has given you a bit of a flavor (ba-dum!) for the types of data cleaning operations that are efficiently enabled by Pandas string methods. Of course, building a very robust recipe recommendation system would require a lot more work! Extracting full ingredient lists from each recipe would be an important piece of the task; unfortunately, the wide variety of formats used makes this a relatively time-consuming process. This points to the truism that in data science, cleaning and munging of real-world data often comprises the majority of the work, and Pandas provides the tools that can help you do this efficiently.

< [Pivot Tables](https://github.com/springboard-curriculum/PythonDataScienceHandbook/blob/13290ad4bf36924bbafde3a5c1b37e7858402e4a/notebooks/03.09-Pivot-Tables.ipynb) | [Contents](https://github.com/springboard-curriculum/PythonDataScienceHandbook/blob/13290ad4bf36924bbafde3a5c1b37e7858402e4a/notebooks/Index.ipynb) | [Working with Time Series](https://github.com/springboard-curriculum/PythonDataScienceHandbook/blob/13290ad4bf36924bbafde3a5c1b37e7858402e4a/notebooks/03.11-Working-with-Time-Series.ipynb) >

# Cleaning Data in Python

* 4 hours
* 13 Videos
* 44 Exercises
* 78,447 Participants
* 3,500 XP

### Course Description



It's commonly said that data scientists spend 80% of their time cleaning and manipulating data and only 20% of their time analyzing it. The time spent cleaning is vital since analyzing dirty data can lead you to draw inaccurate conclusions. Data cleaning is an essential task in data science. Without properly cleaned data, the results of any data analysis or machine learning model could be inaccurate. In this course, you will learn how to identify, diagnose, and treat a variety of data cleaning problems in Python, ranging from simple to advanced. You will deal with improper data types, check that your data is in the correct range, handle missing data, perform record linkage, and more!

1. 1

#### Common data problems

0%

In this chapter, you'll learn how to overcome some of the most common dirty data problems. You'll convert data types, apply range constraints to remove future data points, and remove duplicated data points to avoid double-counting.

##### Data type constraints

50 xp

##### Common data types

100 xp

##### Numeric data or ... ?

100 xp

##### Summing strings and concatenating numbers

100 xp

##### Data range constraints

50 xp

##### Tire size constraints

100 xp

##### Back to the future

100 xp

##### Uniqueness constraints

50 xp

##### How big is your subset?

50 xp

##### Finding duplicates

100 xp

##### Treating duplicates

100 xp

[Hide Chapter Details](https://www.datacamp.com/courses/data-cleaning-in-python?embedded=true)

  2

#### Text and categorical data problems

0%

Categorical and text data can often be some of the messiest parts of a dataset due to their unstructured nature. In this chapter, you’ll learn how to fix whitespace and capitalization inconsistencies in category labels, collapse multiple categories into one, and reformat strings for consistency.

##### Membership constraints

50 xp

##### Members only

100 xp

##### Finding consistency

100 xp

##### Categorical variables

50 xp

##### Categories of errors

100 xp

##### Inconsistent categories

100 xp

##### Remapping categories

100 xp

##### Cleaning text data

50 xp

##### Removing titles and taking names

100 xp

##### Keeping it descriptive

100 xp

[Hide Chapter Details](https://www.datacamp.com/courses/data-cleaning-in-python?embedded=true)

  3

#### Advanced data problems

0%

In this chapter, you’ll dive into more advanced data cleaning problems, such as ensuring that weights are all written in kilograms instead of pounds. You’ll also gain invaluable skills that will help you verify that values have been added correctly and that missing values don’t negatively impact your analyses.

##### Uniformity

50 xp

##### Ambiguous dates

50 xp

##### Uniform currencies

100 xp

##### Uniform dates

100 xp

##### Cross field validation

50 xp

##### Cross field or no cross field?

100 xp

##### How's our data integrity?

100 xp

##### Completeness

50 xp

##### Is this missing at random?

50 xp

##### Missing investors

100 xp

##### Follow the money

100 xp

[Hide Chapter Details](https://www.datacamp.com/courses/data-cleaning-in-python?embedded=true)

  4

#### Record linkage

0%

Record linkage is a powerful technique used to merge multiple datasets together, used when values have typos or different spellings. In this chapter, you'll learn how to link records by calculating the similarity between strings—you’ll then use your new skills to join two restaurant review datasets into one clean master dataset.

##### Comparing strings

50 xp

##### Minimum edit distance

50 xp

##### The cutoff point

100 xp

##### Remapping categories II

100 xp

##### Generating pairs

50 xp

##### To link or not to link?

100 xp

##### Pairs of restaurants

100 xp

##### Similar restaurants

100 xp

##### Linking DataFrames

50 xp

##### Getting the right index

50 xp

##### Linking them together!

100 xp

##### Congratulations!

50 xp

[Hide Chapter Details](https://www.datacamp.com/courses/data-cleaning-in-python?embedded=true)

**Daily XP0**

# Data type constraints

**50 XP**

## 1. Data type constraints

Hi and welcome! My name is Adel, and I'll be your host as we learn how to clean data in Python.

## 2. Course outline

In this course, we're going to understand how to diagnose different problems in our data and how they can can come up during our workflow.

## 3. Course outline

We will also understand the side effects of not treating our data correctly.

## 4. Course outline

and various ways to address different types of dirty data.

## 5. Course outline

In this chapter, we're going to discuss the most common data problems you may encounter and how to address them. So let's get started!

## 6. Why do we need to clean data?

To understand why we need to clean data, let's remind ourselves of the data science workflow. In a typical data science workflow, we usually access our raw data, explore and process it, develop insights using visualizations or predictive models, and finally report these insights with dashboards or reports.

## 7. Why do we need to clean data?

Dirty data can appear because of duplicate values, mis-spellings, data type parsing errors and legacy systems.

## 8. Why do we need to clean data?

Without making sure that data is properly cleaned in the exploration and processing phase, we will surely compromise the insights and reports subsequently generated. As the old adage says, garbage in garbage out.

## 9. Data type constraints

When working with data, there are various types that we may encounter along the way. We could be working with text data, integers, decimals, dates, zip codes, and others. Luckily, Python has specific data type objects for various data types that you're probably familiar with by now. This makes it much easier to manipulate these various data types in Python. As such, before preparing to analyze and extract insights from our data, we need to make sure our variables have the correct data types, other wise we risk compromising our analysis.

## 10. Strings to integers

Let's take a look at the following example. Here's the head of a DataFrame containing revenue generated and quantity of items sold for a sales order. We want to calculate the total revenue generated by all sales orders. As you can see, the Revenue column has the dollar sign on the right hand side. A close inspection of the DataFrame column's data types using the dot-dtypes attribute returns object for the Revenue column, which is what pandas uses to store strings.

## 11. String to integers

We can also check the data types as well as the number of missing values per column in a DataFrame, by using the dot-info() method.

## 12. String to integers

Since the Revenue column is a string, summing across all sales orders returns one large concatenated string containing each row's string. To fix this, we need to first remove the $ sign from the string so that pandas is able to convert the strings into numbers without error. We do this with the dot-str-dot-strip() method, while specifying the string we want to strip as an argument, which is in this case the dollar sign. Since our dollar values do not contain decimals, we then convert the Revenue column to an integer by using the dot-astype() method, specifying the desired data type as argument. Had our revenue values been decimal, we would have converted the Revenue column to float. We can make sure that the Revenue column is now an integer by using the assert statement, which takes in a condition as input, as returns nothing if that condition is met, and an error if it is not.

## 13. The assert statement

For example, here we are testing the equality that 1+1 equals 2. Since it is the case, the assert statement returns nothing. However, when testing the equality 1+1 equals 3, we receive an assertionerror. You can test almost anything you can imagine of by using assert, and we'll see more ways to utilize it as we go along the course.

## 14. Numeric or categorical?

A common type of data seems numeric but actually represents categories with a finite set of possible categories. This is called categorical data. We will look more closely at categorical data in Chapter 2, but let's take a look at this example. Here we have a marriage status column, which is represented by 0 for never married, 1 for married, 2 for separated, and 3 for divorced. However it will be imported of type integer, which could lead to misleading results when trying to extract some statistical summaries.

## 15. Numeric or categorical?

We can solve this by using the same dot-astype() method seen earlier, but this time specifying the category data type. When applying the describe again, we see that the summary statistics are much more aligned with that of a categorical variable, discussing the number of observations, number of unique values, most frequent category instead of mean and standard deviation.

## 16. Let's practice!

Now that we have a solid understanding of data type constrains - let's get to practice!

##### Exercise

#### Common data types

Manipulating and analyzing data with incorrect data types could lead to compromised analysis as you go along the data science workflow.

When working with new data, you should always check the data types of your columns using the .dtypes attribute or the .info() method which you'll see in the next exercise. Often times, you'll run into columns that should be converted to different data types before starting any analysis.

In this exercise, you'll first identify different types of data and correctly map them to their respective types.

##### Instructions

**100XP**

* Assign each card to what type of data you think it is.

Awesome! Correctly identifying what type your data is is one of the easiest ways to avoid hampering your analysis due to data type constraints in the long run.

**Daily XP150**

##### Exercise

##### Exercise

# Numeric data or ... ?

In this exercise, and throughout this chapter, you'll be working with bicycle ride sharing data in San Francisco called ride\_sharing. It contains information on the start and end stations, the trip duration, and some user information for a bike sharing service.

The user\_type column contains information on whether a user is taking a free ride and takes on the following values:

* 1 for free riders.
* 2 for pay per ride.
* 3 for monthly subscribers.

In this instance, you will print the information of ride\_sharing using .info() and see a firsthand example of how an incorrect data type can flaw your analysis of the dataset. The pandas package is imported as pd.

##### Instructions 1/3

**35 XP**

* [1](javascript:void(0))
* [2](javascript:void(0))
* [3](javascript:void(0))
* Print the information of ride\_sharing.
* Use .describe() to print the summary statistics of the user\_type column from ride\_sharing.
* # Print the information of ride\_sharing
* print(ride\_sharing.info())
* # Print summary statistics of user\_type column
* print(ride\_sharing['user\_type'].describe())

# Print the information of ride\_sharing

print(ride\_sharing.info())

# Print summary statistics of user\_type column

print(ride\_sharing['user\_type'].describe())

<class 'pandas.core.frame.DataFrame'>

Int64Index: 25760 entries, 0 to 25759

Data columns (total 9 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 duration 25760 non-null object

1 station\_A\_id 25760 non-null int64

2 station\_A\_name 25760 non-null object

3 station\_B\_id 25760 non-null int64

4 station\_B\_name 25760 non-null object

5 bike\_id 25760 non-null int64

6 user\_type 25760 non-null int64

7 user\_birth\_year 25760 non-null int64

8 user\_gender 25760 non-null object

dtypes: int64(5), object(4)

memory usage: 2.0+ MB

None

count 25760.000

mean 2.008

std 0.705

min 1.000

25% 2.000

50% 2.000

75% 3.000

max 3.000

Name: user\_type, dtype: float64

**Daily XP220**

##### Exercise

##### Exercise

# Numeric data or ... ?

In this exercise, and throughout this chapter, you'll be working with bicycle ride sharing data in San Francisco called ride\_sharing. It contains information on the start and end stations, the trip duration, and some user information for a bike sharing service.

The user\_type column contains information on whether a user is taking a free ride and takes on the following values:

* 1 for free riders.
* 2 for pay per ride.
* 3 for monthly subscribers.

In this instance, you will print the information of ride\_sharing using .info() and see a firsthand example of how an incorrect data type can flaw your analysis of the dataset. The pandas package is imported as pd.

##### Instructions 3/3

**30 XP**

* [3](javascript:void(0))
* Convert user\_type into categorical by assigning it the 'category' data type and store it in the user\_type\_cat column.
* Make sure you converted user\_type\_cat correctly by using an assert statement.
* # Print the information of ride\_sharing
* print(ride\_sharing.info())
* # Print summary statistics of user\_type column
* print(ride\_sharing['user\_type'].describe())
* # Convert user\_type from integer to category
* ride\_sharing['user\_type\_cat'] = ride\_sharing['user\_type'].\_\_\_\_
* # Write an assert statement confirming the change
* assert ride\_sharing['user\_type\_cat'].\_\_\_\_ == '\_\_\_\_'
* # Print new summary statistics
* print(ride\_sharing['user\_type\_cat'].describe())

# Print the information of ride\_sharing

print(ride\_sharing.info())

# Print summary statistics of user\_type column

print(ride\_sharing['user\_type'].describe())

# Convert user\_type from integer to category

ride\_sharing['user\_type\_cat'] = ride\_sharing['user\_type'].astype('category')

# Write an assert statement confirming the change

assert ride\_sharing['user\_type\_cat'].dtype == 'category'

# Print new summary statistics

print(ride\_sharing['user\_type\_cat'].describe())

# Print the information of ride\_sharing

print(ride\_sharing.info())

# Print summary statistics of user\_type column

print(ride\_sharing['user\_type'].describe())

# Convert user\_type from integer to category

ride\_sharing['user\_type\_cat'] = ride\_sharing['user\_type'].astype('category')

# Write an assert statement confirming the change

assert ride\_sharing['user\_type\_cat'].dtype == 'category'

# Print new summary statistics

print(ride\_sharing['user\_type\_cat'].describe())

<class 'pandas.core.frame.DataFrame'>

Int64Index: 25760 entries, 0 to 25759

Data columns (total 9 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 duration 25760 non-null object

1 station\_A\_id 25760 non-null int64

2 station\_A\_name 25760 non-null object

3 station\_B\_id 25760 non-null int64

4 station\_B\_name 25760 non-null object

5 bike\_id 25760 non-null int64

6 user\_type 25760 non-null int64

7 user\_birth\_year 25760 non-null int64

8 user\_gender 25760 non-null object

dtypes: int64(5), object(4)

memory usage: 2.0+ MB

None

count 25760.000

mean 2.008

std 0.705

min 1.000

25% 2.000

50% 2.000

75% 3.000

max 3.000

Name: user\_type, dtype: float64

count 25760

unique 3

top 2

freq 12972

Name: user\_type\_cat, dtype: int64

Awesome work! Take a look at the new summary statistics, it seems that most users are pay per ride users!

**Daily XP250**

##### Exercise

##### Exercise

# Summing strings and concatenating numbers

In the previous exercise, you were able to identify that category is the correct data type for user\_type and convert it in order to extract relevant statistical summaries that shed light on the distribution of user\_type.

Another common data type problem is importing what should be numerical values as strings, as mathematical operations such as summing and multiplication lead to string concatenation, not numerical outputs.

In this exercise, you'll be converting the string column duration to the type int. Before that however, you will need to make sure to strip "minutes" from the column in order to make sure pandas reads it as numerical. The pandas package has been imported as pd.

##### Instructions

**100 XP**

* Use the .strip() method to strip duration of "minutes" and store it in the duration\_trim column.
* Convert duration\_trim to int and store it in the duration\_time column.
* Write an assert statement that checks if duration\_time's **d**ata **type** is now an int.
* Print the average ride duration.
* # Strip duration of minutes
* ride\_sharing['duration\_trim'] = ride\_sharing['duration'].\_\_\_\_.\_\_\_\_()
* # Convert duration to integer
* ride\_sharing['duration\_time'] = \_\_\_\_
* # Write an assert statement making sure of conversion
* assert ride\_sharing['\_\_\_\_'].\_\_\_\_ == '\_\_\_\_'
* # Print formed columns and calculate average ride duration
* print(ride\_sharing[['duration','duration\_trim','duration\_time']])
* print(\_\_\_\_)

# Strip duration of minutes

ride\_sharing['duration\_trim'] = ride\_sharing['duration'].str.strip('minutes')

#print(ride\_sharing['duration\_trim'])

# Convert duration to integer

ride\_sharing['duration\_time'] = ride\_sharing['duration\_trim'].astype('int')

# Write an assert statement making sure of conversion

assert ride\_sharing['duration\_time'].dtype == 'int'

# Print formed columns and calculate average ride duration

print(ride\_sharing[['duration','duration\_trim','duration\_time']])

print(ride\_sharing['duration\_time'].mean())

1.389052795031056

<script.py> output:

duration duration\_trim duration\_time

0 12 minutes 12 12

1 24 minutes 24 24

2 8 minutes 8 8

3 4 minutes 4 4

4 11 minutes 11 11

... ... ... ...

25755 11 minutes 11 11

25756 10 minutes 10 10

25757 14 minutes 14 14

25758 14 minutes 14 14

25759 29 minutes 29 29

[25760 rows x 3 columns]

11.389052795031056

Great work! 11 minutes is really not bad for an average ride duration in a city like San-Francisco. In the next lesson, you're going to jump right ahead into sanity checking the range of values in your data.

**Daily XP100**

# Data range constraints

**50 XP**

## 1. Data range constraints

Hi and welcome back! In this lesson, we're going to discuss data that should fall within a range.

## 2. Motivation

Let's first start off with some motivation. Imagine we have a dataset of movies with their respective average rating from a streaming service. The rating can be any integer between 1 an 5.

## 3. Motivation

After creating a histogram with maptlotlib, we see that there are a few movies with an average rating of 6, which is well above the allowable range. This is most likely an error in data collection or parsing, where a variable is well beyond its range and treating it is essential to have accurate analysis.

## 4. Motivation

Here's another example, where we see subscription dates in the future for a service. Inherently this doesn't make any sense, as we cannot sign up for a service in the future, but these errors exist either due to technical or human error. We use the datetime package's dot-date-dot-today() function to get today's date, and we filter the dataset by any subscription date higher than today's date. We need to pay attention to the range of our data.

## 5. How to deal with out of range data?

There's a variety of options to deal with out of range data. The simplest option is to drop the data. However, depending on the size of your out of range data, you could be losing out on essential information. As a rule of thumb, only drop data when a small proportion of your dataset is affected by out of range values, however you really need to understand your dataset before deciding to drop values. Another option would be setting custom minimums or maximums to your columns. We could also set the data to missing, and impute it, but we'll take a look at how to deal with missing data in Chapter 3. We could also, dependent on the business assumptions behind our data, assign a custom value for any values of our data that go beyond a certain range.

## 6. Movie example

Let's take a look at the movies example mentioned earlier. We first isolate the movies with ratings higher than 5. Now if these values are affect a small set of our data, we can drop them. We can drop them in two ways - we can either create a new filtered movies DataFrame where we only keep values of avg\_rating lower or equal than to 5. Or drop the values by using the drop method. The drop method takes in as argument the row indices of movies for which the avg\_rating is higher than 5. We set the inplace argument to True so that values are dropped in place and we don't have to create a new column. We can make sure this is set in place using an assert statement that checks if the maximum of avg\_rating is lower or equal than to 5.

## 7. Movie example

Depending on the assumptions behind our data, we can also change the out of range values to a hard limit. For example, here we're setting any value of the avg\_rating column in to 5 if it goes beyond it. We can do this using the dot-loc method, which returns all cells that fit a custom row and column index. It takes as first argument the row index, or here all instances of avg\_rating above 5 and as second argument the column index, which is here the avg\_rating column. Again, we can make sure that this change was done using an assert statement.

## 8. Date range example

Let's take another look at the date range example mentioned earlier, where we had subscriptions happening in the future. We first look at the data types of the column with the dot-dtypes attribute. We can confirm that the subscription\_date column is an object and not a date or datetime object. To compare a pandas object to a date, the first step is to convert it to another date. We do so by first converting it into a pandas datetime object with the to\_datetime function from pandas, which takes in as an argument the column we want to convert. We then need to convert the datetime object into a date. This conversion is done by appending dt-dot-date to the code. Could we have converted from an object directly to a date, without the pandas datetime conversion in the middle? Yes! But we'd have had to provide information about the date's format as a string, so it's just as easy to do it this way.

## 9. Date range example

Now that the column is a date, we can treat it in a variety of ways. We first create a today\_date variable using the datetime function date-dot-today, which allows us to store today's date. We can then either drop the rows with exceeding dates similar to how we did in the average rating example, or replace exceeding values with today's date. In both cases we can use the assert statement to verify our treatment went well, by comparing the maximum value in the subscription\_date column. However, make sure to chain it with the dot-date method to return a date instead of a timestamp.

## 10. Let's practice!

Now that you know all about ranges, let's practice!

**Daily XP150**

##### Exercise

##### Exercise

# Tire size constraints

# Convert tire\_sizes to integer

ride\_sharing['tire\_sizes'] = ride\_sharing['tire\_sizes'].astype('int')

# Set all values above 27 to 27

ride\_sharing.\_\_\_\_[\_\_\_\_ > \_\_\_\_, \_\_\_\_] = 27

# Reconvert tire\_sizes back to categorical

ride\_sharing['tire\_sizes'] = \_\_\_\_

# Print tire size description

print(ride\_sharing['tire\_sizes'].\_\_\_\_())

In this lesson, you're going to build on top of the work you've been doing with the ride\_sharing DataFrame. You'll be working with the tire\_sizes column which contains data on each bike's tire size.

Bicycle tire sizes could be either 26″, 27″ or 29″ and are here correctly stored as a categorical value. In an effort to cut maintenance costs, the ride sharing provider decided to set the maximum tire size to be 27″.

In this exercise, you will make sure the tire\_sizes column has the correct range by first converting it to an integer, then setting and testing the new upper limit of 27″ for tire sizes.

##### Instructions

**100 XP**

* Convert the tire\_sizes column from category to 'int'.
* Use .loc[] to set all values of tire\_sizes above 27 to 27.
* Reconvert back tire\_sizes to 'category' from int.
* Print the description of the tire\_sizes.
* # Convert tire\_sizes to integer
* ride\_sharing['tire\_sizes'] = ride\_sharing['tire\_sizes'].astype('int')
* # Set all values above 27 to 27
* ride\_sharing.loc[ride\_sharing['tire\_sizes'] > 27, 'tire\_sizes'] = 27
* # Reconvert tire\_sizes back to categorical
* ride\_sharing['tire\_sizes'] = ride\_sharing['tire\_sizes'].astype('category')
* # Print tire size description
* print(ride\_sharing['tire\_sizes'].describe())

# Convert tire\_sizes to integer

ride\_sharing['tire\_sizes'] = ride\_sharing['tire\_sizes'].astype('int')

# Set all values above 27 to 27

ride\_sharing.loc[ride\_sharing['tire\_sizes'] > 27, 'tire\_sizes'] = 27

# Reconvert tire\_sizes back to categorical

ride\_sharing['tire\_sizes'] = ride\_sharing['tire\_sizes'].astype('category')

# Print tire size description

print(ride\_sharing['tire\_sizes'].describe())

count 25760

unique 2

top 27

freq 13274

Name: tire\_sizes, dtype: int64

<script.py> output:

count 25760

unique 2

top 27

freq 13274

Awesome work! You can look at the new maximum by looking at the top row in the description. Notice how essential it was to convert tire\_sizes into integer before setting a new maximum.

**Daily XP250**

##### Exercise

##### Exercise

# Back to the future

A new update to the data pipeline feeding into the ride\_sharing DataFrame has been updated to register each ride's date. This information is stored in the ride\_date column of the type object, which represents strings in pandas.

A bug was discovered which was relaying rides taken today as taken next year. To fix this, you will find all instances of the ride\_date column that occur anytime in the future, and set the maximum possible value of this column to today's date. Before doing so, you would need to convert ride\_date to a datetime object.

The datetime package has been imported as dt, alongside all the packages you've been using till now.

##### Instructions

**100 XP**

* Convert ride\_date to a datetime object using to\_datetime(), then convert the datetime object into a date and store it in ride\_dt column.
* Create the variable today, which stores today's date by using the dt.date.today() function.
* For all instances of ride\_dt in the future, set them to today's date.
* Print the maximum date in the ride\_dt column.
* # Convert ride\_date to date
* ride\_sharing['ride\_dt'] = pd.\_\_\_\_(\_\_\_\_['\_\_\_\_']).dt.date
* # Save today's date
* today = \_\_\_\_
* # Set all in the future to today's date
* ride\_sharing.\_\_\_\_[\_\_\_\_['\_\_\_\_'] > \_\_\_\_, '\_\_\_\_'] = \_\_\_\_
* # Print maximum of ride\_dt column
* print(ride\_sharing['ride\_dt'].\_\_\_\_())

# Convert ride\_date to date

ride\_sharing['ride\_dt'] = pd.to\_datetime(ride\_sharing['ride\_date']).dt.date

# Save today's date

today = dt.date.today()

# Set all in the future to today's date

ride\_sharing.loc[ride\_sharing['ride\_dt'] > today, 'ride\_dt'] = today

# Print maximum of ride\_dt column

print(ride\_sharing['ride\_dt'].max())

# Convert ride\_date to date

ride\_sharing['ride\_dt'] = pd.to\_datetime(ride\_sharing['ride\_date']).dt.date

# Save today's date

today = dt.date.today()

# Set all in the future to today's date

ride\_sharing.loc[ride\_sharing['ride\_dt'] > today, 'ride\_dt'] = today

# Print maximum of ride\_dt column

print(ride\_sharing['ride\_dt'].max())

2023-02-27

<script.py> output:

2023-02-27

Great job! Imagine counting the number of rides taken today without having cleaned your ranges correctly. You would have wildly underreported your findings!

# Uniqueness constraints

**50 XP**

## 1. Uniqueness constraints

Hi and welcome to the final lesson of this chapter. Let's discuss another common data cleaning problem, duplicate values.

## 2. What are duplicate values?

Duplicate values can be diagnosed when we have the same exact information repeated across multiple rows, for a some or all columns in our DataFrame. In this example DataFrame containing the names, address, height, and weight of individuals, the rows presented have identical values across all columns.

## 3. What are duplicate values?

In this one, there are duplicate values for all columns except the height column -- which leads us to think it's more likely a data entry error than an actual other person.

## 4. Why do they happen?

Apart from data entry and human errors alluded to in the previous slide,

## 5. Why do they happen?

duplicate data can also arise because of bugs and design errors whether in business processes or data pipelines.

## 6. Why do they happen?

However they oftenmost arise from the necessary act of joining and consolidating data from various resources, which could retain duplicate values.

## 7. How to find duplicate values?

Let's first see how to find duplicate values. In this example, we're working with a bigger version of the the height and weight data seen earlier in the video.

## 8. How to find duplicate values?

We can find duplicates in a DataFrame by using the dot-duplicated() method. It returns a Series of boolean values that are True for duplicate values, and False for non-duplicated values.

## 9. How to find duplicate values?

We can see exactly which rows are affected by using brackets as such. However, using dot-duplicated() without playing around with the arguments of the method can lead to misleading results, as all the columns are required to have duplicate values by default, with all duplicate values being marked as True except for the first occurrence. This limits our ability to properly diagnose what type of duplication we have, and how to effectively treat it.

## 10. How to find duplicate rows?

To properly calibrate how we go about finding duplicates, we will use 2 arguments from the dot-duplicated() method. The subset argument lets us set a list of column names to check for duplication. For example, it allows us to find duplicates for the first and last name columns only. The keep argument lets us keep the first occurrence of a duplicate value by setting it to the string first, the last occurrence of a duplicate value by setting it the string last, or keep all occurrences of duplicate values by setting it to False. In this example, we're checking for duplicates across the first name, last name, and address variables, and we're choosing to keep all duplicates.

## 11. How to find duplicate rows?

We see the following results -- to get a better bird's eye view of the duplicates,

## 12. How to find duplicate rows?

We sort the duplicate rows using the dot-sort\_values method, choosing first\_name to sort by.

## 13. How to find duplicate rows?

We find that there are four sets of duplicated rows, the first 2 being complete duplicates of each other across all columns, highlighted here in red.

## 14. How to find duplicate rows?

The other 2 being incomplete duplicates of each other highlighted here in blue with discrepancies across height and weight respectively.

## 15. How to treat duplicate values?

The complete duplicates can be treated easily. All that is required is to keep one of them only and discard the others.

## 16. How to treat duplicate values?

This can be done with the dot-drop\_duplicates() method, which also takes in the same subset and keep arguments as in the dot-duplicated() method, as well as the inplace argument which drops the duplicated values directly inside the height\_weight DataFrame. Here we are dropping complete duplicates only, so it's not necessary nor advisable to set a subset, and since the keep argument takes in first as default, we can keep it as such. Note that we can also set it as last, but not as False as it would keep all duplicates.

## 17. How to treat duplicate values?

This leaves us with the other 2 sets of duplicates discussed earlier, which are the same for first\_name, last\_name and address, but contain discrepancies in height and weight. Apart from dropping rows with really small discrepancies, we can use a statistical measure to combine each set of duplicated values.

## 18. How to treat duplicate values?

For example, we can combine these two rows into one by computing the average mean between them, or the maximum, or other statistical measures, this is highly dependent on a common sense understanding of our data, and what type of data we have.

## 19. How to treat duplicate values?

We can do this easily using the groupby method, which when chained with the agg method, lets you group by a set of common columns and return statistical values for specific columns when the aggregation is being performed. For example here, we created a dictionary called summaries, which instructs groupby to return the maximum of duplicated rows for the height column, and the mean duplicated rows for the weight column. We then group height\_weight by the column names defined earlier, and chained it with the agg method, which takes in the summaries dictionary we created. We chain this entire line with the dot-reset\_index() method, so that we can have numbered indices in the final output. We can verify that there are no more duplicate values by running the duplicated method again, and use brackets to output duplicate rows.

## 20. Let's practice!

Now that we have a solid grasp of duplication, let's practice.

Correct! Subsetting on metadata and keeping all duplicate records gives you a better bird-eye's view over your data and how to duplicate it! You can even subset the loans DataFrame using bracketing and sort the values so you can properly identify the duplicates.

**Daily XP100**

# How big is your subset?

You have the following loans DataFrame which contains loan and credit score data for consumers, and some metadata such as their first and last names. You want to find both complete and incomplete duplicates using .duplicated().

| **first\_name** | **last\_name** | **credit\_score** | **has\_loan** |
| --- | --- | --- | --- |
| Justin | Saddlemeyer | 600 | 1 |
| Hadrien | Lacroix | 450 | 0 |

Choose the **correct** usage of .duplicated() below:

##### Answer the question

**50XP**

#### Possible Answers

* 

loans.duplicated()   
  Because the default method returns both complete and incomplete duplicates.

press1

* 

loans.duplicated(subset = 'first\_name')   
  Because constraining the duplicate rows to the first name lets me find incomplete duplicates as well.

press2

* ****

**loans.duplicated(subset = ['first\_name', 'last\_name'], keep = False)   
  Because subsetting on consumer metadata and not discarding any duplicate returns all duplicated rows.**

press3

* 

loans.duplicated(subset = ['first\_name', 'last\_name'], keep = 'first')   
  Because this drops all duplicates.

press4

**Daily XP100**

##### Exercise

##### Exercise

# Finding duplicates

A new update to the data pipeline feeding into ride\_sharing has added the ride\_id column, which represents a unique identifier for each ride.

The update however coincided with radically shorter average ride duration times and irregular user birth dates set in the future. Most importantly, the number of rides taken has increased by 20% overnight, leading you to think there might be both complete and incomplete duplicates in the ride\_sharing DataFrame.

In this exercise, you will confirm this suspicion by finding those duplicates. A sample of ride\_sharing is in your environment, as well as all the packages you've been working with thus far.

##### Instructions

**100 XP**

* Find duplicated rows of ride\_id in the ride\_sharing DataFrame while setting keep to False.
* Subset ride\_sharing on duplicates and sort by ride\_id and assign the results to duplicated\_rides.
* Print the ride\_id, duration and user\_birth\_year columns of duplicated\_rides in that order.
* # Find duplicates
* duplicates = \_\_\_\_.\_\_\_\_(\_\_\_\_, \_\_\_\_)
* # Sort your duplicated rides
* duplicated\_rides = ride\_sharing[\_\_\_\_].\_\_\_\_('\_\_\_\_')
* # Print relevant columns of duplicated\_rides
* print(duplicated\_rides[['\_\_\_\_','\_\_\_\_','\_\_\_\_']])

# Find duplicates

duplicates = ride\_sharing.duplicated(subset=['ride\_id'],keep=False)

# Sort your duplicated rides

duplicated\_rides = ride\_sharing[duplicates].sort\_values(by='ride\_id')

# Print relevant columns of duplicated\_rides

print(duplicated\_rides[['ride\_id','duration','user\_birth\_year']])

# Find duplicates

duplicates = ride\_sharing.duplicated(subset=['ride\_id'],keep=False)

# Sort your duplicated rides

duplicated\_rides = ride\_sharing[duplicates].sort\_values(by='ride\_id')

# Print relevant columns of duplicated\_rides

print(duplicated\_rides[['ride\_id','duration','user\_birth\_year']])

ride\_id duration user\_birth\_year

22 33 10 1979

39 33 2 1979

53 55 9 1985

65 55 9 1985

74 71 11 1997

75 71 11 1997

76 89 9 1986

77 89 9 2060

<script.py> output:

ride\_id duration user\_birth\_year

22 33 10 1979

39 33 2 1979

53 55 9 1985

65 55 9 1985

74 71 11 1997

75 71 11 1997

76 89 9 1986

77 89 9 2060

Great job! Notice that rides 33 and 89 are incomplete duplicates, whereas the remaining are complete. You'll learn how to treat them in the next exer

**Daily XP200**

##### Exercise

##### Exercise

# Treating duplicates

In the last exercise, you were able to verify that the new update feeding into ride\_sharing contains a bug generating both complete and incomplete duplicated rows for some values of the ride\_id column, with occasional discrepant values for the user\_birth\_year and duration columns.

In this exercise, you will be treating those duplicated rows by first dropping complete duplicates, and then merging the incomplete duplicate rows into one while keeping the average duration, and the minimum user\_birth\_year for each set of incomplete duplicate rows.

##### Instructions

**100 XP**

* Drop complete duplicates in ride\_sharing and store the results in ride\_dup.
* Create the statistics dictionary which holds **min**imum aggregation for user\_birth\_year and **mean** aggregation for duration.
* Drop incomplete duplicates by grouping by ride\_id and applying the aggregation in statistics.
* Find duplicates again and run the assert statement to verify de-duplication.
* # Drop complete duplicates from ride\_sharing
* ride\_dup = \_\_\_\_.\_\_\_\_()
* # Create statistics dictionary for aggregation function
* statistics = {'user\_birth\_year': \_\_\_\_, 'duration': \_\_\_\_}
* # Group by ride\_id and compute new statistics
* ride\_unique = ride\_dup.\_\_\_\_('\_\_\_\_').\_\_\_\_(\_\_\_\_).reset\_index()
* # Find duplicated values again
* duplicates = ride\_unique.\_\_\_\_(subset = 'ride\_id', keep = False)
* duplicated\_rides = ride\_unique[duplicates == True]
* # Assert duplicates are processed
* assert duplicated\_rides.shape[0] == 0

# Drop complete duplicates from ride\_sharing

ride\_dup = ride\_sharing.drop\_duplicates()

# Create statistics dictionary for aggregation function

statistics = {'user\_birth\_year': 'min', 'duration': 'mean'}

# Group by ride\_id and compute new statistics

ride\_unique = ride\_dup.groupby('ride\_id').agg(statistics).reset\_index()

# Find duplicated values again

duplicates = ride\_unique.duplicated(subset = 'ride\_id', keep = False)

duplicated\_rides = ride\_unique[duplicates == True]

# Assert duplicates are processed

assert duplicated\_rides.shape[0] == 0

# Drop complete duplicates from ride\_sharing ride\_dup = ride\_sharing.drop\_duplicates() # Create statistics dictionary for aggregation function statistics = {'user\_birth\_year': 'min', 'duration': 'mean'} # Group by ride\_id and compute new statistics ride\_unique = ride\_dup.groupby('ride\_id').agg(statistics).reset\_index() # Find duplicated values again duplicates = ride\_unique.duplicated(subset = 'ride\_id', keep = False) duplicated\_rides = ride\_unique[duplicates == True] # Assert duplicates are processed assert duplicated\_rides.shape[0] == 0

Awesome work! You can bet after this fix that ride sharing KPIs will come back to normal.

**Daily XP300**

# Membership constraints

**50 XP**

## 1. Membership constraints

Fantastic work on Chapter 1! You're now equipped to treat more complex, and specific data cleaning problems.

## 2. In this chapter

In this chapter, we're going to take a look at common data problems with text and categorical data, so let's get started.

## 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.

## 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.

## 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.

## 6. An example

Let's first look at an example. Here's a DataFrame named study\_data containing a list of first names, birth dates, and blood types. Additionally, a DataFrame named categories, containing the correct possible categories for the blood type column has been created as well.

## 7. An example

Notice the inconsistency here? There's definitely no blood type named Z+. Luckily, the categories DataFrame will help us systematically spot all rows with these inconsistencies. It's always good practice to keep a log of all possible values of your categorical data, as it will make dealing with these types of inconsistencies way easier.

## 8. A note on joins

Now before moving on to dealing with these inconsistent values, let's have a brief reminder on joins. The two main types of joins we care about here are anti joins and inner joins. We join DataFrames on common columns between them. Anti joins, take in two DataFrames A and B, and return data from one DataFrame that is not contained in another. In this example, we are performing a left anti join of A and B, and are returning the columns of DataFrames 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 DataFrames. For example, an inner join of A and B, would return columns from both DataFrames for values only found in A and B, of the common column between them being joined on.

## 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,

## 10. An inner join on blood types

and an inner join returns all the rows containing consistent blood types signs.

## 11. Finding inconsistent categories

Now let's see how to do that in Python. 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.

## 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.

## 13. Let's practice!

Now that we know about treating categorical data, let's practice!

**Daily XP350**

##### Exercise

#### Members only

Throughout the course so far, you've been exposed to some common problems that you may encounter with your data, from data type constraints, data range constrains, uniqueness constraints, and now membership constraints for categorical values.

In this exercise, you will map hypothetical problems to their respective categories.

##### Instructions

**100XP**

* Map the data problem observed with the correct type of data problem

Tremendous work. You're becoming an elite member of categorical variable experts!

**Daily XP450**

##### Exercise

##### Exercise

# Finding consistency

In this exercise and throughout this chapter, you'll be working with the airlines DataFrame which contains survey responses on the San Francisco Airport from airline customers.

The DataFrame contains flight metadata such as the airline, the destination, waiting times as well as answers to key questions regarding cleanliness, safety, and satisfaction. Another DataFrame named categories was created, containing all correct possible values for the survey columns.

In this exercise, you will use both of these DataFrames to find survey answers with inconsistent values, and drop them, effectively performing an outer and inner join on both these DataFrames as seen in the video exercise. The pandas package has been imported as pd, and the airlines and categories DataFrames are in your environment.

##### Instructions 1/4

**35 XP**

* [1](javascript:void(0))
* [2](javascript:void(0))
* [3](javascript:void(0))
* [4](javascript:void(0))
* Print the categories DataFrame and take a close look at all possible correct categories of the survey columns.
* Print the unique values of the survey columns in airlines using the .unique() method.
* # Print categories DataFrame
* print(\_\_\_\_)
* # Print unique values of survey columns in airlines
* print('Cleanliness: ', airlines['cleanliness'].\_\_\_\_, "\n")
* print('Safety: ', \_\_\_\_, "\n")
* print('Satisfaction: ', \_\_\_\_, "\n")

# Print categories DataFrame

print(categories)

# Print unique values of survey columns in airlines

print('Cleanliness: ', airlines['cleanliness'].unique(), "\n")

print('Safety: ', airlines['safety'].unique(), "\n")

print('Satisfaction: ', airlines['satisfaction'].unique(), "\n")

cleanliness safety satisfaction

0 Clean Neutral Very satisfied

1 Average Very safe Neutral

2 Somewhat clean Somewhat safe Somewhat satisfied

3 Somewhat dirty Very unsafe Somewhat unsatisfied

4 Dirty Somewhat unsafe Very unsatisfied

Cleanliness: ['Clean', 'Average', 'Unacceptable', 'Somewhat clean', 'Somewhat dirty', 'Dirty']

Categories (6, object): ['Average', 'Clean', 'Dirty', 'Somewhat clean', 'Somewhat dirty', 'Unacceptable']

Safety: ['Neutral', 'Very safe', 'Somewhat safe', 'Very unsafe', 'Somewhat unsafe']

Categories (5, object): ['Neutral', 'Somewhat safe', 'Somewhat unsafe', 'Very safe', 'Very unsafe']

Satisfaction: ['Very satisfied', 'Neutral', 'Somewhat satisfied', 'Somewhat unsatisfied', 'Very unsatisfied']

Categories (5, object): ['Neutral', 'Somewhat satisfied', 'Somewhat unsatisfied', 'Very satisfied', 'Very unsatisfied']

<script.py> output:

cleanliness safety satisfaction

0 Clean Neutral Very satisfied

1 Average Very safe Neutral

2 Somewhat clean Somewhat safe Somewhat satisfied

3 Somewhat dirty Very unsafe Somewhat unsatisfied

4 Dirty Somewhat unsafe Very unsatisfied

Cleanliness: ['Clean', 'Average', 'Unacceptable', 'Somewhat clean', 'Somewhat dirty', 'Dirty']

Categories (6, object): ['Average', 'Clean', 'Dirty', 'Somewhat clean', 'Somewhat dirty', 'Unacceptable']

Safety: ['Neutral', 'Very safe', 'Somewhat safe', 'Very unsafe', 'Somewhat unsafe']

Categories (5, object): ['Neutral', 'Somewhat safe', 'Somewhat unsafe', 'Very safe', 'Very unsafe']

Satisfaction: ['Very satisfied', 'Neutral', 'Somewhat satisfied', 'Somewhat unsatisfied', 'Very unsatisfied']

Categories (5, object): ['Neutral', 'Somewhat satisfied', 'Somewhat unsatisfied', 'Very satisfied', 'Very unsatisfied']

#### Question

Take a look at the output. Out of the cleanliness, safety and satisfaction columns, which one has an inconsistent category and what is it?

##### Possible Answers

* 

**cleanliness because it has an Unacceptable category**.

* 

cleanliness because it has a Terribly dirty category.

* 

satisfaction because it has a Very satisfied category.

* 

safety because it has a Neutral category.

##### Instructions 3/4

**25 XP**

* [3](javascript:void(0))
* [4](javascript:void(0))
* Create a set out of the cleanliness column in airlines using set() and find the inconsistent category by finding the **difference** in the cleanliness column of categories.
* Find rows of airlines with a cleanliness value not in categories and print the output.

# Print categories DataFrame

print(categories)

# Print unique values of survey columns in airlines

print('Cleanliness: ', airlines['cleanliness'].unique(), "\n")

print('Safety: ', airlines['safety'].unique(), "\n")

print('Satisfaction: ', airlines['satisfaction'].unique(), "\n")

cleanliness safety satisfaction

0 Clean Neutral Very satisfied

1 Average Very safe Neutral

2 Somewhat clean Somewhat safe Somewhat satisfied

3 Somewhat dirty Very unsafe Somewhat unsatisfied

4 Dirty Somewhat unsafe Very unsatisfied

Cleanliness: ['Clean', 'Average', 'Unacceptable', 'Somewhat clean', 'Somewhat dirty', 'Dirty']

Categories (6, object): ['Average', 'Clean', 'Dirty', 'Somewhat clean', 'Somewhat dirty', 'Unacceptable']

Safety: ['Neutral', 'Very safe', 'Somewhat safe', 'Very unsafe', 'Somewhat unsafe']

Categories (5, object): ['Neutral', 'Somewhat safe', 'Somewhat unsafe', 'Very safe', 'Very unsafe']

Satisfaction: ['Very satisfied', 'Neutral', 'Somewhat satisfied', 'Somewhat unsatisfied', 'Very unsatisfied']

Categories (5, object): ['Neutral', 'Somewhat satisfied', 'Somewhat unsatisfied', 'Very satisfied', 'Very unsatisfied']

<script.py> output:

cleanliness safety satisfaction

0 Clean Neutral Very satisfied

1 Average Very safe Neutral

2 Somewhat clean Somewhat safe Somewhat satisfied

3 Somewhat dirty Very unsafe Somewhat unsatisfied

4 Dirty Somewhat unsafe Very unsatisfied

Cleanliness: ['Clean', 'Average', 'Unacceptable', 'Somewhat clean', 'Somewhat dirty', 'Dirty']

Categories (6, object): ['Average', 'Clean', 'Dirty', 'Somewhat clean', 'Somewhat dirty', 'Unacceptable']

Safety: ['Neutral', 'Very safe', 'Somewhat safe', 'Very unsafe', 'Somewhat unsafe']

Categories (5, object): ['Neutral', 'Somewhat safe', 'Somewhat unsafe', 'Very safe', 'Very unsafe']

Satisfaction: ['Very satisfied', 'Neutral', 'Somewhat satisfied', 'Somewhat unsatisfied', 'Very unsatisfied']

Categories (5, object): ['Neutral', 'Somewhat satisfied', 'Somewhat unsatisfied', 'Very satisfied', 'Very unsatisfied']

print(categories)

cleanliness safety satisfaction

0 Clean Neutral Very satisfied

1 Average Very safe Neutral

2 Somewhat clean Somewhat safe Somewhat satisfied

3 Somewhat dirty Very unsafe Somewhat unsatisfied

4 Dirty Somewhat unsafe Very unsatisfied

# Find the cleanliness category in airlines not in categories

cat\_clean = set(airlines['cleanliness']).difference(categories['cleanliness'])

# Find rows with that category

cat\_clean\_rows = airlines['cleanliness'].isin(cat\_clean)

#print(cat\_clean\_rows)

# Print rows with inconsistent category

print(airlines[cat\_clean\_rows])

id day airline destination dest\_region dest\_size boarding\_area dept\_time wait\_min cleanliness safety satisfaction

4 2992 Wednesday AMERICAN MIAMI East US Hub Gates 50-59 2018-12-31 559.0 Unacceptable Very safe Somewhat satisfied

18 2913 Friday TURKISH AIRLINES ISTANBUL Middle East Hub Gates 91-102 2018-12-31 225.0 Unacceptable Very safe Somewhat satisfied

100 2321 Wednesday SOUTHWEST LOS ANGELES West US Hub Gates 20-39 2018-12-31 130.0 Unacceptable Somewhat safe Somewhat satisfied

<script.py> output:

id day airline destination dest\_region dest\_size boarding\_area dept\_time wait\_min cleanliness safety satisfaction

4 2992 Wednesday AMERICAN MIAMI East US Hub Gates 50-59 2018-12-31 559.0 Unacceptable Very safe Somewhat satisfied

18 2913 Friday TURKISH AIRLINES ISTANBUL Middle East Hub Gates 91-102 2018-12-31 225.0 Unacceptable Very safe Somewhat satisfied

100 2321 Wednesday SOUTHWEST LOS ANGELES West US Hub Gates 20-39 2018-12-31 130.0 Unacceptable Somewhat safe Somewhat satisfied

# Find the cleanliness category in airlines not in categories

cat\_clean = set(airlines['cleanliness']).difference(categories['cleanliness'])

# Find rows with that category

cat\_clean\_rows = airlines['cleanliness'].isnot(categories['cleanliness'])

# Print rows with inconsistent category

print(airlines['cleanliness'])

Print the rows with the consistent categories of cleanliness only.

# Find the cleanliness category in airlines not in categories

cat\_clean = set(airlines['cleanliness']).difference(categories['cleanliness'])

# Find rows with that category

cat\_clean\_rows = airlines['cleanliness'].isin(cat\_clean)

# Print rows with inconsistent category

print(airlines[cat\_clean\_rows])

# Print rows with consistent categories only

print(airlines[\_\_\_\_])

# Find the cleanliness category in airlines not in categories

cat\_clean = set(airlines['cleanliness']).difference(categories['cleanliness'])

# Find rows with that category

cat\_clean\_rows = airlines['cleanliness'].isin(cat\_clean)

# Print rows with inconsistent category

print(airlines[cat\_clean\_rows])

# Print rows with consistent categories only

print(airlines[~cat\_clean\_rows])

# Find the cleanliness category in airlines not in categories

cat\_clean = set(airlines['cleanliness']).difference(categories['cleanliness'])

# Find rows with that category

cat\_clean\_rows = airlines['cleanliness'].isin(cat\_clean)

# Print rows with inconsistent category

print(airlines[cat\_clean\_rows])

# Print rows with consistent categories only

print(airlines[~cat\_clean\_rows])

id day airline destination dest\_region dest\_size boarding\_area dept\_time wait\_min cleanliness safety satisfaction

4 2992 Wednesday AMERICAN MIAMI East US Hub Gates 50-59 2018-12-31 559.0 Unacceptable Very safe Somewhat satisfied

18 2913 Friday TURKISH AIRLINES ISTANBUL Middle East Hub Gates 91-102 2018-12-31 225.0 Unacceptable Very safe Somewhat satisfied

100 2321 Wednesday SOUTHWEST LOS ANGELES West US Hub Gates 20-39 2018-12-31 130.0 Unacceptable Somewhat safe Somewhat satisfied

id day airline destination dest\_region dest\_size boarding\_area dept\_time wait\_min cleanliness safety satisfaction

0 1351 Tuesday UNITED INTL KANSAI Asia Hub Gates 91-102 2018-12-31 115.0 Clean Neutral Very satisfied

1 373 Friday ALASKA SAN JOSE DEL CABO Canada/Mexico Small Gates 50-59 2018-12-31 135.0 Clean Very safe Very satisfied

2 2820 Thursday DELTA LOS ANGELES West US Hub Gates 40-48 2018-12-31 70.0 Average Somewhat safe Neutral

3 1157 Tuesday SOUTHWEST LOS ANGELES West US Hub Gates 20-39 2018-12-31 190.0 Clean Very safe Somewhat satisfied

5 634 Thursday ALASKA NEWARK East US Hub Gates 50-59 2018-12-31 140.0 Somewhat clean Very safe Very satisfied

... ... ... ... ... ... ... ... ... ... ... ... ...

2804 1475 Tuesday ALASKA NEW YORK-JFK East US Hub Gates 50-59 2018-12-31 280.0 Somewhat clean Neutral Somewhat satisfied

2805 2222 Thursday SOUTHWEST PHOENIX West US Hub Gates 20-39 2018-12-31 165.0 Clean Very safe Very satisfied

2806 2684 Friday UNITED ORLANDO East US Hub Gates 70-90 2018-12-31 92.0 Clean Very safe Very satisfied

2807 2549 Tuesday JETBLUE LONG BEACH West US Small Gates 1-12 2018-12-31 95.0 Clean Somewhat safe Very satisfied

2808 2162 Saturday CHINA EASTERN QINGDAO Asia Large Gates 1-12 2018-12-31 220.0 Clean Very safe Somewhat satisfied

[2474 rows x 12 columns]

<script.py> output:

id day airline destination dest\_region dest\_size boarding\_area dept\_time wait\_min cleanliness safety satisfaction

4 2992 Wednesday AMERICAN MIAMI East US Hub Gates 50-59 2018-12-31 559.0 Unacceptable Very safe Somewhat satisfied

18 2913 Friday TURKISH AIRLINES ISTANBUL Middle East Hub Gates 91-102 2018-12-31 225.0 Unacceptable Very safe Somewhat satisfied

100 2321 Wednesday SOUTHWEST LOS ANGELES West US Hub Gates 20-39 2018-12-31 130.0 Unacceptable Somewhat safe Somewhat satisfied

id day airline destination dest\_region dest\_size boarding\_area dept\_time wait\_min cleanliness safety satisfaction

0 1351 Tuesday UNITED INTL KANSAI Asia Hub Gates 91-102 2018-12-31 115.0 Clean Neutral Very satisfied

1 373 Friday ALASKA SAN JOSE DEL CABO Canada/Mexico Small Gates 50-59 2018-12-31 135.0 Clean Very safe Very satisfied

2 2820 Thursday DELTA LOS ANGELES West US Hub Gates 40-48 2018-12-31 70.0 Average Somewhat safe Neutral

3 1157 Tuesday SOUTHWEST LOS ANGELES West US Hub Gates 20-39 2018-12-31 190.0 Clean Very safe Somewhat satisfied

5 634 Thursday ALASKA NEWARK East US Hub Gates 50-59 2018-12-31 140.0 Somewhat clean Very safe Very satisfied

... ... ... ... ... ... ... ... ... ... ... ... ...

2804 1475 Tuesday ALASKA NEW YORK-JFK East US Hub Gates 50-59 2018-12-31 280.0 Somewhat clean Neutral Somewhat satisfied

2805 2222 Thursday SOUTHWEST PHOENIX West US Hub Gates 20-39 2018-12-31 165.0 Clean Very safe Very satisfied

2806 2684 Friday UNITED ORLANDO East US Hub Gates 70-90 2018-12-31 92.0 Clean Very safe Very satisfied

2807 2549 Tuesday JETBLUE LONG BEACH West US Small Gates 1-12 2018-12-31 95.0 Clean Somewhat safe Very satisfied

2808 2162 Saturday CHINA EASTERN QINGDAO Asia Large Gates 1-12 2018-12-31 220.0 Clean Very safe Somewhat satisfied

Great \_consistent\_ work! Keep it up! In the next lesson, we'll be looking at more in depth solutions to dealing with dirty categorical data.

# Categorical variables

**50 XP**

## 1. Categorical variables

Awesome work on the last lesson. Now let's discuss other types of problems that could affect categorical variables.

## 2. What type of errors could we have?

In the last lesson, we saw how categorical data has a value membership constraint, where columns need to have a predefined set of values. However, this is not the only set of problems we may encounter. When cleaning categorical data, some of the problems we may encounter include value inconsistency, the presence of too many categories that could be collapsed into one, and making sure data is of the right type.

## 3. Value consistency

Let's start with making sure our categorical data is consistent. A common categorical data problem is having values that slightly differ because of capitalization. Not treating this could lead to misleading results when we decide to analyze our data, for example, let's assume we're working with a demographics dataset, and we have a marriage status column with inconsistent capitalization. Here's what counting the number of married people in the marriage\_status Series would look like. Note that the dot-value\_counts() methods works on Series only.

## 4. Value consistency

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

## 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.

## 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.

## 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.

## 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.

## 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.

## 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.

## 11. Let's practice!

Now that we know about treating categorical data, let's practice!

**Daily XP600**

##### Exercise

#### Categories of errors

In the video exercise, you saw how to address common problems affecting categorical variables in your data, including white spaces and inconsistencies in your categories, and the problem of creating new categories and mapping existing ones to new ones.

To get a better idea of the toolkit at your disposal, you will be mapping functions and methods from pandas and Python used to address each type of problem.

##### Instructions

**100XP**

* Map each function/method to the categorical data problem it solves.

**Daily XP700**

##### Exercise

##### Exercise

# Inconsistent categories

In this exercise, you'll be revisiting the airlines DataFrame from the previous lesson.

As a reminder, the DataFrame contains flight metadata such as the airline, the destination, waiting times as well as answers to key questions regarding cleanliness, safety, and satisfaction on the San Francisco Airport.

In this exercise, you will examine two categorical columns from this DataFrame, dest\_region and dest\_size respectively, assess how to address them and make sure that they are cleaned and ready for analysis. The pandas package has been imported as pd, and the airlines DataFrame is in your environment.

##### Instructions 1/4

**25 XP**

* [1](javascript:void(0))
* [2](javascript:void(0))
* [3](javascript:void(0))
* [4](javascript:void(0))
* Print the unique values in dest\_region and dest\_size respectively.
* # Print unique values of both columns
* print(airlines['dest\_region'].\_\_\_\_())
* print(airlines['\_\_\_\_'].\_\_\_\_())

# Print unique values of both columns

print(airlines['dest\_region'].unique())

print(airlines['dest\_size'].unique())

# Print unique values of both columns

print(airlines['dest\_region'].unique())

print(airlines['dest\_size'].unique())

['Asia' 'Canada/Mexico' 'West US' 'East US' 'Midwest US' 'EAST US'

'Middle East' 'Europe' 'eur' 'Central/South America'

'Australia/New Zealand' 'middle east']

['Hub' 'Small' ' Hub' 'Medium' 'Large' 'Hub ' ' Small'

'Medium ' ' Medium' 'Small ' ' Large' 'Large ']

#### Question

From looking at the output, what do you think is the problem with these columns?

##### Possible Answers

* 

The dest\_region column has only inconsistent values due to capitalization.

* 

The dest\_region column has inconsistent values due to capitalization and has one value that needs to be remapped.

* 

The dest\_size column has only inconsistent values due to leading and trailing spaces.

* ****

**Both 2 and 3 are correct**

# Print unique values of both columns

print(airlines['dest\_region'].unique())

print(airlines['dest\_size'].unique())

['Asia' 'Canada/Mexico' 'West US' 'East US' 'Midwest US' 'EAST US'

'Middle East' 'Europe' 'eur' 'Central/South America'

'Australia/New Zealand' 'middle east']

['Hub' 'Small' ' Hub' 'Medium' 'Large' 'Hub ' ' Small'

'Medium ' ' Medium' 'Small ' ' Large' 'Large ']

<script.py> output:

['Asia' 'Canada/Mexico' 'West US' 'East US' 'Midwest US' 'EAST US'

'Middle East' 'Europe' 'eur' 'Central/South America'

'Australia/New Zealand' 'middle east']

['Hub' 'Small' ' Hub' 'Medium' 'Large' 'Hub ' ' Small'

'Medium ' ' Medium' 'Small ' ' Large' 'Large ']

* Change the capitalization of all values of dest\_region to lowercase.
* Replace the 'eur' with 'europe' in dest\_region using the .replace() method.

# Print unique values of both columns

print(airlines['dest\_region'].unique())

print(airlines['dest\_size'].unique())

# Lower dest\_region column and then replace "eur" with "europe"

airlines['dest\_region'] = airlines['dest\_region'].str.lower()

airlines['dest\_region'] = airlines['dest\_region'].replace({'eur':'europe'})

#print(airlines['dest\_region'])

# Print unique values of both columns

print(airlines['dest\_region'].unique())

print(airlines['dest\_size'].unique())

# Lower dest\_region column and then replace "eur" with "europe"

airlines['dest\_region'] = airlines['dest\_region'].str.lower()

airlines['dest\_region'] = airlines['dest\_region'].replace({'eur':'europe'})

print(airlines['dest\_region'])

['asia' 'canada/mexico' 'west us' 'east us' 'midwest us' 'middle east'

'europe' 'central/south america' 'australia/new zealand']

['Hub' 'Small' ' Hub' 'Medium' 'Large' 'Hub ' ' Small'

'Medium ' ' Medium' 'Small ' ' Large' 'Large ']

0 asia

1 canada/mexico

2 west us

3 west us

4 east us

...

2804 east us

2805 west us

2806 east us

2807 west us

2808 asia

Name: dest\_region, Length: 2477, dtype: object

* Strip white spaces from the dest\_size column using the .strip() method.
* Verify that the changes have been into effect by printing the unique values of the columns using .unique() .

# Print unique values of both columns

print(airlines['dest\_region'].unique())

print(airlines['dest\_size'].unique())

# Lower dest\_region column and then replace "eur" with "europe"

airlines['dest\_region'] = airlines['dest\_region'].str.lower()

airlines['dest\_region'] = airlines['dest\_region'].replace({'eur':'europe'})

# Remove white spaces from `dest\_size`

airlines['dest\_size'] = airlines['dest\_size'].\_\_\_\_.\_\_\_\_()

# Verify changes have been effected

print(\_\_\_\_)

print(\_\_\_\_)

# Print unique values of both columns

print(airlines['dest\_region'].unique())

print(airlines['dest\_size'].unique())

# Lower dest\_region column and then replace "eur" with "europe"

airlines['dest\_region'] = airlines['dest\_region'].str.lower()

airlines['dest\_region'] = airlines['dest\_region'].replace({'eur':'europe'})

# Remove white spaces from `dest\_size`

airlines['dest\_size'] = airlines['dest\_size'].str.strip()

# Verify changes have been effected

print(airlines['dest\_region'].unique())

print(airlines['dest\_size'].unique())

# Print unique values of both columns

print(airlines['dest\_region'].unique())

print(airlines['dest\_size'].unique())

# Lower dest\_region column and then replace "eur" with "europe"

airlines['dest\_region'] = airlines['dest\_region'].str.lower()

airlines['dest\_region'] = airlines['dest\_region'].replace({'eur':'europe'})

# Remove white spaces from `dest\_size`

airlines['dest\_size'] = airlines['dest\_size'].str.strip()

# Verify changes have been effected

print(airlines['dest\_region'].unique())

print(airlines['dest\_size'].unique())

['Asia' 'Canada/Mexico' 'West US' 'East US' 'Midwest US' 'EAST US'

'Middle East' 'Europe' 'eur' 'Central/South America'

'Australia/New Zealand' 'middle east']

['Hub' 'Small' ' Hub' 'Medium' 'Large' 'Hub ' ' Small'

'Medium ' ' Medium' 'Small ' ' Large' 'Large ']

['asia' 'canada/mexico' 'west us' 'east us' 'midwest us' 'middle east'

'europe' 'central/south america' 'australia/new zealand']

['Hub' 'Small' 'Medium' 'Large']

Great work! Notice how all categories have been properly treated?

**Daily XP800**

##### Exercise

##### Exercise

# Remapping categories

To better understand survey respondents from airlines, you want to find out if there is a relationship between certain responses and the day of the week and wait time at the gate.

The airlines DataFrame contains the day and wait\_min columns, which are categorical and numerical respectively. The day column contains the exact day a flight took place, and wait\_min contains the amount of minutes it took travelers to wait at the gate. To make your analysis easier, you want to create two new categorical variables:

* wait\_type: 'short' for 0-60 min, 'medium' for 60-180 and long for 180+
* day\_week: 'weekday' if day is in the weekday, 'weekend' if day is in the weekend.

The pandas and numpy packages have been imported as pd and np. Let's create some new categorical data!

##### Instructions

**100 XP**

* Create the ranges and labels for the wait\_type column mentioned in the description.
* Create the wait\_type column by from wait\_min by using pd.cut(), while inputting label\_ranges and label\_names in the correct arguments.
* Create the mapping dictionary mapping weekdays to 'weekday' and weekend days to 'weekend'.
* Create the day\_week column by using .replace()

# Create ranges for categories

label\_ranges = [0, 60, \_\_\_\_, np.inf]

label\_names = ['short', \_\_\_\_, \_\_\_\_]

# Create wait\_type column

airlines['wait\_type'] = pd.\_\_\_\_(\_\_\_\_, bins = \_\_\_\_,

                                labels = \_\_\_\_)

# Create mappings and replace

mappings = {'Monday':'weekday', 'Tuesday':'\_\_\_\_', 'Wednesday': '\_\_\_\_',

            'Thursday': '\_\_\_\_', '\_\_\_\_': '\_\_\_\_',

            'Saturday': 'weekend', '\_\_\_\_': '\_\_\_\_'}

airlines['day\_week'] = airlines['day'].\_\_\_\_(mappings)

# Create ranges for categories

label\_ranges = [0, 60, 180, np.inf]

label\_names = ['short', 'medium', 'long']

# Create wait\_type column

airlines['wait\_type'] = pd.cut(airlines['wait\_min'], bins = label\_ranges, labels = label\_names)

# Create mappings and replace

mappings = {'Monday':'weekday', 'Tuesday':'weekday', 'Wednesday': 'weekday', 'Thursday': 'weekday', 'Friday': 'weekday', 'Saturday': 'weekend', 'Sunday': 'weekend'}

airlines['day\_week'] = airlines['day'].replace(mappings)

Awesome work! You just created two new categorical variables, that when combined with other columns, could produce really interesting analysis. Don't forget, you can always use an assert statement to check your changes passed.

**Daily XP900**

# Cleaning text data

**50 XP**

## 1. Cleaning text data

Good job on the previous lesson. In the final lesson of this chapter, we'll talk about text data and regular expressions.

## 2. What is text data?

Text data is one of the most common types of data types. Examples of it range from names, phone numbers, addresses, emails and more. Common text data problems include handling inconsistencies, making sure text data is of a certain length, typos and others.

## 3. Example

Let's take a look at the following example. Here's a DataFrame named phones containing the full name and phone numbers of individuals. Both are string columns. Notice the phone number column.

## 4. Example

We can see that there are phone number values, that begin with 00 or +. We also see that there is one entry where the phone number is 4 digits, which is non-existent. Furthermore, we can see that there are dashes across the phone number column. If we wanted to feed these phone numbers into an automated call system, or create a report discussing the distribution of users by area code, we couldn't really do so without uniform phone numbers.

## 5. Example

Ideally, we'd want to the phone number column as such. Where all phone numbers are aligned to begin with 00, where any number below the 10 digit value is replaced with NaN to represent a missing value, and where all dashes have been removed. Let's see how that's done!

## 6. Fixing the phone number column

Let's first begin by replacing the plus sign with 00, to do this, we use the dot str dot replace method which takes in two values, the string being replaced, which is in this case the plus sign and the string to replace it with which is in this case 00. We can see that the column has been updated.

## 7. Fixing the phone number column

We use the same exact technique to remove the dashes, by replacing the dash symbol with an empty string.

## 8. Fixing the phone number column

Now finally we're going to replace all phone numbers below 10 digits to NaN. We can do this by chaining the Phone number column with the dot str dot len method, which returns the string length of each row in the column. We can then use the dot loc method, to index rows where digits is below 10, and replace the value of Phone number with numpy's nan object, which is here imported as np.

## 9. Fixing the phone number column

We can also write assert statements to test whether the phone number column has a specific length,and whether it contains the symbols we removed. The first assert statement tests that the minimum length of the strings in the phone number column, found through str dot len, is bigger than or equal to 10. In the second assert statement, we use the str dot contains method to test whether the phone number column contains a specific pattern. It returns a series of booleans for that are True for matches and False for non-matches. We set the pattern plus bar pipe minus, the bar pipe here is basically an or statement, so we're trying to find matches for either symbols. We chain it with the any method which returns True if any element in the output of our dot-str-contains is True, and test whether the it returns False.

## 10. But what about more complicated examples?

But what about more complicated examples? How can we clean a phone number column that looks like this for example? Where phone numbers can contain a range of symbols from plus signs, dashes, parenthesis and maybe more. This is where regular expressions come in. Regular expressions give us the ability to search for any pattern in text data, like only digits for example. They are like control + find in your browser, but way more dynamic and robust.

## 11. Regular expressions in action

Let's a look at this example. Here we are attempting to only extract digits from the phone number column. To do this, we use the dot str dot replace method with the pattern we want to replace with an empty string. Notice the pattern fed into the method. This is essentially us telling pandas to replace anything that is not a digit with nothing. We won't get into the specifics of regular expressions, and how to construct them, but they are immensely useful for difficult string cleaning tasks, so make sure to check out DataCamp's course library on regular expressions.

## 12. Let's practice!

Now that we know how to clean text data, let's get to practice!

**Daily XP950**

##### Exercise

##### Exercise

# Removing titles and taking names

While collecting survey respondent metadata in the airlines DataFrame, the full name of respondents was saved in the full\_name column. However upon closer inspection, you found that a lot of the different names are prefixed by honorifics such as "Dr.", "Mr.", "Ms." and "Miss".

Your ultimate objective is to create two new columns named first\_name and last\_name, containing the first and last names of respondents respectively. Before doing so however, you need to remove honorifics.

The airlines DataFrame is in your environment, alongside pandas as pd.

##### Instructions

**100 XP**

* Remove "Dr.", "Mr.", "Miss" and "Ms." from full\_name by replacing them with an empty string "" in that order.
* Run the assert statement using .str.contains() that tests whether full\_name still contains any of the honorifics.
* # Replace "Dr." with empty string ""
* airlines['full\_name'] = airlines['full\_name'].\_\_\_\_.\_\_\_\_("\_\_\_\_","")
* # Replace "Mr." with empty string ""
* airlines['full\_name'] = \_\_\_\_
* # Replace "Miss" with empty string ""
* \_\_\_\_
* # Replace "Ms." with empty string ""
* \_\_\_\_
* # Assert that full\_name has no honorifics
* assert airlines['full\_name'].str.contains('Ms.|Mr.|Miss|Dr.').any() == False

# Replace "Dr." with empty string ""

airlines['full\_name'] = airlines['full\_name'].str.replace("Dr.","")

# Replace "Mr." with empty string ""

airlines['full\_name'] = airlines['full\_name'].str.replace("Mr.","")

# Replace "Miss" with empty string ""

airlines['full\_name'] = airlines['full\_name'].str.replace("Miss","")

# Replace "Ms." with empty string ""

airlines['full\_name'] = airlines['full\_name'].str.replace("Ms.","")

# Assert that full\_name has no honorifics

assert airlines['full\_name'].str.contains('Ms.|Mr.|Miss|Dr.').any() == False

# Replace "Dr." with empty string "" airlines['full\_name'] = airlines['full\_name'].str.replace("Dr.","") # Replace "Mr." with empty string "" airlines['full\_name'] = airlines['full\_name'].str.replace("Mr.","") # Replace "Miss" with empty string "" airlines['full\_name'] = airlines['full\_name'].str.replace("Miss","") # Replace "Ms." with empty string "" airlines['full\_name'] = airlines['full\_name'].str.replace("Ms.","") # Assert that full\_name has no honorifics assert airlines['full\_name'].str.contains('Ms.|Mr.|Miss|Dr.').any() == False

Great work! By normalizing full names this way, you can now easily split them into first names and last names!

**Daily XP1050**

##### Exercise

##### Exercise

# Keeping it descriptive

To further understand travelers' experiences in the San Francisco Airport, the quality assurance department sent out a qualitative questionnaire to all travelers who gave the airport the worst score on all possible categories. The objective behind this questionnaire is to identify common patterns in what travelers are saying about the airport.

Their response is stored in the survey\_response column. Upon a closer look, you realized a few of the answers gave the shortest possible character amount without much substance. In this exercise, you will isolate the responses with a character count higher than **40** , and make sure your new DataFrame contains responses with **40** characters or more using an assert statement.

The airlines DataFrame is in your environment, and pandas is imported as pd.

##### Instructions

**100 XP**

* Using the airlines DataFrame, store the length of each instance in the survey\_response column in resp\_length by using .str.len().
* Isolate the rows of airlines with resp\_length higher than 40.
* Assert that the smallest survey\_response length in airlines\_survey is now bigger than 40.

# Store length of each row in survey\_response column

resp\_length = \_\_\_\_

# Find rows in airlines where resp\_length > 40

airlines\_survey = airlines[\_\_\_\_ > \_\_\_\_]

# Assert minimum survey\_response length is > 40

assert \_\_\_\_.str.len().\_\_\_\_ > \_\_\_\_\_

# Print new survey\_response column

print(airlines\_survey['survey\_response'])

# Store length of each row in survey\_response column

resp\_length = airlines['survey\_response'].str.len()

# Find rows in airlines where resp\_length > 40

airlines\_survey = airlines[resp\_length > 40]

# Assert minimum survey\_response length is > 40

assert airlines\_survey['survey\_response'].str.len().min() > 40

# Print new survey\_response column

print(airlines\_survey['survey\_response'])

# Store length of each row in survey\_response column

resp\_length = airlines['survey\_response'].str.len()

# Find rows in airlines where resp\_length > 40

airlines\_survey = airlines[resp\_length > 40]

# Assert minimum survey\_response length is > 40

assert airlines\_survey['survey\_response'].str.len().min() > 40

# Print new survey\_response column

print(airlines\_survey['survey\_response'])

18 The airport personnell forgot to alert us of d...

19 The food in the airport was really really expe...

20 One of the other travelers was really loud and...

21 I don't remember answering the survey with the...

22 The airport personnel kept ignoring my request...

23 The chair I sat in was extremely uncomfortable...

24 I wish you were more like other airports, the ...

25 I was really unsatisfied with the wait times b...

27 The flight was okay, but I didn't really like ...

28 We were really slowed down by security measure...

29 There was a spill on the aisle next to the bat...

30 I felt very unsatisfied by how long the flight...

Name: survey\_response, dtype: object

Phenomenal work! These types of feedbacks are essential to improving any service. Coupled with some wordcount analysis, you can find common patterns across all survey responses in no time!

**Daily XP1150**

# Uniformity

**50 XP**

## 1. Uniformity

Stellar work on chapter 2! You're now an expert at handling categorical and text variables.

## 2. In this chapter

In this chapter, we're looking at more advanced data cleaning problems, such as uniformity, cross field validation and dealing with missing data.

## 3. Data range constraints

In chapter 1, we saw how out of range values are a common problem when cleaning data, and that when left untouched, can skew your analysis.

## 4. Uniformity

In this lesson, we're going to tackle a problem that could similarly skew our data, which is unit uniformity. For example, we can have temperature data that has values in both Fahrenheit and Celsius, weight data in Kilograms and in stones, dates in multiple formats, and so on. Verifying unit uniformity is imperative to having accurate analysis.

## 5. An example

Here's a dataset with average temperature data throughout the month of March in New York City. The dataset was collected from different sources with temperature data in Celsius and Fahrenheit merged together. We can see that unless a major climate event occurred,

## 6. An example

this value here is most likely Fahrenheit, not Celsius. Let's confirm the presence of these values visually.

## 7. An example

We can do so by plotting a scatter plot of our data. We can do this using matplotlib.pyplot, which was imported as plt. We use the plt dot scatter function, which takes in what to plot on the x axis, the y axis, and which data source to use. We set the title, axis labels with the helper functions seen here, show the plot with plt dot show,

## 8. Insert title here...

and voila.

## 9. Insert title here...

Notice these values here? They all must be fahrenheit.

## 10. Treating temperature data

A simple web search returns the formula for converting Fahrenheit to Celsius. To convert our temperature data, we isolate all rows of temperature column where it is above 40 using the loc method. We chose 40 because it's a common sense maximum for Celsius temperatures in New York City. We then convert these values to Celsius using the formula above, and reassign them to their respective Fahrenheit values in temperatures. We can make sure that our conversion was correct with an assert statement, by making sure the maximum value of temperature is less than 40.

## 11. Treating date data

Here's another common uniformity problem with date data. This is a DataFrame called birthdays containing birth dates for a variety of individuals. It has been collected from a variety of sources and merged into one.

## 12. Treating date data

Notice the dates here? The one in blue has the month, day, year format, whereas the one in orange has the month written out. The one in red is obviously an error, with what looks like a day day year format. We'll learn how to deal with that one as well.

## 13. Datetime formatting

We already discussed datetime objects. Without getting too much into detail, datetime accepts different formats that help you format your dates as pleased. The pandas to datetime function automatically accepts most date formats, but could raise errors when certain formats are unrecognizable. You don't have to memorize these formats, just know that they exist and are easily searchable!

## 14. Treating date data

You can treat these date inconsistencies easily by converting your date column to datetime. We can do this in pandas with the to\_datetime function. However this isn't enough and will most likely return an error, since we have dates in multiple formats, especially the weird day/day/format which triggers an error with months. Instead we set the infer\_datetime\_format argument to True, and set errors equal to coerce. This will infer the format and return missing value for dates that couldn't be identified and converted instead of a value error.

## 15. Treating date data

This returns the birthday column with aligned formats, with the initial ambiguous format of day day year, being set to NAT, which represents missing values in Pandas for datetime objects.

## 16. Treating date data

We can also convert the format of a datetime column using the dt dot strftime method, which accepts a datetime format of your choice. For example, here we convert the Birthday column to day month year, instead of year month day.

## 17. Treating ambiguous date data

However a common problem is having ambiguous dates with vague formats. For example, is this date value set in March or August? Unfortunately there's no clear cut way to spot this inconsistency or to treat it. Depending on the size of the dataset and suspected ambiguities, we can either convert these dates to NAs and deal with them accordingly. If you have additional context on the source of your data, you can probably infer the format. If the majority of subsequent or previous data is of one format, you can probably infer the format as well. All in all, it is essential to properly understand where your data comes from, before trying to treat it, as it will make making these decisions much easier.

## 18. Let's practice!

Now let's make our data uniform!

# Ambiguous dates

You have a DataFrame containing a subscription\_date column that was collected from various sources with different Date formats such as YYYY-mm-dd and YYYY-dd-mm. What is the best way to unify the formats for ambiguous values such as 2019-04-07?

##### Answer the question

**50XP**

#### Possible Answers

* 

Set them to NA and drop them.

press1

* 

Infer the format of the data in question by checking the format of subsequent and previous values.

press2

* 

Infer the format from the original data source.

press3

* 

**All of the above are possible, as long as we investigate where our data comes from, and understand the dynamics affecting it before cleaning it.**

press4

+50 XP

Great work! Like most cleaning data tasks, ambiguous dates require a thorough understanding of where your data comes from. Diagnosing problems is the first step in finding the best solution!

**Daily XP1250**

##### Exercise

##### Exercise

# Uniform currencies

In this exercise and throughout this chapter, you will be working with a retail banking dataset stored in the banking DataFrame. The dataset contains data on the amount of money stored in accounts (acct\_amount), their currency (acct\_cur), amount invested (inv\_amount), account opening date (account\_opened), and last transaction date (last\_transaction) that were consolidated from American and European branches.

You are tasked with understanding the average account size and how investments vary by the size of account, however in order to produce this analysis accurately, you first need to unify the currency amount into dollars. The pandas package has been imported as pd, and the banking DataFrame is in your environment.

##### Instructions

**100 XP**

* Find the rows of acct\_cur in banking that are equal to 'euro' and store them in the variable acct\_eu.
* Find all the rows of acct\_amount in banking that fit the acct\_eu condition, and convert them to USD by multiplying them with 1.1.
* Find all the rows of acct\_cur in banking that fit the acct\_eu condition, set them to 'dollar'.

# Find values of acct\_cur that are equal to 'euro'

acct\_eu = banking['\_\_\_\_'] == '\_\_\_\_'

# Convert acct\_amount where it is in euro to dollars

banking.loc[\_\_\_\_, '\_\_\_\_'] = banking.loc[\_\_\_\_, '\_\_\_\_'] \* \_\_\_\_

# Unify acct\_cur column by changing 'euro' values to 'dollar'

banking.loc[\_\_\_\_, '\_\_\_\_'] = \_\_\_\_

# Assert that only dollar currency remains

assert banking['acct\_cur'].unique() == 'dollar'

# 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'

# 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[banking['acct\_amount'], 'acct\_amount'] = banking.loc[banking['acct\_amount'], 'acct\_amount'] \* 1.1 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'

Crafty currency conversion! With just a few lines of code, you made this column ready for analysis!

**Daily XP1350**

##### Exercise

##### Exercise

# Uniform dates

After having unified the currencies of your different account amounts, you want to add a temporal dimension to your analysis and see how customers have been investing their money given the size of their account over each year. The account\_opened column represents when customers opened their accounts and is a good proxy for segmenting customer activity and investment over time.

However, since this data was consolidated from multiple sources, you need to make sure that all dates are of the same format. You will do so by converting this column into a datetime object, while making sure that the format is inferred and potentially incorrect formats are set to missing. The banking DataFrame is in your environment and pandas was imported as pd.

##### Instructions 1/4

**25 XP**

* [1](javascript:void(0))
* [2](javascript:void(0))
* [3](javascript:void(0))
* [4](javascript:void(0))
* Print the header of account\_opened from the banking DataFrame and take a look at the different results.

# Print the header of account\_opened

print(\_\_\_\_)

##### s 2/4

**25 XP**

* [2](javascript:void(0))
* [3](javascript:void(0))
* [4](javascript:void(0))

#### Question

Take a look at the output. You tried converting the values to datetime using the default to\_datetime() function without changing any argument, however received the following error:

ValueError: month must be in 1..12

Why do you think that is?

##### Possible Answers

* 

The to\_datetime() function needs to be explicitly told which date format each row is in.

* 

The to\_datetime() function can only be applied on YY-mm-dd date formats.

* ****

**The 21-14-17 entry is erroneous and leads to an error**.

# Uniform dates

After having unified the currencies of your different account amounts, you want to add a temporal dimension to your analysis and see how customers have been investing their money given the size of their account over each year. The account\_opened column represents when customers opened their accounts and is a good proxy for segmenting customer activity and investment over time.

However, since this data was consolidated from multiple sources, you need to make sure that all dates are of the same format. You will do so by converting this column into a datetime object, while making sure that the format is inferred and potentially incorrect formats are set to missing. The banking DataFrame is in your environment and pandas was imported as pd.

##### Instructions 3/4

**25 XP**

* [3](javascript:void(0))
* [4](javascript:void(0))
* Convert the account\_opened column to datetime, while making sure the date format is inferred and that erroneous formats that raise error return a missing value.

# Print the header of account\_opened

print(banking['account\_opened'].head())

# Convert account\_opened to datetime

banking['account\_opened'] = pd.to\_datetime(\_\_\_\_,

                                           # Infer datetime format

                                           infer\_datetime\_format = \_\_\_\_,

                                           # Return missing value for error

                                           errors = \_\_\_\_)

# Print the header of account\_opened

print(banking['account\_opened'].head())

# Convert account\_opened to datetime

banking['account\_opened'] = pd.to\_datetime(banking['account\_opened'],

                                           # Infer datetime format

                                           infer\_datetime\_format = True,

                                           # Return missing value for error

                                           errors = 'coerce')

# Print the header of account\_opened

print(banking['account\_opened'].head())

# Convert account\_opened to datetime

banking['account\_opened'] = pd.to\_datetime(banking['account\_opened'],

# Infer datetime format

infer\_datetime\_format = True,

# Return missing value for error

errors = 'coerce')

0 2018-03-05

1 21-01-18

2 January 26, 2018

3 21-14-17

4 05-06-17

Name: account\_opened, dtype: object

* Convert the account\_opened column to datetime, while making sure the date format is inferred and that erroneous formats that raise error return a missing value.
* Extract the year from the amended account\_opened column and assign it to the acct\_year column.
* Print the newly created acct\_year column.

# Print the header of account\_opend

print(banking['account\_opened'].head())

# Convert account\_opened to datetime

banking['account\_opened'] = pd.to\_datetime(banking['account\_opened'],

                                           # Infer datetime format

                                           infer\_datetime\_format = True,

                                           # Return missing value for error

                                           errors = 'coerce')

# Get year of account opened

banking['acct\_year'] = banking['account\_opened'].dt.strftime('%Y')

# Print acct\_year

print(banking['acct\_year'])

# Print the header of account\_opend

print(banking['account\_opened'].head())

# Convert account\_opened to datetime

banking['account\_opened'] = pd.to\_datetime(banking['account\_opened'],

# Infer datetime format

infer\_datetime\_format = True,

# Return missing value for error

errors = 'coerce')

# Get year of account opened

banking['acct\_year'] = banking['account\_opened'].dt.strftime('%Y')

# Print acct\_year

print(banking['acct\_year'])

0 2018-03-05

1 21-01-18

2 January 26, 2018

3 21-14-17

4 05-06-17

Name: account\_opened, dtype: object

0 2018

1 2018

2 2018

3 NaN

4 2017

...

92 2017

93 2018

94 2018

95 2017

96 2017

Name: acct\_year, Length: 97, dtype: object

Cunning calendar cleaning! Now that the acct\_year column is created, a simple .groupby() will show you how accounts are opened on a yearly!

# Cross field validation

**50 XP**

## 1. Cross field validation

Hi and welcome to the second lesson of this chapter! In this lesson we'll talk about cross field validation for diagnosing dirty data.

## 2. Motivation

Let's take a look at the following dataset. It contains flight statistics on the total number of passengers in economy, business and first class as well as the total passengers for each flight. We know that these columns have been collected and merged from different data sources, and a common challenge when merging data from different sources is data integrity, or more broadly making sure that our data is correct.

## 3. Cross field validation

This is where cross field validation comes in. Cross field validation is the use of multiple fields in your dataset to sanity check the integrity of your data. For example in our flights dataset, this could be summing economy, business and first class values and making sure they are equal to the total passengers on the plane. This could be easily done in Pandas, by first subsetting on the columns to sum, then using the sum method with the axis argument set to 1 to indicate row wise summing. We then find instances where the total passengers column is equal to the sum of the classes. And find and filter out instances of inconsistent passenger amounts by subsetting on the equality we created with brackets and the tilde symbol.

## 4. Cross field validation

Here's another example containing user IDs, birthdays and age values for a set of users. We can for example make sure that the age and birthday columns are correct by subtracting the number of years between today's date and each birthday.

## 5. Cross field validation

We can do this by first making sure the Birthday column is converted to datetime with the pandas to datetime function. We then create an object storing today's date using the datetime package's date dot today function. We then calculate the difference in years between today's date's year, and the year of each birthday by using the dot dt dot year attribute of the user's Birthday column. We then find instances where the calculated ages are equal to the actual age column in the users DataFrame. We then find and filter out the instances where we have inconsistencies using subsetting with brackets and the tilde symbol on the equality we created.

## 6. What to do when we catch inconsistencies?

So what should be the course of action in case we spot inconsistencies with cross-field validation? Just like other data cleaning problems, there is no one size fits all solution, as often the best solution requires an in depth understanding of our dataset. We can decide to either drop inconsistent data, set it to missing and impute it, or apply some rules due to domain knowledge. All these routes and assumptions can be decided upon only when you have a good understanding of where your dataset comes from and the different sources feeding into it.

## 7. Let's practice!

Now that you know about cross field validation, let's get to practice!

# Cross field validation

**50 XP**

## 1. Cross field validation

Hi and welcome to the second lesson of this chapter! In this lesson we'll talk about cross field validation for diagnosing dirty data.

## 2. Motivation

Let's take a look at the following dataset. It contains flight statistics on the total number of passengers in economy, business and first class as well as the total passengers for each flight. We know that these columns have been collected and merged from different data sources, and a common challenge when merging data from different sources is data integrity, or more broadly making sure that our data is correct.

## 3. Cross field validation

This is where cross field validation comes in. Cross field validation is the use of multiple fields in your dataset to sanity check the integrity of your data. For example in our flights dataset, this could be summing economy, business and first class values and making sure they are equal to the total passengers on the plane. This could be easily done in Pandas, by first subsetting on the columns to sum, then using the sum method with the axis argument set to 1 to indicate row wise summing. We then find instances where the total passengers column is equal to the sum of the classes. And find and filter out instances of inconsistent passenger amounts by subsetting on the equality we created with brackets and the tilde symbol.

## 4. Cross field validation

Here's another example containing user IDs, birthdays and age values for a set of users. We can for example make sure that the age and birthday columns are correct by subtracting the number of years between today's date and each birthday.

## 5. Cross field validation

We can do this by first making sure the Birthday column is converted to datetime with the pandas to datetime function. We then create an object storing today's date using the datetime package's date dot today function. We then calculate the difference in years between today's date's year, and the year of each birthday by using the dot dt dot year attribute of the user's Birthday column. We then find instances where the calculated ages are equal to the actual age column in the users DataFrame. We then find and filter out the instances where we have inconsistencies using subsetting with brackets and the tilde symbol on the equality we created.

## 6. What to do when we catch inconsistencies?

So what should be the course of action in case we spot inconsistencies with cross-field validation? Just like other data cleaning problems, there is no one size fits all solution, as often the best solution requires an in depth understanding of our dataset. We can decide to either drop inconsistent data, set it to missing and impute it, or apply some rules due to domain knowledge. All these routes and assumptions can be decided upon only when you have a good understanding of where your dataset comes from and the different sources feeding into it.

## 7. Let's practice!

Now that you know about cross field validation, let's get to practice!

+100 XP

Awesome work! In the next couple of exercises, you'll be seeing some of these techniques in action!

**Daily XP250**

##### Exercise

##### Exercise

# How's our data integrity?

New data has been merged into the banking DataFrame that contains details on how investments in the inv\_amount column are allocated across four different funds A, B, C and D.

Furthermore, the age and birthdays of customers are now stored in the age and birth\_date columns respectively.

You want to understand how customers of different age groups invest. However, you want to first make sure the data you're analyzing is correct. You will do so by cross field checking values of inv\_amount and age against the amount invested in different funds and customers' birthdays. Both pandas and datetime have been imported as pd and dt respectively.

##### Instructions 1/2

**50 XP**

* [1](javascript:void(0))
  + Find the rows where the sum of all rows of the fund\_columns in banking are equal to the inv\_amount column.
  + Store the values of banking with consistent inv\_amount in consistent\_inv, and those with inconsistent ones in inconsistent\_inv.

 [2](javascript:void(0))

* Store today's date into today, and manually calculate customers' ages and store them in ages\_manual.
* Find all rows of banking where the age column is equal to ages\_manual and then filter banking into consistent\_ages and inconsistent\_ages.

# Store fund columns to sum against

fund\_columns = ['fund\_A', 'fund\_B', 'fund\_C', 'fund\_D']

# Find rows where fund\_columns row sum == inv\_amount

inv\_equ = banking[\_\_\_\_].\_\_\_\_(\_\_\_\_) == \_\_\_\_

# Store consistent and inconsistent data

consistent\_inv = \_\_\_\_[\_\_\_\_]

inconsistent\_inv = \_\_\_\_[\_\_\_\_]

# Store consistent and inconsistent data

print("Number of inconsistent investments: ", inconsistent\_inv.shape[0])

# Store fund columns to sum against

fund\_columns = ['fund\_A', 'fund\_B', 'fund\_C', 'fund\_D']

# Find rows where fund\_columns row sum == inv\_amount

inv\_equ = banking[fund\_columns].sum(axis=1) == banking['inv\_amount']

# Store consistent and inconsistent data

consistent\_inv = banking[inv\_equ]

inconsistent\_inv = banking[~inv\_equ]

# Store consistent and inconsistent data

print("Number of inconsistent investments: ", inconsistent\_inv.shape[0])

Store fund columns to sum against

fund\_columns = ['fund\_A', 'fund\_B', 'fund\_C', 'fund\_D']

# Find rows where fund\_columns row sum == inv\_amount

inv\_equ = banking[fund\_columns].sum(axis=1) == banking['inv\_amount']

# Store consistent and inconsistent data

consistent\_inv = banking[inv\_equ]

inconsistent\_inv = banking[~inv\_equ]

# Store consistent and inconsistent data

print("Number of inconsistent investments: ", inconsistent\_inv.shape[0])

Number of inconsistent investments: 8

* Store today's date into today, and manually calculate customers' ages and store them in ages\_manual.
* Find all rows of banking where the age column is equal to ages\_manual and then filter banking into consistent\_ages and inconsistent\_ages.

# Store today's date and find ages

today = dt.date.today()

ages\_manual = today.year - banking['age'].dt.year

# Find rows where age column == ages\_manual

age\_equ = banking['age'] == ages\_manual

# Store consistent and inconsistent data

consistent\_ages = banking['age']

inconsistent\_ages = \_\_\_\_

# Store consistent and inconsistent data

print("Number of inconsistent ages: ", inconsistent\_ages.shape[0])

# 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 = banking['age'] == ages\_manual

# 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])

# 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 = banking['age'] == ages\_manual

# 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])

Number of inconsistent ages: 4

Awesome work! There are only 8 and 4 rows affected by inconsistent inv\_amount and age values, respectively. In this case, it's best to investigate the underlying data sources before deciding on a course of action!

**Daily XP350**

# Completeness

**50 XP**

## 1. Completeness

Hi and welcome to the last lesson of this chapter. In this lesson, we're going to discuss completeness and missing data.

## 2. What is missing data?

Missing data is one of the most common and most important data cleaning problems. Essentially, missing data is when no data value is stored for a variable in an observation. Missing data is most commonly represented as NA or NaN, but can take on arbitrary values like 0 or dot. Like a lot of the problems that we've seen thus far in the course, it's commonly due to technical or human errors. Missing data can take many forms, so let's take a look at an example.

## 3. Airquality example

Let's take a look at the airquality dataset. It contains temperature and CO2 measurements for different dates.

## 4. Airquality example

We can see that the CO2 value in this row is represented as NaN

## 5. Airquality example

We can find rows with missing values by using the dot is na method, which returns True for missing values and False for complete values across all our rows and columns.

## 6. Airquality example

We can also chain the isna method with the sum method, which returns a breakdown of missing values per column in our dataframe. We notice that the CO2 column is the only column with missing values - let's find out why and dig further into the nature of this missingness by first visualizing our missing values.

## 7. Missingno

The missingno package allows to create useful visualizations of our missing data. Digging into its details is not part of the course, but you can also check out other courses on missing data in DataCamp's course library. We visualize the missingness of the airquality DataFrame with the msno dot matrix function, and show it with pyplot's show function from matplotlib, which returns

## 8. Insert title here...

the following image. This matrix essentially shows how missing values are distributed across a column. We see that missing CO2 values are randomly scattered throughout the column, but is that really the case? Let's dig deeper.

## 9. Airquality example

We first isolate the rows of airquality with missing CO2 values in one DataFrame, and complete CO2 values in another.

## 10. Airquality example

Then, let's use the describe method on each of the created DataFrames.

## 11. Airquality example

We see that for all missing values of CO2, they occur at really low temperatures, with the mean temperature at minus 39 degrees and a minimum and maximum of -49 and -30 respectively. Let's confirm this visually with the missngno package.

## 12. Insert title here...

We first sort the DataFrame by the temperature column. Then we input the sorted dataframe to the matrix function from msno. This leaves us with this matrix.

## 13. Insert title here...

Notice how all missing values are on the top? This is because values are sorted from smallest to largest by default. This essentially confirms that CO2 measurements are lost for really low temperatures. Must be a sensor failure!

## 14. Missingness types

This leads us to missingness types. Without going too much into the details, there are a variety of types of missing data. It could missing completely at random, missing at random, or missing not at random.

## 15. Missingness types

Missing completely at random data is when there missing data completely due to randomness, and there is no relationship between missing data and remaining values, such data entry errors.

## 16. Missingness types

Despite a slightly deceiving name, Missing at random data is when there is a relationship between missing data and other observed values, such as our CO2 data being missing for low temperatures.

## 17. Missingness types

When data is missing not at random, there is a systematic relationship between the missing data and unobserved values. For example, when it's really hot outside, the thermometer might stop working, so we don't have temperature measurements for days with high temperatures. However, we have no way to tell this just from looking at the data since we can't actually see what the missing temperatures are.

## 18. How to deal with missing data?

There's a variety of ways of dealing with missing data, from dropping missing data, to imputing them with statistical measures such as mean, median or mode, or imputing them with more complicated algorithmic approaches or ones that require some machine learning. Each missingness type requires a specific approach, and each type of approach has drawbacks and positives, so make sure to dig deeper in DataCamp's course library on dealing with missing data.

## 19. Dealing with missing data

In this lesson, we'll just explore the simple approaches to dealing with missing data. Let's grab another look at the header of airquality.

## 20. Dropping missing values

We can drop missing values, by using the dot dropna method, alongside the subset argument which lets us pick which column's missing values to drop.

## 21. Replacing with statistical measures

We can also replace the missing values of CO2 with the mean value of CO2, by using the fillna method, which is in this case 1.73. Fillna takes in a dictionary with columns as keys, and the imputed value as values. We can even feed custom values into fillna pertaining to our missing data if we have enough domain knowledge about our dataset.

## 22. Let's practice!

Now that you know how to tackle missing data, let's get started!

# Is this missing at random?

You've seen in the video exercise how there are a variety of missingness types when observing missing data. As a reminder, missingness types can be described as the following:

* ***Missing Completely at Random:*** No systematic relationship between a column's missing values and other or own values.
* ***Missing at Random:*** There is a systematic relationship between a column's missing values and other ***observed*** values.
* ***Missing not at Random:*** There is a systematic relationship between a column's missing values and ***unobserved*** values.

You have a DataFrame containing customer satisfaction scores for a service. What type of missingness is the following?   
  
                              A customer satisfaction\_score column with missing values for highly dissatisfied customers.

##### Answer the question

**50XP**

#### Possible Answers

* 

Missing completely at random.

press1

* 

Missing at random.

press2

* ****

**Missing not at random.**

press3

Awesome work! This is a clear example of missing not at random, where low values of satisfaction\_score are missing because of inherently low satisfaction!

# Missing investors

Dealing with missing data is one of the most common tasks in data science. There are a variety of types of missingness, as well as a variety of types of solutions to missing data.

You just received a new version of the banking DataFrame containing data on the amount held and invested for new and existing customers. However, there are rows with missing inv\_amount values.

You know for a fact that most customers below 25 do not have investment accounts yet, and suspect it could be driving the missingness. The pandas, missingno and matplotlib.pyplot packages have been imported as pd, msno and plt respectively. The banking DataFrame is in your environment.

##### Instructions 1/4

**25 XP**

* [1](javascript:void(0))
* [2](javascript:void(0))
* [3](javascript:void(0))
* [4](javascript:void(0))
* Print the number of missing values by column in the banking DataFrame.
* Plot and show the missingness matrix of banking with the msno.matrix() function.
* # Print number of missing values in banking
* print(\_\_\_\_)
* # Visualize missingness matrix
* \_\_\_\_
* \_\_\_\_

# Print number of missing values in banking

print(banking.isna().sum())

# Visualize missingness matrix

msno.matrix(banking)

plt.show()

# Print number of missing values in banking

print(banking.isna().sum())

# Visualize missingness matrix

msno.matrix(banking)

plt.show()

cust\_id 0

age 0

acct\_amount 0

inv\_amount 13

account\_opened 0

last\_transaction 0

dtype: int64

* Isolate the values of banking missing values of inv\_amount into missing\_investors and with non-missing inv\_amount values into investors.

# Print number of missing values in banking

print(banking.isna().sum())

# Visualize missingness matrix

msno.matrix(banking)

plt.show()

# Isolate missing and non missing values of inv\_amount

missing\_investors = \_\_\_\_

investors = \_\_\_\_

# Print number of missing values in banking

print(banking.isna().sum())

# Visualize missingness matrix

msno.matrix(banking)

plt.show()

# Isolate missing and non missing values of inv\_amount

missing\_investors = banking[banking['inv\_amount'].isna()]

investors = banking[~banking['inv\_amount'].isna()]

# Print number of missing values in banking

print(banking.isna().sum())

# Visualize missingness matrix

msno.matrix(banking)

plt.show()

# Isolate missing and non missing values of inv\_amount

missing\_investors = banking[banking['inv\_amount'].isna()]

investors = banking[~banking['inv\_amount'].isna()]

cust\_id 0

age 0

acct\_amount 0

inv\_amount 13

account\_opened 0

last\_transaction 0

dtype: int64

#### Question

Now that you've isolated banking into investors and missing\_investors, use the .describe() method on both of these DataFrames in the IPython shell to understand whether there are structural differences between them. What do you think is going on?

##### Possible Answers

* 

The data is missing completely at random and there are no drivers behind the missingness.

* 

**The inv\_amount is missing only for young customers, since the average age in missing\_investors is 22 and the maximum age is 25.**

* 

The inv\_amount is missing only for old customers, since the average age in missing\_investors is 42 and the maximum age is 59.

n [1]:

print(missing\_investors.describe())

age acct\_amount inv\_amount

count 13.000 13.000 0.0

mean 21.846 73231.238 NaN

std 1.519 25553.327 NaN

min 20.000 21942.370 NaN

25% 21.000 66947.300 NaN

50% 21.000 86028.480 NaN

75% 23.000 89855.980 NaN

max 25.000 99998.350 NaN

In [2]:

print(investors.describe())

age acct\_amount inv\_amount

count 84.000 84.000 84.000

mean 43.560 75095.273 44717.885

std 10.411 32414.506 26031.246

min 26.000 12209.840 3216.720

25% 34.000 57373.062 22736.037

50% 45.000 83061.845 44498.460

75% 53.000 94165.965 66176.803

max 59.000 250046.760 93552.690

* Sort the banking DataFrame by the age column and plot the missingness matrix of banking\_sorted.
* # Print number of missing values in banking
* print(banking.isna().sum())
* # Visualize missingness matrix
* msno.matrix(banking)
* plt.show()
* # Isolate missing and non missing values of inv\_amount
* missing\_investors = banking[banking['inv\_amount'].isna()]
* investors = banking[~banking['inv\_amount'].isna()]
* # Sort banking by age and visualize
* banking\_sorted = banking.sort\_values(by= 'age')
* msno.matrix(banking\_sorted)
* plt.show()
* n [1]:
* print(missing\_investors.describe())
* age acct\_amount inv\_amount
* count 13.000 13.000 0.0
* mean 21.846 73231.238 NaN
* std 1.519 25553.327 NaN
* min 20.000 21942.370 NaN
* 25% 21.000 66947.300 NaN
* 50% 21.000 86028.480 NaN
* 75% 23.000 89855.980 NaN
* max 25.000 99998.350 NaN
* In [2]:
* print(investors.describe())
* age acct\_amount inv\_amount
* count 84.000 84.000 84.000
* mean 43.560 75095.273 44717.885
* std 10.411 32414.506 26031.246
* min 26.000 12209.840 3216.720
* 25% 34.000 57373.062 22736.037
* 50% 45.000 83061.845 44498.460
* 75% 53.000 94165.965 66176.803
* max 59.000 250046.760 93552.690
* <script.py> output:
* cust\_id 0
* age 0
* acct\_amount 0
* inv\_amount 13
* account\_opened 0
* last\_transaction 0
* dtype: int64

Great job! Notice how all the white spaces for inv\_amount are on top? Indeed missing values are only due to young bank account holders not investing their money! Better set it to 0 with .fillna().

**Daily XP550**

##### Exercise

##### Exercise

# Follow the money

In this exercise, you're working with another version of the banking DataFrame that contains missing values for both the cust\_id column and the acct\_amount column.

You want to produce analysis on how many unique customers the bank has, the average amount held by customers and more. You know that rows with missing cust\_id don't really help you, and that on average acct\_amount is usually 5 times the amount of inv\_amount.

In this exercise, you will drop rows of banking with missing cust\_ids, and impute missing values of acct\_amount with some domain knowledge.

##### Instructions

**100 XP**

* Use .dropna() to drop missing values of the cust\_id column in banking and store the results in banking\_fullid.
* Use inv\_amount to compute the estimated account amounts for banking\_fullid by setting the amounts equal to inv\_amount \* 5, and assign the results to acct\_imp.
* Impute the missing values of acct\_amount in banking\_fullid with the newly created acct\_imp using .fillna().
* # Drop missing values of cust\_id
* banking\_fullid = banking.\_\_\_\_(subset = ['\_\_\_\_'])
* # Compute estimated acct\_amount
* acct\_imp = \_\_\_\_
* # Impute missing acct\_amount with corresponding acct\_imp
* banking\_imputed = banking\_fullid.\_\_\_\_({'\_\_\_\_':\_\_\_\_})
* # Print number of missing values
* print(banking\_imputed.isna().sum())

**Daily XP550**

# Comparing strings

**50 XP**

## 1. Comparing strings

Awesome work on chapter 3! Welcome to the final chapter of this course,

## 2. In this chapter

where we'll discover the world of record linkage. But before we get deep dive into record linkage, let's sharpen our understanding of string similarity and minimum edit distance.

## 3. Minimum edit distance

Minimum edit distance is a systematic way to identify how close 2 strings are. For example, let's take a look at the following two words: intention, and execution. The minimum edit distance between them is the least possible amount of steps, that could get us from the word intention to execution, with the available operations being

## 4. Minimum edit distance

inserting new characters, deleting them, substituting them, and transposing consecutive characters.

## 5. Minimum edit distance

To get from intention to execution,

## 6. Minimum edit distance

We first start off by deleting I from intention, and adding C between E and N. Our minimum edit distance so far is 2, since these are two operations.

## 7. Minimum edit distance

Then we substitute the first N with E, T with X, and N with U, leading us to execution! With the minimum edit distance being 5.

## 8. Minimum edit distance

The lower the edit distance, the closer two words are. For example, the two different typos of reading have a minimum edit distance of 1 between them and reading.

## 9. Minimum edit distance algorithms

There's a variety of algorithms based on edit distance that differ on which operations they use, how much weight attributed to each operation, which type of strings they're suited for and more, with a variety of packages to get each similarity.

## 10. Minimum edit distance algorithms

For this lesson, we'll be comparing strings using Levenshtein distance since it's the most general form of string matching by using the thefuzz package.

## 11. Simple string comparison

thefuzz is a package to perform string comparison. We first import fuzz from thefuzz, which allow us to compare between single strings. Here we use fuzz's WRatio function to compute the similarity between reading and its typo, inputting each string as an argument. For any comparison function using thefuzz, our output is a score from 0 to 100 with 0 being not similar at all, 100 being an exact match. Do not confuse this with the minimum edit distance score from earlier, where a lower minimum edit distance means a closer match.

## 12. Partial strings and different orderings

The WRatio function is highly robust against partial string comparison with different orderings. For example here we compare the strings Houston Rockets and Rockets, and still receive a high similarity score. The same can be said for the strings Houston Rockets vs Los Angeles Lakers and Lakers vs Rockets, where the team names are only partial and they are differently ordered.

## 13. Comparison with arrays

We can also compare a string with an array of strings by using the extract function from the process module from fuzzy wuzzy. Extract takes in a string, an array of strings, and the number of possible matches to return ranked from highest to lowest. It returns a list of tuples with 3 elements, the first one being the matching string being returned, the second one being its similarity score, and the third one being its index in the array.

## 14. Collapsing categories with string similarity

In chapter 2, we learned that collapsing data into categories is an essential aspect of working with categorical and text data, and we saw how to manually replace categories in a column of a DataFrame. But what if we had so many inconsistent categories that a manual replacement is simply not feasible? We can easily do that with string similarity!

## 15. Collapsing categories with string matching

Say we have DataFrame named survey containing answers from respondents from the state of New York and California asking them how likely are you to move on a scale of 0 to 5. The state field was free text and contains hundreds of typos. Remapping them manually would take a huge amount of time. Instead, we'll use string similarity. We also have a category DataFrame containing the correct categories for each state. Let's collapse the incorrect categories with string matching!

## 16. Collapsing all of the state

We first create a for loop iterating over each correctly typed state in the categories DataFrame. For each state, we find its matches in the state column of the survey DataFrame, returning all possible matches by setting the limit argument of extract to the length of the survey DataFrame. Then we iterate over each potential match, isolating the ones only with a similarity score higher or equal than 80 with an if statement. Then for each of those returned strings, we replace it with the correct state using the loc method.

## 17. Record linkage

Record linkage attempts to join data sources that have similarly fuzzy duplicate values, so that we end up with a final DataFrame with no duplicates by using string similarity. We'll cover record linkage in more detail in the next couple of lessons.

## 18. Let's practice!

But for now, let's clean some data using string similarity!

# Minimum edit distance

In the video exercise, you saw how minimum edit distance is used to identify how similar two strings are. As a reminder, minimum edit distance is the minimum number of steps needed to reach from **String A** to **String B**, with the operations available being:

* **Insertion** of a new character.
* **Deletion** of an existing character.
* **Substitution** of an existing character.
* **Transposition** of two existing consecutive characters.

                    What is the minimum edit distance from 'sign' to 'sing', and which operation(s) gets you there?

##### Answer the question

**50XP**

#### Possible Answers

* 

2 by substituting 'g' with 'n' and 'n' with 'g'.

press1

* 

**1 by transposing 'g' with 'n'.**

press2

* 

1 by substituting 'g' with 'n'.

press3

* 

2 by deleting 'g' and inserting a new 'g' at the end.

press4

Correct! Transposing the last two letters of 'sign' is the easiest way to get to 'sing' - in the next exercise, you'll use edit distance at scale to remap categories!

**Daily XP**

# The cutoff point

In this exercise, and throughout this chapter, you'll be working with the restaurants DataFrame which has data on various restaurants. Your ultimate goal is to create a restaurant recommendation engine, but you need to first clean your data.

This version of restaurants has been collected from many sources, where the cuisine\_type column is riddled with typos, and should contain only italian, american and asian cuisine types. There are so many unique categories that remapping them manually isn't scalable, and it's best to use string similarity instead.

Before doing so, you want to establish the cutoff point for the similarity score using the thefuzz's process.extract() function by finding the similarity score of the most distant typo of each category.

##### Instructions 1/2

**50 XP**

* [1](javascript:void(0))
* [2](javascript:void(0))
* Import process from thefuzz.
* Store the unique cuisine\_types into unique\_types.
* Calculate the similarity of 'asian', 'american', and 'italian' to all possible cuisine\_types using process.extract(), while returning all possible matches.
* # Import process from thefuzz
* \_\_\_\_
* # Store the unique values of cuisine\_type in unique\_types
* unique\_types = \_\_\_\_
* # Calculate similarity of 'asian' to all values of unique\_types
* print(process.\_\_\_\_('\_\_\_\_', \_\_\_\_, limit = len(\_\_\_\_)))
* # Calculate similarity of 'american' to all values of unique\_types
* print(\_\_\_\_('\_\_\_\_', \_\_\_\_, \_\_\_\_))
* # Calculate similarity of 'italian' to all values of unique\_types
* print(\_\_\_\_)
* # Drop missing values of cust\_id
* banking\_fullid = banking.dropna(subset= ['cust\_id'])
* # Compute estimated acct\_amount
* acct\_imp = banking\_fullid['inv\_amount'] \* 5
* # Impute missing acct\_amount with corresponding acct\_imp
* banking\_imputed = banking\_fullid.fillna({'acct\_amount':acct\_imp})
* # Print number of missing values
* print(banking\_imputed.isna().sum())

In [2]:

print(banking.shape)

(97, 5)

In [3]:

print(banking\_fullid.shape)

(88, 5)

# Drop missing values of cust\_id

banking\_fullid = banking.dropna(subset= ['cust\_id'])

# Compute estimated acct\_amount

acct\_imp = banking\_fullid['inv\_amount'] \* 5

# Impute missing acct\_amount with corresponding acct\_imp

banking\_imputed = banking\_fullid.fillna({'acct\_amount':acct\_imp})

# Print number of missing values

print(banking\_imputed.isna().sum())

cust\_id 0

acct\_amount 0

inv\_amount 0

account\_opened 0

last\_transaction 0

dtype: int64

<script.py> output:

cust\_id 0

acct\_amount 0

inv\_amount 0

account\_opened 0

last\_transaction 0

dtype: int6

Awesome work! As you can see no missing data left, you can definitely \_bank\_ on getting your analysis right!