**Cleaning Data in Python**

**Data type constraints**

# 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**

* 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())

## Instructions 2/3

## Question

By looking at the summary statistics - they don't really seem to offer much description on how users are distributed along their purchase type, why do you think that is?

### Possible answers



The user\_type column is not of the correct type, it should be converted to str.



The user\_type column has an infinite set of possible values, it should be converted to category.



The user\_type column has an finite set of possible values that represent groupings of data, it should be converted to category.

**Instructions 3/3**

* 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'].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())

# 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

* 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'].str.strip('minutes')

# 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())

**Data range constraints**

**duration station\_A\_id station\_A\_name station\_B\_id station\_B\_name bike\_id user\_type user\_birth\_year user\_gender \**

**0 12 minutes 81 Berry St at 4th St 323 Broadway at Kearny 5480 Subscriber 1959 Male**

**1 24 minutes 3 Powell St BART Station (Market St at 4th St) 118 Eureka Valley Recreation Center 5193 Subscriber 1965 Male**

**2 8 minutes 67 San Francisco Caltrain Station 2 (Townsend St... 23 The Embarcadero at Steuart St 3652 Subscriber 1993 Male**

**3 4 minutes 16 Steuart St at Market St 28 The Embarcadero at Bryant St 1883 Subscriber 1979 Male**

**4 11 minutes 22 Howard St at Beale St 350 8th St at Brannan St 4626 Subscriber 1994 Male**

**... ... ... ... ... ... ... ... ... ...**

**25755 11 minutes 15 San Francisco Ferry Building (Harry Bridges Pl... 34 Father Alfred E Boeddeker Park 5063 Subscriber 2000 Male**

**25756 10 minutes 15 San Francisco Ferry Building (Harry Bridges Pl... 34 Father Alfred E Boeddeker Park 5411 Subscriber 1998 Male**

**25757 14 minutes 15 San Francisco Ferry Building (Harry Bridges Pl... 42 San Francisco City Hall (Polk St at Grove St) 5157 Customer 1995 Male**

**25758 14 minutes 15 San Francisco Ferry Building (Harry Bridges Pl... 42 San Francisco City Hall (Polk St at Grove St) 4438 Customer 1995 Male**

**25759 29 minutes 16 Steuart St at Market St 115 Jackson Playground 1705 Customer 1990 Male**

**tire\_sizes**

**0 27.0**

**1 26.0**

**2 26.0**

**3 29.0**

**4 27.0**

**... ...**

**25755 29.0**

**25756 26.0**

**25757 29.0**

**25758 26.0**

**25759 27.0**

**[25760 rows x 10 columns]**

# Tire size constraints

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

* 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())

# 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

* 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.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())

2023-05-30

# 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

* 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 = ride\_sharing.duplicated(subset='ride\_id', keep=False)

# Sort your duplicated rides

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

# Print relevant columns of duplicated\_rides

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

# 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

* 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 = 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(by='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